Behaviour-Oriented Concurrency

Luke Cheeseman

Department of Computing
Imperial College London
August 2024

Supervised by Professor Sophia Drossopoulou

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in Computing of Imperial College London and the Diploma of Imperial College London

Statement of Originality

I have collaborated closely with Microsoft/Azure Research and Uppsala University to research Behaviour-Oriented Concurrency (BoC) as part of Project Verona[1].

This thesis, whilst written by me, is the result of many discussions with Matthew Parkinson, Sophia Drossopoulou, Sylvan Clebsch, Marios Kogias, David Chisnall, Tobias Wrigstad, and Paul Liétar.

Where work is not my own, it is appropriately referenced. I will now summarise the influences in my thesis chapter-by-chapter:

- Chapter 3: This chapter was written together as the introductory chapter to our co-authored paper[1]. An initial outline was proposed by Paul Liétar and I have refined it from thereon.
- Chapter 4: This chapter was also collaboratively written as part of our paper[1]. Sophia Drossopoulou and Paul Liétar constructed an early proposal for the operational semantics for BoC. I began with these semantics and heavily revised them, and the final version that appears in this thesis was proposed by Matthew Parkinson to streamline and simplify my operational semantics.
- Chapter 5: This chapter takes inspiration from Dinsdale-Young et al. [2], a suggestion made by the co-author Matthew Parkinson. The contents of this chapter are entirely my own work that has been shared and discussed with those mentioned earlier.
- Chapter 6: This chapter takes inspiration from Alglave et al. [3] as suggested by Matthew Parkinson and Sylvan Clebsch. The contents of this chapter are entirely my own work that has been shared and discussed with those mentioned earlier.
- Chapter 7: The C++ runtime implementation of BoC is attributed entirely to Microsoft Research (I have made contributions to this runtime). The quantitative evaluation is a rework of the evaluation chapter to our co-authored paper[1], which was a collaboration between Matthew Parkinson, Marios Kogias and myself. To break this down further: I ported the Savina benchmarks to BoC, constructed the `ReasonableBanking` benchmark, and identified behaviour-ordering patterns in the dining philosophers that lead to the dining philosophers scaling benchmark. The performance data was collected by Matthew Parkinson and we collaborated to identify the cause of performance characteristics. The qualitative analysis is entirely my own work.

[1] https://www.microsoft.com/en-us/research/project/project-verona/

Acknowledgements

I extend my gratitude to my dedicated supervisor, Sophia Drossopoulou, for her unwavering support, endless time, and wealth of knowledge. I also extend my thanks to Matthew Parkinson, whose reservoir of expertise not only answered my questions but also enriched my research with many more questions in return. Their contributions have been instrumental in shaping this endeavour.

I am grateful for the privilege of collaborating with numerous individuals throughout my research; Sylvan Clebsch, Marios Kogias, David Chisnall, Wes Filardo, Tobias Wrigstad, Elias Castegren, Ellen Arvidsson, and Paul Liétar have week-on-week provided discussion, expertise and support.

I am grateful to my examiners, Alastair Donaldson and Elisa Gonzalez Boix, for their insightful feedback and amendments.

I thank also the many friends and family who made it their purpose to lure me outdoors, away from my desk. Their unwavering efforts not only added joy to my life but also played a crucial role in maintaining my balance throughout this journey. Their support has been a vital part of my success.

Copyright

Abstract

Behaviour-oriented concurrency (BoC) is a novel concurrency paradigm which provides programmers with expressive synchronisation whilst avoiding deadlocks and data-races. This thesis focuses on the design of BoC and how it provides isolated and ordered behaviours.

BoC is based on *behaviours* and *concurrent owners*. Concurrent owners provide exclusive access to mutable resources. Behaviours are units of work and are dispatched based on the availability of one or more concurrent owners, providing a simple mechanism to coordinate access to multiple resources. In this thesis, I explore the design of BoC by way of examples, and establish an operational semantics which captures how BoC can enrich an underlying language with concurrency.

BoC guarantees behaviours are deadlock free, data-race free and atomic through behaviour isolation; to ensure this, the underlying language must satisfy key isolation properties. In this thesis, I define these isolation properties which constitute the interface between BoC and the underlying language. Using these properties, I prove that BoC provides atomic behaviours.

BoC behaviours are implicitly ordered based on spawn order and cown acquisition; if one behaviour is spawned after another and the behaviours require any of the same cowns, then the behaviour spawned first will execute first. This ordering allows programmers to construct ordered concurrency without introducing their own control-flow constructs.

In this thesis, I construct an axiomatic model of BoC to explore the design of behaviour ordering. This model allows us to judge the validity of candidate executions, independent of the intricacies of the mechanisms used to execute a program (*e.g.,* an operational semantics). I compare this axiomatic model and the operational semantics to investigate the two models' expressiveness, finding that the operational semantics is slightly too restrictive.

To demonstrate the expressiveness and applicability of BoC, I have implemented well-known concurrency challenges and benchmarks using the paradigm.

Contents

List of Figures

List of Tables

Listings

List of Theorems

1 Introduction

In this thesis I explore the novel concurrency paradigm Behaviour-oriented Concurrency (BoC). I investigate the semantics of this paradigm, the design space for the interface between the paradigm and a programming language, the ordering inherent in the concurrency and when a programmer may want more ordering, and the expressivity of the paradigm. This paradigm is part of the larger project Verona, which is being developed by Azure Research[4].

What is BoC? The BoC programming model relies on a decomposition of state that is akin to actors—a program's state is a set of resources (called *cowns*). The term "cown" is a portmanteau of concurrent owner. In the concurrent setting of BoC, there is a single entry point to data: the owner.

Behaviours are asynchronous units of work that explicitly state their required set of cowns. Behaviours execute with exclusive access to their required cowns. Thus, cowns serve the dual role of data owner and subject of coordination; together, these roles provide data-race freedom for BoC.

A key feature of BoC is that *behaviours can access multiple cowns.* Thus, operations that require synchronous access to multiple cowns can be easily expressed. This distinguishes BoC from actors where behaviours can access only one resource.

To construct those behaviours, a new keyword is used, **when**, which enumerates the set of necessary cowns for the said behaviour and spawns an asynchronous unit of compute. So, a BoC program is a collection of behaviours. Each acquires zero or more cowns, performs computation on them, which typically involves reading and updating them, and spawning new behaviours, before releasing the cowns.

Listing 1.1: A transfer using BoC

```
1 transfer(src: cown[Account], dst: cown[Account], amount: U64) {
2   when (src, dst) { // withdraw and deposit
3     src.balance -= amount;
4     dst.balance += amount;
5   }
6 }
```

Listing 1.1 is a simple bank transfer example that demonstrates behaviour-oriented concurrency. The **when** expression creates the synchronisation between two bank accounts, `src` and `dst`, and allows a behaviour to transfer money from one account to the other.

Despite its apparent simplicity, this model has considerable expressive power. A wide range of concurrent schedules can be achieved by simply nesting behaviours and/or sequencing them in combination with the cowns they require.

While the design of BoC is deceptively simple, the notions of concurrency and isolation are of particular interest to the programmer who wants to create programs using BoC, and the language designer who wants to adopt BoC in their language. If the programmer wants to understand BoC programs, which includes when a behaviour will execute, then they need better understanding of the provided concurrency. If the language designer wants behaviours to be isolated from one another whilst they execute, then they need better understanding of the properties that are required by a programming language.

Concurrency Parallelism and coordination are central for concurrent programming. Parallelism empowers programs to perform multiple tasks simultaneously, while coordination provides them with the control necessary to exclude the concurrent schedules that do not meet our desired outcomes.

BoC recognises the interdependence of parallelism and coordination, and achieves both through a single, powerful primitive (namely **when**). The implicit parallelism that is inherent in a concurrent program stems from the choice of how to coordinate behaviours. Thus, to understand the parallelism in programs, better understanding of behaviour coordination is required.

Listing 1.2: A collection of behaviours to coordinate

```
1  when (src) { /* b1: deposit */ };
2  when (dst) { /* b2: freeze */ };
3  when (src, dst) { /* b3: transfer */ };
4  when (dst) { /* b4: withdraw */ };
```

Consider Listing 1.2. Which of these behaviours can run in parallel? What are the options in the design space? Perhaps it is clear that `b1` and `b2` can run in parallel but neither can run in parallel with `b3`, but by what mechanism is that achieved? Moreover, should there be any order in their execution? Should `b3` be allowed to execute before `b2`? Should `b4` be allowed to execute before `b3`?

To demonstrate how BoC provides concurrency to an underlying language, and ensures that a running behaviour has unique access to the cowns it requires, I construct a semantic model of BoC assuming an axiomatically defined abstract underlying language.

To explore the coordination in BoC, I take inspiration from the weak memory models literature and construct a causal *happens before* relation. This relates which behaviours must happen before which other behaviours. I explore the choices in the design of this relation and the ramifications on parallelism that each choice has. I also investigate scenarios where a programmer may want more than the ordering provided by BoC, and take inspiration from the existing work on promises to explore how to provide more ordering.

Isolation BoC ensures data-race free behaviours by coordinating access to shared cowns. However, data-race freedom requires that behaviours, and therefore cowns, are isolated from one another.

To achieve this behaviour isolation, BoC needs to be composed with an underlying programming language that provides certain properties. I will explore the interface between BoC and an underlying programming language, and the isolation properties which constitute the interface. I have chosen to investigate an interface, instead of a concrete programming language, as it demonstrates how BoC can be composed with a number of programming languages.

In Chapters 4 and 5, I will look at what a cown is, and the conditions required to execute a behaviour and ensure that it remains isolated. These chapters will demonstrate that `b1` and `b2` can execute in parallel,

as long as `src` and `dst` are different cowns.

1.1 When is BoC the tool to use?

Like any tool, there are times when BoC is the right fit, and times when BoC is not the right fit. Here I will discuss *some* programming patterns and broadly categorise them in those where, BoC is optimal, or just effective, or suboptimal. I will revisit these categories in Chapter 7 to more deeply investigate when BoC works well or not.

Optimal: atomic operations over multiple cowns BoC shines when we want to perform *atomic* operations over multiple cowns (as in the bank accounts example of Listing 1.1).

Historically, BoC was created as part of Project Verona to address exactly these kinds of problems in the Confidential Consortium Framework (CCF) [5]. The CCF developers wanted to easily write many atomic database-like operations, and have them effortlessly execute in parallel.

Other concurrency paradigms can achieve such atomic operations which can execute in parallel, but I feel BoC has been constructed to solve this problem very elegantly. BoC avoids the issues of: scaling work to meet available parallelism, deadlocks and data-races as encountered with locks & threads; repeating operations and rollbacks in the presence of conflict as encountered with transactions; constructing necessary communication to coordinate operations over multiple actors as encountered with actor model programming.

BoC creates a concise and simple model for the programmer to write atomic and parallel operations, as I demonstrate in Chapter 3.

Effective: coordinated access to cowns BoC is effective at coordinating exclusive access to cowns, even when operations do not need to atomically update multiple cowns. Moreover, BoC guarantees an implicit ordering to these accesses which can make some programming patterns simpler.

Consider again Listing 1.2. Recall that the deposit (`b1`) and freeze (`b3`) must not run in conjunction with other behaviours that use `src` and `dst`, and they must happen before the transfer (`b3`). Furthermore, deposit and freeze should be able to run in parallel.

Ensuring the mutually exclusive access to `src` is a property which can be achieved by many concurrency paradigms. Achieving this alongside the parallelism and ordering requirements is not so common nor simple. In Chapter 2, I will discuss other paradigms which can achieve combinations of these requirements but BoC provides a concise package for achieving all of them.

Leveraging behaviour ordering may require adjustment for the programmer new to BoC, but ordering can be a very useful tool. In Chapter 7, I will demonstrate design patterns using ordering in BoC to implement known programming patterns such as divide-and-conquer.

Suboptimal: hand-over-hand "locking" A programmer may find it difficult to achieve hand-over-hand "locking" of fine-grained cowns in BoC. This is due to the fundamental choice that behaviours cannot acquire more cowns during their execution. If a programmer requires a behaviour to have *more* cowns, then they must spawn a new behaviour that requires all of the cowns acquired so far, *as well as* the additional necessary cowns. The original behaviour will release all cowns as it terminates, and the new behaviour will acquire the additional cowns when it starts. As such, the programmer must then also be aware of how this new behaviour will be ordered with respect to other behaviours on the same cowns.

In Chapter 7, I will examine why this programming task is difficult to accomplish using BoC.

1.2 Content and Contribution

In this thesis I make the following contributions:

Chapter 2 This chapter provides a background in concurrency, reasoning about concurrent programs, and challenges in concurrency. This includes a survey of the relevant literature and how it relates to BoC.

Chapter 3 This chapter provides an introduction to behaviour-oriented concurrency informally and illustrates it through examples.

Chapter 4 This chapter introduces a semantic model for behaviour-oriented concurrency. I investigate how BoC brings concurrency to an underlying language and how behaviours are coordinated.

Chapter 5 This chapter dives further into the interface between BoC and an underlying programming language. I state the guarantees that BoC can provide and the isolation properties that the programming language must provide to acquire the guarantees.

Chapter 6 This chapter investigates the construction of an axiomatic model for BoC; focussing on the causal "happens before" order for behaviours. Using this model I investigate the design space for alternative causal orders.

Chapter 7 This chapter evaluates BoC. I evaluate the BoC design through a sequence of concurrency case studies, a benchmark suite comparing performance of Verona's BoC implementation with actor based programming, and the discussion of some scenarios for which BoC is not the optimal fit.

Chapter 8 This chapter looks at open research questions in BoC, including: instantiating our model with the co-designed type system for Verona (namely Reggio); and investigating the design and use of read-only cowns.

Chapter 9 This chapter concludes the thesis.

2

Background

In this chapter I will explore the background and literature for BoC, focussing on existing concurrency paradigms, techniques for modelling concurrency, and concurrency case-studies.

I focus on these topics to place BoC relative to other concurrency paradigms, to introduce methodologies I have used to build BoC, and discuss the examples and benchmarks that I have used to evaluate BoC.

Concurrency paradigms I will provide an overview of some notable related concurrency paradigms, and how their features relate to BoC. This discussion includes threads & mutexes, transactions, actors, and the join calculus. This will help to highlight the key design decisions in BoC in comparison with existing programming languages, influencing the design and semantics of BoC discussed in Chapters 3 and 4.

Modelling concurrency I will discuss techniques for modelling concurrent programs and how why they are helpful in modelling BoC. This discussion includes separation logic and views frameworks for modelling state separation; this has influenced my research in demonstrating BoC as data-race free in Chapter 5. This discussion also includes weak memory models for validating execution traces, which has influenced my research in reasoning about BoC execution and causal orders in Chapter 6.

Concurrency case-studies I will undertake a preliminary investigation of the expressiveness and ergonomics of BoC. For this I apply BoC to a collection of case-studies which are explored in Chapter 7.

This collection includes: Savina, the actor model benchmark suite; the dining philosophers problem; the fibonacci sequence through divide-and-conquer programming; channels for inter-behaviour communication; barrier communication; the Santa problem; the Boids program; binary search trees; and building BoC using Rust.

2.1 Concurrency Paradigms

Concurrent programming is a vast topic, and the numerous concurrency paradigms employ different approaches for parallelising and coordinating work. In this section, I will discuss threads and mutexes, transactions, actors and the join calculus.

I focus on these paradigms for the following reasons: they permit exclusive access to shared resources in a concurrent program; they achieve concurrency and exclusive accesses via notably different designs; and, their designs can be compared with the design of BoC. This criteria means I do not discuss paradigms such as SIMD, where there is ideally no shared resources requiring exclusive access, nor do I discuss paradigms such as fork-join, which does not differ from threads & locks in a way that motivates this background. I order the discussion of paradigms in this section, with the intention of transitioning from the most well-known to the least.

To discuss these paradigms and their designs, I will illustrate a scenario representative of tasks I want to achieve using BoC, and implement this scenario in each paradigm.

2.1.1 Bank Transfer

The bank transfer example introduced in Listing 1.1 will be our recurring representative scenario.

I want the following criteria for the bank transfer scenario. There must be two bank accounts which have a balance, and from which funds can be withdrawn and deposited into. I must be able to transfer money from one account and into the other account. The bank accounts must be shared resources accessible by units of concurrent execution. The accesses to the bank accounts must be data-race free and, moreover, the transfer operation must be atomic.

2.1.2 Threads and Mutexes

Perhaps the most recognisable and familiar paradigm comes in the form of threads and mutexes[6]. I begin with this paradigm as almost every programmer will be familiar with it.

The unit of parallelism is a thread; a program can have many threads, each following its own flow of execution. A programmer creates and destroys threads in their program and enjoys the abstraction that all threads could be executing at once.

Threads, typically, interact with one another through shared memory. Execution becomes difficult to understand and error-prone without a means to coordinate, or synchronise, access to shared memory. Mutexes prove a means to achieve coordination by mutual exclusion. When a thread wants access to a resource it acquires the associated lock, uses the resource and releases the lock. These thread and lock primitives are expected to be cheap and lightweight, costing little runtime overhead to a programmer. Listing 2.1 expresses the banking example from Listing 1.1 in terms of C++ threads and locks.

Listing 2.1: Example of threads and locks

```
1  // transfer money between two shared bank accounts
2  void transfer(shared_ptr<Account> acc1, shared_ptr<Account> acc2, int amount) {
3      // acquire the locks for both accounts
4      acc1->lock();
5      acc2->lock();
6
7      // perform the transfer
8      acc1->withdraw(amount);
9      acc2->deposit(amount);
10
11     // release the locks in reverse order
```

```
12      acc2->unlock();
13      acc1->unlock();
14  }
15
16  int main() {
17      // create two bank accounts
18      auto acc1 = make_shared<BankAccount>(500);
19      auto acc2 = make_shared<BankAccount>(300);
20
21      // transfer the money in a separate thread
22      thread t0 ([acc1, acc2] { transfer(acc1, acc2, 100); });
23
24      // wait for the thread to finish
25      t0.join();
26  }
```

In our example, each account has an associated lock. To ensure data-race free access to the accounts, all uses of an account must adhere to the same access pattern. Namely, lock the account lock, use the account, unlock the account lock (Line 4 - Line 13). If an account's lock is already locked, the thread trying to lock the account is blocked until the lock is unlocked.

Locks are an imperative form of data coordination. A programmer expresses where and when locking occurs in their program. Alone, locking does not guarantee freedom from data races. In C++ the onus is on the programmer to ensure their locking convention avoids data races. However, locking can be paired with a type system that ensures locks own their contents, as in Rust[7] or Boyapati et al. [8]. This ownership means that only through the lock can access be acquired to the contents, enforcing the locking convention. Neither Rust nor C++ guarantee freedom from deadlocks or livelocks, preservation of these properties is entirely the programmer's responsibility. However, deadlock avoidance algorithms provided by programming libraries exist as a convenience to programmers. One example of this is C++'s `std::lock`, which will attempt to lock an arbitrary number of locks, through an unspecified series of calls to `try_lock`, `lock` and `unlock`.[1]

Monitors[6] build on locks by attaching intrinsic locks to objects; a thread must acquire the intrinsic lock before calling methods that require coordination. These make a programmer's life easier, less coordination code is required and all users of an object automatically adhere to the same access patterns. These are still locks and, therefore, have many of the same pros and cons. Deadlocking is still a concern, a new concern is that a programmer must decide whether an object requires a monitor lock and which methods require the lock. If they overuse the lock then a program's performance suffers, if they do not use the lock then data races become a concern. A useful take away from this data structure is a coordination design that does not require explicit coordination in code.

Signals[6] are a mechanism that allows threads to communicate. Multiple threads can await a signal, blocking until another thread sends them the signal. This is a useful design pattern to avoid threads taking processing time whilst idle. I have a similar concern in BoC: I do not want to repeatedly reschedule behaviours in the hope that some state change has occurred; I would prefer to only run a behaviour when necessary.

Relation to BoC Threads and locks provide a means for creating parallel programs and coordinating data access. The imperative design, together with a permissive type system as in C++, allow much flexibility in how programs can be constructed. However, this flexibility has outcomes I do not find desirable; data-races and deadlocks are concerns from which I want to free a programmer in BoC.

[1] `https://en.cppreference.com/w/cpp/thread/lock`

2.1.3 Transactions

Software transactional memory is a means to achieve atomic operations on shared objects[9][10]. I include transactions next as they are a well-known paradigm which provide a sequential abstraction to concurrency. Transactions allow programmers to easily acquire access to multiple resources to create atomic operations, whilst avoiding the potential for deadlock. I start the discussion on transactions by describing *optimistic* transactions and then look at *pessimistic* transactions.

Shared objects are encoded as transactional objects and transactions coordinate access to these objects. Only during a transaction can these objects be manipulated. However, whilst the transaction provides a sequential abstraction to the programmer, the execution is more complicated. A transactional thread manipulates a transactional object as follows:

1. A thread begins a transaction.
2. The thread attempts to *open* a transactional object.
3. If the object has not already been opened in a conflicting way, e.g. no other thread has opened the object for writing, the thread gets a *copy* of the object.
4. If the object has been opened in a conflicting way, then the thread is *denied* access.
5. Once computation finishes, a thread attempts to *commit* a transaction. Committing can succeed or fail (e.g. attempt to access an object was denied) and the thread can react accordingly. Successfully committing a transaction makes the updates to used transactional objects globally visible.
6. A thread can also choose to *abort* a transaction, at which point all changes are discarded.

A transactional thread can attempt to open as many objects as it wants (steps 2-4 in the above). The copied version of a transactional object is only visible, and makes sense, during a transaction. The changes to a transactional object are not visible outside of a transaction until the transaction commits.

An example of a transaction can be seen in Listing 2.2, using Java-like syntax described by Herlihy et al. [9].

Listing 2.2: Example of transactions

```
1  public bool transfer(TMObject account1, TMObject account2, int amount) {
2     TMThread thread = (TMThread) Thread.currentThread();
3     while (true) {
4        thread.beginTransaction();
5        try {
6           // get a copy of the bank accounts
7           Account acc1 = (Account) account1.open(WRITE);
8           Account acc2 = (Account) account2.open(WRITE);
9
10          if (acc1.withdraw(amount)) {
11             // if the withdraw from acc1 succeeds then deposit the amount in acc2
12             acc2.deposit(amount);
13          } else {
14             // otherwise abort and return failure
15             thread.abortTransaction();
16             return false;
17          }
18       } catch (Denied d) {
19          // if either object could not be opened then a Denied exception is thrown
20       }
21
```

```
22          // attempt to commit the transaction, if it fails then retry the transfer
23          if (thread.commitTransaction())
24              return false;
25      }
26  }
```

The example using transactions attempts to atomically transfer money between two accounts. Line 7 and Line 8 open the accounts for writing returning a copy on success; the accesses are defined if they conflict with another open instance. The transfer is then performed on Line 10 and Line 12. The transfer is not visible outside of this thread until the transaction commit succeeds, on Line 23.

Transactions allow us to program without locking coordination. Transactions atomically update multiple objects at once. This allows us to use existing functionality of objects to achieve more complicated tasks in a concurrent setting, such as composing `withdraw` and `deposit` to perform `transfer`. Transactions do not suffer from deadlocks and data-races as they only work on local copies of objects. Without deadlocks and data-races, transactions provide a simple mechanism for programming without managing coordination.

However, there are drawbacks to their use. One drawback is the memory overhead; a new copy of an object is made each time it is opened. Although deadlocks and data-races are not issues, another drawback is that there are no guarantees of non-interference. Consider two transfers occurring at the same time, one from `acc1` to `acc2` and another from `acc2` to `acc1`. Each thread could attempt to open their sender at the same time and then their receiver. Opening the receiver would fail as it would conflict with the previous open. This would mean that the transaction could repeatedly fail and restart; no deadlock occurs but no progress is made, otherwise known as a livelock.

More seriously, not all effects of a transaction can be reversed, e.g. I/O. As a result [11] propose *irrevocable*, or *pessimistic*, transactions, i.e. transactions which cannot be rolled back, with the restriction that at most one such transaction may be running at a time – thus reducing parallelism. A similar approach is taken by Harris et al. [12]. Fully pessimistic transactions[13] expand on irrevocable transactions, eliminating the need for any roll-backs; however, they require either a single writing transaction (with versioning) or allowing for potential deadlock.

Considerable effort has been devoted to conflict avoidance and resolution in transactions[9, 14, 15]. Ordering transactions can be used to alleviate contention and conflict; even better, ordering can improve overall parallelism and throughput[15]. Ordering also appears in deterministic databases systems where it is not enough that transactions will commit in some order, but they must commit in a single predetermined order[16]. BoC implements out of the box causal ordering of behaviours.

I desire some of the properties of transactions for our behaviours. I want behaviours that are atomic like committed transactions[9]. I also want deadlock and data-race freedom. However, I do not want to pay the cost of memory overhead. Furthermore, I aim to avoid the issue of livelocking transactions. I avoid this kind of interference as behaviours acquire all of the required cowns at once.

Furthermore, the space for handling nested transactions is vast and seemingly unresolved[17, 18]. Contrast this with BoC, which provides a single clear semantics for nested **`when`**s, that is to say they spawn a new behaviour. I will cover this in detail for BoC in Chapter 3.

Relation to BoC In contrast to BoC behaviours, transactions are not units of concurrent execution, rather they coordinate concurrent execution, e.g., threads. As such, and in contrast to BoC, they do not specify an order of execution. Moreover, nesting in transactions is synchronous, and therefore roll-back of the enclosing and the nested transaction are intertwined, leading to a large design space, with implications for the semantics and the implementations [18]. I will discuss in Listing 3.3, how nested behaviors differ in BoC.

In summary, BoC and transactions are at different ends of the design spectrum when it comes to how much explicit work is required: *BoC is explicit*, in that it enlists the help of the type system to achieve

isolation, and requires the programmer to declare upfront which cowns are required by a behaviour; I will discuss this in Chapters 3 and 5. *Transactions are implicit*, on the other hand, as the programmer need only declare where a transaction begins and ends, and all the scheduling and conflict resolution is expected to be done by the implementation. The implicit nature of the transactions model makes programming easier, at the expense of more demands on the implementation, and more complex semantics.

2.1.4 Actors

Next I discuss the actor model paradigm. BoC began as an evolution of the actor model paradigm, designed to tackle the complications of atomic operations over multiple actors. As a result, I believe the actor model is the closest paradigm to BoC, and thus the paradigm merits inclusion in this thesis. I will demonstrate in Chapter 7, that it is possible to obtain an actor paradigm from BoC, by following a few programming conventions.

The defining feature of actor model programming is that computation proceeds via communication[19, 20]. Tasks, the computation, in an actor system are constructed from:

1. A tag, which uniquely identifies a task in the system.
2. A target, the mail address to which a communication is sent.
3. A communication, the information that is conveyed to the target.

The targets of a communication are actors. Actors have a mail queue, to which they append incoming messages and from which they remove messages for processing. Actors also have behaviours, defining what the actor will do upon processing messages. The state of an actor is known only by the actor; any change in state of actor must be the result of that actor processing a message.

The actor model's design allows for extremely concurrent systems. In a threaded system threads are the unit of concurrency and the resource for work; this means that a thread is performing a continuous flow of execution. A thread continues to execute until it must wait to synchronise or due to some time sharing resource management. With an actor system, behaviours are typically short-lived responses to an incoming message. Actors do not wait to synchronise with other actors. Once a message has been processed, any processing resources used during a behaviour become available.

Consider a naive use of actors to construct the banking example in Listing 2.3. This example has been written in Pony, an actor-model language with deny-capabilities[21].

Listing 2.3: Naive example using actors in Pony

```
1  actor Account
2     var _balance: U64
3
4     new create(balance: U64) =>
5        _balance = balance
6
7     // a behaviour that deposits amount into the account
8     be deposit(amount: U64) =>
9        _balance = _balance + amount
10
11    // a behaviour to try to withdraw amount from account
12    // – if the balance is large enough call the continuation with amount
13    // – otherwise the continuation is not called
14    be withdraw(amount: U64, continuation: {(U64)} val) =>
15       if amount < _balance then
16          _balance = _balance - amount
```

```
17            continuation(amount)
18        end
19
20  actor Main
21      // withdraw amount from account1 with a continuation that deposits
22      // the amount into account2
23      fun transfer(acc1: Account, acc2: Account, amount: U64) =>
24          acc1.withdraw(amount, {(amount: U64) =>
25              acc2.deposit(amount)
26          })
27
28      new create(env: Env) =>
29          var acc1 = Account.create(500)
30          var acc2 = Account.create(200)
31
32          transfer(acc1, acc2, 100)
```

In this example our bank accounts are represented by actors. Depositing to and withdrawing from an account are behaviours of an account, see Line 8 and Line 14 respectively. Behaviours cannot return a result; withdrawing from an account also requires a continuation that executes only if a withdrawal is successful. To transfer between two accounts, `acc1` and `acc2`, requires sending a `withdraw` message to `acc1` and a `deposit` message to `acc2`, see Line 23 - Line 26. This transfer operation is not atomic nor are `acc1` and `acc2` held exclusively during the transfer. Importantly, `acc2` is free to process a `withdraw` message before the amount from `acc1` has been deposited. The program in Listing 2.3 does not have the same semantics as the program using BoC in Listing 1.1. The transfer in Listing 1.1 is atomic; another thread cannot read the two accounts and see that the money has left the source but not entered the destination. In Listing 2.3 this scenario is entirely possible. Furthermore, if I added `acc2.withdraw(100)` after the transfer, on Line 32, the withdraw could happen before the transfer deposits funds into `acc2`. Again, this is unlike Listing 1.1.

This is problematic as the interleaving of the actors processing messages changes the outcome. Whilst the execution of a behaviour in one actor is atomic, I cannot create an atomic operation that has access to the state of two actors. If the programmer wants to ensure that the actors do not interleave messages from different operations, then they need to create more communication and synchronisation. One approach to provide more synchronisation is to implement distributed locking, another approach is to use *two-phase commit*[22] (I discuss this approach more in Section 2.3.1).

Order in Actors

Execution ordering guarantees are valuable to programmers as they allow programmers to reason about their programs. I will demonstrate programs which rely on ordering in Chapter 7 (*e.g.,* Section 7.1.6).

The numerous actor programming languages incorporate various message ordering guarantees in their designs. In this section I will look at the range of messaging guarantees that are provided by actor programming languages.

Causal ordering Pony guarantees causal messaging[23]: [2]

> Given two actors, actor A and actor B. All messages sent from actor A will arrive at actor B in the order they were sent by Actor A and will be processed by Actor B in the same order.
>
> Causal ordering of messages applies to a chain of message sends as well. If actor A sends M1 to B, and after that, sends M2 to C, then the messages at B and C can run in any order and in

[2] `https://www.ponylang.io/faq/runtime/`

> parallel. However, if C sends a message to B, even through another chain of actors, then it will always arrive after M1. [24]

Thus as the messages are received in the sent order, the behaviours to process the messages will execute in the same order. This reflects the programmer-written order of message sends in the program.

Message delivery guarantee Erlang, Akka, and AmbientTalk provide the same message delivery guarantee [25]. [3]

> Signals between two processes are guaranteed to arrive in the order they were sent. In other words, if process A sends signal 1 and then 2 to process B, signal 1 is guaranteed to arrive before signal 2. [26]

However, note that Erlang (and other actor programming languages) also provide *selective receive* which means the receiver can choose which messages to process; thus ignoring the above delivery guarantee. This feature provides a tool for programmers when their actors are not ready to receive certain messages. Consider the two-phase commit; an actor can only be in one transaction at a time, and thus it does not make sense for the actor to receive any new transactions in the mean time. Moreover, we can build this pattern in any actor model language by creating a buffer per actor for messages to *actually* process later (I will demonstrate this later in `ReasonableBanking` Section 7.1.6).

No guarantees SALSA does not provide any message delivery guarantees or ordering of behaviours, however it does provide a mechanism for programmers to achieve order themselves.

> A message to an actor may contain a *customer* actor to which the token should be sent after message processing. SALSA allows the programmer to specify the customer, the message to send to such customer and the position of the token in the arguments of such customer message. Finally, the continuation for a token itself may have another continuation, thus enabling chains of continuations for tokens.
>
> `token` is a special keyword with the scope provided by the last token-passing continuation. For example, in the expression $a_1 \leftarrow m_1(\textit{args})@a_2 \leftarrow m_2(\textit{token})@a_3 \leftarrow m_3(\textit{token})$;, the first token refers to the result of evaluating $a_1 \leftarrow m_1$ while the second token refers to the result of evaluating $a_2 \leftarrow m_2$ [27]

These *tokens* are a form of promise between behaviours; when one behaviour completes, then a promise is fulfilled and enables the behaviour waiting on that promise.

Extensions to Actors

To elaborate on the pain point of actors, it is *often* remarked that actors are not a good fit for operating over multiple actors atomically[28, 29, 30, 31].

De Koster et al. propose extending actors and behaviours with *domains* and *views* as a solution to the shortcomings of isolation in the actor model[32, 33].

Domains enrich actors with shared state, which behaviours can synchronously and exclusively access using views. An arbitrary number of objects are collected into a shared domains. Views can acquire access to a domain during the execution of a behaviour. This view will execute asynchronously, *i.e.,* it will not block the executing behaviour.

[3] `https://doc.akka.io/docs/akka/current/general/message-delivery-reliability.html#discussion-message-ordering`

Listing 2.4: Example of domains and views

```
// create a new shared domain for an account
let account = domain {
   deposit(amount) { ... }
   withdraw(amount) { ... }
}

let teller1 = actor {
   deposit(account, amount) {
      // create a view to exclusively, and asynchronously, deposit into the account
      whenExclusive(account) { account.deposit(amount); }
   }
}

let teller2 = actor {
   withdraw(account, amount) {
      // create a view to exclusively, and asynchronously, withdraw from the account
      whenExclusive(account) { account.withdraw(amount); }
   }
}
```

[32] discuss views with access to a single domain, but propose future work where multiple domains can be acquired. Instead of actors sending asynchronous messages to synchronise updates over multiple actors, one actor can create a view with access to multiple domains.

In De Koster's work, actors remain both the parallelism and coordination mechanism. Actors maintain state and the program progresses by asynchronous execution of behaviours which have exclusive access to this state. However, on top of this is a second coordination mechanism to control access to domains. This means a programmer needs to decide ahead of time what is an actor and what is a domain. Moreover, there is no clear discussion on the order of execution of view operations.

Dynamically nested views do not have access to the domain(s) acquired by the textually enclosing view. The views programmer may wrongly assume a nested view has access to the outer view, and may obtain runtime errors if the outer view is accessed but has expired. Such an error can be prevented statically via a type system.

Multi-actor programming as in Aeon [31, 34] has a similar remit to that of BoC: that programmers often want to reason about the composition of several messages for collaboration. It proposes that actors are placed in an acyclic ownership graph, messages are grouped into units which can be executed in a serializable manner. Events are asynchronous and actors are organized in a DAG, which is used to enforce serialization by locking the actors as the events traverse the DAG.

In the case where a large number of messages is sent concurrently, and where each of these messages has more than one different actors receivers, Aeon requires the receiver actors to be dominated by one single actor. In that case, all these essentially concurrent messages need to be coordinated by that single actor, thus introducing a single point of contention. Such a pattern is discussed in the `ReasonableBanking` example in Section 7.1.6.

Transactors, as they appear by Field and Varela [35], address the consistency of the state of several actors involved in processes with several stages. Transactors are concerned with networks, and their failures. Thus, transactors extend the actor model by explicitly modelling node failures, network failures, persistent storage, and state immutability. Messages are sent to/executed by a single actor.

The term *transactors* has also featured in Akka, but with slightly different meaning.[4] These transactors address the flexibility of coordination across actors by enabling STM across multiple actors. Transactors do not decouple the state and behaviour of actors and, so, an actor must present an interface through which

[4] `https://doc.akka.io/api/akka/2.0.5/akka/transactor/Transactor.html`

transactions can be used. Contrast this with the ah-hoc creation of behaviours that acquire cowns and use them as necessary. Transactors do not so much present a single abstraction; rather the unification of two. When a multi-actor message/operation needs to update state, then that state will need to be transactional, even if part of it logically belonged to the first, and another part logically belonged to the second actor – I discussed such an example in Section 7.1.6. Transactions on that state would need to either each start on separate threads, thus losing ordering guarantees, or we would have to run all the transactions operating on that state on the same thread, thus losing concurrency.

On a similar note, the unification of futures, actors, and transactions has been studied by Swalens et al. [36] with a model and with an implementation on top of Closure. The remit here is to offer a model that supports all three paradigms in a faithful manner, i.e. to preserve the guarantees of each constituent paradigm whenever possible. Again, multiple parallelism and coordination abstractions are presented to a programmer instead of one. This means the programmer must be aware of which intersection they are using and the guarantees it provides. With respect to the pattern of a multi-actor message updating some state, the same limitations apply as in the earlier paragraph: If the state is made transactional, then one will either lose concurrency or lose ordering guarantees - more under the discussion of `ReasonableBanking` in section Section 7.1.6.

Relation to BoC Like actors, BoC cowns provide a single entry point into protected data. Unlike actors, behaviours can access the state of multiple cowns. Furthermore, BoC provides a behaviour ordering guarantee; the order in which behaviours are spawned is reflected in the order in which behaviours execute.

2.1.5 Join Calculus

The join calculus is a paradigm that has been inspired by the chemical abstract machine[37]. The Join calculus[38][39] exposes asynchronous programming as an explicit language feature. The paradigm aims to better suit concurrent, distributed, programming compared with other paradigms, such as threading and locks. This calculus is founded in communication; execution is not constructed as a linear sequence of tasks, as with threads, but instead execution happens through message passing and continuations.

I include this discussion on the join calculus, as it is an asynchronous programming model that allows us to access multiple shared resources within a unit of execution. Unlike the previous paradigms, coordination is based on the presence of data on channels, instead of by coordinating access to the data itself.

A program consists of named channels, join patterns, continuations and processes. Processes execute in parallel and produce messages on named channels. Join patterns define the continuations that should run when there are messages on a number of channels, consuming the messages. Thus the coordination of the paradigm. These concepts are more easily understood through an example. Listing 2.5 shows the banking example written using JoCaml, an implementation of the join calculus in OCaml[40].

Listing 2.5: Example of the Join calculus and JoCaml

```
type account = {
   deposit: int -> unit;
   requestWithdrawal: int -> unit -> int;
}

let create_account opening =
   def balance(m) & deposit(n) = balance(m + n) & reply to deposit
   or balance(m) & requestWithdrawal(n) =
      reply (fun() -> spawn balance(m - n); n) to requestWithdrawal
   spawn balance(opening);
   { deposit = deposit; requestWithdrawal = requestWithdrawal; };;

```

```
13 let transfer acc1 acc2 amount = acc2.deposit(acc1.requestWithdrawal(amount)())
14
15 let acc1 = create_account(500);;
16 let acc2 = create_account(300);;
17
18 transfer acc1 acc2 50;;
```

Calling `create_account` creates a new private channel, `balance`, and two public channels `deposit` and `requestWithdrawal`, that are exposed in a new `account` record, Line 11. In fact, the state of the account is captured only in `balance` messages; creating an account emits a message containing the opening amount on the new `balance` channel. Line 7 - Line 9 define the deposit and withdraw operations of an account. Line 7 defines the join pattern for a message on the channel `deposit` and a message on `balance` using the `&` operator. When both messages are present, they are consumed and the message `m + n` is produced on the channel `balance`. In parallel, also using `&`, execution returns to the producer of the message on `deposit`. Execution returning to the producer is encoded in the `reply to` keyword.

We can build synchronisation primitives such as locks using the join calculus, but we can also restrict access to methods by consuming the messages that are required to enable methods. This is how we restrict access to `acc1` when we emit a message on `requestWithdrawal`; the join pattern on Line 8 consumes the `balance` message without producing a replacement. No further `requestWithdrawal` or `deposit` can be made until the withdrawal has been completed. The withdrawal completes by calling the anonymous function, Line 9, which emits a message on `balance`.

A transfer requests a withdrawal from `acc1` and deposits the result of calling the returned callback into `acc2`. This creates a message of `m + amount` on `acc2`'s `balance` channel and `m - amount` on `acc1`s `balance` channel. This design prevents access to `acc1` until we are ready to perform the withdrawal, claiming exclusive access to `acc1`. Whilst the withdraw is being performed, the state of `acc1` is unavailable to any other process, and whilst the deposit is being performed, the state of `acc2` is unavailable. This means no withdraw from `acc1` can occur before the deposit from `acc2` has finished.

However, this is not quite the semantics as in the example constructed using BoC in Listing 1.1. In the join calculus, we do not achieve maintaining exclusive access to both resources until the transfer is finished. Once the withdrawal is made, any join pattern using `balance` is possible again. In this way, the transfer is not an atomic operation.

Note that the interaction between `deposit` and `requestWithdrawal` has been carefully constructed to coordinate internal messages; preventing undesired access to accounts during a `transfer`. We need to know how we expect accounts to interact with one another, limiting the possible interactions without some other coordination method. In other paradigms we are able to write more general functionality, unaware of whether it will be used in a concurrent setting. Users of this functionality leverage the paradigm to achieve atomic interactions between objects. BoC is an example of one such paradigm.

There are alternative ways to encode the bank transfer and obtain atomicity. We could build locks using join patterns and lock the state of the accounts; preventing data races as long as all accesses adhere to the same access protocol. However, this is very similar to how we were programming using threads and locks; failing to utilise the benefits of the Join Calculus.

We can encode the bank transfer by exploiting messages as state and emitting accounts as messages. Consider a channel where each message on the channels is the state of an account. By joining on many of these channels, we gain exclusive access multiple accounts. Refer to Listing 2.6 for an example using `cell` channels.

Listing 2.6: Message as state

```
1 let create_cell contents =
2    def cell(c) = 0 in
3    spawn (cell(contents));
```

```
    cell;;

// create two cell channels
let c1 = create_cell 10;;
let c2 = create_cell 20;;

// join on swap, c1 and c2 and swap the contents on c1 and c2
def swap() & c1(a) & c2(b) = c1(b) & c2(a);;
```

Here we need to ensure we put the states back on the respective channels and that we don't generate multiple messages on either `cell` channel. Neither of these are prohibited by the paradigm.

We are not restricted to a single synchronous message in a join pattern; synchronous continuations are asynchronous messages with a continuation channel. Leveraging this freedom we are able to encode a rendezvous pattern, whereby two processes are able to wait on each other and exchange information. In Listing 2.7, the join pattern waits for `p1` and `p2` and exchanges the messages. The two threads running these methods have to wait until the other is ready.

Listing 2.7: Rendezvous pattern

```
def p1(m) & p2(n) = reply m to p2 & reply n to p1;;
spawn (print_int (p1(12)); 0);;
print_int (p2(10)); 0;;
```

This pattern does not always integrate well in languages that adopt multiple concurrency paradigms. Chords is an implementation of the join calculus in C♯, it diverges from the join calculus as it allows only one synchronous method in a join pattern. This change has been made to avoid the problem of deciding which thread performs the rendezvous[41]. The outcome of the rendezvous may change depending on the thread that executes the rendezvous; this problem arises as threads have local state. BoC does not maintain thread local state.

The join calculus does not suffer from data races, each message can only be consumed by a single process. We can have deadlocks, a synchronous call made without the asynchronous messages available will wait indefinitely. We also have non-determinism and ambiguity; if multiple messages are available on a channel, we do not know which message will be consumed in a join pattern. Like any paradigm, care is required when building programs, we could end up in a very confusing world if a bank account accidentally generated two balances.

Relation to BoC The join calculus relies heavily on channels and message passing, both of which we can construct in BoC. The join calculus gives us a paradigm where communication and asynchronous computation is explicit. To draw a comparison to BoC, behaviours represent joining on resources and messages signal the availability of cowns. In BoC, coordinating many shared resources is simple.

We can encode the join calculus using cowns and behaviours, see Listing C.1 for this encoding (see Appendix C and [42]). This encoding is inspired by the API for the join calculus in C♯[41]. Each join channel is modelled by a data structure containing a queue of data and list of observers, protected by a cown. Join patterns create new observers for these channels, and a callback to execute. When data is written to a channel, all observers of the channel are notified. If all channels which a pattern is observing have notified the pattern, the pattern acquires all of the cowns and attempts to read data from the channels, executing the patterns callback. The pattern may fail to read data if an intervening pattern has emptied the channel.

2.2 Modelling Concurrency

An important result of my research is establishing and formalising BoC; this includes stating a proving data-race freedom, atomicity, and ordering guarantees. In this section I explore the literature to discuss methods I have used to model BoC.

2.2.1 Separation Logic

The calculus for BoC, which I will discuss in Chapter 4, is built as a parametric calculus atop an underlying language with a specified structure. I want to reason about the separation of behaviours in this calculus and demonstrate that a behaviour only changes the state, or the cowns, to which that behaviour has access.

Separation logic allows us to discuss distinct areas of memory by the predicates that hold in each area[43]. The intention of this language is to reason about the data that is and is not changed by an operation.

I am most interested in the following from separation logic, where h is a heap:

- $h \# h'$ indicates the domains of h and h' are disjoint
- $h * h'$ indicates the union of disjoint heaps

To ensure preservation of disjointedness, I cannot exclusively consider the local execution of behaviours. I must consider the possible global interactions of execution. Assume the following:

$$h \# h'' \text{ and } e, h \rightsquigarrow e', h'$$

Where e, e' are expressions, h, h', h'' are heaps and $\rightsquigarrow$ is an execution relation. What I would like to be able to assume is that disjointedness is preserved:

$$\text{If } h \# h'' \text{ and } e, h \rightsquigarrow e', h' \text{ then } h' \# h''$$

This property says: if I take two disjoint heaps h and h'' and execute an expression using the heap h, then the resulting heap, h', and the original heap h'', will be disjoint. However, this property does not hold. Creating a new object in h during e violates this property; further discussed by Yang and O'Hearn [44]. This means that we should not assume it as an invariant of BoC, I need a different property. Hongseok Yang discusses the frame property that follows. C, s, h is a triple of expression, stack and heap; C representing the actions of one thread.

Lemma 1 (Frame Property)**.** *Suppose* C, s, h_0 *is safe, and* $C, s, h_0 * h_1 \rightsquigarrow^* s', h'$.
Then there is h'_0 *where* $C, s, h_0 \rightsquigarrow^* s', h'_0$ *and* $h' = h'_0 * h_1$

This property says that the result of execution in a large state can be tracked to some execution in a smaller state. By assuming this property I can describe the invariant of disjointedness in our paradigm, allowing us to prove that our semantics guarantee our definition of deadlock and data-race free.

In Chapter 5 I construct similar separation relations to build a notion of well-formed states and an operational semantics that preserves the well-formed property. I introduce a property similar to Lemma 1 – namely, Property 2 in Chapter 5 – to preserve the separation of behaviours during execution in BoC programs.

To summarise: separation logic introduces terminology and proves properties that are fundamental influences to our model.

2.2.2 Concurrent Views Framework

I want the model of BoC to be parametric with an underlying programming language. This allows BoC to be adopted by many different programming languages. Moreover, I want BoC to be able to guarantee that behaviours remain isolated from one another throughout execution, no matter the underlying language. To achieve this I need to define the parameters and properties of the underlying language that must be satisfied. These parameters will allow us to describe how one behaviour executes independently of others and, as long as the behaviours *compose* (according to a defined composition relation), they will remain isolated throughout execution.

The concurrent views framework[2] distils the core principles of compositional reasoning. To summarise compositional reasoning, I focus on some state of interest and abstract the remaining state; as long as the remaining state composes, the properties of the state of interest are upheld. One example of such reasoning is Separation logic[2, 43]. The logic uses some formulae to describe part of the state, and abstracts the remaining state (*i.e.,* framing off part of the heap). Two threads can be composed as long as their state remains disjoint.

The concurrent views framework defines parameters and properties that are required to instantiate their programming language. Separation logic can be achieved through an instantiation of the concurrent views framework. In fact, this parametric definition allows many forms of compositional reasoning to be achieved through instantiations of the framework, such as concurrent separation logic, type systems, and rely-guarantee reasoning[2].

The parameters defined in the framework include *atomic commands*, *machine states*, *interpretation of commands*, and more, mostly importantly to us *view semi-group* (parameter 1) and *axiomatisation* (parameter 2).

Parameter 1 (view semi-group)**.** *Assume a commutative semigroup* $(View, *)$.

Parameter 2 (axiomatisation)**.** *Assume a set of axioms* $Axiom \subseteq View \times Label \times View$.

Parameter 1 allows us to express knowledge about the state, and when states compose. Parameter 2 defines the pre- and post-conditions for atomic commands.

Consider a corresponding notion in BoC; the "view" of a particular behaviour is the cowns it can access, its state, and whether it is running or not. Whether two behaviours compose is decided by the disjointedness of their state and cowns. The pre- and post- conditions for BoC, corresponds to preserving composition of state regardless of the underlying languages step.

In Chapter 5, I will look to the concurrent views framework to guide us in the design of an interface between BoC and an underlying programming language. This interface will define the properties that the underlying programming language must provide, and that BoC will guarantee are maintained, and upon which stronger properties will be proven.

2.2.3 Weak Memory Models

I want to model and investigate the order of behaviours that BoC permits. Constructing operational semantics is one way to achieve this goal. Another method is to construct an axiomatic model of execution which abstracts away the intricacies of an implementation or operational semantics. Inspired by weak memory models, in Chapter 6 I will construct an axiomatic model to explain and investigate the order of behaviours in BoC.

Weak memory models capture the complex interactions of multiprocessors to define how they execute programs. Processors do not, necessarily, execute programs in the sequential order in which the instructions of the program appear; they may optimise the execution of the program, on the fly, to obtain better

performance. Processors are allowed to reorder instructions, buffer reads and writes, and so on, as long as the result of the execution is the same as if the instructions were executed in order.

It is less clear what should happen, and more importantly what *will* happen, when a multiprocessor executes threads which access shared memory. Consider the example in Figure 2.1 from [3]. `x` and `y` are memory locations available to both threads and `r1` and `r2` are registers local to T_1. The two threads execute the instructions from top to bottom. What are the possible values that `r1` and `r2` can read? More to the point, can `r1=1` and `r2=0`? Without a well-defined specification of execution this outcome is entirely possible. T_0 may locally buffer the write of `1` to `x` but not buffer the write to `y`, writing the value back to the shared memory location for `y`.

initially `x=0; y=0`	
T_0	T_1
(a) `x` ← `1`	(c) `r1` ← `y`
(b) `y` ← `1`	(d) `r2` ← `x`

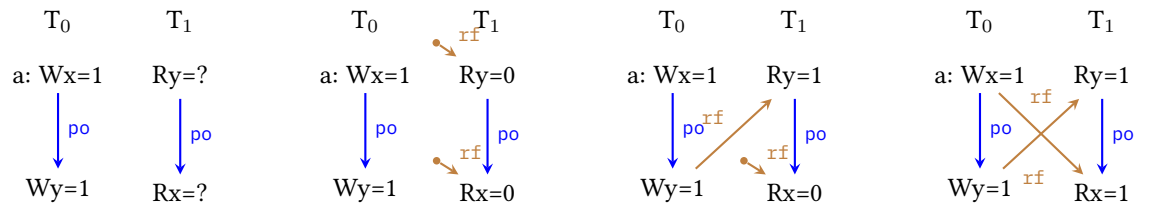
Figure 2.1: Multicore message passing pattern

Memory models are specifications for valid executions of programs. Sequential consistency is one such specification[45]; a multiprocessor is sequentially consistent if "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." This specification rules out the scenario where `r1=1` and `r2=0`.

However, real multiprocessors do not implement sequential consistency; sequential consistency is too strong and restricts program optimisation. Real multiprocessors have relaxed memory models which are more permissive in the executions which they allow[46]. In fact, there are numerous memory models; these models are defined both operationally and axiomatically. Operational models capture the implementation of the hardware they model to explain how programs unfold, whilst axiomatic models distinguish valid and invalid executions of a program by constraining relations on memory access[3]. Axiomatic models can be faster for simulation and verification whilst operational models are considered more intuitive, and so demonstrating their equivalence is ideal[3].

Axiomatic models are defined as follows: instructions are mapped to mathematical objects, for example read and write events and corresponding relations such as program order, allowing for construction of control flow semantics; Candidate executions are constructed from these control flow semantics; Constraint specifications judge whether candidate executions are *valid* or *invalid*.

Consider Figure 2.1; the mathematical model for a program is defined by the events of reading and write to shared memory, the sequential *program order* of a thread, the *reads from* order from a write event to a read event, and the *coherence order* from a write event to a write event[3]. Figure 2.2 presents graphical representations for candidate executions of Figure 2.1. Different constraint specifications will judge different executions valid and invalid; as I stated before, sequential consistency judges Figure 2.2d to be invalid.



(a) control flow (b) data flow 1 (c) data flow 2 (d) data flow 3

Figure 2.2: Control flow and candidate executions for Figure 2.1[3]

Alglave et al. [3] propose an axiomatic generic framework for modelling weak memory. This model is built from four axioms: SC PER LOCATION, NO THIN AIR, OBSERVATION and PROPAGATION. These axioms judge which candidate executions are valid or invalid by, for example, ensuring the order of events is acyclic.

2.2.4 Atomicity and Linearizability

I have mentioned that behaviours should be considered atomic, but it is unclear what I mean by this term. Atomic behaviours could mean that the behaviour is truly uninterruptible. Atomic behaviours could also meant that the execution behaviour *appears* uninterrupted. A clearer interpretation of atomicity is Linearizability[47]. Linearizability of behaviours means that executing a behaviour concurrently with other behaviours, is equivalent to executing the behaviour serially and sequentially. Sequentially, being that any invocation is immediately met with a response, and concurrently being anything that is not sequentially. Linearizability is an important result as it allows us to consider operations on objects uninterrupted, that is that they could be considered to complete instantaneously. Linearizability makes reasoning about programs significantly easier. In Chapter 5 I will revisit linearizability for BoC.

2.2.5 Simulation and Equivalence

I am interested in establishing simulations and bisimulations between different operational semantics for BoC. I will discuss creating a bisimulation between differently grained semantics in Section 5.5. Simulations allow us to create equivalences between programs[48]. A simulation is a relation, $\mathcal{S}$, between states and transitions in two state transition systems, $\mathcal{P}$, defined as follows[49][50]:

Definition 1 (Simulation). *A relation $\mathcal{S} \subset \mathcal{P} \times \mathcal{P}$ is a simulation if $(P, Q) \in S$, implies for all $\alpha \in Act$ Whenever $P \xrightarrow{\alpha} P'$ then, for some Q', $Q \xrightarrow{\alpha} Q'$ and $(P', Q') \in \mathcal{S}$.*

This definition states that if two states P and Q are similar by S, then for all actions from P to P' the same action can take Q to some Q' where P' and Q' are similar. This creates and equivalence class between programs and semantics. We can consider P and Q to be states from two programs and $\xrightarrow{\alpha}$ the execution relation. We can choose what it means for two states to be similar, thus defining our $\mathcal{S}$.

A property stronger than simulation is bisimulation:

Definition 2 (Bisimulation). *A relation $\mathcal{S} \subset \mathcal{P} \times \mathcal{P}$ is a bisimulation if $(P, Q) \in S$, implies for all $\alpha \in Act$*

- *Whenever $P \xrightarrow{\alpha} P'$ then, for some Q', $Q \xrightarrow{\alpha} Q'$ and $(P', Q') \in \mathcal{S}$*
- *Whenever $Q \xrightarrow{\alpha} Q'$ then, for some P', $P \xrightarrow{\alpha} P'$ and $(P', Q') \in \mathcal{S}$*

This property requires that two systems can simulate each other.

In my research I deal with a highly concurrent language with many possible but equivalent executions. Reasoning about and testing many different executions becomes problematic. In Section 5.5.1 I discuss a semantics that has fewer possible executions, a coarser grain semantics. Such a semantics is only useful if anything that can be achieved in the finer-grain semantics, can also be achieved in the coarser-grain semantics. I am interested in establishing a simulation from a coarse grain to a fine grain semantics.

Another interesting research avenue, outside of the scope of this thesis, is to show equivalence between BoC and other concurrency paradigms. This would construct a rudimentary translation of programs using one paradigm into another paradigm. Such a relation would go some way to compare the expressivity of the related languages. Of course, bisimulation does not quantify how well one paradigm is suited to a class of problems compared to another.

2.3 Concurrency case-studies

In Chapter 7 I present an evaluation of the expressiveness and ergonomics of BoC. I do this through a collection of benchmark suites, and concurrency patterns challenges that can be found in various literature[51][52][53].

This collection includes: Savina, the actor model benchmark suite; coordinating asynchronous processes; the dining philosophers problem; the fibonacci sequence through divide-and-conquer programming; channels for inter-behaviour communication; barrier communication; the Santa problem; the Boids program; binary search trees; and building BoC using Rust.

I revisit these in Section 7.2 and many of these case-studies can be found, with BoC implementations, in my github repository[42].

2.3.1 Atomic updates over multiple resources

The recurring atomic bank transfer example demonstrates an important and real-world pattern. Scenarios which require us to perform atomic updates over multiple resources; this may be, for example, in a distributed database or operations over multiple actors.

In this section, I will cover case-studies where this feature of BoC can be well utilised.

Two-Phase Commit

Two-phase commit provides a real-world protocol to achieve atomic operations over multiple resources. This is a costly protocol but allows coordination when resources may be distributed, or when multiple resources cannot be acquired by a single unit of execution (*e.g.,* when using actors).

Two-phase commit, without failures, proceeds as follows:

1. The coordinator sends a vote request, or *Prepare*, to all participants.
2. When a participant receives a vote request, they reply by sending the coordinator a *Yes* or *No*.
3. Once the coordinator receives a response from all participants, the responses are inspected. If all responses are *Yes* then the coordinator sends *Commit* to all participants. Otherwise the coordinator sends *Abort* to all participants.
4. Each participant waits for a *Commit* or *Abort* response from the coordinator. Once received the participant reacts accordingly.

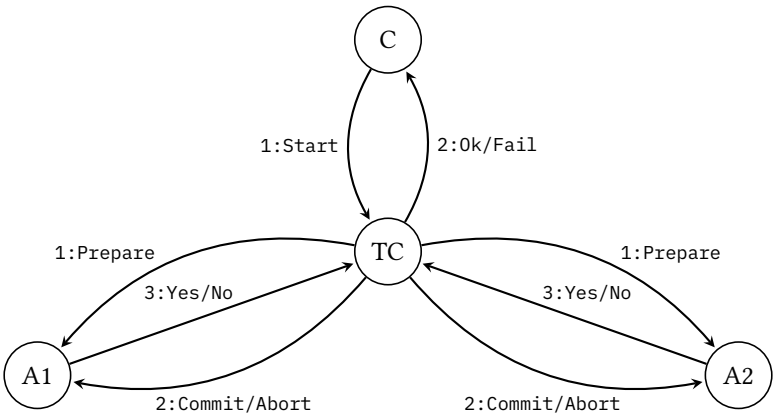


Figure 2.3: Two-Phase Commit for Bank Account Transfer

Two-phase commit is demonstrated in the following bank account example (Figure 2.3)[54]. C is a client that wants to perform a transfer, TC is the transaction coordinator and A1 and A2 are accounts.

The coordinator requests whether A1 is willing to withdraw and whether A2 is willing to deposit. If both A1 and A2 vote yes, a commit message is sent to both and the state of both are updated. If either sends no, which may occur if A1 does not have enough money for the withdrawal or either account is involved in another transaction, then an abort message is sent to both.

Using two-phase commit we can achieve an atomic transfer where no intervening operations involving either account can occur. However, a lot of communication is required to perform the transfer. Furthermore, the commit may have to back off and try again if either account votes no. Appendix D presents an implementation of the bank account transfer using two-phase commit in Pony.

In Section 3.4, I demonstrate why this pattern is not necessary using BoC. In Section 7.1.6, I demonstrate how actors, with two-phase commit, compares with BoC alternative. This protocol is great for demonstrating a key feature of BoC; namely, acquiring multiple resources in a single behaviour for atomic operations. The protocol helps to demonstrate scenarios where BoC can really help.

Boids

Boids is an artificial life program designed to model the flocking motion of birds or schools of fish[55]. Each boid has a position and velocity, and their basic motion is made up of three actions:

1. Separation: steer to avoid crowding local boids
2. Alignment: steer towards the average heading of local boids
3. Cohesion: steer to move toward the average position of local boids

This requires that each boid is able to read the position of any other boid, and update its own position and velocity based on *local* boids.

In Reynolds [55] original design, each boid is naturally modelled as an actor. However, as stated, the motion algorithm requires knowing information about all other boids. This means the boids need to either: send their position to all other boids; or, send their position to an "environment" actor which stores the global state and can be queried for neighbours.

This program provides an interesting and relevant problem, as we need to be able to read the state of multiple Boids to atomically update the state of one.

2.3.2 Coordinated access to cowns

BoC is effective at coordinating exclusive access to cowns, even when operations do not need to atomically update multiple cowns. Moreover, BoC guarantees an implicit ordering to these accesses. This ordering can be used to solve problems in novel ways, such as for divide-and-conquer style programs. The ordering can also be used to simulate existing concurrency patterns, such as process synchronisation in barriers. In this section, I will cover case-studies where BoC can utilised in this manner.

The Dining Philosophers Problem

The Dining Philosophers problem is a well-known classic concurrency problem[51]. The philosophers want to atomically grab two forks, eat, and then release the forks. However, the state of the forks do not change, so there is no atomic updated. The forks are used as a coordination mechanism for the philosophers eating. Thus, the problem does not exemplify the best of what BoC achieves.

The Santa Problem

The Santa Problem is an interesting challenge where a task requires a number of resources, of a certain type, from a larger collection of such resources[56].

More precisely, we have a system with Santa, 9 Reindeer and 7 Elves. When Santa and 9 Reindeer are available, Santa and the Reindeer deliver toys and then the Reindeer temporarily go on holiday. When Santa and 3 Elves are available, Santa and the Elves meet in the study for R&D and then the elves temporarily go to work.

This problem has a solution using Chords[56]. The solution encodes the elves as processes repeatedly queueing, being let into a room by Santa, meeting with Santa and then being let out again. Join patterns are used to allow the first 3 elves to queue, but then blocking other elves when they attempt to queue. Queuing can resume again once Santa has allowed the 3 elves into the room. Further join patterns are used to synchronise the elves having all entered the room before performing R&D. Similarly, the elves synchronise on leaving the room. This pattern is repeated for Reindeer. Whilst this design elegantly uses join patterns, neither the elves nor reindeer nor Santa are objects in the system. Instead elves and reindeer are represented as a loop of tasks, their availability by an increasing counter in a message and Santa by the presence of a message. When the elves and Santa are ready, it is not one process coordinating interaction between objects but several processes executing the same function at a time. This solves the Santa problem as the parties need only synchronise before and after collaborative tasks, such as R&D. This solution uses synchronous messages to create barriers to synchronise tasks.

Typical solutions to the Santa problem[56] represents the elves, reindeer and Santa as processes. However, I want to explore a solution where the elves, reindeer and Santa are resources and not processes. Representing the workers as processes allows them to synchronise at execution points. This representation does not easily permit mutating multiple resources, consider a scenario where Santa could promote an Elf for good work. This pattern is one I expect to be more common using BoC. I believe to achieve this using Chords would require significant and complex redesign of the solution.

To have the Santa and his helpers as resources presents an interesting challenge in BoC. Behaviours are scheduled when specific named resources become available. There is not an immediate way to select N arbitrary resources from a collection of those resources.

Like the dining philosophers, this problem focusses around scheduling work based on resource availability. However, the tasks do not change the state of the resources.

Therefore, the Santa problem introduces an interesting pattern. Yet, does not fully utilise the atomicity of behaviours which BoC provides.

Fibonacci: divide-and-conquer

Divide and conquer programming is a well-known paradigm for recursively decomposing a problem into smaller sub-problems. The sub-problems can then naturally be solved in parallel. Once the sub-problems are solved the results must be combined. I have chosen the Fibonacci sequence as it means I can work with integers, and not need to consider complex data-structures such as lists and trees.

This divide-and-conquer problem requires a natural ordering point; the combination stage must occur after the sub-problems finish. With a synchronous fork/join model this could be achieved by forking threads to solve the sub-problems, and the waiting for them to join before continuing to combine their results. With an asynchronous actor model, actors must solve sub-problems and communicate their computed results to a subsequent actors to combine the results (this can be simplified if promises are available).

This problem is interesting for BoC as it provides a case-study to explore behaviour ordering. The problem does not require BoC's atomic updates over multiple resources; the updates inherently have an order that must be followed, each step enabling the next. Thus the case-study is useful for exploring the ordering feature of BoC.

Barriers

In concurrent paradigms, a programmer often desires a way of coordinating execution and, if possible, synchronising many processes. Barriers are a common means to achieve this goal. Several processes can be created and operate in parallel until they reach a *barrier*, at which point a process must wait until enough processes have reached the barrier[57]. To motivate why one may want this consider the example in Listing 2.8, written in C.

Listing 2.8: Example of barrier in C

```
1  void *run(void* data) {
2     thread_data_t *thread_data = data;
3     printf("thread %d: performing startup...\n", thread_data->id);
4     pthread_barrier_wait(thread_data->barrier); /* wait for threads to finish starting */
5
6     printf("thread %d: running main code...\n", thread_data->id);
7
8     pthread_barrier_wait(thread_data->barrier); /* wait for threads to synch before finishing */
9     printf("thread %d: finishing up...\n", thread_data->id);
10 }
11
12 int main() {
13    /* create a barrier for 4 threads */
14    pthread_barrier_t barrier;
15    pthread_barrier_init(&barrier, NULL, 4);
16
17    /* create a thread that has access to the barrier */
18    pthread_t t1;
19    thread_data_t data1 = {1, &barrier};
20    pthread_create(&t1, NULL, &run, &data1);
21
22    /* more threads start */
23
24    pthread_join(t1, NULL);
25    /* join more threads */
26 }
```

Multiple threads are executing concurrently but they all need to have finished some startup routine before moving on to the main process, Line 4. The barrier allows us to encode this requirement.

There are various algorithms in the literature for building barriers. In Section 7.2.2 I explore the algorithms for building barriers in BoC. One method is to rely on the dispatch ordering of behaviours. This is a lightweight approach to ensure synchronisation, however it requires understanding of behaviour ordering. Another approach is data-oriented and uses callbacks and counters to communicate and synchronise behaviours; however this is a more complex, memory heavy approach.

As for the divide-and-conquer case-study, barriers are a good fit for exploring ordering in BoC, whilst not requiring atomic updates over multiple resources.

Channels

Inter-process communication is long-standing and useful tool in the concurrency toolbox[51, 52]. Channels enable this communication by reading allowing processes to read from and write to shared channels.

Being able to build a mechanism for inter-process communication goes some way to show the expressivity of our paradigm. Building this data structure also provides another tool for programmers to leverage. Channels fit into the category of coordinated access to cowns, as we need only order the reads and writes to a single channel.

2.3.3 Binary search tree: hand-over-hand "locking"

Consider a binary search tree (BST) which can be operated on in parallel. Each node of the tree has a lock. As operations descend the tree, they acquire more and more locks, effectively locking a subtree. Later operations will have to descend the tree in the same order and so will naturally be blocked. This makes sequential synchronous operations over the parallel data-structure straightforward for the programmer.

A, perhaps naive, analogue in BoC replaces the nodes and locks, with nodes protected by cowns. To read a node, a behaviour must be spawned to first acquire the node. As each node is protected by a cown, acquiring a whole subtree requires: spawning a behaviour to acquire a node, reading the node to find the subtree, spawning a new behaviour to acquire to original node and the subtree. The release and re-acquire makes sequential synchronous very difficult to achieve.

This example is very helpful as it demonstrates a programming pattern which is ill-fitting for BoC.

2.3.4 Savina: the actor model benchmark suite

Savina is a benchmark suite designed to compare implementations of the actor paradigm [58]. This suite is widely known in the actors community and has been used to compare the performance of several actor languages[59]. I chose Savina for our evaluation for two reasons:

1. Actors model programs have close relation to BoC and there is a straightforward translation from actors to BoC. Thus, I can compare performance of our runtime against existing runtimes.

2. There are patterns in actor model programs which I have identified as places where BoC is a better fit; thus, I can compare tradeoffs between these paradigms.

One limitation in using Savina is that the benchmarks do not have a reference output (or often any output); nor clear design criteria. This means the programmer must compare two implementations of a benchmark and decide whether they represent the same task. In Section 7.1.6 I demonstrate how I have distilled a more appropriate benchmark from one of the Savina benchmarks.

2.3.5 Rust and Ownership

A foundation of BoC is that behaviours are isolated from one another. One way of achieving this is to ensure that a cown uniquely owns that data it protects, and behaviours uniquely own any local variables that they access. That is to say, there is only a single entry point in to some object or region of memory. This means that behaviours cannot access any memory being accessed by another behaviour.

Rust is an example of a language that provides and preserves this idea through ownership. In Rust, ownership preservation is achieved through move semantics, reference management[60] and lifetimes[61]; Listing 2.9 demonstrates reference management in Rust.

Listing 2.9: Ownership and Borrowing in Rust

```
1  fn print_string(arg: &String) {
2      // borrow a read-only reference to a string
3      println!("string: {}", arg);
4  }
5
6  fn change_string(arg: &mut String) {
7      // borrow a writeable reference to a string
8      arg.make_ascii_lowercase();
9  }
10
11 fn consume_string(arg: String) {
12     // take ownership of a string
13     print_string(&arg);
14 }
15
16 fn main() {
17     let mut mystring = String::from("Hello, world!");
18
19     print_string(&mystring); // temporarily pass a read-only reference
20
21     change_string(&mut mystring); // temporarily pass a writeable reference
22
23     consume_string(mystring); // transfer ownership
24
25     print_string(&mystring); // this will fail as 'mystring' has been moved
26 }
```

Rust uses move semantics to ensure that data only has one owner. Data is moved and not copied between owners when reassigned (except when types implement copy), this means there is only a single alias to mutable data. Variables and arguments are annotated with their mutability and whether they are borrowing or taking ownership of data. For example `change_string`, Line 6, borrows a mutable reference to its argument, the ownership of `arg` is returned to the caller when the function ends. Whilst `consume_string`, Line 11, takes ownership of the string; once `consume_string` is called on Line 23, the original reference `mystring` goes out of scope.

Lifetimes are the final important part of ownership management in Rust. Lifetimes ensure that there are no references to data that no longer exists; see Listing 2.10, an example extended from [61].

Listing 2.10: Lifetimes in Rust

```
1  // cell whose contents is a reference to a u32 with lifetime 'a
2  struct Cell<'a> {
3      data: &'a u32,
4  }
5
```

```
// function that borrows a mutable cell reference instantiated with lifetime 'a
// and a u32 reference with lifetime 'a. Assigns y to data field of Cell.
fn put<'a>(cell: &mut Cell<'a>, y: &'a u32) {
    cell.data = y;
}

fn main() {
    // create a Cell with a reference to startdata
    let startdata = 12;
    let mut cell = Cell { data: &startdata };

    // call put with a reference to cell and
    let gooddata = 14;
    put(&mut cell, &gooddata);

    // create a new scope and a stack for that scope
    // create baddata in this scope
    {
        let baddata = 16;
        put(&mut cell, &baddata); // This line will cause an error at compile time

        // baddata is deallocated when exiting this scope
    }

    // print the cell contents, referencing the deallocated data
    println!("data: {}", cell.data);
}
```

The lifetimes example will fail to compile as the lifetime of `baddata` is shorter than the lifetime of `cell`. Lifetimes allow programmers to ensure that references always refer to allocated data.

Verona provides an implementation of BoC but the paradigm is not tightly coupled to the type system of Verona. I provide a semantics of BoC that is parametric with an underlying language in Chapter 5. I then instantiate BoC with a simple underlying calculus $\lambda_{\texttt{when}}$ that preserves isolation dynamically. To demonstrate this same claim more practically I have built an implementation of BoC using Rust.[5] I will detail my investigation into this in Chapter 7. This idea has been started anew and investigated more deeply by the Imperial College London student Alona Enraght-Moony[6].

[5] https://github.com/lukecheeseman/rust-cowns
[6] https://github.com/aDotInTheVoid/boxcars

3

Design

Behaviour-oriented concurrency (or BoC) is intended as the sole concurrency feature of an underlying programming language. The underlying language is expected to provide a means to separate the heap into disjoint sets with unique entry points, for example through a type system as described by Clarke and Wrigstad [62]. Provided that the type system satisfies the requirements outlined here and in the semantics section, its exact design is orthogonal to BoC. Several such type systems exist already[63, 64, 65]. A collaborating component of project Verona, [66], proposes a new, more powerful type system which is to be adopted for the language implementing the BoC paradigm.

3.1 BoC in a Nutshell

BoC augments the underlying language with two fundamental concepts: the *concurrent owner* or *cown*, and the *behaviour*.

Cowns protect pieces of separated data, meaning they provide unique entry points to data in a program. A cown is in one of two states: *available*, or *acquired* by a behaviour.

Behaviours are the unit of concurrent execution. They are *spawned* with a list of required cowns and a closure. I define *happens before*, as an ordering relation which strengthens the "spawned before" relation by also requiring that the sets of cowns required by the two behaviours have a non-empty intersection:

Definition 3 (Happens Before)**.** *A behaviour b will* happen before *another behaviour b' iff b and b' require overlapping sets of cowns, and b is spawned before b'.*

Once spawned, a behaviour can be *run* (*i.e.,* its closure starts executing) only when all its required cowns are available, and all other behaviours which *happen before* it have been run. Once available, all the cowns are acquired by the behaviour atomically, and become unavailable to other behaviours. Throughout

execution of the closure the behaviour retains exclusive access to its cowns, and to their data. Moreover, the behaviour cannot acquire more cowns, nor can it prematurely release any cown it is holding. Upon termination of the closure, the behaviour terminates, and all its cowns become available.

BoC is datarace-free and deadlock-free BoC provides the principles to ensure data-race freedom. It expects the underlying language to ensure that the state of each cown, and any behaviour-local state, remains isolated throughout program execution (we will see precisely what is expected for isolation in Chapter 5). Since BoC maintains that there can be only one behaviour running with access to a cown's state at any time, there is no way for two different behaviours to have concurrent mutable access to the same state.

Furthermore, as behaviours acquire *all* their required cowns atomically, and because the happens before relation is acyclic, BoC is *deadlock-free* by construction.

I introduce BoC through examples expressed in an imperative, expression-oriented, strongly-typed pseudo-language with isolation. Two language features are of interest:

- The **cown**[T] type: it represents a cown with contents of type T, with a create constructor.
- The **when** expression: it consists of a set of cown identifiers, and a closure, and is used to spawn a behaviour that requires the specified cowns.

It is the remit of the underlying language to ensure that each object in the heap is owned by exactly one entry point, and that any object accessed within the closure of a **when** is owned by one of the acquired cowns.

3.2 Creating Cowns

For our examples I focus on concurrent modifications to bank account objects of type Account with fields balance and frozen. Listing 3.1 shows how a programmer can create a cown to protect an account. The programmer creates and accesses acc1 as an Account object; they can then add and remove funds from acc1 as they want (synchronously). Conversely, acc2 is created as a cown that protects an Account; in other words, a cown of type **cown**[Account]. The cown prevents any direct access to the account, such as that on Line 6. The only way to gain access to the contents of a cown is by spawning a behaviour that acquires it.

Listing 3.1: Creating cowns to protect data

```
1 main() {
2   var acc1 = Account.create();
3   acc1.balance -= 100; // OK
4
5   var acc2 = cown.create(Account.create());
6   acc2.balance += 100 // Access is invalid
7 }
```

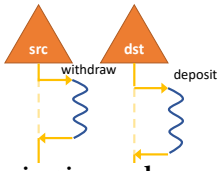
3.3 Spawning and Running Behaviours

Listing 3.2 contains a simple use-case of behaviours. The transfer function transfers amount from src to dst. It consists of two **when** expressions, which spawn behaviours that need mutable access to Account objects. Note that while a **when** always requires resources to be of type **cown**[T] the closure sees the resources as being of type T, *i.e.,* from the *inside*, an acquired **cown**[T] looks like a T. Each **when** expression spawns a behaviour that will, at a later point, execute with exclusive access to the contents of its cown; thus it will

execute without data races, and logically atomically. The `transfer` function returns immediately, without waiting for the two behaviours to execute.

Listing 3.2: Scheduling work on each account

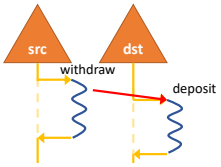
```
1 transfer(src: cown[Account], dst: cown[Account], amount: U64) {
2   when (src) { src.balance -= amount; }; // withdraw
3   when (dst) { dst.balance += amount; } // deposit
4 }
```



Note that spawning and running a behaviour are different events; moreover, **spawning is synchronous**, while **running is asynchronous**. Assuming that `src` and `dst` are not aliases, they can start running in any order: the `withdraw` behaviour might happen before, at the same time as, or after the `deposit` behaviour. In Listing 3.2, each behaviour is executed independently and unconditionally. The right of Listing 3.2 shows a possible execution timeline: each cown (triangles) transitions from available (solid lines) to acquired (dashed lines) as it is acquired by an executing behaviour (squiggly lines), and transitions back again once the behaviour terminates; as these behaviours are independent they may execute together or one at a time (and so may move up and down the timeline). That is, the withdraw behaviour might happen before, at the same time as, or after the deposit behaviour.

Listing 3.3: Nesting spawning behaviours

```
1  transfer(src: cown[Account], dst: cown[Account], amount: U64) {
2    when (src) { // withdraw
3      if (src.balance >= amount) {
4        when (dst) { // deposit
5          dst.balance += amount;
6        }
7        src.balance -= amount;
8      }
9    }
10 }
```

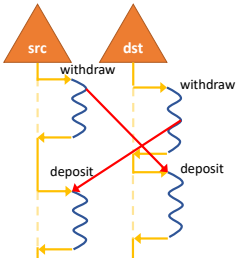


But what if successful transfer is contingent on properties of the `src` cown? Consider the case where accounts may not be overdrawn, *i.e.,* the `balance` has to be positive. This can be achieved by nesting the second **`when`** expression inside a conditional path in the first, as demonstrated in Listing 3.3. Here, the function `transfer` merely spawns one behaviour which requires the `src` cown (the *withdraw* behaviour) and returns immediately. When this *withdraw* behaviour executes, it will check that the `src` has sufficient funds. If it does, the behaviour spawns a new behaviour which requires the `dst` cown (the *deposit* behaviour). The *withdraw* behaviour continues execution after spawning the *deposit* behaviour, the *withdraw* does *not* block and wait for the *deposit* to execute. This means that Line 7 executes after spawning *deposit* regardless of whether *deposit* has executed or not. In fact, *deposit* could begin execution immediately and the operations on Lines 5 and 7 could be executed in parallel. To be clear, **the deposit does *not* have access to** `src`. On the right of Listing 3.3, the *withdraw* behaviour is shown to spawn the *deposit* behaviour (red arrow), the *deposit* behaviour can be run anytime after it was spawned (incidentally, the *deposit* behaviour can move along the timeline as long as the spawning arrow does not point upwards), the *deposit* behaviour may in fact terminate before the *withdraw* behaviour terminates. So, whilst the *deposit* behaviour is textually nested within the *withdraw* behaviour, nested **`when`**s have the same semantics as any other **`when`**s. Contrast this with transactions where nested transactions have special semantics[18].

Consider Listing 3.4, there are two calls of `transfer`: one from `src` to `dst`, and one from `dst` to `src`. Readers accustomed to lock-based programming or nested transactions might think that the example would deadlock, but this is not so: the two `withdraw` behaviours may run in parallel (since they require different cowns), and during their execution they will spawn the `deposit` behaviours. Each `deposit` behaviour will be able to run once its cown is no longer held by the `withdraw` behaviour. Moreover, the two withdraws can happen in either order as can the two deposits. I will revisit this example in Chapter 4.

Listing 3.4: Deadlock-free transfers

```
1 transfer(src, dst, 1);
2 transfer(dst, src, 2);
```



3.4 Behaviours Requiring Multiple Cowns

Listing 3.3 avoids transfers with insufficient funds, but what if I want to guarantee the transfer is not interleaved with other operations on the accounts? For such a guarantee, I need the transfer to be atomic, and Listing 3.3 does not have an atomic transfer.

Listing 3.3 allows money to be withdrawn from `src` and, before the money is deposited into `dst`, an behaviour unrelated to the transfer can access `dst`. In Listing 3.4, the second transfer can be interleaved with the first transfer, and we can see this in the diagram. The first transfer spawns a behaviour to conditionally withdraw from `src`, and then spawn a behaviour to deposit to `dst`; the second transfer does the same with the accounts in the opposite positions. The conditional withdraw from `dst` can be spawned, and therefore executed, before the deposit into `dst`. Consider the two accounts begin with a balance of 1.

If the first transfer completes before the second transfer begins (that is to say both the conditional withdraw from `src` and deposit to `dst` execute before the conditional withdraw from `dst`), then `src` will end up with a balance of 2, and `dst` a balance of 0.

If the transfers are interleaved, such that the conditional withdraw from `dst` executes before the deposit to `dst`, then the condition of the transfer will fail (due to lack of funds) and the deposit to `src` will not be spawned; yet, the deposit to `dst` will still occur. The result in this case is that `src` will end up with a balance of 0, and `dst` a balance of 2 (the opposite result from before).

Furthermore, what if successful transfer is contingent on properties of *both* cowns? For instance, a bank account may be marked as "frozen", meaning any transfer to or from the account should be forbidden. I argue the transfer should be an atomic update over both accounts.

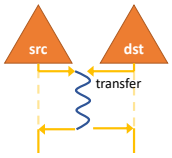
I could try to solve the problem by checking each account in turn, withdraw the money from the `src` and finally deposit the money on the `dst`, requiring further nesting of behaviours. However, by doing so I would lose atomicity guarantees: between checking the account and depositing the money, the `dst` could be frozen by another behaviour spawned by a different part of the program.

Instead, I will employ a key feature of our paradigm which allows behaviours to acquire multiple cowns at once, thus giving *simultaneous* exclusive mutable access to several cowns.

In Listing 3.5, the function `transfer` avoids the earlier atomicity problems. It consists of a **`when`** block that requires access to `src` as well as `dst`. The **`when`** checks that the accounts are not frozen and that there are sufficient funds, and if so, proceeds with transferring the funds.

Listing 3.5: Spawn a behaviour that requires both accounts

```
transfer(src: cown[Account], dst: cown[Account], amount: U64) {
  when (src, dst) { // withdraw and deposit
    if (src.balance >= amount && !src.frozen && !dst.frozen) {
      src.balance -= amount;
      dst.balance += amount;
    }
  }
}
```



This example demonstrates the power of behaviour-oriented concurrency: It allows the programmer to coordinate access to multiple shared resources. A rendezvous of cowns is now a simple task: a single **when** expression creates the synchronisation between multiple cowns and allows a behaviour to read/write the state of all involved cowns. This rendezvous is demonstrated in to the right of Listing 3.5 by a behaviour that acquires both `src` and `dst` at once.

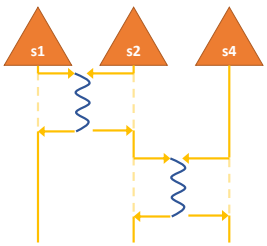
3.5 Order Matters

So far, I have assumed that `src` and `dst` do not alias. For behaviours requiring non-overlapping cowns, the order of execution does not matter. Namely, the effects of behaviour execution demonstrate themselves through modification of the state under their cowns; therefore, cowns cannot observe the effects of execution of behaviours that took place on separate cowns. However, when behaviours execute on overlapping cowns, the order matters, and BoC has rules for such cases.

Behaviour ordering is a dynamic property determined at runtime; it is determined by the order in which behaviours were spawned and the cowns they required (as stated in Definition 3). However, I can sometimes reason about what *will* happen statically.

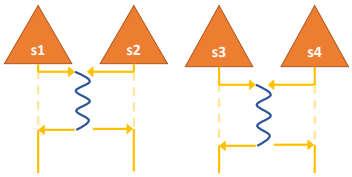
Listing 3.6: Always ordered transfers

```
transfer(s1, s2, 10);
transfer(s2, s4, 20);
```



Listing 3.7: Sometimes ordered transfers

```
transfer(s1, s2, 10);
transfer(s3, s4, 20);
```



For illustration, consider the following scenario: I want to transfer money from one account to another, and then use the money from the second account to pay a third party. This is done in Listing 3.6. The first transfer will be spawned before the second transfer. Here the order of execution of the behaviours matters: e.g., if the second behaviour runs first, it could deplete the funds in `s2`, and incur overdraft fees.

Thus, I would like the first transfer to be completed before the second transfer starts running. I can statically assert that BoC guarantees the desired order of behaviours.

In Listing 3.6, the first transfer requires cowns `s1` and `s2`, the second transfer requires cowns `s2` and `s4`, and {`s1`, `s2`} overlaps with {`s2`, `s4`}. As I said earlier, the first transfer is spawned before the second transfer. Therefore, the first transfer will *happen before* the second transfer.

Consider the slightly modified example from Listing 3.7, where I have four different cown identifiers, `s1`, `s2`, `s3`, `s4`. If `s2` and `s3` are aliases, then the situation is as in Listing 3.6, and the first transfer will complete before the second starts running. But if the four identifiers point to four different cowns, then execution of the first and second transfer may take place in any order, and may overlap. Below Listing 3.7 I show diagrammatically a possible execution, when `s1`, `s2`, `s3`, `s4` differ.

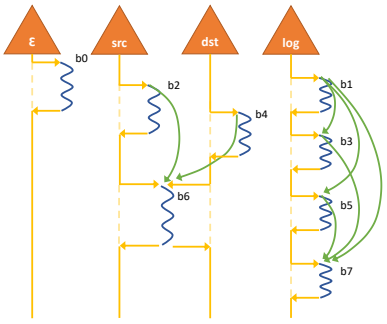
3.6 Putting It All Together

Revisiting our example from section Section 3.3, in Listing 3.2, if `src` and `dst` are aliases, then withdraw will complete before deposit. In Listing 3.8, green arrows represent happens before relations.

This ordering is a deep property that can be used to order nested spawned behaviours. Consider a scenario where, in addition to account operations, I also wanted to create a log of what happened; I augment each behaviour to output strings to an `OutStream` as in Listing 3.8. Notice that $b2$ and $b6$ spawn further behaviours, $b3$ and $b7$, both of which require `log`. Again, I expect the start message, on Line 3, to be logged before the deposit message, on Line 5, and the deposit message before the transfer message on Line 11.

Listing 3.8: Creating an accurate log

```
1  main(src: cown[Account], dst: cown[Account],
2     log: cown[OutStream]) { /* b0 */
3    when(log) { /* b1 */ log.log("begin") }
4    when(src) { /* b2 */ ...
5      when(log) { /* b3 */ log.log("deposit") }
6    }
7    when(dst) { /* b4 */ ...
8      when(log) { /* b5 */ log.log("freeze") }
9    }
10   when(src, dst) { /* b6 */ ...
11     when(log) { /* b7 */ log.log("transfer") }
12   }
13 }
```



The `"begin"` message will be logged before the `"deposit"` message: $b1$ and $b3$ require `log` and $b1$ will be spawned before $b3$ (I know this as `b1` will be spawned before $b2$, and $b2$ spawns $b3$); thus, I know $b1$ will happen before $b3$ from Definition 3.

The `"deposit"` message will be logged before the `"transfer"` message: $b3$ and $b7$ require `log`; I know $b2$ spawns $b3$ and $b6$ spawns $b7$, and I know $b2$ happens before $b6$ (as both require `src` and $b2$ spawns before $b6$); therefore, I know $b3$ will be spawned before $b7$ and so $b3$ will happen before $b7$ (as both require `log` and $b3$ spawns before $b7$).

These constraints are presented in the execution timeline in the right of Listing 3.8: this depicts which behaviours must happen before which others (green arrows), we can see for example that $b2$ and $b4$ can happen in either order, but both must happen before $b6$, and thus $b3$ and $b5$ happen before $b7$. Yet, there is no happens before order between $b3$ and $b5$ and so these behaviours can run in either order.

3.7 Cost of Order

I showed how a desired order can be obtained by construction of a program, but what about when order is not desired? Consider a very simplified version of the Dining Philosophers with 4 forks and 4 philosophers each trying to eat once. If I scheduled them in sequential order, then I would fully sequentialise the program. The overlapping cown sets in Listing 3.9 forces all the operations into a single linear order, whereas

Listing 3.9: Sequential scheduling

```
when (f1, f2) { /* b1 */ }
when (f2, f3) { /* b2 */ }
when (f3, f4) { /* b3 */ }
when (f4, f1) { /* b4 */ }
```

Listing 3.10: Alternating scheduling

```
when (f1, f2) { /* b1 */ }
when (f3, f4) { /* b3 */ }
when (f2, f3) { /* b2 */ }
when (f4, f1) { /* b4 */ }
```

Listing 3.10 enforces that $b1$ and $b3$ must occur before $b2$ and $b4$, but no other constraints. Thus, the second program can execute two things in parallel, whereas first can only execute one. The Philosophers problem can be seen as a generalisation of this pattern.

It is important to note that this degenerate case affects *performance* but not *correctness*. In contrast, the failure modes of a similar error in a lock-based implementation are either data races or deadlock. Implementations using transactional memory would have similar problems, with adjacent philosophers attempting to mutate the same state in a transaction and then rolling back and hitting a slow-path with guaranteed ordering. BoC provides an advantage in that the error causing the performance problem is both observable and fixable in the source language. The happens-before ordering is part of the source-level semantics and so can be broken with source-level constructs, such as the interleaved ordering presented above. I believe that this combination of properties—that failure modes affect performance rather than correctness and that the programmer can reason about performance problems at the source-language level—are key benefits to the BoC model.

3.8 Conclusion

BoC provides a single powerful abstraction for parallelism and *flexible* coordination through the concept of **when**. Like Actors, a BoC program can saturate a system with behaviours which can be run in parallel, but, unlike Actors, BoC programs can flexibly coordinate access such that behaviours access multiple cowns. Moreover, BoC guarantees an *ordering on the execution* of behaviours which enables a runtime to *implicitly parallelise* behaviours as long as the order is respected.

4

Operational Semantics

Having established BoC through example, I move on to give a formal model. In this chapter I will build an operational semantics for BoC, which defines how BoC can enrich a underlying language with BoC, and how behaviours are spawned, executed, started and ended. These semantics provide a simple mechanism for ordering behaviours so that I can focus on the parametric calculus I introduce here. This mechanism is more restrictive than I desire and I will revisit the ordering it provides in Chapter 6.

I first define what is expected of the language, atop which BoC is built, in Definition 4. I will then discuss how I add BoC to this language in Definition 5.

BoC can be built on top of any programming language with the necessary building blocks. Thus, our model is parametric in an underlying programming language. In Definition 4 I define the building blocks upon which BoC can be constructed.

Definition 4 (Simple underlying programming language)**.** *A tuple* $(Context, Heap, Tag, \hookrightarrow, \hookrightarrow_{\mathsf{when}\,(\overline{\kappa})\,\{E\}}, finished)$ *is an* underlying programming language *if all of the points that follow hold. The identifiers* $E, E', \ldots$ *range over elements of* $Context$*,* $h, h' \ldots$ *for* $Heap$*, and* $\kappa, \kappa', \ldots$ *range over elements of* Tag*.*

(1) $Context$*,* $Heap$ *and* Tag *are sets.*

(2) The execution relation $\hookrightarrow$ *has signature* $\hookrightarrow \subseteq (Context \times Heap) \to (Context \times Heap)$*.*

(3) The execution relation $\hookrightarrow_{\mathsf{when}\,(\overline{\kappa})\,\{E\}}$ *has signature* $\hookrightarrow_{\mathsf{when}\,(\overline{\kappa})\,\{E\}} \subseteq Context \to Context$*,*

(4) The unary predicate $finished$ *describes terminal contexts.*

I separate the state of the underlying language into $Context$ and $Heap$ (item 1). This will allow us to refer to the local state of a behaviour, e.g. the current expression to execute and a local stack, more generally as an element of $Context$. I refer to the global state, accessible by all behaviours, as an element of $Heap$. I also require a collection of identifiers, Tag, which I will use as cown identifiers when I define BoC. These must come from the underlying language to ensure that the language has the means to understand them, e.g. in expression evaluations or heap lookups.

I require that the underlying language provides a means to evaluate $Context$ and $Heap$ pairs, namely via $\hookrightarrow$ (item 2).

I also require the evaluation relation $\hookrightarrow_{\texttt{when}\ (\overline{\kappa})\ \{E\}}$ (item 3). This is a side-effecting relation which I will use to spawn behaviours when I define BoC. The relation allows an evaluation step to emit a pair of local context and tags. As we will see in Figure 4.1, these tags represent the cowns required by the spawned behaviour. The emitted context allows the spawning behaviour to pass values, by copy or by ownership transfer, to the spawned behaviour.

As BoC does not know anything about the underlying language's state, it does not know what it means for a context to reach a terminal configuration, thus BoC needs to be provided with a predicate to judge when a behaviour has finished (item 4).

In Definition 5 I show how BoC can extend any underlying language to obtain a BoC language. The extension enriches the underlying language to introduces concurrency to the language by enabling *running* multiple behaviours at a time.

Definition 5 (BoC extensions)**.** *I define the Behaviour-Oriented Concurrency (BoC) extension for an underlying language, ($Context_u$, $Heap_u$, Tag_u, $\hookrightarrow$, $\hookrightarrow_{\texttt{when}\ (\overline{\kappa})\ \{E\}}$, $finished_u$), and obtain a BoC tuple ($Context$, $Heap$, Tag, $\hookrightarrow$, $\hookrightarrow_{\texttt{when}\ (\overline{\kappa})\ \{E\}}$, $finished$, $\rightsquigarrow$):*

(1) Configurations are tuples, $PendingBehaviours \times RunningBehaviours \times Heap$ where

$$P \in PendingBehaviours = (Tag^* \times Context)^*$$
$$R \in RunningBehaviours = (Tag^* \times Context) \to \mathbb{N}^+$$

(2) The evaluation relation $\rightsquigarrow$ is defined in Figure 4.1 and has the following signature.

$$\rightsquigarrow \subseteq (PendingBehaviours \times RunningBehaviours \times Heap) \to (PendingBehaviours \times RunningBehaviours \times Heap)$$

A BoC configuration is formed of three components (item 1): the pending behaviours, the running behaviours, and the heap (which has been lifted from the underlying language).

The elements of the running and pending behaviours have the same signature. Namely, they are a pair of a set of cown identifiers (tags) which the behaviour requires, and the current local context of the behaviour.

The distinction between pending and running collections allow us to refer to behaviours as in either a pending or running state. Intuitively, a pending behaviour has been spawned but is not currently executing, and thus will not be accessing any of the cowns which it requires. Conversely, a running behaviour is currently executing and thus will be accessing cown state.

I want to allow two or more running behaviours to require the same cowns and have identical context. I refer to such behaviours as identical. Identical running behaviours do not necessarily result in a data-race, in Section 4.2 I will demonstrate identical running behaviours which have read-only access to their cowns.

To support identical behaviours, I represent running behaviours as a multiset. The $+$ operator on multisets is the analogue of the disjoint union for sets, and I use it in Figure 4.1 to extract a running behaviour..

I represent pending behaviours as a sequence of behaviours. In the discussion that follows Figure 4.1, I will show how this sequence structure is used as part of the *happens before* order.

I will now discuss each of the rules in Figure 4.1.

STEP describes a step where a running behaviour is able to make a step in the underlying language; this updates the global heap and the local context of the behaviour.

SPAWN describes spawning of a new behaviour. The underlying relation updates the spawning behaviour's context to accommodate the local effects of spawning a behaviour (such as updating stacks to

$$\textsc{step}\ \frac{E, h \hookrightarrow E', h'}{R + (\overline{\kappa}, E), P, h \rightsquigarrow R + (\overline{\kappa}, E'), P, h'} \qquad \textsc{spawn}\ \frac{E \hookrightarrow_{\texttt{when}\ (\overline{\kappa'})\ \{E''\}} E'}{R + (\overline{\kappa}, E), P, h \rightsquigarrow R + (\overline{\kappa}, E'), P : (\overline{\kappa'}, E''), h}$$

$$\textsc{start}\ \frac{\left(\bigcup_{(\overline{\kappa'}, \_) \in (P' \cup R)} \overline{\kappa'}\right) \cap \overline{\kappa} = \varnothing}{R, P' : (\overline{\kappa}, E) : P'', h \rightsquigarrow R + (\overline{\kappa}, E), P' : P'', h} \qquad \textsc{end}\ \frac{\mathit{finished}(E)}{R + (\overline{\kappa}, E), P, h \rightsquigarrow R, P, h}$$

Figure 4.1: Semantics for BoC

reflect captured values and reducing expressions). The cowns required by the new behaviour ($\overline{\kappa}$) and its context (E'') are added to the end of the pending behaviours list.

START describes starting a behaviour. A behaviour can START once two criteria are met: (1) no behaviour that appears earlier in list of pending behaviours has overlapping cowns with this behaviours cowns; (2) the cowns required by the behaviour are not in use by any running behaviour. The $\bigcup_{(\overline{\kappa}, \_) \in P' \cup R}$ notation has two important notes: (1) in an abuse of notation, $P' \cup R$ creates the union of the sequence P' and multiset R to create a multiset of all elements from P' and R; (2) the big $\cup$ notation then ranges over the new multiset using the identifiers $(\overline{\kappa}, \_)$ where _ represents an unnamed element.

END describes terminating a behaviour. This step requires that some running behaviour has reached some terminal state. Then, this behaviour can then be removed from the running behaviours.

Ensuring happens-before SPAWN demonstrates the linear structure of P, spawned behaviours are always appended to the end of the list of pending behaviours. In START a behaviour can only be made running and removed from P if there is no prior behaviour in P that requires the same cowns. This means that whenever two behaviours overlap, the earlier spawned behaviour will always be START first. Consider what this means for the example in Listing 3.8: $b1$ will be spawned before $b2$ and, as $b2$ spawns $b3$, also before $b3$ and so $b1$ will precede $b3$ in P, thus $b1$ will be START first.

Ensuring isolation of behaviours These semantics provide a means to schedule behaviours such that no running behaviours are granted access to the same cowns at once. However, to ensure behaviours are data-race free, I must ensure that they do not share access to state in such a way as to cause a data-race. The state of the underlying language is abstract, and so BoC cannot be responsible for ensuring different cowns do not have access to the same state (consider aliasing pointers into the heap). Instead BoC needs to stipulate requirements of the underlying language to guarantee behaviours cannot access the same state (or are isolated). The requirements of the underlying language to ensure isolation is the topic of Chapter 5, but I will touch on it briefly in now.

Assume the underlying language has a mechanism for isolation, such as a type system enforced statically or dynamically. Consider what it means for a step to be allowed in the underlying language in STEP. One definition of this requires that a step must preserve the state of all memory from which this behaviour is isolated[2]. Thus a behaviour can mutate its cowns and local state, as long as it does not affect other isolated memory.

In more detail, for SPAWN I expect that the type system ensures that any shared data accessed within the new context (E'') is protected (uniquely owned) by the required cowns ($\overline{\kappa}$). For END, I expect that the type system ensures that the data protected (owned by) the cowns being released ($\overline{\kappa}$) are disjoint.

Assuming this provisioned isolation, I can claim that behaviours are data-race free.

Ensuring deadlock freedom BoC is deadlock-free (as stated in the following lemma) by construction.

Lemma 2 (Deadlock Free)**.**

$$\forall P, R, h.\ [\ (\exists P', R', h'.\ P, R, h \rightsquigarrow P', R' h')\ \ \vee\ \ P \cup R = \varnothing\ ]$$

BoC is deadlock free as the semantics in Figure 4.1 can always reduce unless all the behaviours have finished. To show this, I assume the underlying semantics cannot get stuck, that is:

Property 1 (Progress)**.**

$$\forall E.\ [\ (\forall h.\exists E', h'.\ E, h \hookrightarrow E', h')\ \ \vee\ \ (\forall h.\exists E', \kappa, E''.\ E, h \hookrightarrow_{\mathsf{when}\ (\overline{\kappa})\ \{E''\}} E', h)\ \ \vee\ \ \mathit{finished}(E)\ ]$$

Property 1 may seem like too strong an assumption, as I expect that given a context and heap pair progress can always be made, yet this can be satisfied fairly simply by permitting error contexts that satisfy the *finished* predicate. In the presence of a heap in which a context cannot reduce, say some dangling pointer, this will step to an error which will then be *finished*. I can also satisfy this by defining well-formed configurations and proving preservation of such over the execution relation, but this is more involved than I require here.

Proof of Lemma 2. I proceed by case analysis on R being empty. If it is not empty, then I can apply the assumption (Property 1) to an element of R, which gives three cases, one for each disjunct. The three cases can reduce by STEP, SPAWN or END respectively. If R is empty and P is non-empty, then the first element of P can be moved to the running multiset using START. If R and P are both empty, then no rules apply and the program has terminated. Hence, the BoC semantics cannot get stuck before termination. □

4.1 Demonstrating Parallelism and Deadlock Freedom

I now demonstrate that Listing 3.4 is deadlock free through the following potential execution. I use `w1`, `w2`, `d1` and `d2`, as behaviour identifiers for withdrawing from and depositing to `s1` and `s2` respectively. In the transition from Config 2 to Config 3, I use the START rule to start the second behaviour in the queue. This is allowed as it does not require any cowns required by an earlier behaviour. If `s1` and `s2` were the same cown, then the second behaviour could not run as it would require the same cown as the first behaviour. I then let the running behaviour complete. At this point (Config 4), there are two behaviours in the pending queue, but only the first can run, because they both require the same cown. This behaviour must now complete, before the remaining behaviour in the queue can run. Once both withdrawals are complete (Config 6), then both deposits can start in either order, and then they can run in parallel (Config 7).

$$\{(\varnothing, \texttt{transfer(s1, s2, 1); transfer(s2, s1, 2)})\}, [], h \rightsquigarrow^{*} \quad (1)$$

$$\varnothing, [(\{s1\}, \texttt{w1; when(s2) \{ d2 \}}), (\{s2\}, \texttt{w2; when(s1) \{ d1 \}})], h \rightsquigarrow \quad (2)$$

$$\{(\{s2\}, \texttt{w2; when(s1) \{ d1 \}})\}, [(\{s1\}, \texttt{w1; when(s2) \{ d2 \}})], h \rightsquigarrow \quad (3)$$

$$\varnothing, [(\{s1\}, \texttt{w1; when(s2) \{ d2 \}}), (\{s1\}, \texttt{d1})], h' \rightsquigarrow^{*} \quad (4)$$

$$\{(\{s1\}, \texttt{w1; when(s2) \{ d2 \}})\}, [(\{s1\}, \texttt{d1})], h' \rightsquigarrow^{*} \quad (5)$$

$$\varnothing, [(\{s1\}, \texttt{d1}), (\{s2\}, \texttt{d2})], h' \rightsquigarrow^{*} \quad (6)$$

$$\{(\{s1\}, \texttt{d1}), (\{s2\}, \texttt{d2})\}, [], h'' \quad (7)$$

4.2 Generalising cown overlap

In this section I will increase the potential for parallelism by extending the underlying language definition from Definition 4 and adapting the operational semantics from Figure 4.1.

It could be safe to run multiple behaviours accessing the same cown at the same time, as long as all behaviours only read the cown; as yet, BoC will not run such behaviours at the same time.

For this section, consider a partial definition of an underlying language where Tag is a pair of address and read/write capability.

Definition 6 (Read/Write Cowns)**.** *The underlying language provides:*

(1) $\iota, \iota', \ldots$ *ranges over the set of addresses in the heap* $Addr$

(2) $Tag = \mathcal{P}(Addr, \{r, w\})$*, where* r *and* w *are read and write capabilities respectively*

Recall START from Figure 4.1. When starting a behaviour, BoC computes the union of running behaviours and earlier spawned pending behaviours Tag^*, and then checks that the union is disjoint with the starting behaviours Tag^*. This means that the BoC semantics will not admit two behaviours to run in parallel using the same cown, regardless of the capability.

I need to allow the underlying language to describe when behaviours can execute in parallel. Thus I need to weaken the BoC semantics, and utilise knowledge provided by the underlying language to decide when to start behaviours.

The two new pieces of knowledge I will require of the underlying language are presented in Definition 7.

Definition 7 (Simple underlying programming language continued)**.** *I expect the underlying language to provide the following operator and judgement.*

(1) The conjuction operator $\circ : Tag^* \circ Tag^* \to Tag^*$

(2) The disjunction judgement $\# : Tag^* \# Tag^*$*.*

Item 1 tells us how to combine sequences of Tag. More importantly, Item 2 tells us when sequences of Tag are disjoint (or do *not* conflict); this allows us to safely start behaviours that have disjoint accesses as per the underlying languages construction.

Revisiting our read/write capabilities, I can instantiate the operators for Definition 7 as follows.

Definition 8. *Instantiating conjuction and disjunction*

(1) $\overline{(\iota_1, p_1)} \circ \overline{(\iota_2, p_2)} \triangleq \overline{(\iota_1, p_1)} : \overline{(\iota_2, p_2)}$

(2) $\overline{(\iota_1, p_1)} \# \overline{(\iota_1, p_1)} \triangleq \forall (\iota_1, p_1) \in \overline{(\iota_1, p_1)}. \forall (\iota_2, p_2) \in \overline{(\iota_2, p_2)}. [\iota_1 = \iota_2 \implies p_1 \neq w \wedge p_2 \neq w]$

Item 2 tells us that two collections of Tag are disjoint as long, whenever the same cown appears in both sequences, neither sequence has a write access for the cown. This means both sequences can access the cown if they are reading, but only one sequence can access the cown if there is a write.

Now I utilise Definition 7 and plug the definitions into the BoC semantics as in Figure 4.2.

$$\textsc{start}\ \frac{\left(\bigcirc_{(\overline{\kappa'}, \_) \in (P' \cup R)} \overline{\kappa'}\right) \# \overline{\kappa}}{R, P' : (\overline{\kappa}, E) : P'', h \rightsquigarrow R + (\overline{\kappa}, E), P' : P'', h}$$

Figure 4.2: START with more freedom

With such a definition, BoC can execute behaviours that access the same addresses with read capabilities in parallel. This demonstrates how the changes to the operational semantics in this section, have increased the potential parallelism.

4.3 Conclusion

In this chapter I have presented the operational semantics for BoC. This illustrates how behaviours are spawned and executed, whilst ensuring BoC is free of deadlocks and data-races. I have demonstrated that the operational semantics dynamically orders the execution of behaviours through a queue-like data structure and disjoint relation.

I have also demonstrated how BoC can enrich an underlying programming language. By making BoC parametric with an abstract underlying programming language, I have demonstrated that BoC can be adopted by any language that can satisfy the necessary requirements. Whilst simple, these semantics are powerful, able to ensure deadlock freedom, isolation, and provide the happens before ordering for behaviours.

5

Isolation

In this chapter, I define the behaviour isolation that BoC can ensure, given the underlying programming language satisfies isolation properties. These properties, outlined in this chapter, constitute the interface between BoC and an underlying programming language. To aid readability, the proofs for this chapter appear in Appendix B.

BoC composes with an underlying language to construct a new language with concurrency. Defining an interface, instead of pairing BoC with a concrete language, allows BoC to be composed with a number of programming languages.

BoC is responsible for scheduling parallel behaviours with non-overlapping cown accesses. However, it is the responsibility of the underlying language to ensure cowns and behaviours remain isolated.

What is isolation? Isolation is a broad term applied to many areas: memory isolation, thread isolation, process isolation and network isolation name just a few. Yet, all of these forms share a common underlying principle: to separate and restrict access to components of a system, thereby preventing interference between components. Behaviour isolation has, at its core, the same principle: one behaviour cannot interfere with another behaviour.

Listing 5.1: Potential data-race

```
1  // open an account with balance 10
2  var acc = Account.create(10);
3
4  // create two cowns which both protect the account
5  var a1 = cown.create(acc);
6  var a2 = cown.create(acc);
7
8  when(a1) { /* b1 */
9    if (a1.balance >= 10)
10     a1.withdraw(10)
11 }
12
13 when(a2) { /* b2 */
14   if (a2.balance >= 10)
15     a2.withdraw(10)
16 }
```

Listing 5.2: Potential non-atomicity

```
1  when(a1, a2) { /* b1 */
2    a1.deposit(10);
3    a2.withdraw(10);
4  };
5  when(a3, a4) { /* b2 */
6    a3.print_balance();
7    a4.print_balance();
8  }
```

Consider the example in Listing 5.1. What does it mean for `b1` to be isolated? Assume `a1` and `a2` are non-aliasing cown identifiers. However, assume both `a1` and `a2` "own" the same `Account`. This means there is a race condition between `b1` and `b2`. Both behaviours could check their guards, on Lines 9 and 14, and pass (as the opening balance is `10`), and then both withdraw `10`, leaving the account in overdraft.

BoC guarantees running behaviours do not overlap in their required cowns, and so will permit `b1` and `b2` to run in parallel. So far, BoC does *not* guarantee that `a1` and `a2` do not have access to the same area of memory. That is to say, alone, BoC's behaviour scheduling does not ensure behaviours are data-race free. To guarantee this, BoC requires more from the underlying language.

Consider the example in Listing 5.2, is `b1` an atomic operation? can `b2` witness a partial completion of `b1`? Similar to Listing 5.1, currently the cowns `a1, a2` could be owning the same contents as `a3, a4`. Thus, I the contents of the accounts could be printed part way through a transfer, and seemingly "lose" funds in the reporting.

To ensure such scenarios are free of data-races and so BoC can guarantee behaviours are isolated, the underlying language needs to satisfy a number of isolation properties which are defined in this chapter.

Isolation provides the guarantee that behaviours execute as if no other behaviour is executing at the same time.

5.1 Identifying a behaviour

In this section I will enrich the semantics from Figure 4.1 such that *running* behaviours have unique identifiers.

Figure 4.1 captures the execution of BoC program, however if I need to characterise the execution of a particular behaviour then I am going to struggle. Recall from Section 4.2 that it may be that two running behaviours use the same cowns (with read-only access), and note also that behaviours may have the same local context. This means there is no unique element of a behaviour that will allow us to unambiguously identify a behaviour.

Now I enrich the operational semantics from Chapter 4 with behaviour identifiers. Recall the definition of BoC from Definition 5; I now add behaviour identifiers and change the running multi-set to a map of behaviour identifier to behaviour context, with the signature as in Definition 9. I leave the pending list unchanged as pending behaviours do not need to be identified. These pending behaviours cannot execute and so cannot be the source of data-races, and thus I do not focus on their state until they begin executing. I

also want to retain the list structure of the pending behaviours to obtain our behaviour ordering; this order would be more complicated to obtain if the pending structure also became a map. I will also update the evaluation relation to accommodate these changes in Figure 5.1.

Definition 9 (BoC with behaviour identifiers)**.**

$$\beta \in \mathit{BehavID} = \mathrm{N}$$
$$R \in \mathit{RunningBehaviours} = (\mathit{BehavID} \to (\mathit{Tag}^* \times \mathit{Context}))$$

$$\textsc{step}\ \frac{E, h \hookrightarrow E', h'}{R[\beta \mapsto (\overline{\kappa}, E)], P, h \rightsquigarrow R[\beta \mapsto (\overline{\kappa}, E')], P, h'}$$

$$\textsc{spawn}\ \frac{E \hookrightarrow_{\mathsf{when}\ (\overline{\kappa'})\ \{E''\}} E'}{R[\beta \mapsto (\overline{\kappa}, E)], P, h \rightsquigarrow R[\beta \mapsto (\overline{\kappa}, E')], P : (\overline{\kappa'}, E''), h}$$

$$\textsc{start}\ \frac{\left(\bigcirc_{(\overline{\kappa'}, \_) \in (P' \cup rng(R))} \overline{\kappa'}\right) \# \overline{\kappa} \qquad \beta \notin dom(R)}{R, P' : (\overline{\kappa}, E) : P'', h \rightsquigarrow R[\beta \mapsto (\overline{\kappa}, E)], P' : P'', h}$$

$$\textsc{end}\ \frac{\mathit{finished}(E)}{R[\beta \mapsto (\overline{\kappa}, E)], P, h \rightsquigarrow R \backslash \beta, P, h}$$

Figure 5.1: Semantics for BoC with behaviour identifiers

Figure 5.1 is an adaptation of Figure 4.1 to utilise behaviour identifiers. For STEP, SPAWN, and END I have taken each of the original evaluation rules, but now I use a behaviour identifier to select which behaviour to evaluate. For START the rule is the same as the original, but a fresh identifier is selected for the running behaviour. Now, two behaviours with the same cowns and context are still uniquely identifiable as they will have different identifiers.

5.2 Defining an Interface

In this section I define the interface between BoC and the underlying language. I define the constituent properties of the interfere. I prove isolation of behaviours in BoC.

In the operational semantics so far, a behaviour can access anything within its local context and shared heap. To be able to ensure that different behaviours are isolated, I need to ensure different behaviours have different accesses. As I have chosen to make the local context and heap abstract, I need another abstraction to discuss what a behaviour can access. To achieve this, I take inspiration from the views framework [2] and encapsulate the *access rights* of a behaviour in a view in Definition 10.

Definition 10 (Views)**.**

$$s \in \mathit{BehaviourState} = \{p, r\}$$
$$BV \in \mathit{BehaviourView} = (\mathit{Tag}^* \times \mathit{Context} \times \mathit{BehaviourState})$$
$$V \in \mathit{View} = BV \mid BV * V$$

Definition 10 describes the view of a behaviour as the tuple of the cowns required by that behaviour, the local context of that behaviour, and an element to indicate that behaviour's current state (namely, whether it is pending p or running r).

Definition 11 (Extended underlying language)**.** *For a BoC to ensure behaviour isolation, the underlying language must provide definitions for the following:*

(1) A set $Access$ where $\alpha, \alpha', \ldots$ ranges over members of $Access$

(2) An associative and commutative operation which composes members of $Access$

$$* : Access \times Access \to Access$$

(3) A judgement which judges whether a member of $Heap$ and a member of $Access$ model the view of a behaviour:

$$Heap \times Access \vDash BehaviourView$$

Intuitively the judgement on views expresses a heap (h) and access (α) pair that model a behaviour that is running or pending, i.e. the things a behaviour can access when it is running and when it is pending. The access rights of a behaviour is derived from the cowns and local context of this behaviour. A possible definition of $Access$ may simply be the locations of the heap which the behaviour can access. In practice, the behaviour may have more complex access rights (for example read rights of shared immutable data, and differing cown access capabilities).

The judgement should hold when the heap and access are able to model this behaviour; this may require that the behaviour does not have dangling pointers in the heap, or attempt to access memory for which it does not have the necessary cowns. Consider, also, how the definition of these judgements may differ when a behaviour is pending or running. A pending behaviour will have access to all local state in the context, E, but *not* the state of the cowns; whereas, the running behaviour will also have access to the state of the cowns.

BoC builds on these judgements to construct the following isolation definition:

Definition 12 (Isolation)**.**

$$h, \alpha \vDash (\overline{\kappa}, E, s) * V \iff \exists \alpha_1, \alpha_2.[h, \alpha_1 \vDash (\overline{\kappa}, E, s) \wedge h, \alpha_2 \vDash V \wedge \alpha_1 * \alpha_2 = \alpha]$$

The underlying language's judgements on views, in Definition 11, gives us the base case, while Definition 12 gives us the inductive case. Intuitively, Definition 12 says that the view of a behaviour composes with the view of arbitrarily many other behaviours V, if and only if I can split the access α into two composable accesses α_1 and α_2 which model the views $(\overline{\kappa}, E, s)$ and V respectively. Put simply, the views of behaviours compose if and only if the accesses of the behaviours compose.

An interpretation of views and composition

The underlying language must provide the definition of the access set, the heap, the view judgement and access composition. However, to aid understanding of these concepts, I provide an interpretation of views and composition.

A behaviour's access set (α in Definition 11) is the locations in memory which the behaviour can access; these access sets compose as long as the memory locations which they can access are disjoint. Figure 5.2 depicts the view of two behaviours: b_1 can access locations 0 and 1 and b_2 can access locations 3 and 4. As the memory locations are disjoint, the behaviour views compose. Figure 5.3 demonstrates behaviour views which do not compose: b_3 requires accesses which overlap with the accesses of b_1 and b_2.

I will reuse this interpretation throughout the chapter to explain isolation Properties 1 to 5.

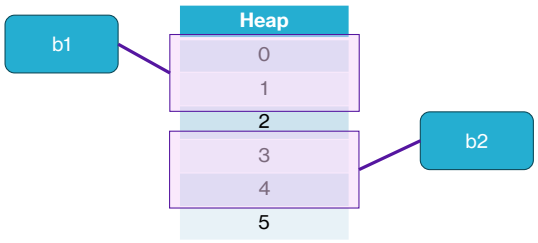


Figure 5.2: Behaviour views

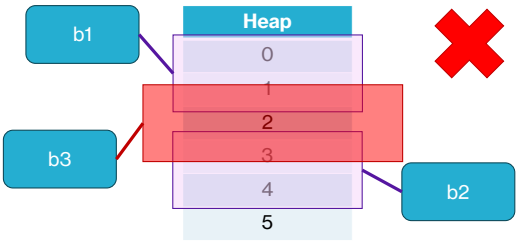


Figure 5.3: Behaviour views which do not compose

5.2.1 Isolation properties of the underlying language

To preserve isolation during the execution of a BoC program, I require that the underlying language satisfy four isolation properties. To motivate theses properties, I revisit the four BoC evaluation steps, and deduce the properties required.

STEP When a running behaviour evaluates, will the new behaviour view compose with the existing behaviour views in the new global state? I define running isolation in Property 2. This property ensures that evaluation in the underlying preserves view composition.

SPAWN When a new behaviour is spawned, will the new running behaviour view and pending behaviour view compose with the existing behaviour views? I define spawn isolation in Property 3.

START When a pending behaviour view becomes a running view, will it compose with the existing behaviour views? I define start isolation in Property 4

END When a behaviour finishes, will removing the view preserve the composition if the existing behaviour views? This is guaranteed by the definition of isolation Definition 12.

The underlying language must satisfy the four isolation properties (all variables are universally quantified unless otherwise stated). Additionally, the underlying language must satisfy the progress property from Chapter 4 (namely, Property 1) repeated here.

Property 1 (Progress)**.**

$$\forall E.\ [\ (\forall h.\, \exists E', h'.\ \ E, h \hookrightarrow E', h') \ \ \vee \ \ (\forall h. \exists E', \kappa, E''.\ \ E, h \hookrightarrow_{\mathsf{when}\ (\overline{\kappa})\ \{E''\}} E', h) \ \ \vee \ \ \mathit{finished}(E)\ ]$$

Property 2 (Running Isolation)**.**

$$E, h \hookrightarrow E', h' \wedge (\exists \alpha. h, \alpha \vDash (\overline{\kappa}, E, r) * V) \implies (\exists \alpha. h', \alpha \vDash (\overline{\kappa}, E', r) * V)$$

This property states that if a behaviour b is isolated from some behaviours, when the behaviour b executes it will not affect other behaviours. An example of this is depicted in Figure 5.4: a step in b results

in the access set of b growing, but the view of everything else remains the same, and composes with the augmented view of b.

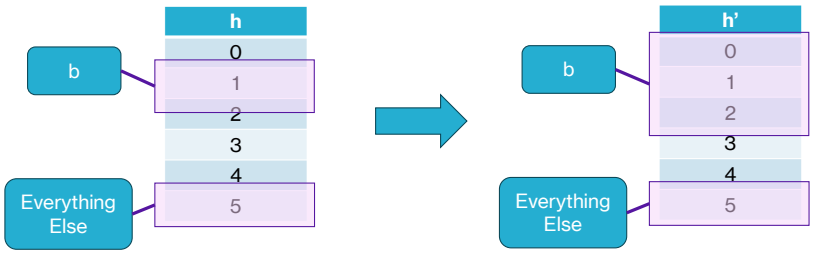


Figure 5.4: Behaviour views maintain composition during execution

Property 3 (Spawn Isolation)**.**

$$E \hookrightarrow_{\mathsf{when}\ (\overline{\kappa'})\ \{E''\}} E' \wedge (\exists \alpha.h, \alpha \vDash (\overline{\kappa}, E, r) * V) \implies (\exists \alpha.h, \alpha \vDash (\overline{\kappa}, E', r) * V * (\overline{\kappa'}, E'', p))$$

This property states that when an isolated behaviour, b, spawns a new behaviour, b', both b and b' will be isolated.

An example of this is depicted in Figure 5.5: b_1 spawns the behaviour b_2 (in purple to represent the behaviour is pending), and none of the behaviour views overlap.

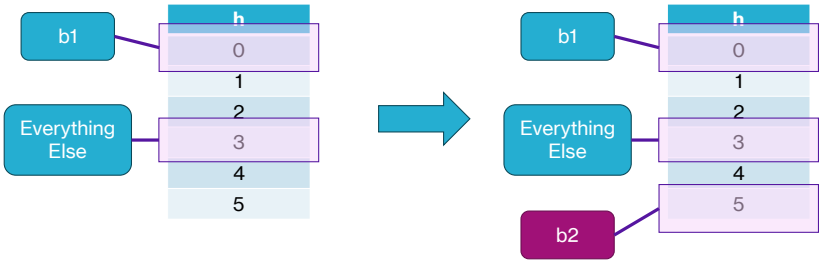


Figure 5.5: Behaviour views maintain composition during spawning

Property 4 (Start Isolation)**.**

$$(\exists \alpha.h, \alpha \vDash (\overline{\kappa}, E, p) * V) \wedge \left(\overline{\kappa} \# \bigcirc_{(\overline{\kappa'}, \_, r) \in V} \overline{\kappa'}\right) \implies (\exists \alpha.h, \alpha \vDash (\kappa, E, r) * V)$$

This property states that if all of the cowns required by an isolated pending behaviour are available, then when that behaviour starts, the behaviour will still be isolated.

Like $\bigcup_{i \in \ldots}$ is the result of multiple unions over a range of elements, $\bigcirc_{i \in \ldots}$ is the result of a multiple $\circ$ operations over a range of sequences of tags. In an abuse of notation, I am using $(\overline{\kappa}, \_, r) \in V$ to destructure V and select all behaviour views where the state is running r,

An example of this is depicted in Figure 5.6: the pending behaviour b starts and remains isolated. Note that the view of b grows, this may happen as the behaviour now has access to the cowns that it requires an can access more of the heap.

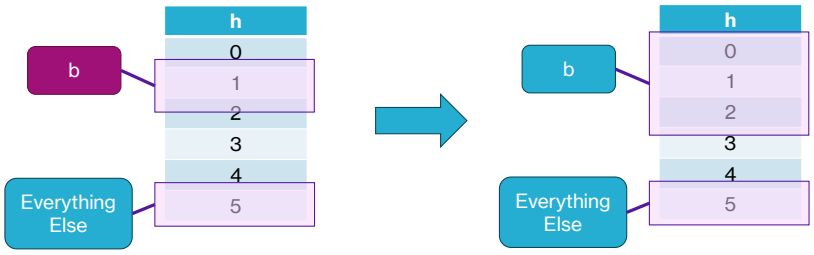


Figure 5.6: Behaviour views maintain composition during starting

Property 5 (Data-race freedom)**.**

$$
\begin{aligned}
&(\exists \alpha. h_1, \alpha \vDash (\overline{\kappa_1}, E_1, r) * (\overline{\kappa_2}, E_2, r)) \\
\wedge\, &E_1, h_1 \hookrightarrow E_1', h_2 \\
\wedge\, &E_2, h_2 \hookrightarrow E_2', h_3 \\
\Longrightarrow\, &\exists h_4. (E_2, h_1 \hookrightarrow E_2', h_4 \wedge E_1, h_4 \hookrightarrow E_1', h_3)
\end{aligned}
$$

This property states that if I take two isolated behaviours and execute one and then the other, I could have executed the behaviours in the opposite order and reach the same final resulting heap.

An example of this is depicted in Figure 5.7 where b_1 can either mutate the memory cell 1 to 42 and then $b2$ can increase its access set (perhaps through some allocation), or vice versa.

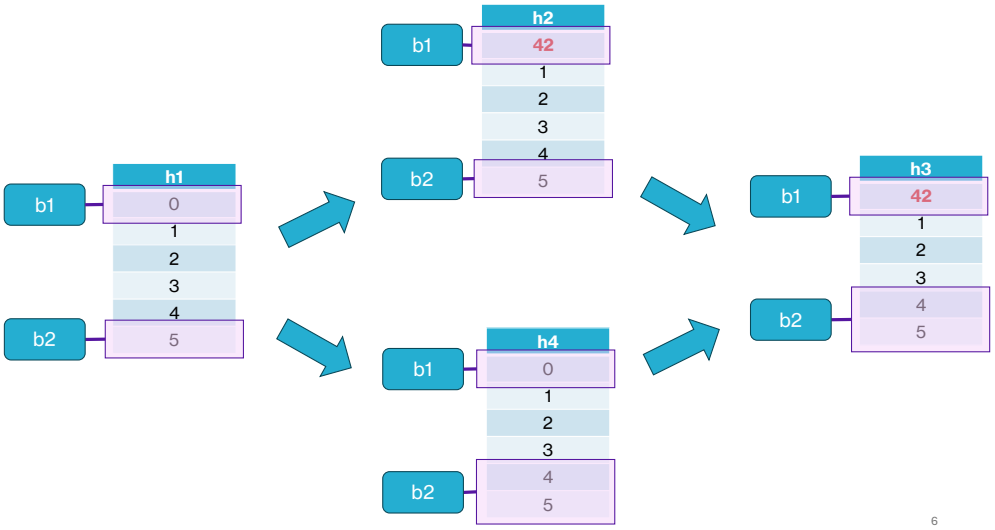


Figure 5.7: Data-race free executions

This is a strong requirement of the underlying language and it has some limitations. For one, the underlying language cannot use an incremental allocator; if both behaviours allocate then swapping the order will result in a different pair of resulting behaviours. This can be overcome in the underlying language using equivalence classes which would relate the equivalent final states (say, for example, equivalent up to renaming of addresses). Moreover, BoC could build a similar property from equivalence classes required to be defined as part of the underlying language. This would require more properties of the underlying language and would make the interface more complex. I use Property 5 in this thesis as it simplifies the lemmas in this chapter, and their proofs.

The rest of this chapter assumes that the underlying language provides Properties 1 to 5.

5.2.2 Well-formed configurations

A BoC configuration is well-formed when the composition of the views of all behaviours compose.

Definition 13 (Well-Formed)**.** *A configuration* P, R, h *is* well-formed *iff:*

$$\exists \alpha.[h, \alpha \vDash *_{(\overline{\kappa},E)\in rng(R)}(\overline{\kappa}, E, r) * *_{(\overline{\kappa},E)\in P}(\overline{\kappa}, E, p)]$$

Definition 13 tells us that a configuration is well-formed if the view of all of the pending and running behaviours compose. That is to say, all of the behaviours are appropriately isolated.

Preservation of well-formedness I now have all the ingredients to demonstrate that, when provided with an underlying language that satisfies Properties 2 to 4, BoC maintains well-formedness of configurations.

Lemma 3 (Preservation of Well-Formed)**.**

$$\forall P, P', R, R', h, h'.[R, P, h \text{ is well-formed} \wedge R, P, h \rightsquigarrow R', P', h' \implies R', P', h' \text{ is well-formed}]$$

5.3 Atomicity

I want that behaviours can be considered as indivisible operations between spawns (or between **when**s). In other words, interleaving the steps of a behaviour with other behaviours is no different than executing the behaviour un-interleaved. This is presented in Lemma 7; in this section I will build to show how this lemma can be proven.

5.3.1 Program evaluation for specific behaviours

To identify the evaluation of a specific behaviour, I need to be able to discuss the evaluation of (or the lack of evaluation of) that behaviour. Definition 14 is a restriction of the BoC evaluation relation such that there must exist some behaviour β that can be evaluated.

Definition 14 (β step)**.** *The evaluation relation $\underset{\beta}{\rightsquigarrow}$, with the following signature, is special case of the evaluation relation from Figure 5.1.*

$$\begin{aligned}\underset{\beta}{\rightsquigarrow} \subseteq &(PendingBehaviours \times RunningBehaviours \times Heap) \\ &\to (PendingBehaviours \times RunningBehaviours \times Heap)\end{aligned}$$

Only the behaviour identifier β can evaluate, and the behaviour can only make steps defined in the underlying language. Importantly, the behaviour cannot, start, nor spawn a behaviour, nor end.

$$\textsc{step}_\beta \frac{E, h \hookrightarrow E', h'}{R[\beta \mapsto (\overline{\kappa}, E)], P, h \;\rightsquigarrow_\beta\; R[\beta \mapsto (\overline{\kappa}, E')], P, h'}$$

Note, I only allow the behaviour β to STEP; the behaviour cannot START, END, nor SPAWN and I will explain why this is necessary in Lemma 5.

I now go on to define the complement, namely the $\sim\beta$ step in Definition 15. This step ensures that β does not change during an evaluation step. This relation can do all of the same steps as Figure 4.1 as long as β doesn't change.

Definition 15 ($\sim\beta$ step)**.** *The evaluation relation $\underset{\sim\beta}{\rightsquigarrow}$, with the following signature, is another special case of the evaluation relation from Figure 5.1, and the complement to Definition 14.*

$$\begin{aligned}\underset{\sim\beta}{\rightsquigarrow} \subseteq &(PendingBehaviours \times RunningBehaviours \times Heap) \\ &\to (PendingBehaviours \times RunningBehaviours \times Heap)\end{aligned}$$

The behaviour with the identifier β can not *evaluate, nor spawn, nor start, nor end. Any other behaviour can perform any of the steps from the BoC semantics with behaviours identifiers in Figure 5.1.*

$$\textsc{step}_{\sim\beta}\ \frac{\beta' \neq \beta \qquad E, h \hookrightarrow E', h'}{R[\beta' \mapsto (\overline{\kappa}, E)], P, h \ \rightsquigarrow_{\sim\beta}\ R[\beta' \mapsto (\overline{\kappa}, E')], P, h'}$$

$$\textsc{spawn}_{\sim\beta}\ \frac{\beta' \neq \beta \qquad E \hookrightarrow_{\mathsf{when}\ (\overline{\kappa'})\ \{E''\}} E'}{R[\beta' \mapsto (\overline{\kappa}, E)], P, h \ \rightsquigarrow_{\sim\beta}\ R[\beta' \mapsto (\overline{\kappa}, E')], P : (\overline{\kappa'}, E''), h}$$

$$\textsc{start}_{\sim\beta}\ \frac{\beta' \neq \beta \qquad \left(\bigcirc_{(\overline{\kappa'}, \_) \in (P' \cup rng(R))} \overline{\kappa'}\right) \# \overline{\kappa} \qquad \beta' \notin dom(R)}{R, P' : (\overline{\kappa}, E) : P'', h \ \rightsquigarrow_{\sim\beta}\ R[\beta' \mapsto (\overline{\kappa}, E)], P' : P'', h}$$

$$\textsc{end}_{\sim\beta}\ \frac{\beta' \neq \beta \qquad \mathit{finished}(E)}{R[\beta' \mapsto (\overline{\kappa}, E)], P, h \ \rightsquigarrow_{\sim\beta}\ R \backslash \beta', P, h}$$

Now I define the evaluation which is the union of these two evaluations in Definition 16, and demonstrate that this new relation preserves well-formed in Lemma 4.

Definition 16 (β related step)**.**

$$R_1, P_1, h_1 \underset{\beta/\sim\beta}{\rightsquigarrow} R_2, P_2, h_2 \triangleq (R_1, P_1, h_1 \underset{\beta}{\rightsquigarrow} R_2, P_2, h_2) \ \vee \ (R_1, P_1, h_1 \underset{\sim\beta}{\rightsquigarrow} R_2, P_2, h_2)$$

Lemma 4 (Preservation of well-formed for restricted operations)**.**

$$R_1, P_1, h_1 \textit{ is well-formed} \ \wedge \ (R_1, P_1, h_1 \underset{\beta/\sim\beta}{\rightsquigarrow} R_2, P_2, h_2) \implies R_2, P_2, h_2 \textit{ is well-formed}$$

Proof. The restricted operations are a subset of the executions for Figure 4.1, which preserves well-formed by Lemma 3. Thus, these restricted semantics also preserve well-formed. □

5.3.2 Deinterleaving executions

Now I get into the meat of the matter, to prove that the effect of interleaving β steps with $\sim\beta$ steps is the same as doing all of the β steps atomically, I need to be able to reorder steps of our evaluation. I do precisely this in Lemma 5, I show that if I first evaluate a $\sim\beta$ step and then a β step, then I can find some new intermediate configuration that allows us to do the two steps in the opposite order but end up in the same final configuration. Lemma 6 builds on this, demonstrating the the property holds for arbitrarily many $\sim\beta$ steps.

Lemma 5 (Swap step)**.**

$$\begin{aligned} & R_1, P_1, h_1 \textit{ is well-formed} \\ \wedge\ & R_1, P_1, h_1 \underset{\sim\beta}{\rightsquigarrow} R_2, P_2, h_2 \underset{\beta}{\rightsquigarrow} R_3, P_3, h_3 \\ \implies & \exists R_4, P_4, h_4. [R_1, P_1, h_1 \underset{\beta}{\rightsquigarrow} R_4, P_4, h_4 \underset{\sim\beta}{\rightsquigarrow} R_3, P_3, h_3] \end{aligned}$$

Lemma 6 (Swap steps generalised)**.**

$$\begin{aligned} & R_1, P_1, h_1 \textit{ is well-formed} \\ \wedge\ & R_1, P_1, h_1 \overset{n}{\underset{\sim\beta}{\rightsquigarrow}} R_2, P_2, h_2 \underset{\beta}{\rightsquigarrow} R_3, P_3, h_3 \\ \implies & \exists R_4, P_4, h_4.[R_1, P_1, h_1 \underset{\beta}{\rightsquigarrow} R_4, P_4, h_4 \overset{n}{\underset{\sim\beta}{\rightsquigarrow}} R_3, P_3, h_3] \end{aligned}$$

Atomicity Lemma 7 sets out our atomicity result. In essence, if take an evaluation with interleaved steps from β and $\sim\beta$ then I can always deinterleave the steps into all the β steps followed by all the $\sim\beta$ steps. (Note that β must initially be running due to the construction of the two evaluation relations). I refer to this property as atomic*ish*; the behaviour β cannot START, SPAWN nor END as these rules affect the global state, so I cannot consider the execution of the behaviour to be completely atomic.

Lemma 7 (Atomicish)**.**

$$\begin{aligned} & R_1, P_1, h_1 \textit{ is well-formed} \\ \wedge\ & R_1, P_1, h_1 \overset{n}{\underset{\beta/\sim\beta}{\rightsquigarrow}} R_2, P_2, h_2 \\ \implies & \exists R_3, P_3, h_3, n_1, n_2.[R_1, P_1, h_1 \overset{n_1}{\underset{\beta}{\rightsquigarrow}} R_3, P_3, h_3 \overset{n_2}{\underset{\sim\beta}{\rightsquigarrow}} R_2, P_2, h_2 \\ & \qquad \wedge\ n = n_1 + n_2] \end{aligned}$$

Increasing the atomicity

A stronger result than Lemma 7 could be achieved if I were to include both the START and END of a behaviour β. This would allow us to deinterleave all of the steps from β when the behaviour doesn't spawn. Currently, I cannot achieve this result as the behaviour identifiers in Figure 5.1 can be reused once a behaviour terminates. Replacing the terminating behaviour with some empty state, say $(\epsilon, \texttt{unit})$ or (ϵ, E), would obtain this stronger result; I leave this as future work.

Partial order reduction

The reader may wonder about the connection between de-interleaving, atomicity, and partial order reduction. Partial order reduction is a technique for program verification. The technique relies on reducing the number of executions that must be verified, by observing that many executions are redundant. Often, different execution paths will result in identical states, as the different paths correspond to reordered execution steps.[67, 68]

The focus of Lemmas 6 and 7, is to demonstrate the behaviours are atomic and isolated. Thus, a behaviour's execution interleaved and deinterleaved with other behaviours, arrives in the same state. This means we can employ partial order reduction for verifying BoC programs; we can coalesce many different behaviour interleavings into fewer, representative, executions.

5.4 Instantiating the interface

In this section I introduce a simple lambda calculus, $\lambda_{\texttt{when}}$, and demonstrate that it satisfies the interface Properties 1 to 5. This demonstrates that the interface properties are realistic and satisfiable for language designers.

Definition 17 defines the $\lambda_{\texttt{when}}$ language, a simple lambda calculus with references derived from [69] but with three noteworthy modifications. Firstly, references are now cowns, this is largely syntactic but instead of syntax for creating a `ref`, one creates a **`cown`** (the `!` operator is still used to dereference cowns). This means *all* heap allocations become cowns in this language. Secondly, there is a new **`when`** syntax for spawning behaviours. Thirdly, contexts carry the cowns which they have permission to access.

A context, $\mathcal{C}$, is a pair of the cowns the execution has access, and the expression which is currently being evaluated.

Definition 17 ($\lambda_{\texttt{when}}$)**.** *(1) The definition of the language λ_{when} is as follows:*

$t ::=$		*terms*	$\mathsf{E} ::=$		*evaluation context*
	x	*variable*		$\lvert[\bullet]$	
	$\lvert\lambda x.t$	*abstraction*		$\lvert\mathsf{E}\ t$	
	$\lvert t\ t$	*application*		$\lvert v\ \mathsf{E}$	
	$\lvert t; t$	*sequence*		$\lvert\mathsf{E}; t$	
	$\lvert$*cown* t	*cown creation*		$\lvert$!E	
	$\lvert$!t	*deref*		$\lvert\mathsf{E} := t$	
	$\lvert t := t$	*assign*		$\lvert v := \mathsf{E}$	
	$\lvert$*when*$(\overline{t})\{t\}$	*when*		$\lvert$*when*$(\overline{v}\ \mathsf{E}\ \overline{t})\{t\}$	
	$\lvert$*error*	*error*		$\lvert$*cown* E	
$v ::=$		*values*			
	$unit$	*constant unit*	$h ::=$		*heap*
	$\lvert\lambda x.t$	*abstraction*		$\lvert\varnothing$	*empty store*
	$\lvert\kappa$	*cown*		$\lvert h, \kappa \mapsto v$	*cown binding*

(2) A context is $\mathcal{C} \in Context = \mathcal{P}(Cown^* \times Term)$.

(3) An evaluation relation $\hookrightarrow \subseteq (Context \times Heap) \times (Context \times Heap)$, this is defined in Definition 18

(4) An evaluation relation $\hookrightarrow_{\texttt{when}\ (\overline{\kappa})\ \{E\}} \subseteq Context \to Context$, this is also defined in Definition 18

(5) The finished predicate finished$(_, t, _) \triangleq t \in \{$unit, error$\}$

(6) The disjoint predicate $\overline{\kappa_1} \# \overline{\kappa_2} \triangleq \overline{\kappa_1} \cap \overline{\kappa_1} = \varnothing$

(7) The compose function $\overline{\kappa_1} \circ \overline{\kappa_2} \triangleq \overline{\kappa_1} \cup \overline{\kappa_1}$

Definition 18 (λ_{when} Evaluation Relation)**.**

$$\textsc{hole}\ \frac{(\overline{\kappa}, t), h \hookrightarrow (\overline{\kappa}, t'), h'}{(\overline{\kappa}, \mathsf{E}[t]), h \hookrightarrow (\overline{\kappa}, \mathsf{E}[t']), h'} \qquad \textsc{application}\ \frac{}{(\overline{\kappa}, (\lambda x.t)v), h \hookrightarrow (\overline{\kappa}, t[x \mapsto v]), h}$$

$$\textsc{sequence}\ \frac{}{(\overline{\kappa}, unit; t), h \hookrightarrow (\overline{\kappa}, t), h} \qquad \textsc{cown}\ \frac{\kappa' \notin dom(h)}{(\overline{\kappa}, \mathit{cown}\ v), h \hookrightarrow (\overline{\kappa}, \kappa'), h[\kappa' \mapsto v]}$$

$$\textsc{deref}\ \frac{\kappa' \in \overline{\kappa}}{(\overline{\kappa}, !\kappa'), h \hookrightarrow (\overline{\kappa}, h(\kappa')), h} \qquad \textsc{assign}\ \frac{\kappa' \in \overline{\kappa}}{(\overline{\kappa}, \kappa' := v), h \hookrightarrow (\overline{\kappa}, unit), h[\kappa' \mapsto v]}$$

$$\textsc{stuck}\ \frac{\begin{array}{l}\neg \mathit{finished}(\overline{\kappa}, t) \\ \neg \exists \overline{\kappa'}, t', t'', h'. \left[\left(t' \neq \mathit{error} \ \wedge\ (\overline{\kappa}, t), h \hookrightarrow (\overline{\kappa}, t'), h'\right) \ \vee\ (\overline{\kappa}, t) \hookrightarrow_{\mathsf{when}\ (\overline{\kappa'})\ \{(\overline{\kappa'}, t'')\}} (\overline{\kappa}, t') \right]\end{array}}{(\overline{\kappa}, t), h \hookrightarrow (\overline{\kappa}, \mathit{error}), h}$$

$$\textsc{when-hole}\ \frac{(\overline{\kappa}, t) \hookrightarrow_{\mathsf{when}\ (\overline{\kappa'})\ \{(\overline{\kappa'}, t)\}} (\overline{\kappa}, t')}{(\overline{\kappa}, \mathsf{E}[t]) \hookrightarrow_{\mathsf{when}\ (\overline{\kappa'})\ \{(\overline{\kappa'}, t)\}} (\overline{\kappa}, \mathsf{E}[t'])} \qquad \textsc{when}\ \frac{}{(\overline{\kappa}, \mathit{when}(\overline{\kappa'})\{t\}) \hookrightarrow_{\mathsf{when}\ (\overline{\kappa'})\ \{(\overline{\kappa'}, t)\}} (\overline{\kappa}, \mathit{unit})}$$

Many of the rules in Definition 18 are standard rules for a lambda calculus with references, so I will only discuss those are of most interest here.

COWN A new heap allocation is created and its value set to that provided. The context does *not* add the cown it to its set of accessible cowns.

DEREF If the cown to dereference is in the set of cowns that can be accessed, then the value is found in the heap and replaces the dereference operation.

ASSIGN Much the same as DEREF but the value of the cown in the heap is updated to the new value.

STUCK A catch-all case for when a context can no longer make progress, this allows the context to change to some error state and be marked as finished. This is needed to satisfy the progress claim of Property 1.

WHEN The behaviour spawning rule that will be used by BoC to create new behaviours. This rule passes the required cowns out to BoC, and keeps track of the cowns that were requested as part of spawned behaviours context. Note that I do not need a premise that the cowns are valid cowns. The syntax of the language prevents a programmer from generating cowns, these can only be acquired by creating a cown, reading a cown address from the heap, or written through some term rewrite abstraction.

5.4.1 A small example

I can write the bank transfer example, from Listing 5.3, using λ_{when}. However, without the BoC concurrency semantics, an evaluation of the program will get stuck spawning the behaviour.

Listing 5.3: Bank Transfer in $\lambda_{\texttt{when}}$

```
λ.src.dst.(
  when(src, dst) {
    src = !src - 10;
    dst = !dst + 10
  }
) (cown 10)(cown 0)
```

5.4.2 Satisfying the interface

I can instantiate BoC with $\lambda_{\texttt{when}}$ as the underlying language with the tuple $(\mathit{Context}, \mathit{Heap}, \hookrightarrow, \hookrightarrow_{\texttt{when}\ (\overline{\kappa})\ \{E\}}, \mathit{finished})$ is an *underlying programming language*. This allows us to execute the transfer (or multiple transfers) concurrently.

Definition 19 defines a function to generate the known cowns of a context.

Definition 19 ($\lambda_{\texttt{when}}$ cowns)**.**

$$
\begin{aligned}
\mathit{cowns}(t, h) &= \mathit{cowns}(t, h, \varnothing) \\
&\textbf{\textit{where}} \\
\mathit{cowns}(x, h, \mathit{seen}) &= \varnothing \\
\mathit{cowns}(\lambda x.t, h, \mathit{seen}) &= \mathit{cowns}(t, h, \mathit{seen}) \\
\mathit{cowns}(t_1 t_1, h, \mathit{seen}) &= \mathit{cowns}(t_1, h, \mathit{seen}) \cup \mathit{cowns}(t_2, h, \mathit{seen}) \\
\mathit{cowns}(\mathit{cown}\ t, h, \mathit{seen}) &= \mathit{cowns}(t, h, \mathit{seen}) \\
\mathit{cowns}(!t, h, \mathit{seen}) &= \mathit{cowns}(t, h, \mathit{seen}) \\
\mathit{cowns}(t_1 := t_2, h, \mathit{seen}) &= \mathit{cowns}(t_1, h, \mathit{seen}) \cup \mathit{cowns}(t_2, h, \mathit{seen}) \\
\mathit{cowns}(\mathit{when}(\overline{t_1})\{t_2\}, h, \mathit{seen}) &= \mathit{cowns}(t_1, h, \mathit{seen}) \cup \mathit{cowns}(t_2, h, \mathit{seen}) \\
\mathit{cowns}(\mathit{error}, h, \mathit{seen}) &= \varnothing \\
\mathit{cowns}(\mathit{unit}, h, \mathit{seen}) &= \varnothing \\
\mathit{cowns}(\kappa, h, \mathit{seen}) &= \begin{cases} \{\kappa\} \cup \mathit{cowns}(h(\kappa), h, \{\kappa\} \cup \mathit{seen}) & \kappa \notin \mathit{seen} \\ \varnothing, & \mathit{otherwise} \end{cases}
\end{aligned}
$$

The judgements of well-formed (from Definition 5) are defined for $\lambda_{\texttt{when}}$ in Definition 20.

Definition 20 ($\lambda_{\texttt{when}}$ Well-formed)**.**

$$
\begin{aligned}
h, \alpha \vDash (\overline{\kappa}, (\overline{\kappa}, t), r) &\triangleq \alpha = \overline{\kappa} \subseteq \mathit{dom}(h) \wedge \mathrm{cowns}(t, h) \subseteq \mathit{dom}(h) \\
h, \alpha \vDash (\overline{\kappa}, (\overline{\kappa}, t), p) &\triangleq \alpha = \varnothing \wedge \overline{\kappa} \subseteq \mathit{dom}(h) \wedge \mathrm{cowns}(t, h) \subseteq \mathit{dom}(h)
\end{aligned}
$$

Note that the cowns in the view and the cowns accessible in the $\lambda_{\texttt{when}}$ context must match for the judgement to be defined.

The composition operator for $\lambda_{\texttt{when}}$ is defined in Definition 21.

Definition 21 ($\lambda_{\texttt{when}}$ Compose)**.**

$$\alpha_1 * \alpha_2 = \alpha_1 \uplus \alpha_2$$

This concludes the definition of $\lambda_{\texttt{when}}$. I prove that $\lambda_{\texttt{when}}$satisfies Properties 1 to 5 in Appendix B.0.5.

5.5 Further work

There are a couple of directions in which this work on isolation can proceed which I will now discuss.

5.5.1 Linearization points

I have shown in Lemma 7 that I can reorder the steps of an execution involving a behaviour β, which doesn't SPAWN or END, such that I can deinterleave the steps of β and end up in the same resulting configuration.

I now take that idea further such that a behaviour has linearization points at SPAWN and END. That is, once a behaviour has started, all of the steps of a behaviour appear to happen instantaneously together with the next SPAWN or END. This would greatly simplify the reasoning of BoC programs whilst preserving the inherent parallelism.

I want that a linearized semantics and the original semantics are simulations of one another, as in Conjecture 1.

The operational semantics for linearizable semantics appears in Figure 5.8.

$$\textsc{spawn}\ \frac{E_1, h_1 \hookrightarrow^* E_2, h_2 \qquad E_2 \hookrightarrow_{\mathsf{when}\ (\overline{\kappa'})\ \{E_4\}} E_3}{R[\beta \mapsto (\overline{\kappa}, E_1)], P, h_1 \overset{\mathtt{atom}}{\rightsquigarrow} R[\beta \mapsto (\overline{\kappa}, E_3)], P : (\overline{\kappa'}, E_4), h_2}$$

$$\textsc{end}\ \frac{E_1, h_1 \hookrightarrow^* E_2, h_2 \qquad \mathit{finished}(E_2)}{R[\beta \mapsto (\overline{\kappa}, E)], P, h_1 \overset{\mathtt{atom}}{\rightsquigarrow} R \backslash \beta, P, h_2}$$

$$\textsc{start}\ \frac{\left(\bigcirc_{(\overline{\kappa'}, \_) \in (P_1 \cup rng(R))} \overline{\kappa'}\right) \# \overline{\kappa} \qquad \beta \notin dom(R)}{R, P_1 : (\overline{\kappa}, E) : P_2, h \overset{\mathtt{atom}}{\rightsquigarrow} R[\beta \mapsto (\overline{\kappa}, E)], P_1 : P_2, h}$$

Figure 5.8: Semantics for BoC with linearization points

I also need to define an equivalence relation between configurations in atomic semantics and the non-atomic semantics. For clarity here, I will subscript the atomic configurations as $\__a$ (This is not technically necessary as I am drawing configurations from the same domain). This equivalence relation is difficult to define as I will need behaviours to be a different stages of execution but parts of global configuration to be the same. As such, defining this precisely is left as future work and for now I will simply name the relation as in Definition 22.

Definition 22 (Equivalence relation for atomic semantics)**.**

$$R_a, P_a, h_a \approx R, P, h \triangleq \ldots$$

I use Definition 22 to define a bisimilarity relation in Conjecture 1 (A superscript ? on an evaluation relation is used to denote 0 or 1 steps). Recall that in the atomic world I am collapsing a number of behaviour-local steps into a single global step. As such, the atomic semantics may not always transition whilst the original semantics catches up.

Conjecture 1 (Bisimulation of atomic semantics)**.**

$$R_a, P_a, h_a \approx R, P, h \wedge R, P, h \rightsquigarrow R', P', h'$$
$$\implies \exists R'_a, P'_a, h'_a.[R_a, P_a, h_a \overset{?}{\overset{\texttt{atom}}{\rightsquigarrow}} R'_a, P'_a, h'_a \wedge R', P', h' \approx R'_a, P'_a, h'_a]$$

and

$$R_a, P_a, h_a \approx R, P, h \wedge R_a, P_a, h_a \overset{\texttt{atom}}{\rightsquigarrow} R'_a, P'_a, h'_a$$
$$\implies \exists R', P', h'.[R, P, h \overset{*}{\rightsquigarrow} R', P', h' \wedge R', P', h' \approx R'_a, P'_a, h'_a]$$

Taking this conjecture further is left as future work. However, I will show that it closely relates to Chapter 6 when I reason about behaviours in terms of only the START, SPAWN and END events.

5.5.2 Cown update atomicity

The atomicish property (Lemma 7) has to consider the effect that executing a behaviour has on the pending queue. However, a behaviour spawning another behaviour does not affect the outcome of the behaviour itself, and the updates to the cowns state can still be viewed as atomic. In this section, I propose a different result in Conjecture 2 that captures this idea.

I need to first redefine our restricted evaluation relations (similar to Definition 14 and Definition 15). The first is Definition 23 which states that there must be a running behaviour with a particular identifier β, and requires that the evaluation relation was a STEP or SPAWN which involved the behaviour β. (Note, now I am allowing β to SPAWN).

Definition 23 (β step)**.**

$$\textsc{step}_\beta \frac{E, h \hookrightarrow E', h'}{R[\beta \mapsto (\overline{\kappa}, E)], P, h \;\rightsquigarrow_\beta\; R[\beta \mapsto (\overline{\kappa}, E')], P, h'}$$

$$\textsc{spawn}_\beta \frac{E \hookrightarrow_{\mathsf{when}\ (\overline{\kappa'})\ \{E''\}} E'}{R[\beta \mapsto (\overline{\kappa}, E)], P, h \;\rightsquigarrow_{\sim\beta}\; R[\beta \mapsto (\overline{\kappa}, E')], P : (\overline{\kappa'}, E''), h}$$

The second restricted relation is Definition 24. This states that some behaviour with some identifier that is not β START, SPAWNS, STEPS, or ENDS.

Definition 24 ($\sim\beta$ step)**.**

$$\textsc{step}_{\sim\beta}\ \frac{\beta' \neq \beta \qquad E, h \hookrightarrow E', h'}{R[\beta' \mapsto (\overline{\kappa}, E)], P, h \ \rightsquigarrow_{\sim\beta}\ R[\beta' \mapsto (\overline{\kappa}, E')], P, h'}$$

$$\textsc{spawn}_{\sim\beta}\ \frac{\beta' \neq \beta \qquad E \hookrightarrow_{\mathsf{when}\ (\overline{\kappa'})\ \{E''\}} E'}{R[\beta' \mapsto (\overline{\kappa}, E)], P, h \ \rightsquigarrow_{\sim\beta}\ R[\beta' \mapsto (\overline{\kappa}, E')], P : (\overline{\kappa'}, E''), h}$$

$$\textsc{start}_{\sim\beta}\ \frac{\beta' \neq \beta \qquad \left(\bigcirc_{(\overline{\kappa'}, \_) \in (P' \cup rng(R))} \overline{\kappa'}\right) \# \overline{\kappa} \qquad \beta' \notin dom(R)}{R, P' : (\overline{\kappa}, E) : P'', h \ \rightsquigarrow_{\sim\beta}\ R[\beta' \mapsto (\overline{\kappa}, E)], P' : P'', h}$$

$$\textsc{end}_{\sim\beta}\ \frac{\beta' \neq \beta \qquad \mathit{finished}(E)}{R[\beta' \mapsto (\overline{\kappa}, E)], P, h \ \rightsquigarrow_{\sim\beta}\ R \backslash \beta', P, h}$$

With Definitions 14 and 15 in mind, I construct a relation with allows β steps n times and $\sim\beta$ steps infinitely many times in Definition 25.

Definition 25 (β related step)**.** *where* $n \in \mathbb{N}$

$$R_1, P_1, h_1 \underset{\beta^n/{\sim\beta}*}{\rightsquigarrow} R_2, P_2, h_2 \triangleq \begin{cases} R_1, P_1, h_1 \overset{*}{\underset{\sim\beta}{\rightsquigarrow}} R_2, P_2, h_2 & \text{if } n = 0 \\ \exists R_3, P_3, h_3 . [R_1, P_1, h_1 \underset{\beta}{\rightsquigarrow} R_3, P_3, h_3 \underset{\beta^{n-1}/{\sim\beta}*}{\rightsquigarrow} R_2, P_2, h_2 & \text{otherwise} \\ \qquad \vee\ R_1, P_1, h_1 \underset{\sim\beta}{\rightsquigarrow} R_3, P_3, h_3 \underset{\beta^n/{\sim\beta}*}{\rightsquigarrow} R_2, P_2, h_2] & \end{cases}$$

I want that no matter how a behaviour gets interleaved with other behaviours, as long as the behaviour makes the same number of steps it will always end up in an equivalent state in the resulting configurations. So I want to be able to consider two configurations similar for some behaviour; this is what I define in Definition 26. But, what does it mean for two states to be considered β-similar? We're going to define two configurations as β-similar if given both configurations agree on the state of the behaviour β, and the behaviours agree on the reachable parts of both heaps.

Property 6 (State, Behaviour projection, and State similarity)**.** *I assume the underlying provides three further properties.*

A collection I refer to as $State$*, a behaviour projection function, and a state similarity function.*

The behaviour projection function that provides so state *visible by a view for a given heap*

$$\Downarrow \subseteq (Heap \times View \times State)$$
$$\sim \subseteq (State \times State)$$

Building on these properties I define the β-similar relation which is intended to consider two configurations similar if they agree on the state of a particular behaviour β under the configurations heap.

Definition 26 (β-similar)**.**

$$R_1, P_1, h_1 \sim_\beta R_2, P_2, h_2 \triangleq R_1(\beta) = (\overline{\kappa_1}, E_1) \ \wedge\ R_2(\beta) = (\overline{\kappa_2}, E_2) \\ \wedge\ (h_1 \Downarrow (\overline{\kappa_1}, E_1, r)) \sim (h_2 \Downarrow (\overline{\kappa_2}, E_2, r))$$

I now define what it means for a behaviour to be atomic using behaviour identifiers. If two β-similar configurations execute β the same number of times, n, interleaved with other behaviors infinitely many times, then the resulting states are also β-similar (recall that β-similar requires the behaviour β to be in the running map). This is presented in Conjecture 2

Conjecture 2 (Behavioural Consistency)**.**

$$
\begin{aligned}
& R_1, P_1, h_1 \textit{ is well-formed} \\
\wedge\ & R_2, P_2, h_2 \textit{ is well-formed} \\
\wedge\ & R_1, P_1, h_1 \sim_\beta R_2, P_2, h_2 \\
\wedge\ & R_1, P_1, h_1 \underset{\beta^n/\sim\beta*}{\rightsquigarrow} R_3, P_3, h_3 \\
\wedge\ & R_2, P_2, h_2 \underset{\beta^n/\sim\beta*}{\rightsquigarrow} R_4, P_4, h_4 \\
\implies & R_3, P_3, h_3 \sim_\beta R_4, P_4, h_4
\end{aligned}
$$

Let us look at how Conjecture 3 differs from Lemma 7. In Lemma 7, I require that the I can separate an execution involving β and $\sim\beta$ steps such that all the β steps occur first, followed by allow of the $\sim\beta$ steps, but eventually I will end up in the same configuration. As I am moving steps around but require that the pending queue has the same resulting state, it must be that β cannot spawn otherwise the shape of the pending would be different. However, in Conjecture 3, I state that from two β similar configurations, if I execute β n times and $\sim\beta$ any number of times, the state of the behaviour β will be the same; what happens to the pending is immaterial. This observation is the key difference Lemma 7 is a global atomicity result whilst Conjecture 2 is a behaviour-local atomicity result.

Note also an important effect of how I have defined our evaluation relations and conjectures, namely that I cannot start the behaviour β, there is no evaluation rule that allows us to do that during the evaluation of the global configuration. Thus, the behaviour β must be running at the start of the evaluation; this is crucial, consider the behaviour β was allowed to start arbitrarily during the full execution, then in each path the behaviour may start after different operations on their required cowns have taken place, and proving their resulting state would rarely if ever be possible.

I also propose an independence result in Conjecture 3.

Conjecture 3 (Behavioural Independence)**.**

$$
\begin{aligned}
& R_1, P_1, h_1 \textit{ is well-formed} \\
\wedge\ & R_2, P_2, h_2 \textit{ is well-formed} \\
\wedge\ & R_1, P_1, h_1 \sim_\beta R_2, P_2, h_2 \\
\wedge\ & R_1, P_1, h_1 \underset{\beta^n/\sim\beta*}{\rightsquigarrow} R_3, P_3, h_3 \\
\implies & \exists R_4, P_4, h_3 . R_2, P_2, h_2 \underset{\beta^n/\sim\beta*}{\rightsquigarrow} R_4, P_4, h_4
\end{aligned}
$$

Proving such conjectures requires further or modified properties of the underlying language. For example, the underlying language must now provide a property that two similar states, both executing the same behaviour, will result in two similar states - in essence, axiomatically defining the behaviour similarity relation. This will remain as work left for the future.

5.6 Conclusion

In this chapter I demonstrated that behaviours in BoC can remain isolated from one another. This is achieved by requiring the isolation properties Properties 1 to 5. I have proven that, if the underlying language satisfies the properties, then BoC guarantees that behaviours are atomic between spawning behaviours. I have also provided a simple untyped underlying language, $\lambda_{\texttt{when}}$, which satisfies these properties. This provides an argument that BoC can, realistically, be adopted as an extension to a whole family of languages.

6

Axiomatic Model

In this chapter, I develop an axiomatic model of behaviour-oriented concurrency to better understand causality, or *happens before*, through behaviour ordering. I will revisit the operational semantics from Chapter 4, explore the earlier claim that they are too restrictive, and look at what causal order I would ideally want. I will demonstrate that it is more concise to explore and define the dynamic causal order by, first, abstracting a concurrent execution to its key events, and, second, constructing relations and constraint specifications over these events.

Ordering is intrinsic to how we, as programmers, build programs. Recall how the BoC program with ordered transactions and logging in Section 3.6. The desired outcome is only possible if the behaviours execute in the order expected.

Many actor-based languages give guarantees about the ordering of behaviours (cf. Section 2.1.4). This is no less the case in BoC and ordering is, in fact, a fundamental concept. I want that the order in which behaviours that use the same cown are spawned, affects the order in which those behaviours are executed and complete. I call such an order a *causal order*.

Consider Listing 6.1: an initial behaviour is spawned which writes `0` to the cowns `x` and `y`; then a second behaviour is spawned which spawns two behaviours to write `1` to `x` and `y`; and then spawns a third behaviour which spawns two behaviours to print `y` and `x`.

Listing 6.1: Concurrent reads and writes of cowns

```
1 when(x, y) { /* b0 */ x = 0; y = 0 };
2 when() { /* b1 */
3   when(x) { /* b2 */ x = 1 };
4   when(y) { /* b3 */ y = 1 }
5 };
6 when() { /* b4 */
7   when(y) { /* b5 */ print("y = " + y) };
8   when(x) { /* b6 */ print("x = " + x) }
9 }
```

Assume that the behaviour $b0$ will execute first in all executions, what are the possible printed values? The reader may assume that there are no constraints on the order of writes and prints, however the operational semantics from Chapter 4 do not allow all permutations of these operations. Table 6.1 demonstrates that the operational semantics will never print `x = 0` and `y = 1` in the same output. I argue that this result is not desired but not a correctness issue; it is simply the case that there should be more possible executions than the operational semantics permits. To add weight to this argument, all of the outcomes of Table 6.1 are permitted by the operational semantics, if I swap the spawn order of $b5$ and $b6$ in Listing 6.1. I argue that swapping the spawn order of $b5$ and $b6$ should be a meaning preserving program transformation. Assume `x` and `y` do not alias. Within the local context of $b4$, swapping the order of $b5$ and $b6$ appears to be a valid program transformation; there is no nested behaviours or other information to tell us that this swap should change the program in any way. However, once I look at $b4$ in the global context, and consider the behaviours that come before it, the transformation is no longer valid. This means I cannot think about behaviours compositionally.

✓	✗	✓	✓	✓	✓	✗	✓
`x = 0` `y = 0`	`x = 0` `y = 1`	`x = 1` `y = 0`	`x = 1` `y = 1`	`y = 0` `x = 0`	`y = 0` `x = 1`	`y = 1` `x = 0`	`y = 1` `x = 1`

Table 6.1: Behaviour orders permitted by operational semantics

I want to describe the ordering I desire from BoC and compare it with the ordering that the operational semantics provides. The operational semantics in Chapter 4 describes the emergence of permitted executions of BoC programs, thus the ordering of behaviours is intrinsic to a configuration and the intricacies of a particular operational semantics. Axiomatic models allow us to abstract the execution of a program to its key events – in this case the spawning, running and completion events of behaviours – and construct relations and constraint specifications over these events. An axiomatic model describes permitted executions from complete candidate executions, so the ordering of behaviours can be validated more concisely by looking at complete histories. The benefits of utilising both operational semantics and axiomatic models has been observed elsewhere in the literature[70].

In this chapter I will take inspiration from weak memory models to axiomatically define behaviour ordering; this will enable me to succinctly explore the design space and outcomes of different choices when constructing the ordering.

6.1 Limited execution orders from operational semantics

I will revisit the operational semantics and see that, whilst these semantics are correct, they guarantee an ordering that is slightly too strong. I will also show that, to weaken the ordering in the operational semantics, would require them to be more complicated. In comparison, the axiomatic model will succinctly capture the intended behaviour ordering.

The semantics presented in Chapter 4 provides *a* causal order known as as happens before, repeated below, and I explained how the order is achieved.

Definition 3 (Happens Before)**.** *A behaviour b will* happen before *another behaviour b' iff b and b' require overlapping sets of cowns, and b is spawned before b'.*

Consider again Listing 6.1. If I assume `x` and `y` to be non-aliasing, what are the happens before relationship between the behaviours $b2$, $b3$, $b5$ and $b6$?

When I execute this program, $b5$ will be spawned before $b6$ and $b2$ before $b3$, and either $b2$ will be spawned before $b6$ or $b6$ before $b2$. This has a repercussion, if $b6$ is spawned before $b2$, for instance, then

$b5$ will be spawned before $b3$. Definition 3 says that, in such a case, $b6$ happens before $b2$ and $b5$ happens before $b3$ due to the spawned before relations and overlapping cowns.

Let us look at this more generally. If I were to start behaviours one at a time, which start orders do the operational semantics from Figure 4.1 actually permit? The answer is presented in Table 6.2. Recall that the behaviours perform the following actions: $b2$ sets `x=1`; $b3$ sets `y=1`; $b5$ prints `y`; and, $b6$ prints `x`.

order				printed	allowed	order				printed	allowed
b2	b3	b5	b6	`y = 1` `x = 1`	✓	b5	b2	b3	b6	`y = 0` `x = 1`	✓
b2	b3	b6	b5	`x = 1` `y = 1`	✓	b5	b2	b6	b3	`y = 0` `x = 1`	✓
b2	b5	b3	b6	`y = 0` `x = 1`	✓	b5	b3	b2	b6	`y = 0` `x = 1`	✓
b2	b5	b6	b3	`y = 0` `x = 1`	✓	b5	b3	b6	b2	`y = 0` `x = 0`	✓
b2	b6	b3	b5	`x = 1` `y = 1`	✓	b5	b6	b2	b3	`y = 0` `x = 0`	✓
b2	b6	b5	b3	`x = 1` `y = 0`	✓	b5	b6	b3	b2	`y = 0` `x = 0`	✓
b3	b2	b5	b6	`y = 1` `x = 1`	✓	b6	b2	b5	b3	`x = 0` `y = 0`	✓
b3	b2	b6	b5	`x = 1` `y = 1`	✓	b6	b2	b3	b5	`x = 0` `y = 1`	✗
b3	b5	b2	b6	`y = 1` `x = 1`	✓	b6	b5	b2	b3	`x = 0` `y = 0`	✓
b3	b5	b6	b2	`y = 1` `x = 0`	✗	b6	b5	b3	b2	`x = 0` `y = 0`	✓
b3	b6	b2	b5	`x = 0` `y = 1`	✗	b6	b3	b2	b5	`x = 0` `y = 1`	✗
b3	b6	b5	b2	`x = 0` `y = 1`	✗	b6	b3	b5	b2	`x = 0` `y = 1`	✗

Table 6.2: Possible sequential executions

I want and expect BoC to permit all executions in Table 6.2. Yet, there are a number of executions the operational semantics will not admit as they violate the happens before relation. As I have shown, this means some outcomes are simply not possible. The operational semantics are too restrictive. More generally, I know that an execution is valid iff observation 1 holds.

Observation 1.

In Listing 6.1 $b2$ *happens before* $b6 \vee b5$ *happens before* $b3$

Let us put observation 1 in the context of the operations of Listing 6.1. The observation states that either, `x = 1` happens before print `x`, or print `y` happens before `y = 1`. This excludes executions where `x = 0` and `y = 1` are printed.

I know what the operational semantics allows, but should this be the canonical truth for any operational semantics for BoC? I argue no, there should not be any causal order between any of the behaviours and that all of the orders in Table 6.2 should, in theory, be allowed. There is no causal order between $b1$ and $b4$, and no execution dependency between $b2$ and $b3$, nor $b5$ and $b6$ (that is to say $b2$ doesn't spawn $b3$, for example,

so the two behaviours should be able to execute in either order). So it does not follow that there should be some executions which are precluded due to the order in which they are spawned. This doesn't mean the operational semantics are bad, they do not admit executions that should be disallowed, they are simply too strong.

Let us consider Listing 6.2, an example where the operational semantics provides a desirable observed ordering derived from an execution choice. The behaviours $b1$ and $b4$ are identical: they spawn a behaviour which accesses a sensor and reads some data into a local variable; the spawned behaviour, spawns another behaviour which accesses a processing unit (passing the read data in by copy); the deepest spawned behaviours then process the read data in some way.

Listing 6.2: An indirect causal order

```
when() { /* b1 */
  when(sensor) { /* b2 */
    data = sensor.read()
    when(processor) { /* b3 */
      processor.process(data)
    }
  }
};
when() { /* b4 */
  when(sensor) { /* b5 */
    data = sensor.read()
    when(processor) { /* b6 */
      processor.process(data)
    }
  }
}
```

The general property of permitted executions for this example is presented in observation 2 In this case, I argue that observation 2 makes sense. The causal order between $b3$ and $b6$ is contingent on what happened between $b2$ and $b5$, i.e. their spawning behaviours. Either $b2$ will execute completely before $b5$ starts, in which case $b3$ will be spawned before $b6$, or, $b5$ will execute completely before $b2$ starts, in which case $b6$ will be spawned before $b3$. As the programmer, I may not know which of the two behaviours will execute first, but I can know something about what will happen once that choice is made. This allows me to rely on the behaviour to process the sensor data in FIFO order, without needing to create some data structure.

Observation 2.

$$\begin{aligned} \textit{In Listing 6.2}\ (b2\ \textit{happens before}\ b5 &\iff b3\ \textit{happens before}\ b6) \\ \wedge\ (b5\ \textit{happens before}\ b2 &\iff b6\ \textit{happens before}\ b3) \end{aligned}$$

I now have a few litmus tests for executions in programs that I would like to be permitted by an operational semantics. Consider the adjustments that would have to be made to the operational semantics to admit all orders for Listing 6.1. I briefly introduce a revised semantics in Figure 6.1 as a means to labour the point of added complexity, thus do not study this model to intricately.

Fundamentally, the issue with the original semantics is that once a behaviour enters the pending queue, it cannot be started after a behaviour, with overlapping cowns, that is later in the queue. This has the intended ordering effect for behaviours spawned by the same behaviour, but for Listing 6.1 the order is incidental and not intentional. To change this I would need some new data structure or supplementary information that retains more spawning information for posterity. I use a tree of pending queues as this new data structure.

I revise the global configuration from Definition 9 to change pending as in Definition 27.

Definition 27 (BoC with a tree of queues)**.**

$$p \in PendingEntry = (BehavID \times Tag^* \times Context)$$
$$P \in PendingBehaviours = BehavID \to (BehavId \times PendingEntry^*)$$

The behaviour identifiers are shared as keys for the pending and the running maps. The pending maps behaviour identifiers to a pair of an identifier, the parent queue in the tree, and queue of pending entries. Each entry contains the behaviour identifier for the entry (in the pending map and eventually in the running map), the cowns the behaviour requires and the behaviour context. The semantics are presented in Figure 6.1

$$\textsc{step}\ \frac{E, h \hookrightarrow E', h'}{R[\beta \mapsto (\overline{\kappa}, E)], P, h \rightsquigarrow R[\beta \mapsto (\overline{\kappa}, E')], P, h'}$$

$$\textsc{spawn}\ \frac{P(\beta_1) = (\beta_2, p) \qquad E \hookrightarrow_{\mathsf{when}\,(\overline{\kappa'})\,\{E''\}} E' \qquad \beta_3 \notin dom(p_1)}{R[\beta_1 \mapsto (\overline{\kappa}, E)], P, h \rightsquigarrow R[\beta \mapsto (\overline{\kappa}, E')], P[\beta_1 \mapsto (\beta_2, p : (, \overline{\kappa'}, E'')), \beta_3 \mapsto (\beta_1, [])], h}$$

$$\textsc{emit}\ \frac{\beta_3 \neq \epsilon \qquad P(\beta_3) = (\beta_4, p_3) \qquad (\bigcirc_{(\_, \overline{\kappa'}, \_)\, \in\, p_1} \overline{\kappa'}) \# \overline{\kappa}}{R, P[\beta_1 \mapsto (\beta_3, p_1 : (\beta_2, \overline{\kappa}, E) : p_2)], h \rightsquigarrow R, P[\beta_1 \mapsto (\beta_3, p_1 : p_2), \beta_3 \mapsto (\beta_4, p_3 : (\beta_2, \overline{\kappa}, E))], h}$$

$$\textsc{start}\ \frac{(\bigcirc_{(\overline{\kappa'}, \_) \in rng(R)} \overline{\kappa'}) \# \overline{\kappa} \qquad (\bigcirc_{(\_, \overline{\kappa'}, \_) \in rng(p_1)} \overline{\kappa'}) \# \overline{\kappa}}{R, P[\beta_1 \mapsto (p_1 : (\beta_2, \overline{\kappa}, E) : p_2, \epsilon)], h \rightsquigarrow R[\beta_2 \mapsto (\overline{\kappa}, E)], P[\beta_1 \mapsto (\epsilon, p_1 : p_2)], h}$$

$$\textsc{end}\ \frac{finished(E)}{R[\beta \mapsto (\overline{\kappa}, E)], P, h \rightsquigarrow R \backslash \beta, P, h}$$

Figure 6.1: Semantics for BoC with trees of pending queues

The main revisions are in SPAWN, START and the new rule EMIT. SPAWN: when a behaviour is spawned a new identifier for the behaviour is created, the new behaviour enters into the local queue of the behaviour that spawned it, and a new leaf queue is added for fresh behaviour identifier. EMIT: this allows a pending behaviour to move up the tree, abiding by the same non-overlapping rules as START did earlier, a candidate behaviour will be appended to the parent level's queue. START: is in essence, emit at the root of the tree, the pending behaviour can start executing as long as their are no earlier behaviours in the queue with overlapping cowns and no require cown is running.

Clearly, this is very complex in both the rules and structure of information. Now, this is not to say this is the *only* way to permit the executions of our litmus tests, but it is one way that describes execution with desired causal order. Indeed there is a clearer way to describe executions which satisfy our litmus tests. What I will now show is that I can much more succinctly and clearly describe what I want using an axiomatic model.

6.2 Ordering axiomatically

In this section, inspired by the weak memory models literature, I construct an axiomatic model of BoC programs to describe *valid* and *invalid* candidate executions. I construct an alternative happens before relation for this axiomatic model that permits the desired executions for our motivating examples Listings 6.1 and 6.2.

Definition 28 (A model of BoC programs)**.** *A tuple of* $(\mathbb{E}, \mathtt{po}, \mathtt{co}, \#)$ *forms an axiomatic model of BoC executions.*

(1) $S_i \in Spawn \subseteq \{S_i \mid i \in \mathbb{N}\}$,

(2) $R_i \in Start \subseteq \{R_i \mid i \in \mathbb{N}\}$

(3) $C_i \in Complete \subseteq \{C_i \mid i \in \mathbb{N}\}$

(4) $\mathbb{E} = Spawn \cup Start \cup Complete$

(5) $\mathtt{po} \subseteq (Start \times Spawn) \cup (Spawn \times Spawn) \cup (Spawn \times Complete)$

(6) $\mathtt{co} \subseteq (Complete \times Start \times Tag)$

(7) $\# \subseteq Tag^* \times Tag^*$

The model consists of three types of events, namely, *spawn*, *start* and *complete*. These correspond to spawning a behaviour, via a **when**, a behaviour starting, and a behaviour terminating.

Compare these events with those in the weak memory models literature, i.e., read and write events for a memory location or register. These read/write events have global, and not local, interactions; the result of reading shared memory requires knowing the order of writes to the location from other threads.

The shared memory locations in BoC programs are protected by cowns, and a cown is accessible by at most one running behaviour, and for the entire duration of that behaviour's lifetime. Thus any write to a cown's memory will only become visible when a behaviour ends, and will only affect the reads of the next behaviour that accesses the cown's memory. So, the reads and writes events of cown's memory are subsumed by the *start* and *complete* events. The *spawn* events immediately affect any global program state, they create a new behaviour that can start running, and so I distinguish these events. I will show how I extend the axiomatic model for BoC with read and write events in Section 6.6.

The model also consists of two kinds of relation: the intra-behaviour *program order* relation, $\mathtt{po}$, which orders the events within a behaviour; and, the inter-behaviour *coherence order* relation which orders the start and end of behaviours which use the same cowns. These orders reflect their counterparts in the weak memory models literature, where *program order* is the order within a thread and *coherence order* is order between threads.

Finally, the model requires a provided disjoint relation for cowns. I will later use this to derive a *happens before* order, much as I did for the operational semantics.

On top of the provided relations, I define a few derived relations:

Definition 29 (Derived Relations)**.** *For a given model,* $(\mathbb{E}, \mathtt{po}, \mathtt{co})$*, the following are derived:*

(1) $\mathtt{r} \triangleq \{(S_i, R_i) \mid S_i \in \mathbb{E}, R_i \in \mathbb{E}\}$

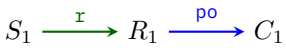
(2) $\mathtt{co}_\kappa \triangleq \{(C_i, R_j) \mid (C_i, R_i, \kappa) \in \mathtt{co}\}$

The *run order* relation, $\mathtt{r}$, orders when behaviours spawn and start, and the $\mathtt{co}_\kappa$ relation is a restriction of the coherence order to refer to the coherence order induced by a particular cown.

Examples of executions Let us investigate executions that this model describes and how they correspond with programs. In the following, a part of a BoC program is on the left and a graph representation of a corresponding execution is on the right. Events are nodes in the graphs, program order is in solid blue,

run order in solid green, coherence order in solid orange, and happens before in dashed purple (this will appear later).

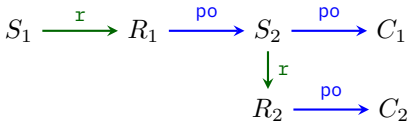
```
1  when(c1) { /* b1 */ }
```



The three events, S_1, R_1, C_1, represent the events of the behaviour $b1$; namely, it is spawned, started and completed. The run order, derived, relates the spawn event and the start event, the program order relates the start event and complete event. So, the behaviour $b1$ is spawned, then starts, and then completes.

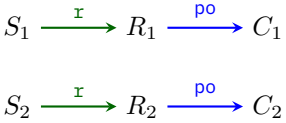
Extending the example above, I can include that $b1$ spawns another behaviour, $b2$, and I can get the program and execution below.

```
1  when(c1) { /* b1 */
2    when(c2) { /* b2 */ }
3  }
```



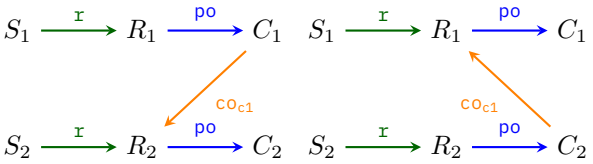
The program order states that the $b1$ starts, then spawns $b2$, and then completes. The run order states $b2$ starts after it spawns and the program order then says it completes. The diagram shows that $b2$ can start at the same time as $b1$ is executing. This can see this more directly in the example that follows where $b1$ and $b2$ are spawned in parallel.

```
1  when() { when(c1) { /* b1 */ } };
2  when() { when(c2) { /* b2 */ } }
```



There is no ordering between the two behaviours whatsoever, so their events can happen in parallel or interleaved. But, what if $b1$ and $b2$ used the same cown? Let us adapt the above program so that $b1$ and $b2$ are spawned in parallel, as in the following example. In this example I have two executions that differ in a coherence order relation.

```
1  when() { when(c1) { /* b1 */ } };
2  when() { when(c1) { /* b2 */ } }
```

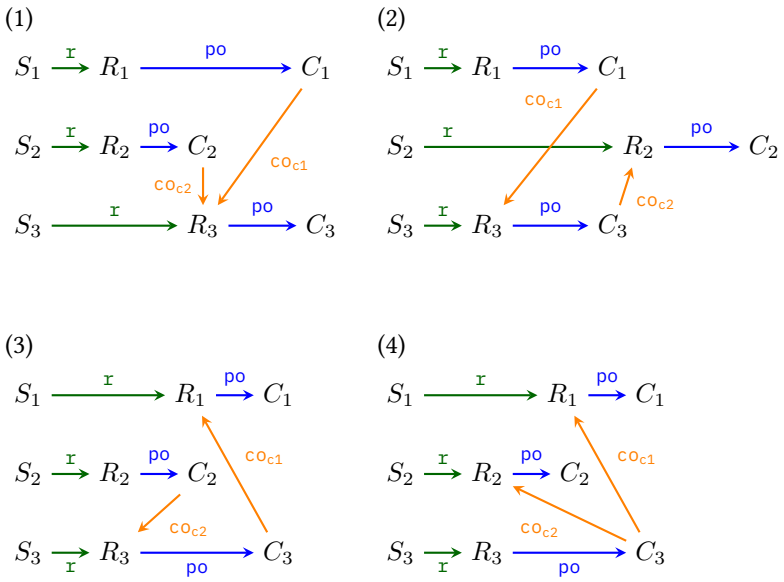


Now, I have a coherence order edge to represent that $b1$ and $b2$ use the same cown, co_{c1}, and one must execute before the other. However, which one happens first is immaterial, thus both $b1$ completes before $b2$ starts or $b2$ completes before $b1$ starts are valid orders.

Remember that one fundamental of BoC is that behaviours can access multiple cowns, so let us modify the previous example such that I have three behaviours spawned in parallel, two requiring a single cown (that do not overlap) and a third that requires the cowns of the previous two.

```
1  when() { when(c1) { /* b1 */ } };
2  when() { when(c2) { /* b2 */ } };
3  when() { when(c1, c2) { /* b3 */ } }
```

Presented below are the four possible executions: in the first, $b3$ executes last, so there two incoming coherence order edges to R_3; in the second, the behaviours execute in $b1, b3, b2$ order, so there is a chain of coherence order edges; in the third, the behaviours execute in $b2, b3, b1$ order, another chain; in the fourth, $b3$ executes first, so there are two outgoing coherence order edges from C_3.



6.3 Valid executions

I have shown a number of executions that make sense or are, more precisely, *valid*. But beyond those examples there are many executions that do not make sense or are *invalid*. I will now define exactly what makes an execution valid.

The weak memory models literature uses *constraint specifications* to decide which candidate executions are valid. Multiple specifications exist for weak memory models, such as Lamport's Sequential Consistency. Each specification can decide differently on whether an execution is valid or not.

I will now create a single constraint specification for our axiomatic model of BoC. Note that item 9 refers to a happens before relation $\mathtt{hb}$ which will be defined in the following section. In keeping with prior terminology in this thesis, I will refer to this specification as *well-formed*, and a well-formed execution is a valid execution.

Definition 30 (Well-formed model)**.** *A model* $(\mathbb{E}, \mathtt{po}, \mathtt{co})$ *is constrained if:*

(1) $\forall e_1, e_2.(e_1 \,\mathtt{po}\, e_2 \;\vee\; e_1 \,\mathtt{co}\, e_2 \implies \{e_1, e_2\} \subseteq \mathbb{E})$

(2) $\forall e_1, e_2 \in \mathbb{E}.(e_1(\mathtt{po} \cup \mathtt{co} \cup \mathtt{r})^* e_2 \implies e_1 \neq e_2)$

(3) $\forall R_i \in \mathbb{E}, C_j \in \mathbb{E}.(R_i \,\mathtt{po}^*\, C_j \implies i = j)$

(4) $\forall e_1, e_2, e_3 \in \mathbb{E}.(e_1 \,\mathtt{po}\, e_3 \;\wedge\; e_2 \,\mathtt{po}\, e_3 \implies e_1 = e_2)$

(5) $\forall e_1, e_2, e_3 \in \mathbb{E}.(e_1 \,\mathtt{po}\, e_2 \;\wedge\; e_1 \,\mathtt{po}\, e_3 \implies e_2 = e_3)$

(6) $\forall e_1, e_2, e_3 \in \mathbb{E}, \kappa.(e_1 \,\mathtt{co}_\kappa\, e_3 \wedge e_1 \,\mathtt{co}_\kappa\, e_3 \implies e_1 = e_2)$

(7) $\forall e_1, e_2, e_3 \in \mathbb{E}, \kappa.(e_1 \,\mathtt{co}_\kappa\, e_2 \wedge e_1 \,\mathtt{co}_\kappa\, e_3 \implies e_2 = e_3)$

(8) $\forall e_1, e_2, e_3, e_4 \in \mathbb{E}, \kappa.(e_1 \,\mathtt{co}_\kappa\, e_2 \wedge e_3 \,\mathtt{co}_\kappa\, e_4 \implies e_2(\mathtt{po} \cup \mathtt{co}_\kappa)^* e_3 \vee e_4(\mathtt{po} \cup \mathtt{co}_\kappa)^* e_1)$

(9) $\forall e_1, e_2 \in \mathbb{E}.(e_1(\mathtt{po} \cup \mathtt{co} \cup \mathtt{hb})^* e_2 \implies e_1 \neq e_2)$

Item 1 ensures the relations only order known events. Item 2 ensures that the transitive closure of the union of orders must be acyclic; this precludes an event being ordered before itself. Item 3 ensures the behaviours start and complete event are uniquely related; the $\mathtt{po}$ relates the behaviour local events, so I only want to relate the start and end of the same behaviour with $\mathtt{po}$. Item 4 ensure an events has only a

unique po predecessor. Item 5 ensures an event has only a unique po successor. These two ensure I have straight line execution without branching or non-determinism (note that conditional spawns will have been resolved when I look at an execution). Item 6 and item 7 say similarly about $\texttt{co}_\kappa$. Item 8 says that $\texttt{co}_\kappa$ creates a single causal order path through the events that use $\overline{\kappa}$, to understand this better lets look at Section 6.3; in this diagram the solid $\texttt{co}_{c1}$ edges exist in the execution, and a one of the dashed $\texttt{co}_{c1}$ edges must also exist to create a single continuous causal order path for $c1$. Finally, I have item 9, which says that the transitive closure of the ordering edges *including* the hb relation is acyclic; this is a derived relation that I will look at in the next section.

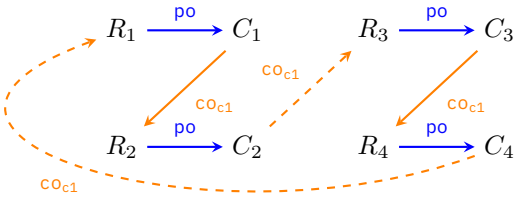


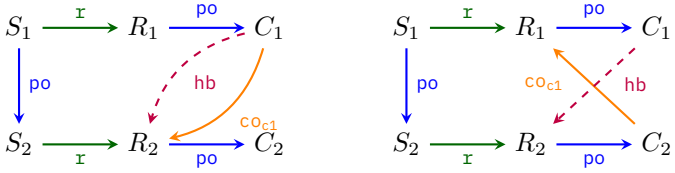
Figure 6.2: $\texttt{co}_{c1}$ must form a single acyclic path

6.4 Constructing happens before

I will introduce happens before into the axiomatic model and use it to describe which candidate executions are valid. I will first reproduce the operational semantics ordering in the axiomatic model, explore why this is too restrictive, and then construct a better, more permissive ordering.

I want a happens before relation that describes which behaviour *should* happen before which other behaviour. I want to use this relation to determine whether an execution is valid or invalid. In the following example I provide an example with two behaviours $b1$ and $b2$ that are spawned, not in parallel, but in program order. I want that, in every *valid* execution of the program, $b1$ completes before $b2$ starts. I also provide two potential executions of this program, the left execution I wish to be valid and the right I wish to be invalid. Desired happens before edges have been added to these executions for illustration purposes.

```
1  when(c1) { /* b1 */ };
2  when(c1) { /* b2 */ }
```



In the left execution, $b1$ completes before $b2$ starts. Here, there is a happens before relation, in dashed orange, which orders the completion of $b1$ before $b2$. In the right execution, the coherence order states that $b2$ completes before $b1$ but the happens before order hasn't changed. In fact, the right execution is invalid.

In this section I will build a happens before relation that disallows executions like the one I have just discussed, whilst allowing the desired executions that I established at the beginning of the chapter.

6.4.1 Happens before for the operational semantics

Let us first look at the ordering provided by the operational semantics. I define a happens before order for the axiomatic model such that only candidate executions that the operational semantics could elicit are

valid.

Recall that the operational semantics in Figure 4.1 use a queue to store pending behaviours, and once a pending behaviour is in the queue, it can only leave the queue when there are no earlier behaviours in the queue that use any of the same cowns. So, conversely, a behaviour can only have started before another behaviour, with overlapping cowns, if the behaviour that started first was also spawned first. If I want only those executions that the operational semantics can elicit to be those which are valid in the axiomatic model, then I need to reflect this spawning relation in the definition of happens before. Thus I define happens before for the axiomatic model as in Definition 31.

Definition 31.

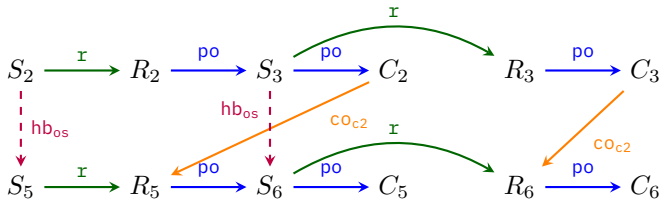
$$\mathtt{hb_{os}} \triangleq \{(S_i, S_j) \mid C_i \,\mathtt{co}\, R_j\}$$

I will now use this definition of happens before in our axiomatic model, and revisit out two motivating examples (Listings 6.1 and 6.2). I will demonstrate that those executions that the operational semantics from Chapter 4 can elicit are considered valid by this axiomatic model, and those that the operational semantics cannot elicit are considered invalid by this axiomatic model. In other words, the operational semantics and axiomatic model agree. However, as I have mentioned, this definition of happens before is too restrictive and motivates the search for a better definition; this will be the subject of the following subsection.

First let us revisit Listing 6.2, reproduced below, and recall observation 2 on its possible executions.

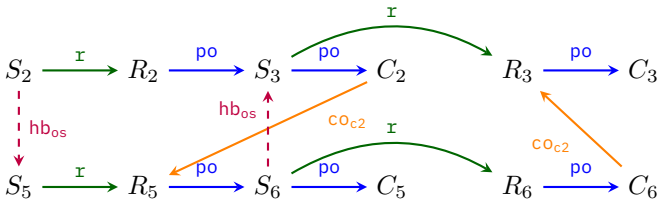
```
1  when() { /* b1 */
2    when(c1) { /* b2 */
3      when(c2) { /* b3 */ }
4    }
5  };
6  when() { /* b4 */
7    when(c1) { /* b5 */
8      when(c2) { /* b6 */ }
9    }
10 }
```

I can draw out the diagram for an execution of this program and add in the derived happens before edges.



This is an execution that the operational semantics can elicit. This is also a valid execution in the axiomatic model if Definition 30 is satisfied. I will focus on whether Definition 30.item 9 is satisfied. From inspection of the execution, I can see there is no cycle, and so the execution is valid.

But, what about an execution that the operational semantics cannot elicit, is such an execution invalid in the axiomatic model? What if the coherence order said that $b6$ completed before $b3$ started, as in the diagram that follows?

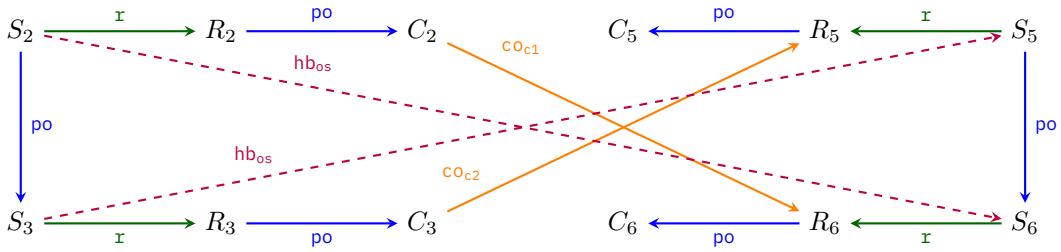


Now, I have introduced a cycle and thus the execution is invalid, thus corroborating with the operational semantics.

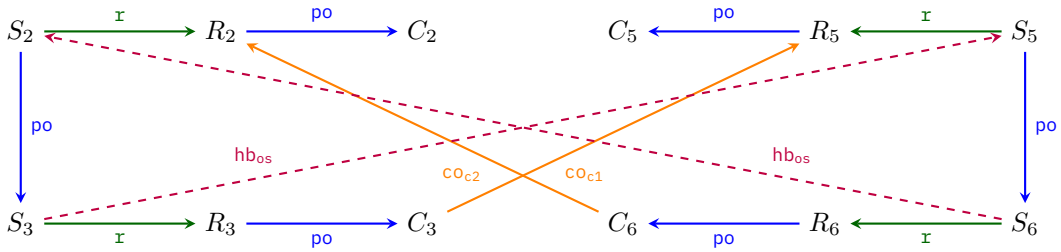
Now, let us revisit Listing 6.1, reproduced below, where the operational semantics were too restrictive.

```
1  when() { /* b1 */
2    when(c1) { /* b2 */ };
3    when(c2) { /* b3 */ }
4  };
5  when() { /* b4 */
6    when(c2) { /* b5 */ };
7    when(c1) { /* b6 */ }
8  }
```

An execution order that is permitted by the operational semantics is $b2; b3; b5; b6$, whilst $b6; b2; b3; b5$ is not. Let us construct the execution for the first execution order.



There are no cycles, and thus the execution is valid. Now, let us inspect the execution for the second execution order.



Now, I have a cycle. Recall, to match the operational semantics, the coherence order reflects the order in which behaviours were spawned into the queue. Furthermore, behaviours are spawned in the order in which they appear in their behaviour. So, one way to read this execution is that $b2$ is spawned before $b3$, and $b5$ is spawned before $b6$ (these are immutable facts), and if $b3$ completed before $b5$ then $b3$ was spawned before $b5$ (which would give us transitively that $b2$ is spawned before $b6$), and if $b6$ completed before $b2$

started then $b6$ must have been spawned before $b2$ (which means, somehow, $b2$ was, transitively, spawned before itself); this is a causal relation that simply cannot exist.

The crux of why this execution is not allowed, is that the happens before relation is on the spawning event. I can create an axiomatic model where this execution is valid if I instead relate the completion and start of behaviours. This is the focus of the following section.

6.4.2 Constructing a better happens before relation

Now, I redefine happens before for the axiomatic model, so that the executions of interest from our litmus tests are valid in our model. I will briefly introduce and discuss the complete happens before relation, and then delve deeper into how I arrive at this answer in the remainder of this subsection. The complete relation is presented in Definition 38.

Definition 38 (Happens Before)**.**

$$\texttt{hb} \triangleq \{(C_i, R_j) \mid C_i \, \texttt{po}^{-1*} \, \texttt{r}^{-1} (\texttt{co} \mid \texttt{po} \mid \texttt{r})^* \, \texttt{r} \, R_j \wedge \text{conflict}(i,j) \wedge i \neq j\}$$

There are three main components to this relation. First, I don't want the causal order to be on the spawning relation, as we've seen that can be too restrictive. What I want is to order which behaviour should complete before which other behaviour starts, in other words to order complete and start events.

Second, I need to define some path from a complete event to a start event.

- $C_i \, \texttt{po}^{-1*} \, \texttt{r}^{-1}$: I start from the completion of a behaviour, say b, and look back to find the event where b was spawned.
- $(\texttt{co} \mid \texttt{po} \mid \texttt{r})^*$: Now, I look forwards from the spawn, to consider any possible event that I know happened after b was spawned. Thus considering (a) any descendants of the parent of b that were spawned after b, (b) any behaviour that started after the parent of b completed.
- $\texttt{r}$: Finally, I need to ensure that behaviours can only happen before behaviours at the same depth of the family tree or deeper (I don't want children happening before aunts/uncles). This means I need to reflect the initial $\texttt{r}^{-1}$ relation to balance this out.

Third, from all the possible candidate pairs of events, I need to know which behaviours actually need ordering. For this, I have a conflict relation. Note that I also have an $i \neq j$ constraint, which only enforces "non-reflexive" happens before relations, this could be part of the conflict.

Building to Definition 38 I will now look more deeply at how I arrive at this definition through further examples.

I define a conflicts relation to capture which behaviours conflict with which other behaviours. I build conflicts on an auxiliary cowns function which finds all of the cowns a behaviour uses.

Definition 32 (Cowns)**.**

$$\text{cowns}(i) \triangleq \{\kappa \mid (\_ \, \texttt{co}_\kappa \, R_i) \ \vee \ (C_i \, \texttt{co}_\kappa \, \_)\}$$

Conflict is then constructed on sets of cowns which are not disjoint w.r.t a provided $\#$ relation.

Definition 33 (Conflicts)**.**

$$\text{conflict}(i,j) \triangleq \neg(\text{cowns}(i) \ \# \ \text{cowns}(j))$$

For this section, I define the disjoint relation to be empty cown intersection as follows.

Definition 34 (Disjoint on Intersection)**.**

$$\overline{\kappa} \# \overline{\kappa'} \triangleq \overline{\kappa} \cap \overline{\kappa'} = \varnothing$$

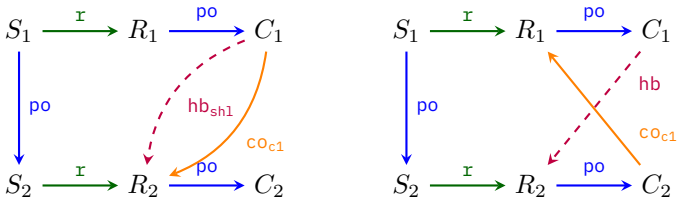
Now I construct the first step towards happens before in Definition 35 and see why it needs improvement through examples. The first conjunct creates a path from a completion event to a start event, *i.e.,* follows the lineage, the second tells us whether order is needed or not, and the last conjunct ensures that the completion of a behaviour, and the start of the same behaviour, are unrelated.

Definition 35.

$$\mathtt{hb}_{\mathtt{shl}} \triangleq \{(C_i, R_j) \mid C_i \, \mathtt{po}^{-1*} \, \mathtt{r}^{-1} \, \mathtt{po}^{*} \, \mathtt{r} \, R_j \wedge \mathrm{conflict}(i, j) \wedge i \neq j\}$$

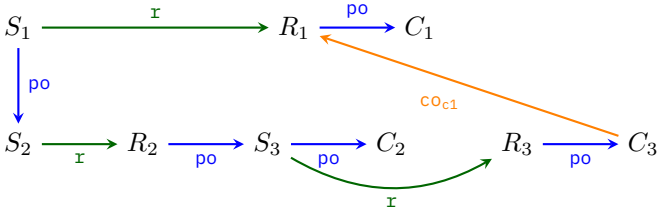
This works for ordering behaviours spawned by the same parent, but this is too shallow, I will show this now. Take the following program and two executions; the first execution says that $b1$ completes before $b2$ starts, and this agrees with our happens before order; whereas the second execution says that $b2$ completes before $b1$ starts, the happens before edge is the same in both executions, but the second execution has a causal cycle and is thus invalid.

```
1  when(c1) { /* b1 */ };
2  when(c1) { /* b2 */ }
```



Definition 35 is too shallow, the relation can only relate behaviours of the same depth, and I want happens before to be a deeper relation. In the following example, I want that C_1 happens before R_3. However, with Definition 35, I do not get this and so the model judges the accompanying execution to be valid (I want it to be invalid).

```
1  when(c1) { /* b1 */} ;
2  when(c2) { /* b2 */
3    when(c1) { /* b3 */ };
4  }
```

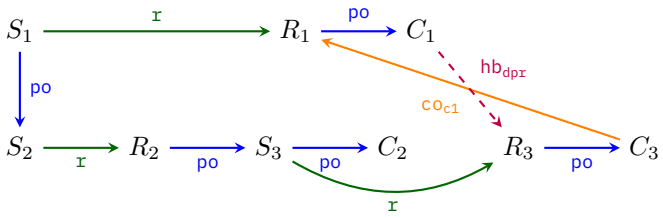


To get C_1 happens before R_3, I need to be able to follow the spawning lineage further through the po and r relations. I update the happens before relation in Definition 36 (the changes are highlighted in red). Importantly, I only want to follow the lineage deeply in a subsequent behaviour, so from a completion event I only follow the behaviour up to its immediate ancestor, look at any sibling behaviours that succeed the behaviour, and follow that behaviour's descendants.

Definition 36.

$$\mathtt{hb}_{\mathtt{dpr}} \triangleq \{(C_i, R_j) \mid C_i \, \mathtt{po}^{-1*} \, \mathtt{r}^{-1} \, (\mathtt{po} \mid \mathtt{r})^{*} R_j \wedge \mathrm{conflict}(bi, bj) \wedge i \neq j\}$$

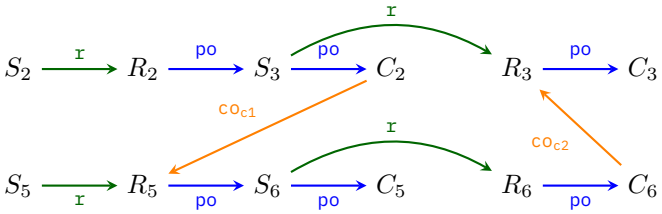
Now, let us revisit the previous example with this new definition of happens before from Definition 36. Indeed, I can now derive $C_1 \, \mathtt{hb}_{\mathtt{dpr}} \, R_3$ and I can see that there is a causal cycle in the graph and, thus, the execution is invalid.



We're still not quite where I want to be, because we're creating causal orders based only on the program order and run order, which is apparent in all executions of a program. So what I have so far is the happens before relations that will happen in every execution of a program. Yet, I know, from the introduction of this chapter, that scheduling decisions induce further causal orders.

Consider that in one of our motivating examples Listing 6.2 (repeated below), in an execution where C_1 is co-before R_3, I also want that C_2 happens before R_4. Our current definition of happens before will not order these two events, which means a valid execution can contain $C_1\ \mathtt{co}_{c1}\ R_3$ and $C_4\ \mathtt{co}_{c2}\ R_2$, as I show below.

```
1  when() { /* b1 */
2    when(c1) { /* b2 */
3      when(c2) { /* b3 */ }
4    }
5  };
6  when() { /* b4 */
7    when(c1) { /* b5 */
8      when(c2) { /* b6 */ }
9    }
10 }
```

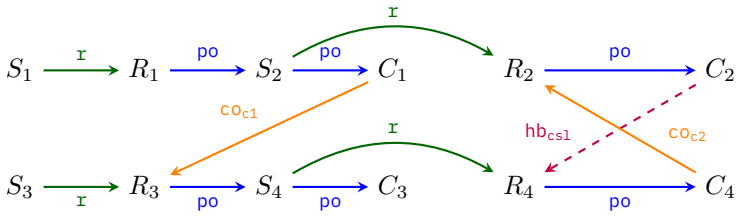


So I enrich our definition of happens before to consider these existing causal edges in Definition 37.

Definition 37.

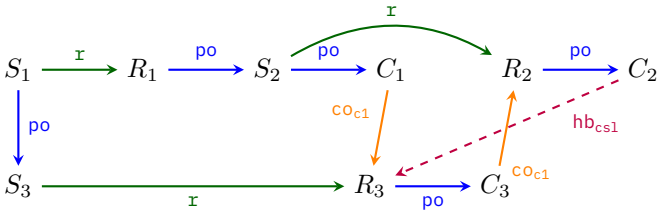
$$\mathtt{hb}_{\mathtt{csl}} \triangleq \{(C_i, R_j) \mid C_i\ \mathtt{po}^{-1*}\ \mathtt{r}^{-1}(\mathtt{co} \mid \mathtt{po} \mid \mathtt{r})^* R_j \wedge \mathrm{conflict}(bi, bj) \wedge i \neq j\}$$

I now revisit the previous example and see that, when $C_1\ \mathtt{co}_{c1}\ R_3$ I indeed induce $C_2\ \mathtt{hb}_{csl}\ R_4$, this introduces a causal cycle and judges the execution as invalid.



I just need one small fix-up to this definition. In the example that follows, I have $C_2\ \mathtt{hb}_{\mathtt{csl}}\ R_3$ which introduces and undesired cycle, and judges the execution as invalid (when I want it to be valid). Intuitively, this would be that nested behaviours must run before outer behaviours, whilst this *could* be the case I do not want to enforce it.

```
1  when(c1) { /* b1 */
2    when(c1) { /* b2 */ };
3  };
4  when(c1) { /* b3 */ }
```



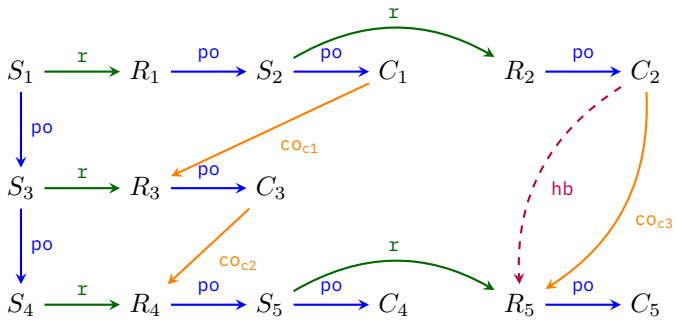
To fix this issue, I require that any happens before relation must end in an $\mathtt{r}$ relation. This prevents a behaviour from happening before the sibling of a parent. Fixing this gives us our final and complete definition for happens before in Definition 38.

Definition 38 (Happens Before)**.**

$$\mathtt{hb} \triangleq \{(C_i, R_j) \mid C_i \, \mathtt{po}^{-1*} \, \mathtt{r}^{-1} (\mathtt{co} \mid \mathtt{po} \mid \mathtt{r})^* \, \mathtt{r} \, R_j \wedge \mathrm{conflict}(i,j) \wedge i \neq j\}$$

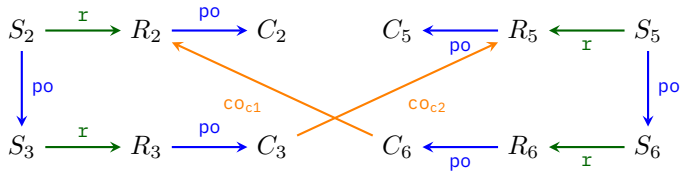
This definition lets us build the transitive ordering relation through ancestors that do not overlap on their required cowns. Consider the following program, I require that $b2$ happens before $b5$, and indeed that requirement is satisfied.

```
1 when(c1) { /* b1 */
2   when(c3) { /* b2 */ }
3 };
4 when(c1, c2) { /* b3 */ };
5 when(c2) { /* b4 */
6   when(c3) { /* b5 */ }
7 };
```



Returning to the original issue. Is the execution $b6; b2; b3; b5$ admitted for the following program? Let us draw out that execution and see.

```
1 when() { /* b1 */ when(c1) { /* b2 */ }; when(c2) { /* b3 */ } };
2 when() { /* b4 */ when(c2) { /* b5 */ }; when(c1) { /* b6 */ } }
```



Indeed that execution is admitted, in fact, there is a stark absence of happens before relations in this execution which is exactly what I was aspiring to. I can start the behaviours in any order and there won't be any cycles.

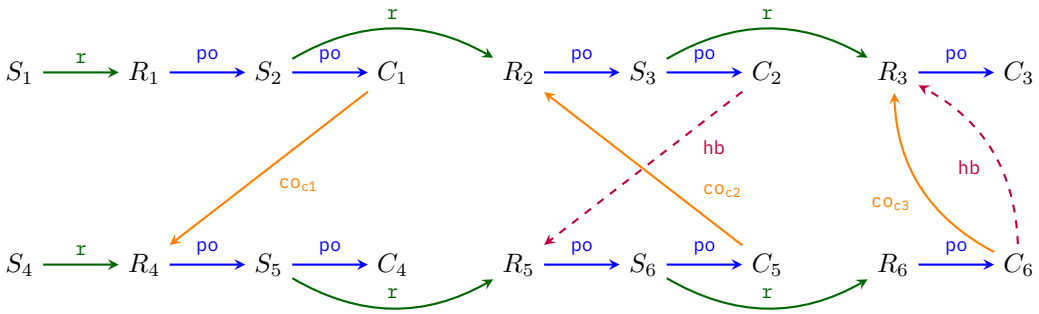
Let us inspect the space of executions again and compare what is permitted using the happens before relation as per the operational semantics (Definition 31), compared with the revised definition (Definition 38). The comparison can be found in Table 6.3.

				Def 31	Def 38					Def 31	Def 38
b2	b3	b5	b6	✓	✓	b5	b2	b3	b6	✓	✓
b2	b3	b6	b5	✓	✓	b5	b2	b6	b3	✓	✓
b2	b5	b3	b6	✓	✓	b5	b3	b2	b6	✓	✓
b2	b5	b6	b3	✓	✓	b5	b3	b6	b2	✓	✓
b2	b6	b3	b5	✓	✓	b5	b6	b2	b3	✓	✓
b2	b6	b5	b3	✓	✓	b5	b6	b3	b2	✓	✓
b3	b2	b5	b6	✓	✓	b6	b2	b5	b3	✓	✓
b3	b2	b6	b5	✓	✓	b6	b2	b3	b5	✗	✓
b3	b5	b2	b6	✓	✓	b6	b5	b2	b3	✓	✓
b3	b5	b6	b2	✗	✓	b6	b5	b3	b2	✓	✓
b3	b6	b2	b5	✗	✓	b6	b3	b2	b5	✗	✓
b3	b6	b5	b2	✗	✓	b6	b3	b5	b2	✗	✓

Table 6.3: Possible sequential executions

Executions are *only* valid or invalid Consider the program and execution that follows. In the execution, the coherence order says that $b1$ completes before $b4$ whilst $b5$ completes before $b2$. This disagrees with our designed causal order and, indeed, creates a cycle with the derived happens before relation. Yet, the execution also says that $b6$ will complete before $b3$; the happens before relation will agree as the relation only looks up into the immediate ancestor. This is not a limitation of the happens before construction - once the order of $b2$ and $b5$ agrees with their ancestors, the happens before order between $b3$ and $b6$ will be different - however, it is a warning that this axiomatic model only determines whether an execution is valid or invalid and does not allow one to reason about behaviour ordering once an execution is found to be invalid.

```
1 when(){ when(c1) { /* b1 */ when(c2) { /* b2 */ when(c3) { /* b3 */ } } } };
2 when(){ when(c1) { /* b4 */ when(c2) { /* b5 */ when(c3) { /* b6 */ } } } }
```

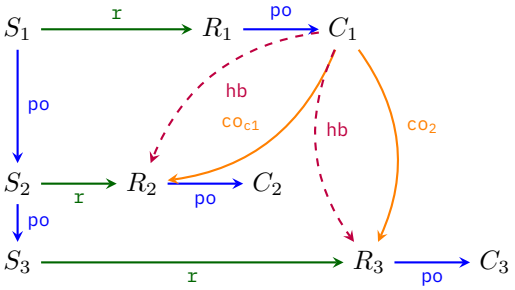


6.4.3 Happens before with multiple cowns

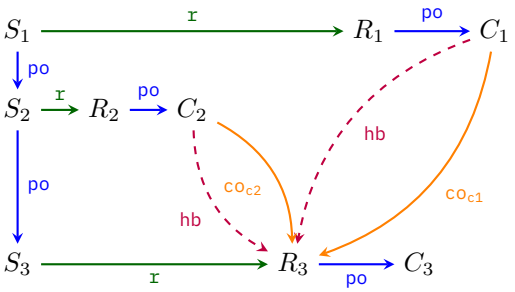
One of the key strengths of BoC is that behaviours can have access to multiple cowns. This means that for a behaviour that use n cowns, there will be up to n ingoing or outgoing `hb` and `co` relations.

In the following examples, I show how the happens before relation can be derived when multiple cowns are in use.

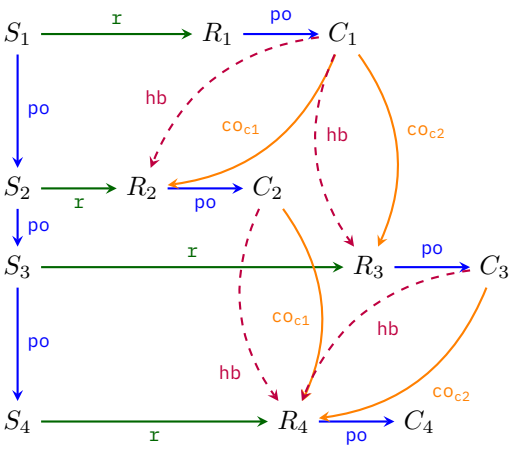
```
when(c1, c2) { /* b1 */ };
when(c1) { /* b2 */ };
when(c2) { /* b3 */ }
```




```
when(c1) { /* b1 */ };
when(c2) { /* b2 */ };
when(c1, c2) { /* b3 */ };
```




```
when(c1, c2) { /* b1 */ };
when(c1) { /* b2 */ };
when(c2) { /* b3 */ };
when(c1, c2) { /* b4 */ };
```



Other than inducing more orders in the executions, not much changes, the same core principles judging whether executions are valid or invalid still apply.

6.5 Design space for ordering behaviours

In our progression towards defining happens before for the axiomatic model, we've seen that there are different ways to define happens before and these elicit different valid executions. Recall the progression from Definition 35 to Definition 38. In this section I look at and discuss other potential definitions for happens before.

Each change I make has trade-offs between parallelism and determinism. More permissive relations allow us more parallelism but less determinism, whilst more restrictive relations do the opposite. Yet this trade-off is nuanced; recall the discussion on the cost of order from Section 3.7. Determinism can ensure both minimal parallelism or maximal parallelism throughout a programs execution; yet, this requires a comprehensive understanding of the program and ordering constraints. Building for maximal parallelism

avoids the necessity of such understanding, but it removes some of the tuning possible for programs and requires more elaborate programmer constructed intervention to recover ordering when it is needed.

First, what can I change? I have seen so far that I can change the permitted executions by changing the definition of happens before. This has involved changing the candidate events for ordering by changing the "type" of the event (*i.e.,* from Spawns to Run and Complete), and the permitted "path" between two candidate events. I will also show that I can change the definition of "conflict" between two events. In summary, the design space I will investigate in this section is:

- Change the type of events which are ordered by happens before
- Change the definition of "conflict" between two events which are ordered by happens before
- Change the definition of the path between events which are ordered by happens before

6.5.1 Type of events which are ordered

I changed the type of event ordered, when I transitioned from the definition of happens before in Definition 35 to that in Definition 38. Initially, the spawning events were ordered by happens before, but I showed that this provided slightly more determinism than I was looking for in a way that, I argue, the programmer couldn't make use of. By changing the happens before to be between the completion of one behaviour and the start of another, I removed this determinism.

Could I change the ordering to be on other types of events? In theory, yes of course, however are there any orders which make practical sense? Ordering two start events is essentially equivalent to ordering complete and start events when behaviours that overlap on cowns cannot execute in parallel (likewise for ordering two complete events).

Ordering spawn events with other types of events also does not look to serve any intuitive purpose. Moreover, in Definition 35, the happens before relating spawning events reflected a specific fact. That fact was, if $b1$ was coherence order before $b2$ then it *must* have been that $b1$ was spawned before $b2$. I used the known relation to work out more about what happened and see if it could possibly have happened.

This brief summary argues why this part of the design space does not have much more potential. So, I move on to look at other parts of the design space.

6.5.2 Conflict

The conflict relation tells us when two candidate events, related by a "path", need to be ordered. There are a few ways I could define this that I will now take a look at.

Recall that conflict is defined as in the following definition, so I can manipulate the definition in the underlying $\#$ relation.

Definition 33 (Conflicts)**.**

$$\mathrm{conflict}(i, j) \triangleq \neg(\mathrm{cowns}(i) \ \# \ \mathrm{cowns}(j))$$

Conflict never simply means I will never have a conflict and, thus, an empty happens before relation. This means there will be minimal determinism and maximal parallelism. I can create this by defining the disjoint relation as follows.

$$\overline{\kappa} \# \overline{\kappa'} \triangleq \top$$

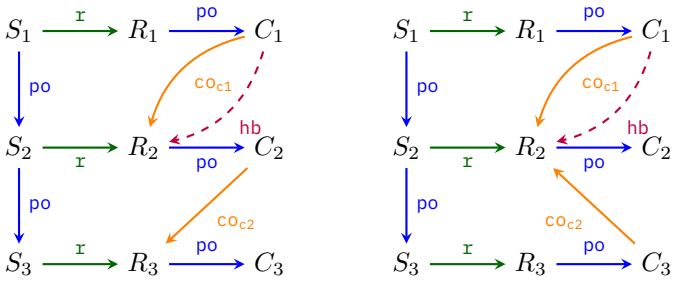
If the programmer wants to enforce order then they will have to build this themselves (for example through continuation passing). This is the choice taken in the programming language SALSA [27].

Conflict on subset means that b will conflict with b' whenever b requires a subset of the cowns required by b'. This introduces more determinism into the program than *conflict never* but not as much as *conflict on intersection*

$$\overline{\kappa} \# \overline{\kappa'} \triangleq \overline{\kappa} \subseteq \overline{\kappa'}$$

In the example that follows there are two possible executions (while conflict on intersection allowed only one). This exchanges determinism for parallelism with respect to the conflict relation I have explored so far.

```
1  when(c1) { /* b1 */ };
2  when(c1, c2) { /* b2 */ }
3  when(c2) { /* b3 */ };
```



I argue that this conflict relation is difficult to work with; consider in the example above, the lack of ordering knowledge between $b2$ and $b3$ means I cannot rely on $b2$ have executed before $b3$, which in a set of transactions I would likely want to do. Moreover, if I introduce permissions to cowns, such as read and write, I now need to extend what subset means for a read and write.

The reader may wonder why I am considering this conflict relation, given its asymmetry. The reason is that, this ordering was, initially and for a long time, the chosen ordering for BoC in project Verona. The rationale behind this was a combination of the programming model evolution, and artifacts of the programming model implementation.

I digress here from the current state of BoC, to explain this historical artifact. Project Verona began as an evolution of actor model programming, and the model treated cowns more like actors that rendezvoused. Each cown had its own message queue. Spawning a behaviour would order the cowns according to a global ordering, and then send a message to the first cown in the order. The message contained the ordered cowns (minus the first cown) and the closure to execute. Cowns would process their queue by removing the first message in the queue, marking itself as participating in a behaviour (and thus would no longer process any messages), finding the next cown in the ordered list, and sending the message to the next cown (removing the next cown from the ordered list). If there were no more cowns in the ordered list, the cown would instead execute the behaviour, which now had data-race free access to multiple cowns. On completion of the behaviour, the involved cowns were signalled to say they could process their next message.

The global order and relayed message results in the subset ordering. Let us see why by investigating the example from above. When I spawn $b1$, it is enqueued on the `c1` message queue. When I spawn $b2$, it is enqueued on either `c1` and then `c2`, or `c2` then `c1`. In either case, $b1$ will end up earlier in the queue of `c1` than $b2$ and thus will execute before $b1$. When I spawn $b3$, it is enqueued on `c2`. If $b2$ was enqueued on `c1` first and had not been relayed to `c2`, then $b3$ will be enqueued on `c2` before $b2$ and will execute before $b2$; conversely, if $b2$ was enqueue on `c2` first, then $b3$ will be enqueued after $b2$.

During the development of project Verona, the design of BoC has shifted away from a cown-centric paradigm, with multiple queues, to a behaviour-centric paradigm, with a centralised graph of behaviours. Thus I shifted to ordering behaviours based on cown overlap.

Conflict on intersection is what I have assumed for the discussion in the construction of happens before in Section 6.4 (repeated here).

Definition 34 (Disjoint on Intersection)**.**

$$\overline{\kappa} \# \overline{\kappa'} \triangleq \overline{\kappa} \cap \overline{\kappa'} = \varnothing$$

I introduce more determinism into the program than *conflict on subset*; and, if I revisit the example from *conflict on subset*, I can now rely on the fact that $b2$ will execute before $b3$.

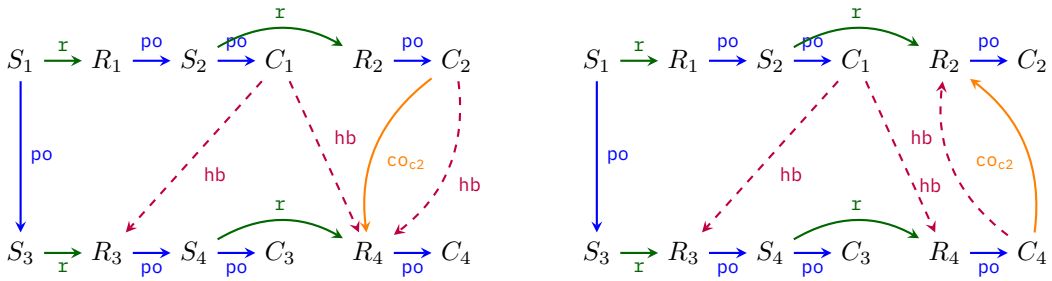
Conflict always means that any two behaviours will conflict, thus any two behaviours which have a "path" between them will conflict and must have a happens before relation between.

Definition 39 (Conflict always)**.**

$$\overline{\kappa} \# \overline{\kappa'} \triangleq \bot$$

This is the maximum determinism, and conversely the minimum parallelism, I can achieve with the conflict relation. However, this is only as strong as *conflict on intersection*, seen in Definition 38. To see why, let us look at an example (assume I have the "path" as I have shown it so far).

```
1 when(c1) { /* b1 */ when(c2) { /* b2 */ }; };
2 when(c3) { /* b3 */ when(c2) { /* b4 */ }; }
```



Both of the executions shown here are valid. There are happens before edges between any behaviours connected by a "path", regardless of the cowns used. However, there are no coherence edges between the behaviours that do not use the same cowns, so these new happens before edges cannot invalidate the non-existent coherence order of such behaviours; furthermore, the path does not incorporate the happens before edges to construct further happens before edges. This means, for the example above, $b2$ and $b4$ can execute in either order.

6.5.3 Path

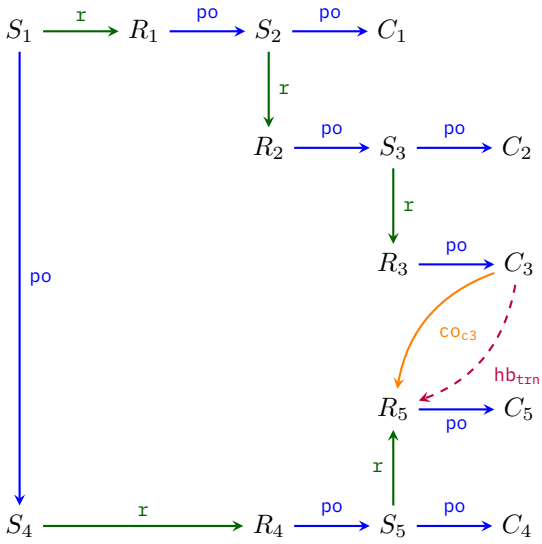
As I discussed, whilst constructing the happens before relation, the "path" linking behaviours affects which events are candidates for conflict. I have to be careful when constructing the "path" otherwise I can end up creating creating ordering relations between events which simply aren't achievable, for example that a child must execute before itself or some ancestor that lead to it being spawned in the first place.

Assume for this section I use *conflict on intersection* (unless stated otherwise).

Depth first

What if I wanted a more "transactional" feel to our order? That is to say, nested behaviours complete before sibling behaviours. Consider the example that follows, what if I want $b3$ to happen before $b4$.

```
1  when(c1) { /* b1 */
2    when(c2) { /* b2 */
3      when(c3) { /* b3 */ };
4    };
5  };
6  when(c5) { /* b4 */
7    when(c3) { /* b5 */ };
8  }
```



To achieve this I must have a path between $b3$ and $b5$. In Definition 40 I construct a new "path" that links sibling behaviours and all of their descendants, such that all of the first behaviours descendants must complete fist.

Definition 40.

$$\mathtt{hb}_{\mathtt{trn}} \triangleq \{(C_i, R_j) \mid \exists S_a, S_b.[C_i\ (\mathtt{po}^{-1} \mid \mathtt{r}^{-1})^*\ S_a\ \ \mathtt{po}^+\ \ S_b\ (\mathtt{po} \mid \mathtt{r})^*\ R_j] \wedge \mathrm{conflict}(i,j) \wedge i \neq j\}$$

In theory, this should allow a large amount of parallelism, in the example above there is only one required ordering edge. However, let us consider how this order correlates with an implementation of BoC.

The ordering in the operational semantics from Chapter 4 is achieved as the ordering follows the incremental spawning order. If b_1 is spawned before b_2 and both use a cown c_1, then b_1 will execute before b_2, then if both spawn a subsequent behaviour, b_1' and b_2', which uses c_2 then I know b_1' will be spawned before b_2' and so appears earlier in the queue and will be executed earlier, thus aligning with a valid execution order as per the happens before definition I defined in this chapter.

However, in this section, I have a completely different ordering to that which I have established in our operational semantics. In the example above, I do not know in which order $b3$ and $b5$ will be spawned, but I am requiring that they execute in a particular order. Thus, I would need to construct an implementation that can account for behaviours being spawned in an order which is more permissive than they order in which they start. I can take inspiration from transactions and go one of two ways:

- Optimistic: I could, like optimistic transactions, allow behaviours to start and potentially rollback their effects when I find a later behaviour conflicts. However, I would then need to create a model for how nested behaviours work, much like for nested transactions which has various answers to this question.
- Pessimistic: I could, like pessimistic transactions, not start other behaviours that could potentially conflict. I could either: dynamically analyse the cowns that all descendents will need, however this is fairly difficult as cowns will only be resolved dynamically; or, depth first spawn and start and behaviour and all of its descendents, which will remove almost all of the parallelism from a program.

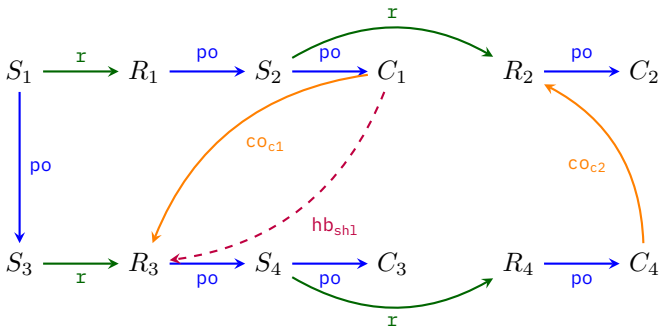
The main difficulty in choosing this ordering relation for BoC compared with transactions, is that transactions are typically used as a coordination mechanism between parallel threads of execution. Behaviours,

however, are asynchronously spawning and executing, and there are no threads of execution; the coordination and parallelism are combined. Thus, a BoC program is permanently in a "transactional context" and so finding parallelism or resolving conflict will be difficult.

Shallow

I could instead only order direct siblings and forgo a deeper relation. For example; the below, where $b1$ executes before $b3$ and $b4$ before $b2$, would be a valid execution. This investigation is purely exploratory, and I do not have a use-case where this would be a preferable ordering.

```
1 when(c1) { /* b1 */
2   when(c2) { /* b2 */ };
3 };
4 when(c1) { /* b3 */
5   when(c2) { /* b4 */ };
6 }
```



To permit this, I change the "path" in happens before to, simply, the following definition.

Definition 41.

$$\mathtt{hb_{shl}} \triangleq \{(C_i, R_j) \mid C_i \, \mathtt{po}^{-1*} \, \mathtt{r}^{-1} \, \mathtt{po}^{*} \, \mathtt{r} \, R_j \wedge \text{conflict}(i,j) \wedge i \neq j\}$$

This removes some of the determinism and increases the available parallelism. In theory, this could allow an implementation to reorder some behaviours as their are fewer orderings. In practice, how one would implement this order in such a way as to not result in the same order I constructed for Definition 38 is unclear. Some queue of behaviours is likely to required, and how $b4$ could end up earlier in that queue than $b2$ isn't obvious. Moreover, adopting this definition would mean for a programmer to restore the $b2$ completes before $b4$, would require either nesting $b3$ within $b1$, or some bespoke callback based signalling data structure.

This shallow ordering is a weaker order than the happens before order BoC provides. Any order provided by the shallow order, is also provided by BoC's happens before order. The *potential* parallelism gain does not seem like a worthwhile trade-off.

6.5.4 Concluding on the design of ordering

We've seen some different ordering possibilities in these sections and touched on the trade-offs of the designs. I feel that the choice made for BoC, *i.e.,* Definition 38, finds a sweet spot between determinism and parallelism. In Chapter 7, I will demonstrate qualitatively how I can create programs using this ordering, and how this ordering affects the performance of programs.

On the principles of order. Whilst I have explored part of the design space for ordering in BoC with respect to parallelism and determinism, a crucial factor I have not considered, in depth, is the intuitiveness and usability of the order. This is much harder to capture and evaluate than answering whether an order precludes or permits a particular execution, or whether one ordering provides better performance, on average, than another.

6.6 Cown reads and writes

So far, I have considered the ordering of behaviours, however the literature on weak memory models orders the reads and writes to specific memory addresses. I now investigate the reads and writes to cowns, and demonstrate that their order can be derived from existing relations in Definition 28.

Let us, loosely, enrich our model to consider the reads and writes of the shared cowns. I will add two new types of event: $Wr_{x=y}$ which states a write to x with value y; and $Rd_{x=y}$ which represents a read of x with value y. Similarly I will add two new types of relations: `cow` (coherence-order on writes) which for $Wr_{x=y}$ `cow` $Wr_{x=z}$ states the $Wr_{x=y}$ precedes $Wr_{x=z}$; and `rf` (read from) which for $Wr_{x=y}$ `rf` $Rd_{x=y}$ states the write event from which the read must take its value. The program order relation is also extended to order these events within a behaviour.

Recall that only one behaviour can acquire a cown at a time, and for the duration of the behaviour, it is the only behaviour that can access the cown (assuming the language ensures the isolation properties I have discussed). So I do not need the execution to provide the `cow` and `rf` orders as they are derivable from the relations I already have. The first read or write of a cown by a behaviour must be ordered after the last write to that cown from a previous behaviour (an event which is uniquely identifiable by following the coherence order backwards). Once the behaviour does write, all reads and writes will be sequential within the behaviour. Thus, I need the read and write events as input to the execution, but I can derive the order in which they happen. This derivation is provided in Definition 42.

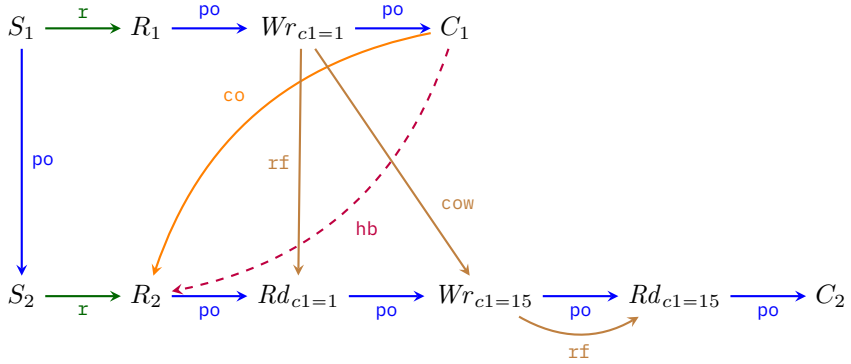
Definition 42 (Coherence Order Writes and Read From)**.**

$$Wr_{c=x}\ \texttt{cow}\ Wr_{c=y} \triangleq Wr_{c=x}\ (\texttt{po} \mid \texttt{co})^{+}\ Wr_{c=y} \wedge \neg\exists\, Wr_{c=z}.[Wr_{c=x}\ (\texttt{po} \mid \texttt{co})^{+}\ Wr_{c=z}\ (\texttt{po} \mid \texttt{co})^{+}\ Wr_{c=y}]$$
$$Wr_{c=x}\ \texttt{rf}\ Rd_{c=y} \triangleq Wr_{c=x}\ (\texttt{po} \mid \texttt{co})^{+}\ Rd_{c=y} \wedge \neg\exists\, Wr_{c=z}.[Wr_{c=x}\ (\texttt{po} \mid \texttt{co})^{+}\ Wr_{c=z}\ (\texttt{po} \mid \texttt{co})^{+}\ Rd_{c=y}]$$

Definition 42 says there is a relation between a write and write, or write and read, if the two events access the same cown and there is no intervening write to that cown.

Let us look at an example of how these orders work, in the examples that follows I write to a cown in one behaviour, and then read and write to that cown in a subsequent behaviour that will execute afterwards. The new `cow` and `rf` edges are show in red.

```
1  when(c1) { /* b1 */
2    c1 = 1;
3  };
4
5  when(c1) { /* b2 */
6    x = c1;
7    c1 = 15;
8    x = c1
9  }
```



I show the `cow` and `rf` relations follow the `co` and `po` edges that I have demonstrated so far, and indeed this makes sense.

The impact of this is that I can simplify the reasoning of executions and programs. I can reason about a behaviour locally, considering only the effects of that behaviour on the cowns, given their starting configuration. However, across behaviours, I need only know the coarser behaviour orderings to reason about the program as whole. Indeed, this is the desired outcome for atomicity in concurrent programming.

6.7 Future Work

One direction for future work is to prove correspondence between an axiomatic model and operational semantics.

I have introduced a number of axiomatic models and claimed they correspond to a particular operational semantics, such as between Definition 35 and Figure 4.1. A desirable future goal is to substantiate the claim that an operational semantics is as permissive as an axiomatic model. Moreover, if the operational semantics are less expressive, having the language to say exactly how much less.

6.8 Conclusion

In this chapter I constructed an axiomatic model of BoC and demonstrated how I use it to define valid complete executions of programs. This axiomatic model abstracts away from the finer-grained details of an operational semantics, considering only a trace of key global events including spawning, starting and completing. This provides us with the flexibility to more concisely define the causal order of BoC and permitted executions. With this axiomatic model, I can more easily consider whether the emergent executions of operational semantics are valid or invalid, and refine them as appropriate.

7 Evaluation

In this chapter, I will evaluate the central contribution of this thesis; namely, the new concurrency paradigm behaviour oriented concurrency. This evaluation is undertaken through both quantitative and qualitative analysis. This chapter will also investigate situations where BoC may not be the right fit, acknowledging potential pitfalls and providing an understanding of its limitations.

Quantitative analysis I establish a baseline using Savina, a benchmark suite for actor model programming, with a specific focus on the Pony programming language. This benchmarking exercise not only validates the performance viability of BoC but also emphasizes its comparability with the established actor model paradigm. To exemplify practical scalability, a `ReasonableBanking` scenario is examined, showcasing BoC's capacity to elicit scalable parallelism.

Qualitative analysis I collect a corpus of diverse program challenges and constructs, including well-known instances such as the Santa problem and synchronization barriers. These examples serve as litmus tests for BoC's expressiveness, illustrating its capability to address a many concurrent programming challenges.

7.1 Quantitative Evaluation

Behaviour-oriented concurrency has been implemented as part of project Verona[4, 71] and I will use this C++ implementation to evaluate BoC in this section, providing context on this implementation. I also demonstrate that BoC is a paradigm that can provide scalable parallelism, and compare its performance against that of the actor model paradigm. Note that I am not aiming to demonstrate the raw performance of the implementation of BoC.

7.1.1 Implementation

First, I will provide some context on the C++ runtime library for BoC to show: where the implementation differs from the operational semantics in Chapter 4; and the API for the C++ implementation. This implementation has been designed and written by Microsoft/Azure Research and a detailed description can be found in our co-authored paper[1].

Behaviour DAGs instead of a behaviour queue

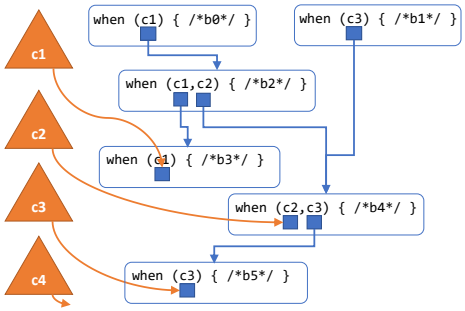
In Chapter 5 I presented an operational semantics built on a single global queue of pending work. In practice this would become a serious point of contention; every spawned behaviour would need to be atomically appended to the queue of work, requiring frequent locking and unlocking. Instead of building this work store as a global queue, the work store is built as a dependency graph which provides an isomorphic representation of work.

Listing 7.1 and Figure 7.1 will be used as an illustrative example for how this dependency graph works.

Listing 7.1: Example of ordering

```
1
2  when (c1) { /* b0 */ }
3
4  when (c3) { /* b1 */ }
5
6  when (c1, c2) { /* b2 */ }
7
8  when (c1) { /* b3 */ }
9
10 when (c2, c3) { /* b4 */ }
11
12 when (c3) { /* b5 */ }
```

Figure 7.1: Dependency graph for Listing 7.1



The graph is constructed from behaviours, each behaviour points to the behaviour(s) which will execute once the behaviour has executed. A behaviour which has no predecessors in the dependency graph can be executed. Once it completes executing, it removes itself from the predecessor set of all it successors in the dependency graph. For instance, in the dependency graph in Figure 7.1, $b0$ may execute, and once it completes it can remove itself from $b2$'s predecessors. Hence, when both $b0$ and $b1$ have completed, $b2$ will have no predecessors, and thus can execute.

When a behaviour is spawned, the graph is extended based on the cowns required. The last behaviour to require a particular cown is pointed to by that cown. For instance, in Figure 7.1, the cown `c1` refers to $b3$. This specifies where the dependency graph must be extended for the next behaviour that uses `c1`.

If I were to schedule `when(c1,c4){/*b6 */}` on the dependency graph in Figure 7.1, I would update the cowns `c1` and `c4` to point to $b6$, and add $b6$ as a successor to $b3$ as that was the last value for `c1`. As `c4` had no behaviours using it, I would not need to add a successor for `c4`.

From the "perspective" of a cown, there is a single queue of behaviours that use this cown.

Building work in this graph representation decentralises the work scheduling for the implementation and allows behaviours to be spawned in parallel.

C++ API

In this thesis so far, I have been using a pseudo-code for building BoC programs. Here, I will explain the C++ API for the BoC runtime library that has been used to build the benchmarks that follow and my repository of BoC examples[42]. I will use the following Listing 7.2 to explain the API.

`cown_ptr` is a smart pointer (thus manages reference counting and automatic memory management) which represents shared access to a cown. The pointer can only be used with `when` to get underlying access. `make_cown` constructs a new `cown_ptr`. So on Lines 2 and 3 I construct new cowns of `Account` (an assumed existing type for bank accounts).

`when(cowns) << [captures](acquired_cowns)` represents the BoC-like "when" and the implementation details are somewhat complicated, so here I will explain the essence of what it achieves. This spawns a behaviour where `cowns` are the list of `cown_ptr` objects that are required for the behaviour, `captures` are local variables holding pass-by-value or moved data (the spawned behaviour must have isolated access), `acquired_cowns` name objects of type `acquired_cown` which have access to the contents of the cown when the behaviour executes (the inner types of `cowns` and `acquired_cowns` must match). So in Line 7 a behaviour is spawned which requires `src` and `dst`, names the acquired contents with the same variable names (this is a convention but not a requirement), captures the variable `amt`, and transfers `amt` from one account to another.

Listing 7.2: Bank transfer

```
1  void transfer() {
2    cown_ptr<Account> src = make_cown<Account>(100);
3    cown_ptr<Account> dst = make_cown<Account>(0);
4
5    const int amt = 50;
6    when(src, dst)
7      << [amt](acquired_cown<Account> src, acquired_cown<Account> dst) mutable {
8      src->balance -= amt;
9      dst->balance += amt;
10   };
11 }
```

Recall that C++ does not, natively, ensure anything akin to the isolation properties defined in Chapter 5. Such invariants must be ensured by the programmer when using this runtime library.

7.1.2 Demonstrating performance

Savina is a benchmark suite designed to compare implementations of the actor paradigm [58]. This suite is widely known in the actors community and has been used to compare the performance of several actor languages[59]. I chose Savina for our evaluation for two reasons:

1. Actors model programs have close relation to BoC and there is a straightforward translation from actors to BoC. Thus, I can compare performance of our runtime against existing runtimes.

2. There are patterns in actor model programs which I have identified as places where BoC is a better fit; thus, I can compare tradeoffs between these paradigms.

All our experiments were run on an Azure F72s v2 instance, which has 72 hardware threads. My, BoC, implementation and the Pony implementation of these benchmarks can be found at [72] and [73] respectively.

7.1.3 Mapping Pony Actors to BoC

Here, I will motivate the correspondence between actors and BoC through a translation from one to the other. In this chapter, I will refer to actor programs mapped to BoC as "BoC (Actor)" programs, and programs utilising the full power of BoC as "BoC (Full)".

Constructing an equivalent BoC program from an actor program is a straightforward process. An actor is constructed from some private state and a message box. In response to messages, the actor mutates its state and sends messages to other actors. Thus the state and behaviour are encapsulated together.

In BoC the state, captured in cowns, and behaviour are separated. I can emulate an actor in BoC, by wrapping actor state in a cown and spawning behaviours that mutate only a single cown.

I now look at this concretely, translating from Pony actors to C++ BoC.

- Actors become structs.
- Actor constructors become a static method that create a new cown containing a struct for the actor, they spawn a new behaviour on the actor which asynchronously runs the initialiser code (due to the ordering of behaviours I know this will be the first behaviour to run), then the cown is returned to the caller.
- Behaviours become static methods which take equivalent arguments as well as an additional cown, this cown is the actor which will run the behaviour, I then spawn a behaviour that will run the code of the original behaviour.

```
actor MyActor
  new make(/* args */) =>
    /* init */

  be my_behaviour(n:U64, o:MyActor) =>
    /* behaviour */
```

The above actor program is translated into the below BoC program:

```
struct MyActor {
  static cown_ptr<MyActor> make(/* args */) {
    MyActor actor = make_cown<MyActor>();
    when(actor) << [/* args */](acquired_cown<MyActor> actor) {
      /* init */
    };
    return actor;
  }

  static void my_behaviour(cown_ptr<MyActor> self,
                           uint64_t n, cown_ptr<MyActor> o) {
    when(self) << [n, o=move(o)] (acquired_cown<MyActor> self) {
      /* behaviour */
}; } };
```

7.1.4 Evaluation of "BoC (Actor)"

To evaluate "BoC (Actor)", I compare the performance of the programs in the Savina suite written in an actor language, and written in "BoC (Actor)" – *i.e.*, BoC where each behaviour runs on exactly one cown. Such a comparison allows us to investigate whether BoC can be implemented with a similar runtime overhead to that of actors', and compare how these two models performance scale with increasing work and cores.

Table 7.1: Average runtime (ms) on Savina benchmarks. Parenthesised values give logarithm relative overheads to single core Pony, $\log(\frac{x}{\text{1 core Pony}})$.

	Pony				BoC (Actor)							
Benchmark	LoC	1 core	8 cores		LoC	1 core		8 cores			cowns	behs
Banking	105	184 ± 1.5	48.5 ± 2.0	(-0.6)	117	**19.8** ± 0.6	(-1.0)	22.6 ± 1.6	(-0.9)		1001	$10^{5.4}$
Big	65	250 ± 0.3	501 ± 0.4	(0.3)	83	247 ± 0.2	(-0.0)	**202** ± 0.3	(-0.1)		121	$10^{6.7}$
Bounded Buffer	132	2984 ± 0.1	406 ± 0.1	(-0.9)	142	1593 ± 0.1	(-0.3)	**342** ± 0.1	(-0.9)		81	$10^{5.2}$
Chameneos	103	110 ± 0.5	522 ± 0.1	(0.7)	105	**46.1** ± 0.2	(-0.4)	94.5 ± 1.4	(-0.1)		101	$10^{5.9}$
Cig Smokers	45	1.9 ± 4.0	2.2 ± 0.7	(0.1)	66	**0.7** ± 0.4	(-0.4)	1.4 ± 0.4	(-0.1)		201	$10^{3.5}$
Conc Dict	65	256 ± 0.4	366 ± 0.4	(0.2)	84	**22.7** ± 0.3	(-1.1)	66.5 ± 1.2	(-0.6)		22	$10^{5.6}$
Conc Sorted List	90	13653 ± 0.2	15059 ± 1.0	(0.0)	108	**6206** ± 0.1	(-0.3)	9913 ± 0.8	(-0.1)		22	$10^{5.5}$
Count	28	**25.1** ± 1.4	120 ± 9.4	(0.7)	40	46.1 ± 0.4	(0.3)	47.7 ± 0.4	(0.3)		2	$10^{6.0}$
Dining Phils	78	142 ± 0.5	560 ± 2.9	(0.6)	94	**73.3** ± 0.2	(-0.3)	165 ± 0.9	(0.1)		21	$10^{6.1}$
Fib	44	322 ± 0.8	46.7 ± 1.4	(-0.8)	51	29.3 ± 0.3	(-1.0)	**19.4** ± 0.7	(-1.2)		150049	$10^{5.5}$
Filterbank	225	2787 ± 0.1	380 ± 4.2	(-0.9)	247	1170 ± 0.1	(-0.4)	**349** ± 1.6	(-0.9)		62	$10^{6.2}$
FJ Create	28	50.2 ± 1.6	31.3 ± 2.2	(-0.2)	42	**10.5** ± 0.4	(-0.7)	12.4 ± 0.4	(-0.6)		40001	$10^{4.9}$
FJ Throughput	42	**50.2** ± 3.6	207 ± 5.9	(0.6)	53	53.4 ± 0.9	(0.0)	101 ± 0.6	(0.3)		61	$10^{6.1}$
Map Series	137	832 ± 0.1	44.1 ± 12	(-1.3)	133	**39.0** ± 0.4	(-1.3)	41.2 ± 4.0	(-1.3)		21	$10^{5.9}$
Ping Pong	29	7.2 ± 0.4	51.3 ± 3.4	(0.9)	43	**4.6** ± 0.5	(-0.2)	5.4 ± 0.4	(-0.1)		2	$10^{4.9}$
Quicksort	126	234 ± 0.2	67.1 ± 5.2	(-0.5)	121	125 ± 0.3	(-0.3)	**58.8** ± 0.4	(-0.6)		1935	$10^{3.6}$
Radixsort	77	180 ± 0.5	210 ± 0.8	(0.1)	105	236 ± 0.2	(0.1)	**149** ± 0.8	(-0.1)		61	$10^{6.8}$
Matrix Mul	142	9825 ± 2.6	1158 ± 9.3	(-0.9)	156	1614 ± 0.1	(-0.8)	**563** ± 1.5	(-1.2)		22	$10^{3.4}$
Sieve	64	222 ± 66	**45.2** ± 9.9	(-0.7)	68	331 ± 0.2	(0.2)	104 ± 3.8	(-0.3)		10	$10^{5.0}$
Sleeping Barber	113	3652 ± 0.3	**261** ± 4.7	(-1.1)	127	1660 ± 0.1	(-0.3)	3544 ± 0.9	(-0.0)		5003	$10^{7.4}$
Thread Ring	39	11.0 ± 1.1	95.1 ± 2.3	(0.9)	46	**4.7** ± 0.6	(-0.4)	5.5 ± 0.3	(-0.3)		100	$10^{5.0}$
Trapezoid	62	842 ± 0.1	**115** ± 0.9	(-0.9)	72	970 ± 0.1	(0.1)	172 ± 0.1	(-0.7)		101	$10^{2.3}$

I chose to compare with the actor language Pony[74] because the latest comparison[59] showed Pony to be comparable in performance with both Akka[75, 76] and CAF[77, 78]. Thus, by being comparable with Pony I are comparable across the space of actor-based languages, and demonstrate that BoC does not introduce high overheads. Furthermore, Pony and BoC share some agreeable language features, *e.g.,* message ordering guarantees, and no need to explicitly terminate actors (poison pills). Finally, many of the implementation design decisions were taken from the Pony implementation which makes for more direct comparison[64, 23].

I present the results in Table 7.1, where I run each program with 1 and 8 cores. The Savina benchmark suite consists of 30 programs. In accordance with Blessing et al. [59], I dropped 8 programs because they relied on language specific, non-standard libraries, and one has problems with termination; thus, I have 22 programs. I ran each benchmark 100 times and report the average and an approximation of the confidence interval (the standard error times 1.96). I use Pony version 0.53.0. To give additional insights into the benchmarks, I present the number of cowns and behaviours in each benchmark (obtained by instrumenting the runtime to detect cown/behaviour allocation and logging the result on termination).

The "BoC (Actor)" implementations are faster than Pony on 17 out of 22 benchmarks. The majority of results are similar between the two implementations.

Where Pony has better performance The worst performance of our runtime relative to Pony is the Sleeping Barber. This involves busy waiting, which causes a lot of pointless work on the runtime. Pony has a mechanism for back pressure that gives the busy waiting a low priority, so the rest of the system makes progress.

The second worst performance is Sieve of Eratosthenes. When analysed, using the profiling information,

it was discovered that Pony was using 32-bit division, whereas BoC was using 64-bit division. It is unclear why Pony used 32-bit division, but if I change the BoC implementation to use 32-bit division, then BoC elicits similar performance to Pony.

The single core Count works better for Pony as the overhead of allocating a message is lower. The example is completely single threaded, there is at most one Actor/Cown that can be executed at a time. For the multi-threaded run, BoC works better on this example than Pony due to a work batching scheme in BoC. Pony moves the work between multiple threads slowing down the processing.

Observe that the LoC for each benchmark is often lower in Pony; yet, the largest delta is 28 LoC in Radix Sort. Here are some reasons for the difference: Pony doesn't use braces to delineate scope; In BoC each **`when`** is wrapped in a function call to match the style of behaviours in Pony; Pony provides union types and parametric polymorphism that C++ and the BoC runtime do not. Radix Sort is strongly affected by these last two issue. However, I will show in Table 7.2 in Section 7.1.6 that BoC improves for multi-cown behaviours.

Where BoC (Actor) has better performance The Bounded Buffer, Concurrent SortedList and Matrix Mult all had poor performance on Pony relative to the BoC runtime. When the profiling information was investigated, each of these examples had generated less optimal inner loops for the core computation.

The Banking, Chameneos and Concurrent Dictionary examples pass references to actors in messages. In the C++ BoC runtime, this is supported with reference counting. In Pony this uses remembered sets and tracing. Profiling showed that these memory management costs caused the overhead. Similarly, Fib's overheads for Pony are primarily due to the cost of deallocating actors, which is cheaper in the BoC runtime.

Overall, the majority of difference are not due to the BoC runtime, so there is no evidence that BoC introduces high overheads.

7.1.5 Evaluation of "BoC (Full)"

For our comparison based on the Savina suite, I considered which of the Savina programs would benefit from the extra power afforded by behaviours that support more than one cown. I then rewrote these programs using BoC to utilise behaviours and cowns; I refer to these versions of the program as the "BoC (Full)" versions. Not all Savina programs offer scope for change (for example, actors sending messages in a ring or pinging each other), but there are several programs which do have scope. They fall broadly into the following two categories (*c.f.,* Table 7.2): Multi-actor operations, where **`when`** helps, and Parallelising workloads where behaviour ordering helps.

Table 7.2: Overview of Savina benchmarks considered.

Pattern	Benchmarks
Multi-actor	Banking, Barber, Dining Philosophers, Chameneos
Parallelism	Logistic Map Series, Count, Fibonacci, Fork-Join Create, Fork-Join Throughput, Quicksort, Trapezoid

Multi-actor operations These are problems such as two-phase commit and rendezvous of actors. Two (or more) actors must atomically update state through message exchange; for example, a transaction manager coordinates updates between two bank accounts, such that an account is only involved in one operation at a time, and either both or neither accounts are updated. In Section 7.1.6 I will discuss this pattern in detail. Using BoC (Full) reduces the required message through a **`when`** that acquires multiple cowns at once.

Parallelising workloads These are problems in the vein of divide and conquer, fork-join, and map reduce, where a problem is decomposed into smaller problems that can be solved in parallel and combined. In actor systems, each parallel solution needs to send a message back to an aggregating actor to recompose the solution. Using BoC (Full) reduces the required message, as causal order ensures sub-solutions are only aggregated once they have been computed, and thus avoids sending messages back to an aggregating actor. The BoC implementation of `Fib` (i.e. fibonacci) using this pattern can be found at Section 7.2.3. Note, this pattern requires the work to be divided in a single behaviour, so whether this is faster depends on the amount of work performed in the asynchronous behaviours. For Fibonacci this change improves performance whereas for Quicksort it does not, thus there are related decisions regarding which algorithms should be used for a program.

In Table 7.3, I compare the performance of these programs written in BoC (Actor) and written in BoC (Full). I see that 6 benchmarks demonstrate significant improvement in performance. I also present the lines of code for the implementations of the benchmarks as an approximation of the complexity of the solutions. Again, I see that 9 of the benchmarks are smaller than the Actor based implementations. I present the behaviour count for each benchmark, and additionally for BoC (Full) I separate this by the number of cowns used in each behaviour. I observe that all the benchmarks with significant speed-ups have a significantly lower behaviour count, and a high percentage of these use two cowns. In some benchmarks the number of cowns required changes when using BoC (Full); often this number gets lower, e.g. Fib, as BoC constructs the computation bottom up and so is able to reuse cowns, whilst in Pony the computation is top down with actors waiting in place for the results of sub-problems; in the Dining Philosophers, in Pony, forks are not modelled as actors whilst they are in BoC, so as to decentralise the fork management and provide a better solution using BoC. I also observe that the total number of behaviours remains the same or decreases from BoC (Actor) to BoC (Full); this always comes from creating a single behaviour that accesses multiple cowns in place of multiple messages to coordinate access over multiple cowns.

Table 7.3: Average runtime (ms) on selected Savina benchmarks

	BoC (Actor)					BoC (Full)						
										behaviours		
Benchmark	LoC	1 core (ms)	8 cores (ms)	cowns	behaviours	LoC	1 core (ms)	8 cores (ms)	cowns	1 cown	2 cowns	total
Banking	117	19.8 ± 0.6	22.6 ± 1.6	1001	$10^{5.4}$	78	**7.9** ± 1.2	9.3 ± 4.5	1001	$10^{4.7}$	$10^{4.7}$	$10^{5.0}$
Chameneos	105	**46.1** ± 0.2	94.5 ± 1.4	101	$10^{5.9}$	85	49.2 ± 0.3	122.1 ± 1.7	101	$10^{5.6}$	$10^{5.3}$	$10^{5.8}$
Count	40	46.1 ± 0.4	47.7 ± 0.4	2	$10^{6.0}$	30	**45.4** ± 0.4	46.9 ± 0.6	2	$10^{6.0}$	$10^{0.0}$	$10^{6.0}$
Dining Phils	94	73.3 ± 0.2	164.9 ± 0.9	21	$10^{6.1}$	61	22.2 ± 0.7	**16.6** ± 0.8	41	$10^{5.3}$	$10^{5.3}$	$10^{5.6}$
Fib	51	29.3 ± 0.3	19.4 ± 0.7	150049	$10^{5.5}$	28	**10.9** ± 0.4	13.5 ± 0.8	75025	0	$10^{4.9}$	$10^{4.9}$
FJ Create	42	10.5 ± 0.4	12.4 ± 0.4	40001	$10^{4.9}$	42	**9.6** ± 0.3	11.8 ± 0.3	40001	$10^{4.6}$	$10^{4.6}$	$10^{4.9}$
FJ Throughput	53	53.4 ± 0.9	100.8 ± 0.6	61	$10^{6.1}$	52	**35.8** ± 1.2	45.8 ± 2.2	61	$10^{5.8}$	$10^{1.8}$	$10^{5.8}$
Map Series	133	39.0 ± 0.4	41.2 ± 4.0	21	$10^{5.9}$	52	**17.4** ± 1.0	19.8 ± 0.9	21	0	$10^{5.4}$	$10^{5.4}$
Quicksort	121	124.7 ± 0.3	**58.8** ± 0.4	1935	$10^{3.6}$	85	105.9 ± 0.3	78.6 ± 0.4	968	$10^{3.0}$	$10^{3.0}$	$10^{3.3}$
Sleeping Barber	127	1660 ± 0.1	3544 ± 0.9	5003	$10^{7.4}$	106	**12.5** ± 7.7	16.5 ± 10.2	5003	$10^{4.8}$	$10^{4.0}$	$10^{4.9}$
Trapezoid	72	970.0 ± 0.1	**172.0** ± 0.1	101	$10^{2.3}$	68	958.3 ± 0.1	172.6 ± 0.1	100	$10^{2.0}$	$10^{2.0}$	$10^{2.3}$

I see improvements in both size and speed when adopting BoC (Full).

7.1.6 `ReasonableBanking`: The Banking Example Revisited

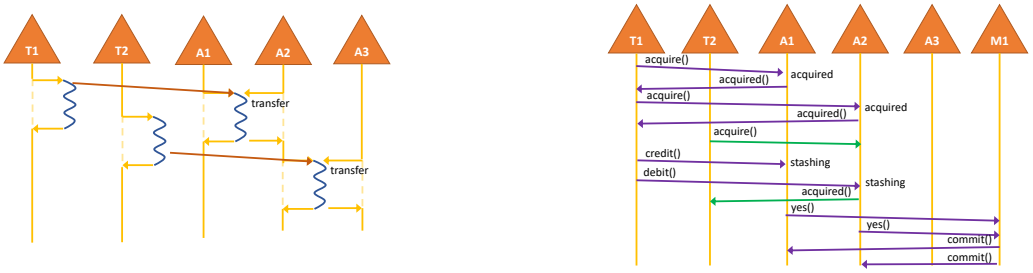
I now revisit the banking example, adding requirements that epitomise those often found in concurrency applications:

- Tellers can issue transactions between any accounts.
- There can be several Tellers issuing transactions to shared accounts.

- No livelocks and no deadlocks.
- Operations over multiple accounts are atomic.
- Any two transactions issued by one teller which involve the same account must be executed in the order they were issued.

I will refer to the collection of these requirements as `ReasonableBanking`. The `ReasonableBanking` guarantees are inherently achieved by the BoC paradigm. As already shown in Chapter 3, the behaviour `transfer` involves two account cowns, and transfers the money. Any number of tellers issue such `transfer` transactions between the accounts. Figure 7.2a shows two such transactions.

Whilst it is possible to achieve the `ReasonableBanking` guarantees in an actor language (e.g. Pony), it is difficult. The Savina banking benchmark does not satisfy the guarantees because (a) it is difficult (b) the benchmarks are micro-benchmarks designed to stress components of an actor system. One design to achieve this is shown in Figure 7.2b (which presents only part of the protocol for two transactions for brevity), and goes as follows: There are three types of actor involved in a transaction, `Tellers`, `Accounts` and `Managers`; `Accounts` have two state flags (`acquired` and `stashing`) and associated queues which are used to track incoming requests. A `Teller` selects two accounts at random and sends a message `acquire()` to the lowest `Account` address of the two, and records the other `Account` to be acquired. When the `Account` processes the `acquire()` message, if `acquired` is set then the acquire request is enqueued, otherwise `acquired` is set and the `Account` replies `acquired()` to the `Teller`. The `Teller` will then repeat this process with the other `Account`. Once both `Accounts` are acquired, the `Teller` will create a `Manager` actor, and send `credit()` and `debit()` messages to the `Accounts`, including a reference to the `Manager` in the message payload. Each `Account` will test whether it is processing an operation through `stashing`, if it is, then the message will be enqueued, otherwise the account will decide whether the operation will succeed or not and inform the `Manager` through a `yes()` or `no()` message. The `Account` will also mark itself as stashing; this also releases the acquisition by a `Teller` and the account will process the next pending acquisition (if one exists). The `Manager` will aggregate the responses and reply `commit()` or `abort()` to the `Accounts` and inform the `Teller` of a completed transaction. The `Accounts` will then commit or rollback the operation.



(a) with BoC (Full)

(b) with Actors

Figure 7.2: `ReasonableBanking`: Protocols for bank transfers

This is, evidently, a complicated protocol that requires careful construction Moreover, it requires the actors (here `Teller`, and `Account`) to be aware of the protocol, thus mixing business logic with the protocol. Furthermore, if a new set of actors had similar protocol requirements, then the behaviours described above would need to be mixed into *their* logic as well. My implementation `ReasonableBanking` in Pony can be found in Appendix E and [79]

Table 7.4 compares `ReasonableBanking` implemented in BoC (Full) and in Pony. Neither of the implementations scales with more cores; this is due to the very little work done by the each `transfer` – a common feature of the Savina suite. The BoC version is over 100 times faster than Pony on 1 core, and over 35 times

Table 7.4: Average runtime (ms) and lines of code (LoC) of ReasonableBanking benchmark

BoC (Full)			Pony		
LoC	1 core (ms)	8 cores (ms)	LoC	1 core (ms)	8 cores (ms)
78	7.9 ± 1.2	9.3 ± 4.5	226	987.9 ± 0.33	344.717 ± 0.35

faster than Pony on 8 cores. The Savina benchmarks focus on the runtime overheads, which causes the larger churn of messages to have a drastic effect on performance. In a realistic application, the improvement would be considerably smaller. The simplicity of the BoC version, already demonstrated in Figure 7.2, results in code that is 35% the length of that of Pony.

`ReasonableBanking` in related work I discuss related work tackling the issue of updating multiple actors atomically in Chapter 2; here I will use the `ReasonableBanking` example to elucidate some of the points more deeply.

In Aeon[31], atomicity is achieved through the use of dominators. Actors are arranged in a DAG, and atomicity is guaranteed by an actor that dominates all the actors required in a particular operation. This unavoidably restricts parallelism as a dominator is responsible for serializing events for a set of actors. If I were to attempt the `ReasonableBankingBanking` example, then I would need to introduce an actor that dominated all the accounts. This is an additional actor that is required by the paradigm, but not by the business logic. More importantly, this additional actor would introduce a single point of contention, which could harm performance. BoC, does not need this single point of contention.

Chocola[36] proposes that actors state is isolated, but actors can manipulate shared transactional state through transactions. This requires the programmer to decide if state should be transaction based or actor based. Transaction based access is synchronous whilst Actor access is asynchronous. Consider a sequence of three transfers: `src1` to `dst1`, `src2` to `dst2` and `dst1` to `dst2` in Chocola; the state of each account would need to be transactional for a behaviour to transfer funds (each transfer accesses two accounts, thus a single account cannot be an internal actor state). Moreover, I need that the third transfer should start executing only after the first two have completed. The first two transactions can execute in parallel as they access disjoint accounts. BoC is able to exploit the implicit parallelism whilst retaining order; in Chocola, the two readily available options are to start each transaction effectively in its own "thread" and lose all order, or execute them sequentially and lose all concurrency. To achieve both parallelism and order is the programmers burden through ad hoc coordination. These concerns affect `ReasonableBanking`, a series of ordered transfers.

The Akka Transactors[80] approach was built on STM, so again required splitting the world in transactional (STM) state and Actor state, and similar to Chocola performs synchronous STM operations. So would suffer the same negatives as described above. [1]

7.1.7 Scalability

Although the Savina benchmark suite is targeting highly concurrent Actor frameworks, many of the benchmarks do not have sufficient parallel work to benefit from multiple threads. For both Pony and BoC (Actor and Full), I see that about half of the benchmarks are faster with 1 core than 8 cores. This is due to the very small amount of work that is actually parallelised in the examples relative to the overhead of scheduling work. Additionally, the BoC runtime (and the Pony runtime) performs work stealing, which can get in the way if there is insufficient work.

[1] Note that Akka has dropped the support of transactors. https://github.com/akka/akka/pull/1878

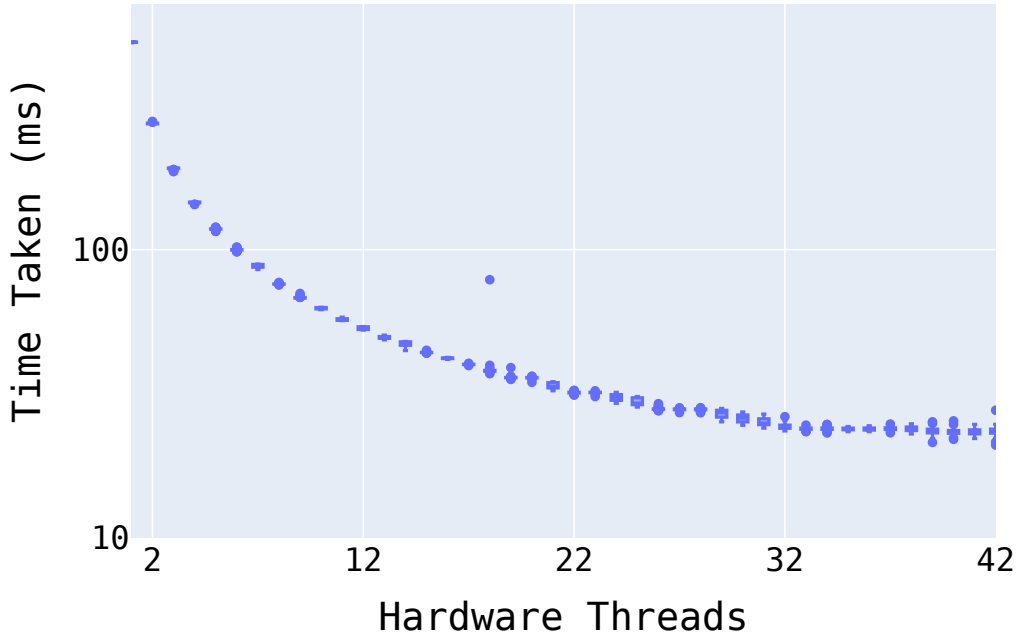


Figure 7.3: `BusyBanking` scaling

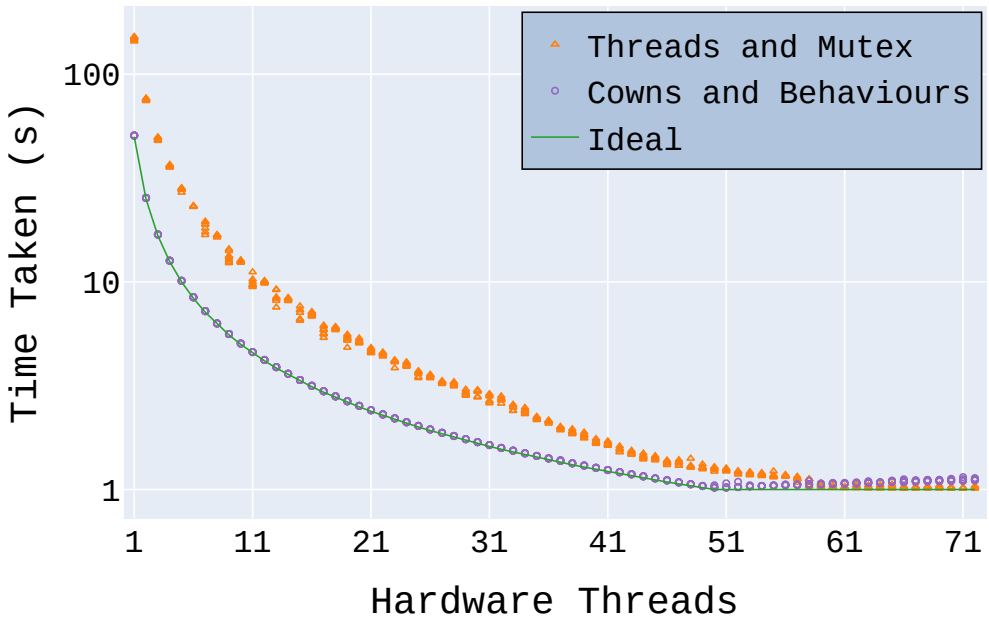


Figure 7.4: Dining Philosophers scaling

To illustrate the BoC runtime's potential for scaling, I use a modified version of the Savina Banking example using BoC (Full), which I call `BusyBanking`. This has a $10\mu s$ busy loop in each transaction. The results are presented in Figure 7.3, and I see the runtime scales well.

Our second scalability investigation is the Dining Philosophers. I compare the BoC runtime with an implementation using standard C++ abstractions (*e.g.,* `std::lock`). The number of Philosophers is set to 100, which means that the maximum parallelism is 50 concurrent eats. While holding both forks, a Philosopher spin-waits for 1ms. This additional processing delay has been added to enable defining the optimal processing time of the experiment on an overhead-free system given the number of philosophers, the number of times they eat, and the available system-level parallelism. The choice of 1ms was made to reduce the impact of system overheads, either coming from the operating system and its scheduler or from the BoC runtime implementation, on the experiment results. I time how long it takes for all the Philosophers to eat 500 times each. I present the time taken for each number of hardware threads in Figure 7.4. I also present the "Ideal", which is calculated by dividing the 50 seconds of busy work by the number of hardware threads up to the maximum parallelism of 50.

The experiment has three main purposes. First, it demonstrates that **BoC can be efficiently implemented on a real system** and achieves close to the optimal performance. Second, it **showcases the implicit parallelism that BoC offers** the programmer compared to plain mutual exclusion and how this can be leveraged to achieve the optimal performance. Third, it compares BoC with vanilla C++ abstractions offering mutual exclusion, *i.e.,* mutexes, and **shows how the OS scheduler that lacks application level-insights can hinder scalability**, as opposed to BoC where happens-before ordering makes this insights explicit to the BoC runtime.

I present "Cowns and Behaviours", in Figure 7.4, as the performance of our runtime. In this configuration BoC's happens-before ordering is leveraged to control the philosopher execution order. By scheduling alternating Philosophers, first the odd and then the even, I am able to ensure a really fair order of eating. All the odd philosophers will have a turn, and then all the even. The happens-before order of the runtime and the fixed and common eat time ensure this continues for the rest of the execution. As you can see, the performance closely matches the ideal.

As the example reaches the limit of the concurrency, I see the runtime starts to slow down slightly. This is believed to be because the current implementation of work-stealing does not backoff when there is insufficient work (harming performance).

Comparison to C++ threads and mutexes The results for C++ implementation using `std::lock` are labelled as "Threads and Mutex". The mutexes are acquired using `std::lock`, which ensures deadlock freedom and uses a back-off strategy, when it fails to acquire the set of locks. The backoff strategy takes a significant amount of the computation time. This is why the code starts significantly above 50s for the single hardware thread case. As the number of hardware threads increases, the performance improves, and the backoff becomes a much smaller percentage of the runtime. This approach does reach the maximum parallelism for some runs with high hardware thread count.

7.1.8 Conclusion

This evaluation suggests the BoC paradigm can support the construction of powerful protocols in a convenient way, and that the BoC runtime is competitive with Pony (and therefore with other actor implementations), and has the potential to be much faster for examples that need the full power of BoC.

7.2 Qualitative: Programming challenges and idioms

The aim of this section is twofold: to demonstrate BoC is expressive enough to solve concurrency challenges, and implement concurrency idioms; and, to demonstrate common program designs for programming using BoC.

In this section, I use following programming challenges and idioms to investigate the above points:

- Dining Philosophers Problem
- Barrier Synchronization
- Fibonacci numbers by divide and conquer
- Channels
- Santa Problem
- Boids

I will present only snippets of programs which demonstrate the salient points of the implementation; these snippets will also utilise the more concise pseudo-language I have been using instead of the C++ API. The full C++ implementation of these problems can be found in the repository of examples at [42].

7.2.1 Dining Philosophers in BoC

The dining philosophers is the classic example that demonstrates the potential for deadlock in concurrent programs using locks[51]. This example also well illustrates the deadlock free and flexibility of BoC. Consider Listing 7.3. I define a `Philosopher` with fields for `hunger`, `left` and `right` forks which are cowns. The core `eat` logic of a philosopher spawns a new behaviour that requires `left` and `right`. When the behaviour is run, it atomically `uses` both forks and decrements the hunger of the philosopher; if the philosopher is still hungry then eat is called again.

Listing 7.3: Dining Philosophers

```
1  class Philosopher
2  {
3    hunger: U64;
4    left: cown[Fork];
5    right: cown[Fork];
6
7    create(hunger: U64, left: cown[Fork], right: cown[Fork])
8    { ... }
9
10   eat(self) {
11     var l = self.left;
12     var r = self.right;
13     when (l, r) {
14       l.use(); r.use();
15       if (--self.hunger != 0) {
16         self.eat();
17 } } } }
```

7.2.2 Barrier Synchronization in BoC

Barriers are a common tool for synchronising execution in threaded programs; these ensure a thread blocks at the barrier, waiting until all other threads reach the barrier. I do not have the ability to block in BoC and, furthermore, BoC does not have threads to synchronise. I can create "barriers" which ensure some behaviours on a set of cowns have completed, before further behaviours using these cowns are spawned; for simple cases I can utilise the behaviour ordering which can be seen in Listing 7.4.

Listing 7.4: Barrier by ordering

```
1  var p1 = cown.create(Parti.create());
2  var p2 = cown.create(Parti.create());
3  when (p1) { /* use p1 on its own */ };
4  when (p2) { /* use p2 on its own */ };
5  when (p1, p2) { /* barrier for using p1 or p2 */ };
6  when (p1) { /* use p1 on its own */ };
7  when (p2) { /* use p2 on its own */ };
```

Listing 7.5: Barrier

```
1  class Barrier {
2   count: U64;
3   participants: Queue[Parti];
4
5   wait(barrier: cown[Barrier], p: Party) {
6    when (barrier) {
7     barrier.participants.add(p);
8     barrier.count = barrier.count - 1;
9     if (barrier.count == 0) {
10     /* iterate over participants
11          and call the next method */
12  } } } }
```

However, I need a different design if I require that any nested behaviour spawned on Line 3 or Line 4 must also complete before the "barrier". Such a design is presented in Listing 7.5: a `Barrier` maintains a counter and a list of participants; each call to `wait` decrements the count and adds a participant that is ready to be used in the next step; once the count hits zero, the list of participants can be emptied and each used in the next step of execution.

7.2.3 Fibonacci numbers by divide and conquer

Divide and conquer programming is a well-known paradigm for recursively decomposing a problem into smaller sub-problems; these sub-problems can then naturally be solved in parallel and their results combined.

In Listing 7.6, I create a divide and conquer approach to computing fibonacci numbers, which relies on behaviour ordering for correctness. For a call to `Fib::parallel(n)`, the size of the problem is measured by checking the value of `n`. If the problem is too large to solve sequentially, I recursively call `parallel` to solve a smaller problem (Lines 13 and 14); this provides two cowns with partial solutions, and I spawn a behaviour that requires both cowns and combines the results into the first (Line 15); finally, I return the cown. Otherwise, if the problem is small enough to solve sequentially, I create a new cown and spawn a behaviour that will fill this cown with the solution (Lines 8–10); this cown is returned to the caller.

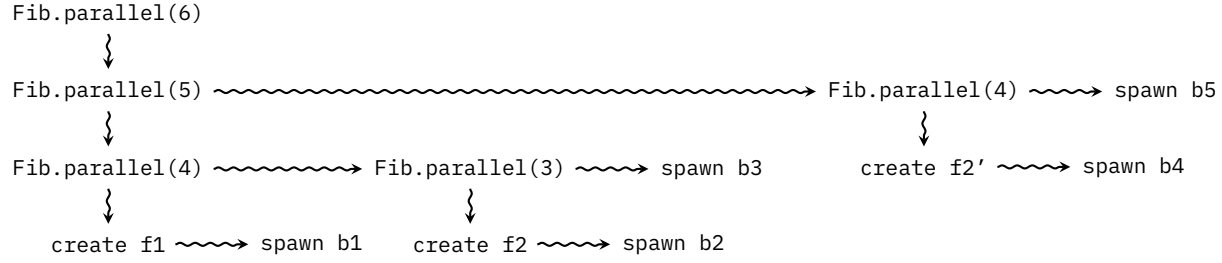
Listing 7.6: Divide and conquer fibonacci numbers

```
1  class Fib {
2    sequential(n: U64): U64 {
3      if n <= 1 { n }
4      else { Fib.sequential(n - 1) + Fib.sequential(n - 2) }
5    }
6
7    parallel(n: U64): cown[U64Obj] {
8      if n <= 4 {
9        var result = cown.create(U64(0));
10       when (result) { result.v = Fib.sequential(n); };
11       result
12     } else {
13       var f1 = Fib.parallel(n - 1);
14       var f2 = Fib.parallel(n - 2);
15       when (f1, f2) { f1.v = f1.v + f2.v; };
16       f1
17 } } }
```

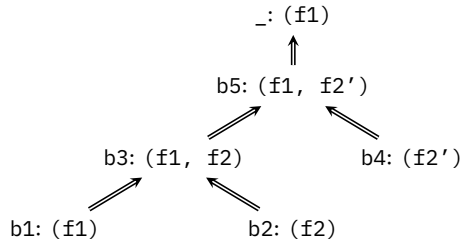
This example demonstrates the importance of the happens before order: Namely, it is crucial that the two sub-behaviours, which will be spawned as a result of the calls in Lines 13 and 14, have finished before the behaviour from Line 15 starts – otherwise the values in `f1.v` and `f2.v` would not be correct. Happens before ordering ensures this crucial requirement is met.

Contrast this implementation using BoC with how one might implement the algorithm using fork/join semantics: each sub-problem would be resolved in a forked thread, and then the two threads would use a blocking join to combine the results.

The diagram in Figure 7.5a shows the execution of the call `Fib.parallel(6)`. The statements shown in the diagram are executed sequentially, and the behaviours `b1`, …, `b5` are spawned in that order. While `b1` and `b2` may run in parallel, it is required that `b3` does not get run before `b1` and `b2` have terminated. The behaviour ordering rules from Definition 3 guarantee that `b3` does not get run before `b1` and `b2` have terminated, and similarly, that `b5` does not get run before `b3` and `b4` have terminated. The complete ordering of these behaviours is given in Figure 7.5b (this also includes a behaviour that will access `f1` once the final result is computed).

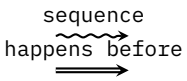


(a) Breakdown of work



(b) Behaviour ordering

```
b1: when (f1) {f1.v = Fib.sequential(4)}
b2: when (f2) {f2.v = Fib.sequential(3)}
b3: when (f1, f2) {f1.v = f1.v + f2.v}
b4: when (f2') {f2'.v = Fib.sequential(4)}
b5: when (f1, f2') {f1.v = f1.v + f2'.v}
```



(c) Behaviours and Ordering Legend

Figure 7.5: Solving `Fib.parallel(6)`

In Chapter 6 I investigated a axiomatic model for BoC; a model used for describing valid and invalid executions. Let us now use this model to take a look at a valid candidate execution of the program in Figure 7.6. This is, in fact, the only valid execution. I can see that the happens before edges enforce the ordering of behaviour starts and completes that lead to the correct result being in `f1` when $b6$ reads the result.

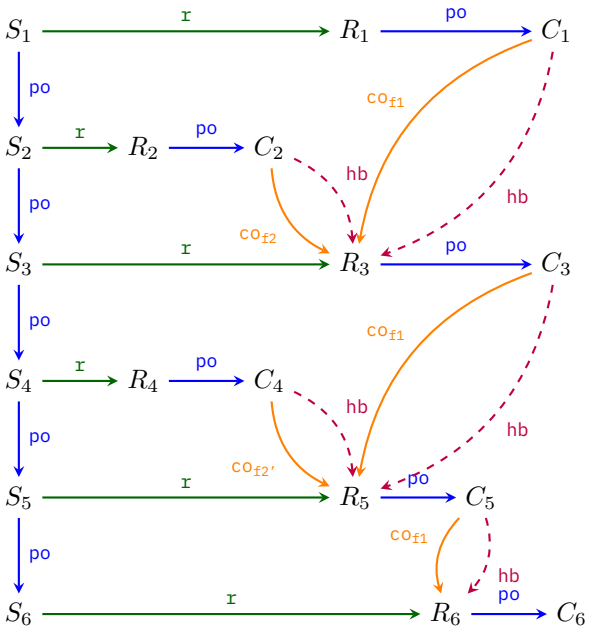


Figure 7.6: Candidate execution for `Fib.parallel(6)`

7.2.4 Channels

Channels are a means for asynchronous processes to communicate[52] and are the fundamental component of the programming language Go.[2] There are variants of channels with both blocking and non-blocking reads and writes; but in BoC, as there is no blocking, I build channels with non-blocking reads and writes. A snippet of the design is presented in Listing 7.7: a channel holds a queue of pending reads and a queue of pending writes; a read of that channel requires a callback that will either be serviced immediately, if there is a pending write, or be stored for later; there is very similar logic for writing but a value is required instead of a callback. This means that both reads and writes of channels are non-blocking.

Listing 7.7: Channel

```
1  class Channel[T] {
2    reads: Queue[Callback[Cell[T]]];
3    writes: Queue[Cell[T]];
4
5    create(): Channel[T] { ... }
6
7    read(channel: cown[Channel[T]],
8         callback: Callback[Cell[T]]) {
9      when (channel) {
10       match channel.writes.remove() {
11         var _: None =>
12           channel.reads.add(callback)
13         var value: Cell[T] =>
14           callback.apply(value);
15   } } }
```

[2] `https://go.dev/ref/spec#Channel_types`

```
16      // Similar logic for writes
17  }
```

7.2.5 Santa

The Santa problem is a challenge for coordinating three different kinds of entities from the concurrency and Chords literature[81, 56]. In summary: there is 1 santa, 9 reindeer and 10 elves; when santa and all 9 reindeer are ready they go and deliver presents, afterwards the reindeer go on holiday before being ready again; when santa and 3 elves are ready they meet in the workshop, afterwards the elves go and work before being ready again; if both reindeer and elves are ready then the reindeer take precedence. In Listings 7.8 to 7.11, I build a `Workshop` to solve the problem using BoC.

A workshop knows of santa, a list of reindeer, a list of waiting groups of reindeer and likewise for elves, see Listing 7.8.

Listing 7.8: Workshop fields

```
1   class Workshop {
2     santa: cown[Santa];
3
4     elves: cown[Queue[Elf]];
5     deer: cown[Queue[Deer]];
6
7     wait_elves: cown[Queue[Queue[Elf]]];
8     wait_deer: cown[Queue[Queue[Deer]]];
9
10    ...
```

Listing 7.9: Adding reindeer

```
1   add_deer(self, r: Deer) {
2     var rs = self.deer;
3     when (rs) {
4       rs.add(r);
5       if rs.length >= 9 {
6         var group = Workshop.take_deer(rs, 9);
7         var wait_deer = self.wait_deer;
8         when (wait_deer) {
9           wait_deer.add(group);
10          self.process();
11  } } } }
```

When entities, *i.e.,* a reindeer or an elf, are added to the workshop, in Listing 7.9, if the number exceeds the relevant threshold then entities are taken from the list to form a group. This group is then added to the appropriate list of waiting groups, and finally the workshop is notified to process this change.

The workshop processes this change, in Listing 7.10, by spawning a behaviour that requires santa and the two lists of groups, checks which of the two lists of groups have an entry (giving preference to reindeer), begins the appropriate task, and finally adds the used resources back to their appropriate list.

Listing 7.10: Processing a change

```
process(self) {
  var s = self.santa;
  var es = self.wait_elves;
  var rs = self.wait_deer;
  when (s, es, rs) {
    /* prioritise working with the reindeer */
    match rs.remove() {
      var d: Queue[Deer] => {
        print("Reindeer and Santa meet\n");
        // add each reindeer back to workshop
        self.return_deer(rs);
      }
      var _: None => {
        match es.remove() {
          var es: Queue[Elf] => {
            print("Elves and Santa meet\n");
            // add each elf back to workshop
            self.return_elves(es);
          }
          var _: None => { /* unreachable */ }
} } } } }
```

Listing 7.11: Creating a workshop

```
create() {
  // create fields
  var result = new Workshop;
  ...

  // make deeply immutable
  var workshop = freeze(result);

  //create the entities
  workshop.create_deer(9);
  workshop.create_elves(10);
}
```

Creating a workshop, in Listing 7.11, initialises the necessary fields and then makes the instance deeply immutable; this is a necessary step as the instance will be referenced in multiple behaviours. To set everything in motion I start creating reindeer and elves, adding them to the workshop as I go and causing groups to form.

The complete solution is long and repeats similar ideas for elves and reindeers; as such, I restricted the discussion to focus on the key aspects of a `Workshop`.

7.2.6 Boids

In this section I will look at the Boids program and show how I can implement the program using BoC. This poses at least two interesting points:

1. The state of an object is updated in response to the state of all other objects in the program. Thus, whilst the objects are encapsulated and updated asynchronously, they must be able to read the state of all other objects.
2. Behaviours require access to many cowns at once (far beyond the 1 or 2 cowns I have been using so far)

Boids is an artificial life program designed to model the flocking motion of birds or schools of fish[55]. Each boid has a position and velocity, and their basic motion is made up of three actions:

1. Separation: steer to avoid crowding local boids
2. Alignment: steer towards the average heading of local boids
3. Cohesion: steer to move toward the average position of local boids

Let us investigate, top-down, how I implement this program in BoC; Listing 7.12 presents the fundamental logic of the program. This program is derived from an available pseudo-code example. [3]

I implement each boid as a cown, *i.e.,* **cown**`[Boid]`; this allows a boid to directly read the position of any other boid by acquiring both boids in a behaviour. I also have a window object, *i.e.,* **cown**`[Window]`, which I have included for completeness; this object encapsulates the functionality of drawing to the users window.

[3] https://vergenet.net/ conrad/boids/pseudocode.html

The rules for computing the update to a boid imply three distinct rules which access all boids. Recall that whilst a behaviour is using a cown no other behaviour which requires that cown can be running; so I want to minimise the time a behaviour has access to all other boids. Thus, I split the boid update into two phases: a phase which computes some partial result for a boid from all other boids; and a phase which updates a boid based on this partial result.

The program evolves by launching an asynchronous behaviour which: allocates space for partial results; computes partial results for each boid through `compute_partial_results` (described in Listing 7.15); updates each boids position and velocity (described in Listing 7.16); and, finally, draws the updated state of the boids (not described here as it is bespoke logic for the drawing library).

Listing 7.12: Main loop of boids

```
void step(cown[Window] window, Array[cown[Boid]] boids) {
  when() {
    Array[cown[Result]] partial_results;
    /* initialise partial_results */

    compute_partial_results(partial_results, boids);
    update_boids(partial_results, boids);
    draw_boids(window, boids);

    step(window, boids);
  };
}
```

The `Boid` and `Result` are represented by structures of `Vector`, demonstrated in Listing 7.13 and Listing 7.14.

Listing 7.13: Boid datatype

```
struct Boid {
  Vector position;
  Vector velocity;
};
```

Listing 7.14: Result datatype

```
struct Result {
  Vector cohesion;
  Vector separation;
  Vector alignment;
};
```

To compute the partial result for a boid update, a behaviour is spawned that requires the particular partial result cown, and *all* of the other boids in the program. The partial results are then calculated in accordance with the rules are the beginning of this section. This means only one boid can be calculating its updates at a time.

Listing 7.15: Boids calculate update for local neighbours

```
void compute_partial_results(Array[cown[Result]] results, Array[cown[Boid]] boids) {
  for (int i = 0; i < results.size(); ++i) {
    when(var partial_result = partial_result, boids) {
      for (size_t j = 0; j < n; ++j) {
        /* collect cohesion, separation and alignment
            information of local boids */
      }
    };
  }
}
```

To update the boids, a behaviour is spawned which requires only the partial result and boid. These updates can be executing in parallel, so any work which does not need access to the global knowledge occurs here.

Listing 7.16: Update Boids

```
void update_boids(Array[cown[Result]] results, Array[cown[Boid]] boids) {
  for (int i = 0; i < results.size(); ++i) {
    when(var partial_result = results[i], var boid = boids[i]) {
      /* compute the boid update from the collected global information */
    };
  }
}
```

Note that this solution intrinsically relies on the behaviour ordering of BoC to ensure computations and updates to boids occur in the intended order. The behaviours which compute partial results will be scheduled before the behaviours that update the boids. Moreover, the behaviours from a "loop" of the program must complete before the behaviours of the next "loop" begin.

Thus, I achieve the boid program using BoC. The boids are updated in two phases in a lock-step solution. I could alternatively implement boids having their own program loops, grabbing all other known cowns to compute their own updates when they get scheduled. This could suffer from some boids being scheduled more frequently than others.

In Chapter 8, I will discuss further the idea of read-only access to a cown. Note that in this program the boids are only globally required in a read-only manner. This means it is safe to access them at the same time, and so the behaviours that compute partial results could be parallelised.

7.2.7 Binary Search Trees with hand-over-hand locking

Consider a binary search tree (BST) which can be operated on in parallel. In Listing 7.17 I provide a definition for a `node_t` with locks; a node consists of a mutex to lock the node, the data, and two pointers to the children. A, perhaps straightforward, "translation" for Boc appears Listing 7.18; there is no mutex as the node is expected to be placed inside a cown, thus the two child nodes are cowns. This data-type exposes a programming pattern which is ill-fitting for BoC. Namely, a programming pattern where parts of the data structure are accessed asynchronously, but operations are expected to behave as if performed synchronously. This is typically embodied by nesting cowns where the structure of the cowns is meaningful; this requires spawning more behaviours with a growing set of resources whilst trying to avoid interleaved operations. In this section, I will demonstrate why a BST should not be designed like this in BoC.

Listing 7.17: node type with locks

```
struct node_t {
  mutex m;
  int data;
  node_t *left;
  node_t *right;
};
```

Listing 7.18: node type for BoC

```
struct node_t {
  int data;
  optional[cown[node_t]] left;
  optional[cown[node_t]] right;
}
```

Let us consider one way I could write the insertion operation for BoC. In Listing 7.19 I start at some node and descend into the tree until I find the empty position where the new node lives; I create the node and link it into the tree. At each node, I acquire the cown and check its contents and access its children; I release the node when I move down into the tree. Assume that the insertion with locks is fairly similar.

Listing 7.19: Insert into a BST with cowns

```
static cown[node_t] insert(optional[cown_ptr[node_t]] node, int v) {
  if (!node) { // check if node has a value
    // if I am at an empty leaf, create a node
    // and return it
```

```
    return cown[node_t](v);
  } else {
    var n = *node; // access the value
    when(node) {
      // descend into the appropriate subtree
      if (v > node->data) {
        node->right = node_t::insert(node->right, v);
      } else if (v < node->data) {
        node->left = node_t::insert(node->left, v);
      }
    };
    return *node;
  }
}
```

Consecutive insertions behave as they do for the locking equivalent; if I write `tree_t::insert(tree, 2); tree_t::insert(tree, 20)` then the first insert will complete before the second insert. This happens due to the causal order of behaviours as I descend into the tree; the first behaviour and second behaviour will access the nodes in the same order. Moreover, if I have two inserts which insert in different directions at a node, they will access different subtrees and can thus be parallelised.

Problems arise when the consecutive operations do not follow the same path of accesses, such as a print followed by an insertion.

Let us consider how to achieve an in-order print of the nodes of the BoC BST. There are at least two distinct ways to achieve this:

1. Traverse the tree, acquiring the nodes and printing them as you go
2. Traverse the tree to collect all of the cowns, then spawn a behaviour to acquire them all and print the nodes

Let us first consider Item 1, presented in Listing 7.20: the operation initially creates an empty stack of nodes; at each node, if the node is non-empty, the node is pushed to the stack; then a behaviour is spawned which continues the print into the left subtree; once an empty leaf is reached, the last node is popped off the stack and a behaviour is spawned which prints the node and continues into the right subtree of that node.

Listing 7.20: Print nodes

```
static void print_impl(optional[cown[node_t]] node, stack[cown[node_t]] nodes) {
  if (node) {
    var n = *node;
    nodes.push_back(n);
    when(n) {
      node_t::print_impl(node->left, nodes);
    };
  } else if (nodes.size() > 0) {
    cown[node_t] n = nodes.back();
    nodes.pop_back();
    when(n) {
      print(n->data);
      node_t::print_impl(n-right, nodes);
    };
  }
}

static void print(optional[cown[node_t]] node) {
  node_t::print_impl(node, stack[cown[node_t]]());
}
```

Note that this print always goes left before going right. If I print a tree and then insert into the tree a value that requires going into the right subtree, I will see that the oder of operations can be interleaved in a way that does not align with the expected synchronous lock-like behaviour. Consider Listing 7.21, the shape of the tree constructed in the first 3 insertions means that `insert(t, 7)` will likely complete before the `print` makes its way into the right subtree from the root.

Listing 7.21: Non-determinstic tree operations

```
var t = cown(tree_t());
tree_t::insert(t, 5);
tree_t::insert(t, 3);
tree_t::insert(t, 6);

tree_t::print(t);
tree_t::insert(t, 7);
```

Let us return to Item 2, this approach will lock the entire tree, so can the same problem with interleaved operations happen? Yes, the tree has to be traversed to collect the nodes, which means that it is entirely possible the insertion executes between discovering the nodes and acquiring all of the nodes. This pattern is demonstrated in Listing 7.22; collecting a node and then its children is simple enough to achieve for one layer down, but when collecting an entire subtree I need to arbitrarily nest the behaviours. Moreover, each nested behaviour will execute at some point in the future, thus allowing for the potential operations to execute in the meantime and transform the tree.

Finally, whilst a behaviour in Listing 7.22 has acquired `node`, `l` and `r`, one cannot access the left node through `node->left`. At the point at which the behaviour runs, `l` and `node->left` may no longer point to the same cown; some other behaviour could have executed first and changed the structure. Thus, in these behaviours, I have access to the data but not the the structure.

Listing 7.22: Collect nodes

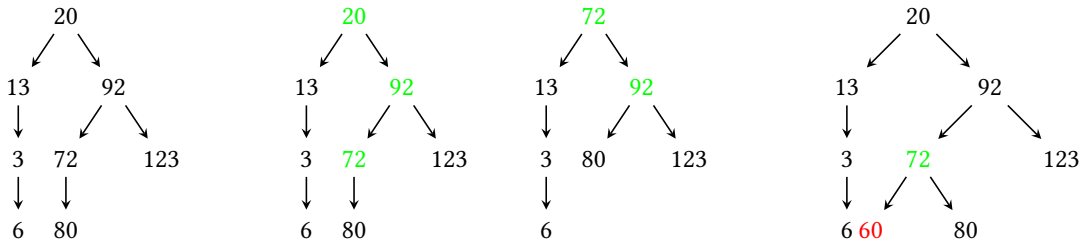
```
void collect_nodes(node: cown[node_t]) {
  when(node) {
    if (node->left && node->right) {
      when(node, var l = node->left, var r = node->right) {
        ...
      }
    }
  }
}
```

So far, the issues I have shown affect the expected outcomes of operations, but the integrity and invariants of the tree have always been preserved. When I introduce deletion into the tree, the integrity and invariants becomes harder to preserve.

The deletion of a node in a BST is a complicated task in general, and is particularly ill-suited for this tree design in BoC. Recall the deletion of a node in a binary search tree:

1. Find the node in the tree to be deleted
2. If the node has no children, delete the node
3. If the node has one child, replace the node with the child
4. If the node has two children, find the node with the minimum key in the right subtree (note the minimum node can have at most one subtree and it must be to the right)
   (a) If the minimum node has no children, replace the node to delete with the minimum node
   (b) If the minimum node has one right child, replace the node to delete with the minimum node and the minimum node with the right subtree

The case where the node has two children is demonstrated in Figures 7.7a to 7.7c; I delete the node with



(a) Tree (b) Nodes required for delete (c) Node deleted (d) Interleaved insertion

20 from the tree, replacing it with value 72 (the green nodes in Figure 7.7b highlight the nodes involved).

For locks the delete operation is only as complicated the above algorithm; it simply needs adapting to lock nodes as the operation descends into the tree, and unlocking the nodes when done. As long as all operations lock the nodes in the same way, other operations cannot be interleaved with delete operation.

For BoC, at some point I will need to be holding, at least, three nodes: the node to delete, the minimum node, and the parent of the minimum node. This means I will have to be releasing and acquiring nodes throughout the operation as I search for the correct nodes. I have to be very careful to account for an insertion part way through the operation. Figure 7.7d demonstrates a potential pitfall: the deletion finds that the node with 72 is the value to replace 20, however it must release 72 to grab 20 and 72; in the meantime an insertion is scheduled for 60 which becomes a child of 72; now the delete cannot replace 20 with 72 and must start again (further changing the oder of operations compared with the locking implementation).

In summary, the straightforward BoC parallel for a program using locks demonstrates clearly that cowns and behaviours are *not* just locks and threads. The data-structure I have designed in this chapter and the expectations of its use do not align well. I am accessing and mutating the structure asynchronously and expecting to obtain the same effect as if operations were performed synchronously.

In BoC: either the parallelism is important for this structure, in which case I ensure the integrity of the data-structure invariants, but allow for arbitrary ordering; or the order of operations is important in which case I should increase the granularity of the cown and place as much of the tree inside the cown as is important.

If one really wanted to construct the BST with the node data-structure I have been doing, this could be achieved with per node work queue (This would similar to selective receive actors/message queues). A node enters in to some operation and enqueues any work that comes in whilst the operation is ongoing; once the operation is finished and the node signalled, the node processes the next item in the work queue. Note the similarity between this and `ReasonableBanking`, for actors, where incoming transactions are stashed until they can be processed (Section 7.1.6).

7.2.8 Conclusion

This evaluation suggests the BoC paradigm is expressive enough to implement a variety of concurrency challenges in a convenient way. Some problems, such as the dining philosophers and boids, have some natural improvements with BoC; other problems, such as fibonacci and barriers, require some mental adjustment to adapt for the paradigm. This evaluation also highlights the novel program designs utilising behaviour ordering, in fibonacci and boids, and suggests BoC may have more avenues for interesting program design.

7.3 Qualitative: BoC in Rust

As a more practical investigation into whether BoC can be adopted by programming languages, I have built a rudimentary implementation of BoC in Rust[4]. This explores whether the Rust type system can provide the isolation guarantees that were set out in Chapter 5 and how BoC can be implemented in another language.

The following example, Listing 7.23, has the same semantics as it would if it were constructed in Verona, see Listing 1.1.

Listing 7.23: Behaviour-oriented concurrency in Rust

```
// transfer takes two references to cowns of accounts
fn transfer(acc1: &Cown<Account>, acc2: Cown<Account>, amount: u64) {
    // outside of a behaviour the state of the cown cannot be be modified.
    // when macro schedules a behaviour that requests the two accounts
    when!(acc1, acc2).run(move |acc1, acc2| {
        // inside of a behaviour, the state of the accounts can be modified
        acc1.withdraw(amount);
        acc2.deposit(amount);
    });
}
```

Let us discuss how to read this example; a `Cown` wraps a shared resource that is inaccessible outside of a behaviour, **`when`**`!` is a macro that accepts an arbitrary number of cowns and an anonymous function that accepts as many arguments, the function has mutable access to the state of the cowns.

Rust ensures that a cown always owns its state. Consider the example in Listing 7.24, it is an important requirement that this example fails to type check in Rust.

Listing 7.24: Ownership for Cowns

```
// Rusts type system prevents this, as such cowns are own their data
struct A<'a>(std::cell::RefCell<Option<&'a B>>);
struct B;

fn break_ownership(a: &Cown<A>, b: &Cown<B>) {
    when!(a, b).run(|a, b| {
        // If the following worked then ownership would be violated
        *a.0.borrow_mut() = Some(b);
    });
}
```

This example attempts to create a mutable reference to the contents of one cown and store it in another cown. If this type-checked then I would not be able to guarantee data race freedom using behaviour-oriented concurrency in Rust; the cowns would not own their data.

Ownership and traits prevent data races when programming in Rust; multiple threads cannot have a mutable reference to the same data at the same time. Rust uses traits `Send` and `Sync`; data that is `Send` allows transfer of ownership between threads and data that is `Sync` allows for access from multiple threads. Both ownership and traits are used to build channels for inter-process communication as well as locks and atomic reference counters. With these a programmer can safely share data and communicate in a concurrent program.

A thread that will run a behaviour becomes the owner of the required cowns, providing a mutable borrowed reference to the state of the cown to the behaviours body. For Rust to understand this, a `Cown` must be `Send` and its contents `Sync`. Building this implementation shows that BoC can be adopted by languages other than Verona.

[4] https://github.com/lukecheeseman/rust-cowns

7.4 Conclusion

This evaluation suggests BoC can be an effective and expressive paradigm, and that their are no obvious showstoppers for further research; BoC can be implemented such that it is comparably performant with actor model languages, and BoC can be used to implement a variety of concurrency challenges and idioms.

8

Further Work

In this chapter I will discuss further work that either crosses multiple chapters, such as read-only cowns, or does not have a clear chapter in which it belongs. I will conclude this chapter by recapping the further work that I have discussed in each chapter.

8.1 Verona: Reggio and BoC

The research programming language Verona introduced both a novel concurrency paradigm, BoC, and also a novel region based type system, Reggio[66] (being researched by Ellen Arvidsson, Elias Castegren, and Tobias Wrigstad at Uppsala University). This type system is inherently built for safe concurrency; however, as yet, the bridge between the type system and BoC has not been established.

Reggio organises objects in a program into a forest of isolated regions. A thread has a "window of mutability" - the single active region which a thread can mutate. Establishing that the type system can provide the isolation properties in Chapter 5, and constructing the bridge between the type system and the concurrency, is a pragmatic next step in the research for Verona.

8.2 Optimising programs and program equivalence

Modern compilers and runtimes employ many tactics in the effort to optimise programs; these tactics necessitate the ability to make meaning preserving transformations to a program. As such, we must understand what it means for two BoC programs to be "equivalent". In this section we will use examples to discuss program transformations, optimising runtime, and program equivalence (when these transformations should be applied is another question to be left for the future).

Recall from Chapter 5 that, for languages that fulfil the isolation properties, behaviours are isolated from one another. Furthermore, we saw that behaviours are linearizable between spawning behaviours.

This means we can apply existing tactics for sequential programs when considering equivalence and transformations between spawns; we can do this as we are dealing with, essentially, sequential programs.

Thus we need to consider how we can transform behaviour spawns and when these transformations result in equivalent programs; this includes merging, splitting, and transposing spawns. We will now look at examples to explore these scenarios. Consider the three examples which follow, Listings 8.1 to 8.3; we now briefly discuss some of the valid and invalid program transformations.

In Listing 8.1 we have two behaviours spawned one after another that acquire a single cown (the following observations still apply if the behaviours acquire multiple cowns).

If `c1` and `c2` do not alias then it is safe to swap the spawn order of $b1$ and $b2$ as there is no happens before order. Another transformation is to merge the two behaviours into a single behaviour that acquires `c1` and `c2`; this is indistinguishable from $b1$ and $b2$ happening in parallel or one after another in every execution, which is a valid execution.

If `c1` and `c2` do alias then the order of $b1$ and $b2$ cannot be swapped. However, the behaviours can still be merged as $b1$ followed immediately by $b2$ is a valid program execution. In fact, $b1$ could be elided altogether; based on the previous observation, if $b1$ is immediately followed by $b2$ then the effects of $b1$ are never observed.

In Listing 8.2 we introduce a function call for `g()` in $b1$. No matter what the function `g()` does, the observations we just made for Listing 8.1 still apply. There can be no happens before edges between $b1$, nor any behaviour it transitively spawns, and $b2$; this means we will not violate the order by moving and merging behaviours. Note that these transformations may introduce new happens before orders.

Conversely, in Listing 8.3 we introduce a function call for `g()` in $b2$. Now we cannot transform the program so liberally. The function `g()` could spawn some behaviour $b3$ which requires `c1` and thus must happen after $b1$; if we were to reorder spawning $b1$ and $b2$ then this required order would be violated. How this program can be transformed depends on what `g()` does.

Listing 8.1: No function calls

```
void f(c1: cown[U64],
       c2: cown[U64]) {
  when(c1) { /* b1 */
    c1 = 10;
  };
  when(c2) { /* b2 */
    c2 = 20;
  }
}
```

Listing 8.2: `b1` calls `g()`

```
void f(c1: cown[U64],
       c2: cown[U64]) {
  when(c1) { /* b1 */
    g(c1, c2);
    c1 = 10;
  };
  when(c2) { /* b2 */
    c2 = 20;
  }
}
```

Listing 8.3: `b2` calls `g()`

```
void f(c1: cown[U64],
       c2: cown[U64]) {
  when(c1) { /* b1 */
    c1 = 10;
  };
  when(c2) { /* b2 */
    g(c1, c2);
    c2 = 20;
  }
}
```

Listings 8.4 and 8.5 demonstrate an example where more potential parallelism can be obtained. If a long running behaviour requires multiple cowns but they are used in distinct phases of the behaviour, as in Listing 8.4 for example, then it is possible to spawn two separate behaviours instead; each behaviour acquires a separate cown and they can execute in parallel. Note that this requires that $b1$ does not spawn a behaviour requiring `c2`; if this is not respected and $b1$ spawns a behaviour, say $b3$, then $b3$ could run before $b2$.

Listing 8.4: Merged behaviours

```
when(c1, c2) {
  /* b1: a long computation involving c1 */
  /* b2: a long computation involving c2 */
}
```

Listing 8.5: Split behaviours

```
when(c1) { /* b1: a long computation involving c1 */ };
when(c2) { /* b2: a long computation involving c2 */ }
```

Finally, let us consider Listing 8.6 and the early release of cowns. This is an idea proposed by Marios Kogias but highlights a vital consideration when introducing new features; we must be vigilant of breaking the existing causal order in programs.[1]

A behaviour acquires `c1` and `c2`, operators on both in one phase, and then operates only on `c1` in the second phase; meanwhile $b2$ is pending waiting for `c2`. In this case `c2` is not in use but also cannot be acquired by another behaviour. Moreover, the programmer has no *simple* way of releasing the cown and continuing with the behaviour atomically. In such a case, it would be ideal if the behaviour could release `c2` in the second phase and allow $b3$ to run; we call this *early release*. This transformation could be programmer directed, through syntax, or automated as part of a program analysis step.

Either way there is a pitfall of which to be wary which is presented in Listing 8.7. For clarity I have introduced a `release()` function which releases the cown provided as argument, this cown cannot be read or written after the release. Under normal circumstances $b2$ happens before $b4$; without careful consideration for the implementation of early release this constraint could be violated. If $b1$ releases `c1` before spawning $b2$, then $b3$ can begin execution and spawn $b4$ before $b2$ is spawned, then $b4$ can execute before $b2$.

Listing 8.6: A candidate for early release

```
1 when(c1, c2) {
2   /* p1: a long computation involving c1 and c2 */
3   /* p2: a long computation involving c1 and not c2 */
4 }
5 when(c2) { /* b3 */ }
```

Listing 8.7: A potential pitfall for early release

```
1 when(c1, c2) { /* b1 */
2   release(c1);
3   when(c2) { /* b2 */ }
4 }
5 when(c1) { /* b3 */
6   when(c2) { /* b4 */}
7 }
```

In this section we have covered patterns where program transformations are possible. I have included the final point on Listing 8.7 to motivate why we need to research further and establish a principled definition of program equivalence in BoC.

8.3 Read-only cowns

In this section I will motivate read-only cowns in BoC and discuss how it can be designed.

Recall the discussion on the design of Boids from Section 7.2.6. During the `compute_partial_results` a behaviour reads the state of all other boids a writes data into a cown that only that behaviour requires (repeated below in Listing 8.8). As behaviours must be isolated from one another, the behaviours must take exclusive access to the boids even though the behaviours will not modify them. In principle, the behaviours could all execute *safely* in parallel, however the underlying language will not allow it; thus we miss out on lots of potential parallelism. The underlying language needs to be enriched with information that a cown is only going to be read in a behaviour and a means to use this information.

Listing 8.8: Boids calculate update for local neighbours

```
1 void compute_partial_results(Array[cown[Result]] results, Array[cown[Boid]] boids) {
2   for (int i = 0; i < results.size(); ++i) {
3     when(var partial_result = partial_result, boids) { ... }
4   }
5 }
```

Note the emphasis that it is the *underlying language* that is prohibiting the parallelism. The paradigm is more than capable of accommodating them after our adjustment in Section 4.2, specifically considering read-only cowns.

[1] https://github.com/microsoft/verona/pull/506

To include read-only cowns, the underlying language requires capabilities for cown access, *i.e.,* read and write. A behaviour can start executing as long as the accesses it requires do not conflict with the running accesses; two behaviours conflict if either behaviour requires write access to a cown required by the other behaviour.

The underlying language will need also syntax and type annotations to denote these capabilities. In Listings 8.9 and 8.10 we achieve this with a `read()` function which take an object of type **`cown`**`[T]` and returns an object of type **`cown`**`[const T]`; denoting that the cown is for reading only. Any normal cown object is writeable.

Listing 8.9: Read annotation

```
1 when(read(c1), c2) { /* b1 */ };
2 when(read(c1), c3) { /* b2 */ }
```

Listing 8.10: Read type annotation

```
1 void f(c1: cown[const U64]) {
2   when(c1) {/* b1 */ };
3   when(c1) {/* b2 */ }
4 }
```

I have submitted a patch to implement this in the Verona runtime; however, due to fundamental implementation changes to the verona runtime, the feature is currently disabled.[2]

Let us consider how this affects our understanding of "happens before". In Listing 8.11 $b1$ and $b2$ can execute in parallel; thus, unlike our previous examples, $b2$ and $b4$ do not have a happens before order.

Listing 8.11: Spawning in behaviours with read-only cowns

```
1 when(read(c1)) { /* b1 */ when(c2) { /* b2 */ } };
2 when(read(c1)) { /* b3 */ when(c2) { /* b4 */ } }
```

The axiomatic model in Chapter 6 has been constructed to accommodate for these different kinds of accesses, however it has not been utilised in practice.

8.4 Recapping further work

Here, I will briefly recap the further work I have outlined throughout this thesis.

Isolation: Linearization points In Chapter 6 I established an atomicish property for the isolated operational semantics; this property stated that we can reorder the steps of an execution involving a behaviour β, which doesn't SPAWN or END, such that we can deinterleave the steps of β and end up in the same resulting configuration.

Linearization points takes that idea further such that once a behaviour has started, all of the steps of a behaviour appear to happen instantaneously together with the next SPAWN or END. This would greatly simplify the reasoning of BoC programs whilst preserving the inherent parallelism.

Isolation: Cown update atomicity The atomicish property (Lemma 7) has to consider the effect that executing a behaviour has on the global state of the program. However, a behaviour spawning another behaviour does not affect the outcome of the spawning behaviour, and the updates to the cowns state can still be viewed as atomic. This work would establish another property the demonstrates that, from a cowns perspective, a behaviour completes atomically.

Axiomatic Model: Correspondence between operational semantics and axiomatic models In Chapter 6, I established an axiomatic model for BoC. One result of this chapter was to demonstrate that the axiomatic model was more permissive than the operational semantics from Chapter 4. I also introduced

[2] https://github.com/microsoft/verona/pull/595

an operational semantics which I claimed was as permissive as the axiomatic model. Ideally, I could demonstrate whether the operational semantics are sound and complete with respect to the axiomatic model; and, if the semantics are not, then being able to precisely demonstrate when they are not.

Beyond Behaviour Ordering In this section I explored the design space for promises. Established the desired use-cases and principles for promises in BoC is the required next step before they can be integrated. This will likely require constructing a large real-world example that adopts BoC.

9 Conclusion

In this thesis, I have introduced BoC, demonstrating that it provides a language with parallelism and *flexible* coordination though a single powerful abstraction. I have demonstrated that BoC ensures behaviours remain isolated throughout execution of a program; BoC provides desirable safety guarantees of data-race freedom, deadlock freedom and atomicity; and, BoC provides an ordering for concurrent behaviours which provides implicit parallelism.

I have introduced both an operational semantics and axiomatic model for reasoning about the execution of BoC programs; the first describes the emergent execution of a program and the second describes behaviour ordering and judges whether candidate executions are valid or invalid.

Also, I have shown the applicability of BoC through examples, challenges and comparative benchmarks. I have shown that BoC compares well with actors: it simplifies the design of many problems, while providing performance comparable with actors.

To reiterate the point made by Cheeseman et al. [1], the **`when`** construct is simple, intuitive, and expressive.

9.1 Reflections on BoC

BoC is an intuitive paradigm for many applications, and the evaluation demonstrates that BoC compares well with other approaches to concurrency.

There are programming patterns for which BoC is intended and seamlessly integrates; there are others where BoC functions effectively, but programmers must navigate and adapt to its idiosyncrasies; and finally, there are programming patterns where BoC is not intended or proves challenging to adapt. Let us reflect on some of these patterns.

9.1.1 Optimal BoC applications

I have shown that, BoC shines when we need to perform atomic operations over multiple resources, and when we need to access multiple resources at once.

Examples of these two patterns are the bank account transfers which should not be interleaved with other operations, and the boids program where an individual boid needs to read the state of all other boids to compute its next step.

9.1.2 Effective BoC applications

I discussed that BoC is effective when we need to perform coordinated operations on resources, but the operation does not need to atomically update multiple cowns. Many of the examples demonstrated in this thesis can be solved as elegantly using BoC as they can be with actors and promises; this includes examples santa, barrier synchronisation and dining philosophers.

We can sometimes improve on these situations by leveraging the ordering guarantees of BoC. This is possible when we spawn behaviours that must happen in sequence, accessing the same cowns, and when we know no interleaving behaviours can occur. Recall the fibonacci example from Section 7.2.3. In these scenarios we can avoid a completion signal from one behaviour to the next, or a continuation that is passed or nested within the behaviour (flattening the code which aids comprehension).

However, it is reasonable to say that this kind of programming requires adjustment; it is not typical to program by relying on the implicit order of behaviours.

9.1.3 Suboptimal BoC applications

I demonstrated, in Section 7.2.7, that BoC does not fit with the goal of building an asynchronous data-structure which must perform operations in a synchronous order. The example I presented was hand-over-hand locking of a tree structure.

In these scenarios, the programmer must revise what they are trying to achieve and decide whether parallelism or order are important.

9.1.4 Unexplored BoC applications

One of the more fundamental open questions around the design of BoC is whether the paradigm can be adapted to distributed programming. The actor model seamlessly extends to distributed programming, as shown by Erlang and Akka. Could BoC be used as the basis for distributed programming? Because of isolation of actor state, and because behaviours only access the state of the particular actor, there is no immediate need to migrate actors – unless you want to co-locate actors which communicate often. But for distributed BoC to be efficient, I would need this migration, so that the collection of cowns needed by a behaviour becomes co-located. This might lead to interesting patterns of migration where commonly used together cowns will reside on the same node. But I am a long way from making any claims in this space.

9.2 Recapping conclusions

Now I will briefly summarise the conclusions made in this thesis.

Operational Semantics I have demonstrated how an underlying programming language can be enriched with concurrency using BoC. By making BoC parametric with an abstract underlying programming language, I have demonstrated that the operational semantics for BoC are simple. Whilst simple, these

semantics are powerful, able to ensure deadlock freedom and provide the happens before ordering for behaviours.

Isolation I have demonstrated that BoC can ensure behaviours remain isolated from one another; this requires that the underlying programming language provides the isolation properties. Moreover, under these circumstances, BoC guarantees that behaviours are atomic between spawning behaviours.

Axiomatic Model I constructed an axiomatic model of BoC and demonstrated how to use it to judge executions of programs as valid or invalid. This axiomatic model abstracts away from the finer-grained details of an operational semantics, providing the flexibility to more concisely define the causal order of BoC and permitted executions.

Beyond Behaviour Ordering I have explained when and why to enrich the implicit ordering of BoC; promises offer this richer ordering, and I discussed a number of different designs.

Evaluation This evaluation suggests BoC can be an effective and expressive paradigm, and that their are no obvious showstoppers for further research; BoC can be implemented such that it is comparably performant with actor model languages, and BoC can be used to implement a variety of concurrency challenges and idioms.

Bibliography

[1] L. Cheeseman, M. J. Parkinson, S. Clebsch, M. Kogias, S. Drossopoulou, D. Chisnall, T. Wrigstad, and P. Liétar, "When concurrency matters: Behaviour-oriented concurrency," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA2, oct 2023. [Online]. Available: https://doi.org/10.1145/3622852

[2] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang, "Views: Compositional reasoning for concurrent programs," in *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 287–300. [Online]. Available: https://doi.org/10.1145/2429069.2429104

[3] J. Alglave, L. Maranget, and M. Tautschnig, "Herding cats: Modelling, simulation, testing, and data mining for weak memory," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 2, jul 2014. [Online]. Available: https://doi.org/10.1145/2627752

[4] "Project verona." [Online]. Available: https://github.com/microsoft/verona

[5] Microsoft, "Confidential consortium framework," 2020. [Online]. Available: https://github.com/microsoft/CCF/tree/v0.9.3

[6] A. D. Birrell, *An introduction to programming with threads*. Digital Systems Research Center, 1989.

[7] A. Turon, "Fearless concurrency with rust," 2015. [Online]. Available: https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html

[8] C. Boyapati, R. Lee, and M. Rinard, "Ownership types for safe programming: preventing data races and deadlocks," *SIGPLAN Not.*, vol. 37, no. 11, p. 211–230, nov 2002. [Online]. Available: https://doi.org/10.1145/583854.582440

[9] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III, "Software transactional memory for dynamic-sized data structures," in *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, 2003, pp. 92–101.

[10] N. Shavit and D. Touitou, "Software transactional memory," *Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997.

[11] A. Welc, B. Saha, and A.-R. Adl-Tabatabai, "Irrevocable transactions and their applications," in *SPAA*, 2008.

[12] T. Harris, S. Marlow, and S. Peyton Jones, "Composable Memory Transactions," in *PPoPP*. ACM Press, 2005, pp. 48–60.

[13] A. Matveev and N. Shavit, "Towards a fully pessimistic stm model," 2012.

[14] Y. Huang, W. Qian, E. Kohler, B. Liskov, and L. Shrira, "Opportunities for optimism in contended main-memory multicore transactions," *The VLDB Journal*, pp. 1–23, 2022.

[15] D. Qin, A. D. Brown, and A. Goel, "Caracal: Contention management with deterministic concurrency control," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 180–194. [Online]. Available: https://doi.org/10.1145/3477132.3483591

[16] D. J. Abadi and J. M. Faleiro, "An overview of deterministic database systems," *Communications of the ACM*, vol. 61, no. 9, pp. 78–88, 2018.

[17] E. Koskinen and M. Herlihy, "Checkpoints and continuations instead of nested transactions," in *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, 2008, pp. 160–168.

[18] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman, "Open nesting in software transactional memory," in *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2007, pp. 68–78.

[19] G. A. Agha, "Actors: A model of concurrent computation in distributed systems." Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab, Tech. Rep., 1985.

[20] C. Hewitt, P. Bishop, and R. Steiger, "Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence," in *Advance Papers of the Conference*, vol. 3. Stanford Research Institute Menlo Park, CA, 1973, p. 235.

[21] S. Clebsch, S. Drossopoulou, S. Blessing, and A. McNeil, "Deny capabilities for safe, fast actors," in *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, ser. AGERE! 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 1–12. [Online]. Available: https://doi.org/10.1145/2824815.2824816

[22] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Addison-wesley Reading, 1987, vol. 370.

[23] S. Clebsch, "'pony': co-designing a type system and a runtime," Ph.D. dissertation, Imperial College London, 2017.

[24] T. Pony Developers, "Causal message." [Online]. Available: https://www.ponylang.io/faq/runtime/

[25] T. Van Cutsem, E. Gonzalez Boix, C. Scholliers, A. Lombide Carreton, D. Harnie, K. Pinte, and W. De Meuter, "Ambienttalk: programming responsive mobile peer-to-peer applications with actors," *Computer Languages, Systems & Structures*, vol. 40, no. 3, pp. 112–136, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1477842414000335

[26] J. Högberg, "A few notes on message passing," 2021. [Online]. Available: https://www.erlang.org/blog/message-passing/

[27] C. Varela and G. Agha, "Programming dynamically reconfigurable open systems with salsa," *ACM SIGPLAN Notices*, vol. 36, no. 12, pp. 20–34, 2001.

[28] P. A. Bernstein, "Actor-oriented database systems," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 2018, pp. 13–14.

[29] P. Kraft, F. Kazhamiaka, P. Bailis, and M. Zaharia, "Data-Parallel actors: A programming model for scalable query serving systems," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022, pp. 1059–1074. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/kraft

[30] H. Plociniczak and S. Eisenbach, "Jerlang: Erlang with joins," in *International Conference on Coordination Languages and Models*. Springer, 2010, pp. 61–75.

[31] B. Sang, G. Petri, M. S. Ardekani, S. Ravi, and P. Eugster, "Programming scalable cloud services with AEON," in *Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016*. ACM, 2016, p. 16. [Online]. Available: http://dl.acm.org/citation.cfm?id=2988352

[32] J. De Koster, T. Van Cutsem, and T. D'Hondt, "Domains: safe sharing among actors," in *Proceedings of the 2nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions*, ser. AGERE! 2012. New York, NY, USA: Association for Computing Machinery, 2012, p. 11–22. [Online]. Available: https://doi.org/10.1145/2414639.2414644

[33] J. De Koster, "Domains: Language abstractions for controlling shared mutable state in actor systems," Ph.D. dissertation, PhD thesis, Vrije Universiteit Brussel, 2015.

[34] B. Sang, P. Eugster, G. Petri, S. Ravi, and P.-L. Roman, "Scalable and serializable networked multi-actor programming," *Proc. ACM Program. Lang.*, nov 2020.

[35] J. Field and C. Varela, "Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments," in *POPL*, 2005.

[36] J. Swalens, J. D. Koster, and W. D. Meuter, "Chocola: Composable concurrency language," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 42, no. 4, pp. 1–56, 2021.

[37] G. Berry and G. Boudol, "The chemical abstract machine," in *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '90. New York, NY, USA: Association for Computing Machinery, 1989, p. 81–94. [Online]. Available: https://doi.org/10.1145/96709.96717

[38] C. Fournet and G. Gonthier, "The reflexive cham and the join-calculus," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1996, pp. 372–385.

[39] ——, "The join calculus: A language for distributed mobile programming," in *International Summer School on Applied Semantics*. Springer, 2000, pp. 268–332.

[40] S. Conchon and F. Le Fessant, "Jocaml: mobile agents for objective-caml," in *Proceedings. First and Third International Symposium on Agent Systems Applications, and Mobile Agents*, 1999, pp. 22–29.

[41] N. Benton, L. Cardelli, and C. Fournet, "Modern concurrency abstractions for c#," *ACM Trans. Program. Lang. Syst.*, vol. 26, no. 5, p. 769–804, sep 2004. [Online]. Available: https://doi.org/10.1145/1018203.1018205

[42] L. Cheeseman, "Boc examples," 2023. [Online]. Available: https://github.com/ic-slurp/boc-examples

[43] P. W. O'Hearn, "Resources, concurrency and local reasoning," in *CONCUR 2004 - Concurrency Theory*, P. Gardner and N. Yoshida, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 49–67.

[44] H. Yang and P. O'Hearn, "A semantic basis for local reasoning," in *Foundations of Software Science and Computation Structures*, M. Nielsen and U. Engberg, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 402–416.

[45] Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, 1979.

[46] S. Owens, S. Sarkar, and P. Sewell, "A better x86 memory model: x86-tso," in *Theorem Proving in Higher Order Logics*, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 391–407.

[47] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, p. 463–492, jul 1990. [Online]. Available: https://doi.org/10.1145/78969.78972

[48] R. Milner, *An algebraic definition of simulation between programs*. Citeseer, 1971.

[49] ——, *Communication and concurrency*. Prentice hall New York etc., 1989, vol. 84.

[50] D. Park, "Concurrency and automata on infinite sequences," in *Theoretical Computer Science*, P. Deussen, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1981, pp. 167–183.

[51] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[52] ——, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, p. 666–677, aug 1978. [Online]. Available: https://doi.org/10.1145/359576.359585

[53] J. Magee and J. Kramer, *State models and java programs*. wiley Hoboken, 1999.

[54] B. Karp, "Two-phase commit," 2011. [Online]. Available: http://www0.cs.ucl.ac.uk/staff/B.Karp/gz03/f2011/lectures/gz03-lecture6-2PC.pdf

[55] C. W. Reynolds, *Flocks, herds, and schools: a distributed behavioral model*. New York, NY, USA: Association for Computing Machinery, 1998, p. 273–282. [Online]. Available: https://doi.org/10.1145/280811.281008

[56] N. Benton, "Jingle bells: Solving the santa claus problem in polyphonic C#," Microsoft Research, Tech. Rep., 2003.

[57] D. Hensgen, R. Finkel, and U. Manber, "Two algorithms for barrier synchronization," *International Journal of Parallel Programming*, vol. 17, no. 1, pp. 1–17, 1988.

[58] S. M. Imam and V. Sarkar, "Savina - an actor benchmark suite: Enabling empirical evaluation of actor libraries," in *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, ser. AGERE! '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 67–80. [Online]. Available: https://doi.org/10.1145/2687357.2687368

[59] S. Blessing, K. Fernandez-Reyes, A. M. Yang, S. Drossopoulou, and T. Wrigstad, "Run, actor, run: towards cross-actor language benchmarking," in *Proceedings of the 9th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, ser. AGERE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 41–50. [Online]. Available: https://doi.org/10.1145/3358499.3361224

[60] "References and borrowing." [Online]. Available: https://doc.rust-lang.org/1.9.0/book/references-and-borrowing.html

[61] "Validating references with lifetimes." [Online]. Available: https://doc.rust-lang.org/1.9.0/book/lifetimes.htmll

[62] D. Clarke and T. Wrigstad, "External uniqueness is unique enough," in *ECOOP 2003 – Object-Oriented Programming*, L. Cardelli, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 176–200.

[63] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy, "Uniqueness and reference immutability for safe parallelism," *SIGPLAN Not.*, vol. 47, no. 10, p. 21–40, oct 2012. [Online]. Available: https://doi.org/10.1145/2398857.2384619

[64] S. Clebsch, S. Drossopoulou, S. Blessing, and A. McNeil, "Deny capabilities for safe, fast, actors," in *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, 2015, pp. 1–10.

[65] J. Noble, D. Clarke, and J. Potter, "Object ownership for dynamic alias protection," in *Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 32*, 1999, pp. 176–187.

[66] E. Arvidsson, E. Castegren, S. Clebsch, S. Drossopoulou, J. Noble, M. J. Parkinson, and T. Wrigstad, "Reference Capabilities for Flexible Memory Management," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, 2023.

[67] J. H. Siddiqui, A. Rauf, and M. A. Ghafoor, "Chapter two - advances in software model checking," ser. Advances in Computers, A. M. Memon, Ed. Elsevier, 2018, vol. 108, pp. 59–89. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0065245817300505

[68] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," *SIGPLAN Not.*, vol. 40, no. 1, p. 110–121, jan 2005. [Online]. Available: https://doi.org/10.1145/1047659.1040315

[69] B. C. Pierce, *Types and programming languages*. MIT press, 2002.

[70] A. Hammond, Z. Liu, T. Pérami, P. Sewell, L. Birkedal, and J. Pichon-Pharabod, "An axiomatic basis for computer programming on the relaxed arm-a architecture: The axsl logic," *Proc. ACM Program. Lang.*, vol. 8, no. POPL, jan 2024. [Online]. Available: https://doi.org/10.1145/3632863

[71] "Verona Github Repo." [Online]. Available: https://github.com/microsoft/verona-rt

[72] L. Cheeseman, "Boc benchmarks," 2023. [Online]. Available: https://github.com/ic-slurp/verona-benchmarks

[73] S. Blessing, "Pony savina," 2023. [Online]. Available: https://github.com/sblessing/pony-savina/tree/boc

[74] D. PonySite, "Pony Github Repo." [Online]. Available: https://github.com/ponylang/

[75] P. Haller and M. Odersky, "Scala actors: Unifying thread-based and event-based programming," *Theoretical Computer Science*, vol. 410, no. 2, pp. 202–220, 2009, distributed Computing Techniques. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397508006695

[76] D. AkkaSite, "Akka repo." [Online]. Available: https://akka.io/docs/

[77] R. Hiesgen, D. Charousset, and T. C. Schmidt, "Reconsidering reliability in distributed actor systems," in *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, ser. SPLASH Companion 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 31–32. [Online]. Available: https://doi.org/10.1145/2984043.2989218

[78] D. CAFSite, "Caf repo." [Online]. Available: https://www.actor-framework.org

[79] L. Cheeseman, "Reasonablebanking," 2023. [Online]. Available: https://github.com/sblessing/pony-savina/blob/boc/savina-pony/concurrency/banking2pc/banking.pony

[80] "Transactors." [Online]. Available: https://doc.akka.io/docs/akka/2.2/scala/transactors.html

[81] J. A. Trono, "A new exercise in concurrency," *ACM SIGCSE Bulletin*, vol. 26, no. 3, pp. 8–10, 1994.

[82] H. C. Baker and C. Hewitt, "The incremental garbage collection of processes," *SIGART Bull.*, no. 64, p. 55–59, aug 1977. [Online]. Available: https://doi.org/10.1145/872736.806932

[83] E. Ábrahám, I. Grabe, A. Grüner, and M. Steffen, "Behavioral interface description of an object-oriented language with futures and promises," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 7, pp. 491–518, 2009, the 19th Nordic Workshop on Programming Theory (NWPT 2007). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1567832609000022

[84] P. Haller, A. Prokopec, H. Miller, V. Klang, R. Kuhn, and V. Jovanovic, "Futures and promises," 2012. [Online]. Available: http://docs.scala-lang.org/overviews/core/futures.html

[85] "Promises/a+." [Online]. Available: https://promisesaplus.com/

[86] K. Fernandez-Reyes, D. Clarke, L. Henrio, E. B. Johnsen, and T. Wrigstad, "Godot: All the Benefits of Implicit and Explicit Futures," in *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), A. F. Donaldson, Ed., vol. 134. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, pp. 2:1–2:28. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2019.2

[87] M. S. Miller, E. D. Tribble, and J. Shapiro, "Concurrency among strangers," in *Trustworthy Global Computing*, R. De Nicola and D. Sangiorgi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 195–229.

[88] "Async/await." [Online]. Available: https://javascript.info/async-await

Appendix A

Promises

In this appendix, I explore promises in BoC. This chapter appears as an appendix chapter as I currently do not have clear use-case scenarios to aid my design goals. Thus, this chapter is largely exploratory and poses different questions and designs for promises in BoC. I do not investigate too deeply how these designs integrate with my operational semantics and axiomatic models.

Much like JavaScript, BoC programs are, in essence, constructed from many asynchronous callbacks. This means BoC programs are exposed to the same ergonomic issues from which JavaScript programs suffer.

These issues include deeply-nested callback pyramids, known as callback hell, and complicated communication patterns when asynchronous behaviours must pass results to subsequent behaviours.

I want to alleviate programmers from these ergonomic issues. I take inspiration from JavaScript, and many other asynchronous languages, and investigate promises. I demonstrate that promises can be adopted as library feature, similar to JavaScript. I also demonstrate that the intersection of cowns and promises provides an interesting design discussion; there exists the potential to unify the two concepts.

A.1 Background

In this section we will look at promises, predominantly as they appear in Javascript, and how they benefit asynchronous programming.

Returning a value from a synchronous function is straightforward. Execution transfers from the call-site of a function to the start of the function, computes a value, and then execution continues from the call-site of the function with the computed value.

Returning a value from an asynchronous function is not straightforward. The unit of execution which called the function, will continue to execute immediately and not wait for the function to complete or even start. Moreover, one does not even know when the asynchronous function will be executed or have some value to return.

Prior to Javascript ES6, the way to "return" a value from an asynchronous function was by passing a callback to the asynchronous function.[1] Listing A.1 demonstrates nesting callback functions to create a data processing pipelines

Listing A.1: Callbacks used in Javascript

```
1 // Three functions which act as the stages of a data pipeline
2 // Each asynchronously waits for a timeout to expire before
```

[1] https://www.w3schools.com/js/js_es6.asp#mark_promise

```
3  // completing their work and calling the next stage if necessary.
4  function fetchData(next) {
5    setTimeout(() => {
6      next(Date.now() % 10);
7    }, 100);
8  }
9
10 function processData(data, next) {
11   setTimeout(() => {
12     next(data * 10);
13   }, 100);
14 }
15
16 function renderData(data) {
17   setTimeout(() => {
18     console.log(data);
19   }, 100);
20 }
21
22 // Build the data processing pipeline by nesting callbacks
23 const executeDataPipeline = () => {
24   fetchData((data) => {
25     processData(data, (data) => {
26       renderData(data);
27     })
28   });
29 };
30
31 executeDataPipeline();
```

This style of programming has serious ergonomic issues. Stacking callbacks one inside another leads to a deeply nested pyramid-like structure known as "callback hell" which is fragile, and can quickly become unmanageable and confusing. We can see this beginning to occur in Lines 23–29, this would have been even worse had we not separated each stage into standalone functions.

Another complication to using callbacks to “return” a value is when multiple “return” values are required to launch another task. Consider the case when the data pipeline in Listing A.1 collects data from multiple inputs and processes them together. If we want to parallelise these asynchronous fetch stages, then we need some data-race safe method to ensure all fetch stages have completed before launching the next stage.

The introduction of promises has helped to alleviate JavaScript programmers from these ergonomic issues.[2]

Futures and promises are a common programming feature that enables asynchronous computation and communication. There is some inconsistency in the use, and naming, of futures and promises in the literature. The term future, as a name for the result of some delayed computation, can be traced to [82]. Hewitt and Baker used futures to run multiple computations in parallel, select the first computation to complete and free processing resources by garbage collecting non-required computations.

In this thesis we will adopt the definitions as presented by Ábrahám et al. [83], Haller et al. [84]. A future represents a result which is, potentially, yet to be computed; it is a read-only reference to a result of some computation that might not yet have finished. A process can dispatch some asynchronous task to compute a value whilst performing some other synchronous task, only waiting on the result when is necessary. A promise is a generalisation of this idea; it is a writeable reference to a result yet to be computed, decoupling the creation of a promise and the creation of the computation that will fulfil the promise.

We will study promises as they appear in Javascript. An open standard for JavaScript promises provides a

[2] https://www.geeksforgeeks.org/what-to-understand-callback-and-callback-hell-in-javascript/

design for implementers[85]. A promise constructor takes an executor function which takes two parameters, one to resolve the promise and one to reject the promise. This executor function runs *synchronously* but may *launch* asynchronous functions which resolve or reject the promise. Callbacks to handle the resolve or rejected promise can be registered on the promise object using `then`. These callbacks will not execute until the promise has been resolved or rejected. Once the value is ready, the callbacks will asynchronously execute.

There is also a "reject" mechanism which allows a reader of the promise to differentiate between success and unsuccessful promise results. We will not consider this reject mechanism in this chapter.

Consider the example in Listing A.2; this is a revision of Listing A.1 to introduce promises.

Listing A.2: Promises as in Javascript

```
1  // The stages return a promise which is resolved when they are complete
2  function fetchData() {
3    return new Promise((resolve, reject) => {
4      setTimeout(() => {
5        resolve(Date.now() % 10);
6      }, 100);
7    });
8  }
9
10 function processData(data) {
11   return new Promise((resolve, reject) => {
12     setTimeout(() => {
13       resolve(data * 10);
14     }, 100);
15   });
16 }
17
18 function renderData(data) {
19   return new Promise((resolve, reject) => {
20     setTimeout(() => {
21       console.log(data);
22     }, 100);
23   });
24 }
25
26
27 // Chain promises together to create the data pipeline
28 const executeDataPipeline = () => {
29   fetchData()
30     .then((data) => processData(data))
31     .then((data) => renderData(data));
32 };
33
34 executeDataPipeline();
```

Each stage returns a promise for a value, *e.g.*, Line 3, this enables chaining the stages of the data pipeline, registering the next stage to start once the promised value has been resolved. These promises help to flatten the callback structure as we can see in Lines 29–31. Note that promises in Javascript are not a de facto fix for callback hell, the promises still require a callback and so it is still possible to end up in a similar deeply nested stack.

We can also wait for the result of multiple promises using the `Promise.all()` constructor for promises in JavaScript.

[86] discusses the trade-offs between explicit and implicit futures. Explicit futures require make the

distinction between a type and a future of a type, and one cannot be appear in place of another; this means a programmer is always aware whether a value has been computed or not. Implicit futures blur the line between a value and future, and the future of a type can be use in place for a value of that type. This means a function may have to wait for a value to be computed, or a new implicit promise can be created which will be fulfilled when the initial promise is fulfilled, this creates a pipeline.

These pipelined promises are one way to avoid callback hell. We need only create a callback of point at which to wait when we actually need the value, say when we want to display a value or write it to disk, all intermediate steps can be substituted for promises that will propagate as they complete.

In the language E, promises represent the pending delivery of a value [87]. The language is used to write programs that communicate across networks, and so promises are able to amortise some of the round-trip time by substituting values for eventually delivered values. Moreover, the promises can be pipelined and sent across the network with the initial request message, such that the intermediate values do not need to be sent back across the network. This pipelining is able to amortise much of the latency that comes from network communication.

A.2 Promises in BoC

BoC ensures the ordering of behaviours which the programmer can leverage to achieve data flow, but the programmer will sometimes want to enrich this to guarantee outcomes beyond what BoC can guarantee. One such scenario is provided in Listing A.3. Assume we have some channel data type, reading from a channel takes some callback function that will be called when a value is available, and writing to a channel fulfils a pending read if one exists and otherwise enqueues the value. In this example, it is clear that the read must happen after the write, but $b2$ and $b4$ can execute in either order. So, the programmer needs to either build this logic into the channel, or we can introduce a new general purpose data structure to achieve such ordering, *i.e.,* a promise, and a channel can be built upon this.

Listing A.3: Ordering reads and writes of a channel

```
1 when() { /* b1 */ when(channel) { /* b2 */ channel.read((v) => { ... } ); } }
2 when() { /* b3 */ when(channel) { /* b4 */ channel.write(10); } }
```

In Section 7.2, when I evaluate BoC, we will see that there are often occasions when we need more ordering that the implicit behaviour ordering. These include building barrier synchronisation primitives, solving some divide and conquer style computations, and, as we have alluded to, constructing channels. Thus, it is pragmatic to investigate how we can provide the programmer with even more order.

Why do we want promises? We have seen futures and promises in the literature and other paradigms in Appendix A.1, and illustrated their desirability in asynchronous programming.

Promises are widespread in asynchronous programming, alleviating programmers from "callback hell" by providing an intuitive method for constructing asynchronous programs. They signify the eventual success or failure of some event, internal or external to the program.

Promises allows programmers to flow data between units of asynchronous execution, ensuring that an execution unit does not start until the data that it requires has been constructed. This could be used to enrich the implicit ordering of BoC, enabling a programmer to construct complex ordering of behaviours where the implicit ordering is not enough, or to ensure that no intervening behaviour accesses the data.

The design of a promise I want a design of promises that has similar characteristics to how they appear in Javascript (see Appendix A.1).

I choose promises like in Javascript and not those like in C++ intentionally. In C++, the retrieval of a promises value is a blocking operation [3]. I do not want to introduce synchronous blocking calls to BoC.

To summarise I want the foundational design of a promise to be the following:

- A promise object represents the eventual success or failure of a computation.
- The object can be either pending, fulfilled, or rejected.
- Callbacks can be registerd on the promise object.
- When the promise object is fulfiled, then the registers callbacks (and any later register callbacks) will be asynchronously resolved.

To reduce the scope of this investigation, I will not consider the failure of a computation nor the rejected state of a promise.

A myriad of design questions arise from this seemingly innocuous design outline.

- Are we seeking functionality beyond what we can achieve with cowns and behaviours?
- Do we want to provide programs with a single powerful concept? or multiple simple concepts?
- If we have the choice, should we build promises as a language feature or a supporting library?
- Are we willing to weaken our claim on deadlock freedom?
- Do we need to enrich our notion of causality in the presence of a new explicit ordering?
- Can a promise be written to multiple times?
- Is the read value of a promise mutable or immutable?

In this chapter, we look at variations of how we can design promises and outline the trade-offs that we are making. To provide a final design for promises would require testing it against a large collection of examples which are representative of the problems I am trying to solve; and, programmers who can independently assess the design. At the time of writing I do not have either of these; so, we will explore all of the above questions to provide guiding feedback for the future.

Intrinisic promises or library promises We can broadly divide the design space in two, constructing promises as either an intrinsic paradigm feature or as a supporting library.

Introducing new language features allows a program language designer to either introduce *new* functionality to the programmer, or present existing functionality in a refined format. Meanwhile, building libraries for common programming patterns from the existing paradigm means we can rely on existing results.

Construction as a paradigm feature provides more design choices we can look into. Should cowns and promises be presented as a unified or separate concepts? We could provide a single powerful concept by extending the states in which a cown can be. However, now the programmer must understand something more complex and be aware of any pitfalls. We could, instead, provide two simple concepts; this would simplify the programmers understanding of each concept but may create less desirable programming patterns when managing both cowns and promises together.

Construction as a library makes us question whether we can provide programmers with a solution which is both elegant and powerful enough?

Let us now look into these questions in detail.

A.3 Promises as a library

Building promises as a library means we must build the structure out of existing features. As long as everything we want from promises is achievable by existing features, then this is possible. An open standard for JavaScript promises for implementers provides a starting point[85]. A `then` method registers callbacks

[3] https://en.cppreference.com/w/cpp/thread/promise

to receive a promise's value or reason for why it could not be fulfilled. A similar design is achievable in BoC and we build this promise in Listing A.4.

```
class Promise[T] {
  // The inner pair of a queue of callbacks
  // and a value that may or may not exist.
  // Initialised to an empty Queue and Optional
  class Inner {
    queue: Queue[T -> ()];
    val: Optional[T];
  };

  // The inner cown is the actual promise.
  inner: cown[Inner];

  // Append a callback to the queue
  // and if there is a value
  // attempt to resolve the callback
  void then(f: T -> ()) {
    when(inner) {
      inner.queue.add(f);
      if (inner.val)
        resolve();
    }
  }

  // Write a value into the Optional[T]
  // and fulfill any waiting callbacks
  void fulfill(v: T) {
    when(inner) {
      if(!inner.val) {
        inner.val = v
        resolve();
      }
    }
  }

  // resolve a pending queue item
  // and spawn a behaviour which
  // attempts to resolve another
  void resolve() {
    when(inner) {
      if (inner.val) {
        inner.queue.pop()(inner.val);
        resolve();
      }
    }
  }
}
```

Listing A.4: Promises through cowns

The promise, or some copy, will be accessed by multiple behaviours at once, thus we need to use a cown to coordinate the access, *i.e.,* the `inner`. In essence, the promise is a wrapper around a cown of a queue of callbacks, and a value that may or may not have been produced. Listing A.5 demonstrates use of this library. Here, we are going to assume that values can be implicitly copied and passed to behaviours. A promise is created and copied into two behaviours; recall that under-the-hood the copies will point to the same cown. $b1$ registers a callback on on the promise and $b2$ fulfills the promise (enabling the registered callback). Note the similarity to our motivating example Listing A.3.

Listing A.5: Using a library promise

```
var p = Promise[int]();
when() { /* b1 */ p.then((v: int) -> { ... }) }
when() { /* b2 */ p.fulfill(10); }
```

In this design, we still have all of the properties of atomicity and deadlock freedom that we have seen in previous chapters.

Yet we have to ask, if we do not have deadlocks, then how do we describe the outcomes of Listings A.6 and A.7?

Listing A.6: Callbacks that will never run

```
p1 = Promise[int];
p2 = Promise[int];
p1.then((x: int) -> { p2.fulfill(x) });
p2.then((x: int) -> { p1.fulfill(x) });
```

Listing A.7: Callbacks that may never run

```
// c is a cown that holds:
// struct { done: boolean; promise: Promise[int] }

when() { /* b1 */
  when(c) { c.done = true; }
}

when() { /* b2 */
  when(c) {
    if (c.done) c.promise.fulfill(0)
  }
}
```

In Listing A.6, a callback is registered on `p1` that will fulfill `p2`, and, likewise, a callback is registered on `p2` that will fulfill `p1`. So, as both promises require the other promise to be fulfilled to execute the registered callbacks which fulfill them, neither promise will ever be fulfilled. This means that no behaviours will be spawned to read the value of the `inner` cown. In Listing A.7, the order in which $b1$ and $b2$ run affects whether the promise in `c` is ever fulfilled.

Importantly, these types of programs were already constructable in BoC, and we did not consider them deadlocks then as they cannot create a scenario where progress cannot be made. Any behaviour that is spawned will eventually execute. These are programmer errors, errors that will mean code does not execute, and may incur potential memory leaks. Yet, not an issue with BoC.

A.4 Promises as a paradigm feature

If we want to achieve more than what is available in the existing paradigm, or we want to refine the experience of programming with promises, then we need to alter the paradigm. We will explore two avenues: the first is to redesign a cown as a more powerful construct with more state, so as to build a promise-like cown; the second is separate cowns and promises and build new promise-specific features.

A.4.1 Unifying cowns and promises

We can consider a promise to be a cown that does not yet have a value. When we create these promises, we get two objects: an *empty* cown which behaviours, instead of callbacks, can be registered on; and, an object that allows writing to the promise.

Listing A.8: Cowns as promises

```
// creating a promise creates
// an empty cown and writeable object
pr, pw = Promise[int]();
// when pw is written this behaviour can start
when(pr) { /* b1 */ };
// a promise and a cown can be taken together
when(c, pr) { /* b2 */ };
when() { /* b3 */ pw.fulfill(10); };
when(c) { /* b4 */}
```

As promises are just cowns, we use **when** to register behaviours on promises, we can see this for behaviour $b1$ in Listing A.8. The difference now is that those behaviours will not start until a value is written to the promise. Once the promise has a value, behaviours are scheduled as they would be for any regular cown. For now, assume the promise can only be written to once; to allow multiple writes would open up the possibility for race conditions. Moreover, cowns and promises can be taken together as for $b2$, which means that $b2$ can run once c and pr are both available and have a value. The desired behaviour ordering is the same as it was in Chapter 6, thus, $b1$ happens before $b2$ and $b2$ happens before $b4$.

Note, we could not easily achieve this example with a library. To achieve that would require something like that which appears in Listing A.9. The behaviours $b1$ and $b3$ are simple enough, we register a callback for the promise that will asynchronously run when the promise is fulfilled (note this means we do not need a **when** for $b1$ as the callback will run asynchronously and the behaviour does not require cowns). However, $b2$ has been be transformed into a lambda function which spawns a behaviour that captures, via copy, the result of the promise. If we want multiple promises then we need to join their results somehow. This example also makes some assumptions about the design of the library, namely that callbacks are ordered in the same way as behaviours are ordered, so as to achieve that $b1$ happens before $b2$ (of course, this does not have to be a requirement of library). Yet, the ordering between $b2$ and $b4$ is completely lost. We could move the spawn of $b4$ inside of the callback on Line 4, but this begins a snowball effect of continuation passing style transformation, moving the remaining code inside of the callback of the promise. If $b3$ also spawned a behaviour that used c, then we would be in a difficult situation. If me moved the spawn of $b3$ in side of $b2$, then $b2$ would need to execute to enable itself. Yet, this is also true of Listing A.8!

Listing A.9: Recreating Listing A.8 with a library

```
1 var p = Promise[int]();
2 p.then((x: int) -> { /* b1 */ });
3 p.then((x: int) -> {
4   when(c) { /* b2 captures and uses x */ }
5 });
6 when() { /* b3 */ p.fulfill(10); }
7 when(c) { /* b4 */ }
```

We have changed the paradigm and so we cannot rely on old conclusions. We have to reassess whether BoC is deadlock free. Recall the definition of deadlock free from Chapter 4.

Lemma 2 (Deadlock Free)**.**

$$\forall P, R, h.\ [\ (\exists P', R', h'.\ P, R, h \rightsquigarrow P', R' h') \quad \vee \quad P \cup R = \varnothing\ ]$$

Putting aside any adaptations we would need to make for a new semantics and well-formedness condition, this lemma states the global configuration can make progress, or there are no pending or running behaviours. So, if this is what it means to be deadlock free, then we can clearly violate this lemma with the introduction of promises as we have defined them so far. Consider Listings A.10 and A.11 which do not have this property. In Listing A.10, we have exchanged promises such that we have created a dependency cycle; neither of the behaviours will ever be able to run. In Listing A.11, we see that $b1$ happens before $b2$, but $b2$ enables $b1$ and so neither will ever run. This is in fact a much worse outcome than Listing A.10 as we have now blocked use of c, so no behaviour which $b2$ must happen before will ever run. Indeed, if we consider the queue semantics for BoC in Chapter 4, no subsequent behaviour *at all* that uses c will ever run.

Listing A.10: Promise cycle

```
1 p1r, p1w = Promise[int]();
2 p2r, p2w = Promise[int]();
3 when(p1r) { p2w.fulfill(p1r) };
4 when(p2r) { p1w.fulfill(p2r) };
```

Listing A.11: Promise and happens before cycle

```
1 pr, pw = Promise[int]();
2 when(c, pr) { /* b1 */ };
3 when(c) { /* b2 */ pw.fulfill(c) };
```

It is at this point that we need to consider the principles of our design. If we believe the flexibility of these kinds of promises is worth weakening our deadlock freedom claim, then we can adopt this unification. Or, would we consider such programs to be an error, and dynamically reporting such errors to the programmer for them to handle. Or, we could discredit the idea altogether and adopt either the library or separate cowns and promises as we will see in the next section.

Taking a brief step away from correctness and principles, practically, this is an issue of not just progress, but also of memory. If these dependency cycles exist, they represent a memory leak, and the more behaviours caught by this cycle, the bigger the leak.

Operational semantics We will now look at how we can incorporate these promises into the operational semantics from Chapter 4. Much like in Figure 4.1, an underlying language is responsible for creating the promises (or cowns), but we now need another predicate that indicates when a promise has been fulfilled.

Property 7 (Fulfilled)**.**

$$\mathit{fulfilled} : (Heap \times Tag \times Context)$$

Now, we need to update the START rule from Figure 4.1 to ensure behaviours only start when all promises are fulfilled. STEP, SPAWN and END remain the same as before.

Definition 43 (BoC Semantics with Promises)**.**

$$\text{START}\ \frac{\forall \kappa \in \overline{\kappa}.\mathit{fulfilled}(h, \kappa, E) \qquad \left(\bigcirc_{(\overline{\kappa'},\_)\in(P'\cup R)} \overline{\kappa'}\right) \# \overline{\kappa}}{R, P' : (\overline{\kappa}, E) : P'', h \rightsquigarrow R + (\overline{\kappa}, E), P' : P'', h}$$

Unfulfilling a promise We redesigned a cown such that it could start empty, now lets digress briefly and consider a further change so that cowns can also be emptied during program execution. Consider the example in Listing A.12: we construct an empty cown, spawn a behaviour that requires it, and fulfill the promise. Next, we unfulfill the promise; the behaviour takes the contents of the cown and leaves the cown empty. This behaviour is enqueued as a pending behaviour that requires p to be available and to have a value, but when the behaviour runs it will take the value out of p, leaving p empty. This means $b3$ cannot run until the subsequent fulfill has executed. In essence, we are turning the cown into a channel where behaviours run once a message has arrived. This requires consideration of how writes, or fulfills, are buffered. This also complicates understanding of the program further as we have happens before, fulfilling and unfulling affecting the order of program execution.

Listing A.12: Unfulfill a promise

```
1 var p = Promise[int]();
2 when(p) { /* b1 */ }
3 p.fulfill(10);
4 when(unfulfill(p)) { /* b2 */ }
5 when(p) { /* b3 */ }
6 p.fulfill(20);
```

Looking any deeper into this is beyond the scope of my thesis, however, it is interesting to see how me could increase the remit of constructs of BoC and see the trade-off for doing this.

A.4.2 Separating cowns and promises

Once again, we will introduce a promise as a language feature. A promise is constructed with a read and write end, we can fulfill the promise, but now promises are not cowns. A promise is a new distinct kind of paradigm-intrinsic object. We will see how this makes the use of promises more streamlined than building them as a library, avoids blocking cowns as for unified cowns and promises, and provides new design choices compared with unified cowns and promises.

Let us take inspiration from the Async/Await feature that appears in many programming languages, and introduce a construct **`await`** [88]. We will separate waiting for a cown and waiting for a promise to be fulfilled, by using **`when`** for cowns, and **`await`** for waiting for a promise value.

Listing A.13: Await a promise

```
1 p1r, p1w = Promise[int]();
2 await (p1r) { /* f1 */ };
3 p1w.fulfill(10);
```

Listing A.14: Await multiple promises

```
p1r, p1w = Promise[int]();
p2r, p2w = Promise[int]();
await (p1r, p2r) { /* f1 */ };
p1w.fulfill(10);
p2w.fulfill(20);
```

Typically, **await**, transforms the program such that all code that follows the **await** becomes an asynchronous function that executes when the promise is fulfilled. We will do similarly for BoC in this section, **await** creates an asynchronous *thunk* that waits for all of the require promises to have a value. We can see examples of this in Listings A.13 and A.14. The thunk $f1$ will only execute once the promises have been written to.

Let us revisit Listing A.11 and see how we avoid the blocked cown problem. In the unified cowns and promises this program would prevent the cown c from running any future behaviours. We create the program in Listing A.15. The behaviour $b1$ is spawned before $b2$ and thus must execute first, however, this now only spawns the thunk $f1$ that waits until the promise is fulfilled. When $b2$ runs after $b1$ terminates, the promise will be fulfilled and $f1$ will be able to run. Separating the waiting on cowns and promises has broken the dependency cycle in this case.

Listing A.15: No dependency cycle

```
pr, pw = Promise[int]();
when(c) { /* b1 */
  await(pr) { /* f1 */ };
};
when(c) { /* b2 */ pw.fulfill(c) };
```

Listing A.16: Cown is not blocked

```
pr, pw = Promise[int]();
await(pr) { /* f1 */
  when(c) { /* b1 */ pw.fulfill(c)}
};
```

Note, we can still create dependency cycles, but we will never block a cown. Take the clearly problematic example Listing A.16, to enable the thunk $f1$, $f1$ must run, so it will never run. However, this means that behaviour $b1$ will also never be spawned, and so c will not be prevented from partaking in future behaviours.

A.5 Common design questions

Whether we choose to include promises as a library feature or paradigm feature, there are design questions which apply nonetheless. We will now look at these questions.

A.5.1 when creates a promise

Let us revisit Async/Await programming; typically, calling an `async` implicitly returns a promise, this is the only value an `async` function could return as the value is not yet computed. We can incorporate this convenience into promises in BoC. For a library, we can wrap **when**s with the construction of a promise and return it (as in Listing A.17). For integration into the paradigm, we can implicitly return a promise and fulfill the promise with the return value of the behaviour (as in Listing A.18).

Listing A.17: Library wrapping **when**

```
promise = Promise[int]();
when(c1) { promise.fulfill(c1.value) };
promise.then((x: int) -> { /* f1 */ })
```

Listing A.18: Implicitly return a promise from **when**

```
pr = when (c1) { return c1.value };
pr.then((x: int) -> { /* f1 */ })
```

In this case, the paradigm-intrinsic promises win in their elegance, as it is clunkier and less convenient to achieve this via a library.

A.5.2 Capability of the read and write ends

What a reader can do with the promised value, or a writer can do with a provided value, changes the parallelism and reasoning in the paradigm. Let us briefly touch on the capabilities of promises and how this

affects BoC.

Write once versus write multiple If a promise is more than a write-once permission, then a receiver can receive multiple values from a writer, in essence a form of channel. This introduces a few design alternatives. When can a writer write a value? This would need to be scheduled outside of any writing behaviour otherwise we create the potential for race conditions. Do readers wait for new values or do they execute with any value available? We would want to avoid "busy waiting" behaviours which reschedule themselves waiting for new data.

Read mutable versus read immutable If a promised value is mutable, then only a single behaviour can access the value at a time. Whereas, if a promised value is immutable, then there is no danger in multiple behaviours executing with access to the data as one.

A.6 Conclusion

Promises are a ubiquitous addition to asynchronous programming languages, and their value is no less present in BoC than they are elsewhere, as I have demonstrated through the motivating example Listing A.3. In this chapter I have explained why we would want promises in BoC, and presented a number of different designs and considerations for promises, as either a library or a paradigm-intrinsic feature. To progress this to inclusion into BoC requires a number of clear design objectives and examples. As a final point, if one were to pursue promises as paradigm-intrinsic feature, then it would be ideal to revisit the axiomatic model and enrich it to include these new causal relations.

Appendix B

Proofs

B.0.1 Proof of Lemma 3

Proof of Lemma 3. Take P, P', R, R', h, h' arbitrarily

Assumption 1. R, P, h is *well-formed* and $R, P, h \rightsquigarrow R', P', h'$

Case 1 (STEP). We know from the premise of STEP that there exists $\beta, \overline{\kappa}, E, E'$ such that: $R(\beta) = (\overline{\kappa}, E)$, $R'(\beta) = (\overline{\kappa}, E')$, $E, h \hookrightarrow E', h'$, and $R \backslash \beta = R' \backslash \beta$.

We know by assumption 1 that $\exists \alpha.[h, \alpha \vDash *_{(\overline{\kappa},E) \in rng(R)}(\overline{\kappa}, E, r) * *_{(\overline{\kappa},E) \in P}(\overline{\kappa}, E, p)]$

Thus, we know by Property 2 that $\exists \alpha.[h', \alpha \vDash (\overline{\kappa}, E', r) * *_{(\overline{\kappa},E) \in rng(R \backslash \beta)}(\overline{\kappa}, E, r) * *_{(\overline{\kappa},E) \in P}(\overline{\kappa}, E, p)]$

So the lemma holds in this case.

Case 2 (SPAWN). We know from the premise of SPAWN that there exists $\beta, \overline{\kappa}, \overline{\kappa'}, E, E', E''$ such that: $R(\beta) = (\overline{\kappa}, E)$, $R'(\beta) = (\overline{\kappa}, E')$, $E \hookrightarrow_{When \overline{\kappa} E''} E'$.

We know by assumption 1 that $\exists \alpha.[h, \alpha \vDash *_{(\overline{\kappa},E) \in rng(R)}(\overline{\kappa}, E, r) * *_{(\overline{\kappa},E) \in P}(\overline{\kappa}, E, p)]$.

Thus, we know by Property 3 that $\exists \alpha. h, \alpha \vDash (\overline{\kappa}, E', r) * (\overline{\kappa'}, E'', p) * *_{(\overline{\kappa},E) \in rng(R \backslash \beta)}(\overline{\kappa}, E, r) * *_{(\overline{\kappa},E) \in P}(\overline{\kappa}, E, p)$

So the lemma holds in this case.

Case 3 (START). We know from the premise of START that there exists $\beta, \overline{\kappa}, E, P', P''$ such that: $P = P' : (\overline{\kappa}, E) : P'$, $\left(\bigcirc_{(\overline{\kappa'},\_) \in (P' \cup rng(R))} \overline{\kappa'}\right) \# \overline{\kappa}$

We know by assumption 1 that $\exists \alpha.[h, \alpha \vDash *_{(\overline{\kappa},E) \in rng(R)}(\overline{\kappa}, E, r) * *_{(\overline{\kappa},E) \in P}(\overline{\kappa}, E, p)]$.

We know $\left(\bigcirc_{(\overline{\kappa'},\_) \in (rng(R))} \overline{\kappa'}\right)$ and $\overline{\kappa} \# \left(\bigcirc_{(\overline{\kappa'},\_) \in (rng(R))} \overline{\kappa'}\right)$

Thus we know by Property 4 that $\exists \alpha.[h, \alpha \vDash *_{(\overline{\kappa''},E'') \in rng(R)}(\overline{\kappa}, E'', r) * *_{(\overline{\kappa''},E'') \in P' : P''}(\overline{\kappa}, E'', p) * (\overline{\kappa}, E, r)]$

So the lemma holds in this case.

Case 4 (END). We know by assumption 1 that $\exists \alpha.[h, \alpha \vDash *_{(\overline{\kappa},E) \in rng(R)}(\overline{\kappa}, E, r) * *_{(\overline{\kappa},E) \in P}(\overline{\kappa}, E, p)]$.

Thus we know $\exists \alpha.[h, \alpha \vDash *_{(\overline{\kappa},E) \in rng(R \backslash \beta)}(\overline{\kappa}, E, r) * *_{(\overline{\kappa},E) \in P}(\overline{\kappa}, E, p)]$.

So the lemma holds in this case.

As the lemma holds in all cases, the lemma holds. □

B.0.2 Proof of Lemma 5

Proof of Lemma 5.

Assumption 1.

$$R_1, P_1, h_1 \text{ is well-formed}$$
$$\wedge\ R_1, P_1, h_1 \underset{\sim\beta}{\rightsquigarrow} R_2, P_2, h_2 \underset{\beta}{\rightsquigarrow} R_3, P_3, h_3$$

We need to show that:

$$\exists R_4, P_4, h_4.[R_1, P_1, h_1 \underset{\beta}{\rightsquigarrow} R_4, P_4, h_4 \underset{\sim\beta}{\rightsquigarrow} R_3, P_3, h_3]$$

There are four cases for $R_1, P_1, h_1 \rightsquigarrow_{\sim\beta} R_2, P_2, h_2$. We prove the lemma holds for each of these cases.

Case 1 (STEP$_{\sim\beta}$). We know from the premise of STEP$_{\sim\beta}$ and STEP$_{\beta}$ there exists $\beta', \overline{\kappa_1}, \overline{\kappa_2}, E_1, E_1', E_2, E_2'$ such that: $\beta' \neq \beta$, $R_1(\beta') = (\overline{\kappa_1}, E_1)$ $R_2 = R_1[\beta' \mapsto ((\overline{\kappa_1}, E_1'))]$, $E_1, h_1 \hookrightarrow E_1', h_2$, $R_2(\beta) = (\overline{\kappa_2}, E_2)$, $R_3 = R_2[\beta \mapsto \overline{\kappa_2}, E_2']$, $E_2, h_2 \hookrightarrow E_2', h_3$,

We know R_1, P_1, h_1 is well-formed and thus R_2, P_2, h_2 and R_3, P_3, h_3 are also well-formed by Lemma 3.

Thus we know by Property 5 $\exists h_4.(E_2, h_1 \hookrightarrow E_2', h_4 \wedge E_1, h_4 \hookrightarrow E_1', h_3)$

We chose $R_4 = R_1[\beta \mapsto \overline{\kappa_2}, E_2']$.

We know $P_1 = P_2 = P_3$ and we chose $P_4 = P_1$.

Thus, we know that: $R_1, P_1, h_1 \rightsquigarrow_{\beta} R_4, P_4, h_4$, $R_4, P_4, h_4 \rightsquigarrow_{\sim\beta} R_3, P_3, h_3$

Thus, the lemma holds in this case.

Case 2 (SPAWN$_{\sim\beta}$). We know from the premise of SPAWN$_{\sim\beta}$ and STEP$_{\beta}$ there exists β', $\overline{\kappa_1}$, $\overline{\kappa_2}$, $\overline{\kappa_3}$, E_1, E_1', E_2, E_2', E_3 such that: $R_1(\beta') = ((\overline{\kappa_1}, E_1))$, $R_2 = R_1[\beta' \mapsto ((\overline{\kappa_1}, E_1'))]$, $P_2 = P_1 : (\overline{\kappa_3}, E_3)$, $E_1 \hookrightarrow_{\mathsf{when}\ (\overline{\kappa_3})\ \{E_3\}} E_1'$, $R_2(\beta) = (\overline{\kappa_2}, E_2)$, $R_3 = R_2[\beta \mapsto \overline{\kappa_2}, E_2']$, $E_2, h_2 \hookrightarrow E_2', h_3$, $h_1 = h_2$

The first step does not have access to the heap, thus we reflect that by choosing $h4 = h3$. We choose $R_4 = R_1[\beta \mapsto \overline{\kappa_2}, E_2']$. We choose $P_4 = P_2$.

We can demonstrate: $R_1, P_1, h_1 \rightsquigarrow_{\beta} R_4, P_4, h_4$, $R_4, P_4, h_4 \rightsquigarrow_{\sim\beta} R_3, P_3, h_3$

Thus, the lemma holds in this case.

Case 3 (START$_{\sim\beta}$). We know from the premise of START$_{\sim\beta}$ and STEP$_{\beta}$ that there exists β', $\overline{\kappa_1}$, $\overline{\kappa_2}$, E_1, E_2, E_2', P_1', P_1'' such that: $P_1 = P_1' : (\overline{\kappa_1}, E_1) : P_1''$, $R_2 = R_1[\beta' \mapsto (\overline{\kappa_1}, E_1)]$, $P_2 = P_1' : P_1''$ $R_2(\beta) = (\overline{\kappa_2}, E_2)$, $R_3 = R_2[\beta \mapsto \overline{\kappa_2}, E_2']$, $P_3 = P_2$

We choose: $R_4 = R_1[\beta \mapsto \overline{\kappa_2}, E_2']$, $P_4 = P_1$, $h_4 = h_3$,

We can demonstrate: $R_1, P_1, h_1 \rightsquigarrow_{\beta} R_4, P_4, h_4$, $R_4, P_4, h_4 \rightsquigarrow_{\sim\beta} R_3, P_3, h_3$

Thus, the lemma holds in this case.

Case 4 (END$_{\sim\beta}$). We know from the premise of END$_{\sim\beta}$ and STEP$_{\beta}$ that there exists β', $\overline{\kappa_1}$, $\overline{\kappa_2}$, E_1, E_2, E_2' such that: $R_2 = R_1 \backslash \beta$, $R_2(\beta) = (\overline{\kappa_2}, E_2)$, $R_3 = R_2[\beta \mapsto (\overline{\kappa_2}, E_2')]$

We choose: $R_4 = R_1[\beta \mapsto (\overline{\kappa_2}, E_2')]$ $P_4 = P_1$, $h_4 = h_3$,

We can demonstrate: $R_1, P_1, h_1 \rightsquigarrow_{\beta} R_4, P_4, h_4$, $R_4, P_4, h_4 \rightsquigarrow_{\sim\beta} R_3, P_3, h_3$

Thus, the lemma holds in this case.

As we have found R_4, P_4, h_4 in each case, we have demonstrated that the lemma holds. □

B.0.3 Proof of Lemma 6

Proof of Lemma 6.

Case 1 (Base cases). For $n = 0$: this is trivial.

For $n = 1$: this is the case for Lemma 5.

Case 2 (Inductive case). For $n > 1$.

Assumption 1 (Inductive hypothesis). Assume for $n = k$ the property holds.

To show for $n = k + 1$.

Assumption 2.

$$R_1, P_1, h_1 \text{ is well-formed} \wedge \ R_1, P_1, h_1 \overset{k+1}{\underset{\sim\beta}{\rightsquigarrow}} R_2, P_2, h_2 \underset{\beta}{\rightsquigarrow} R_3, P_3, h_3$$

Thus there exists R_4, P_4, h_4 such that:

$$R_1, P_1, h_1 \overset{k+1}{\underset{\sim\beta}{\rightsquigarrow}} R_2, P_2, h_2 = R_1, P_1, h_1 \overset{k}{\underset{\sim\beta}{\rightsquigarrow}} R_4, P_4, h_4 \underset{\sim\beta}{\rightsquigarrow} R_2, P_2, h_2$$

Thus by Lemma 5 there exists R_5, P_5, h_5 such that:

$$R_4, P_4, h_4 \underset{\beta}{\rightsquigarrow} R_5, P_5, h_5 \underset{\sim\beta}{\rightsquigarrow} R_3, P_3, h_3$$

Thus by assumption 1 there exists R_6, P_6, h_6 such that:

$$R_1, P_1, h_1 \underset{\beta}{\rightsquigarrow} R_6, P_6, h_6 \overset{k}{\underset{\sim\beta}{\rightsquigarrow}} R_5, P_5, h_5$$

Thus:

$$R_1, P_1, h_1 \underset{\beta}{\rightsquigarrow} R_6, P_6, h_6 \overset{k+1}{\underset{\sim\beta}{\rightsquigarrow}} R_3, P_3, h_3$$

□

B.0.4 Proof of Lemma 7

Proof of Lemma 7. Proof by induction.

Case 1 (n = 0). Trivial

Case 2 (n = 1). Trivial

Case 3 (n = 2).

Assumption 1.

$$R_1, P_1, h_1 \text{ is well-formed} \wedge R_1, P_1, h_1 \overset{2}{\underset{\beta/\sim\beta}{\rightsquigarrow}} R_2, P_2, h_2$$

There are four subcases to prove.

Subcase 3.1 ($\rightsquigarrow_{\sim\beta}$ then $\rightsquigarrow_{\sim\beta}$). Take $R_3, P_3, h_3 = R_1, P_1, h_1, n_1 = 0$ and $n_2 = 2$

Subcase 3.2 ($\rightsquigarrow_{\sim\beta}$ then $\rightsquigarrow_{\beta}$). Thus, $R_1, P_1, h_1 \rightsquigarrow_{\sim\beta} R_4, P_4, h_4 \rightsquigarrow_{\beta} R_2, P_2, h_2$. So, we know by Lemma 5 that $\exists R_5, P_5, h_5$ such that $R_1, P_1, h_1 \rightsquigarrow_{\beta} R_5, P_5, h_5 \rightsquigarrow_{\sim\beta} R_2, P_2, h_2$. So, chose $n_1 = n_2 = 1$.

Subcase 3.3 ($\rightsquigarrow_{\beta}$ then $\rightsquigarrow_{\sim\beta}$). Thus, $R_1, P_1, h_1 \rightsquigarrow_{\beta} R_3, P_3, h_3 \rightsquigarrow_{\sim\beta}$ So, chose $n_1 = n_2 = 1$.

Subcase 3.4 ($\rightsquigarrow_{\beta}$ then $\rightsquigarrow_{\beta}$). Take $R_3, P_3, h_3 = R_2, P_2, h_2, n_1 = 2$ and $n_2 = 0$

The lemma holds in all subcases.

Case 4 (Inductive Case).

Assumption 2 (Inductive Hypothesis). Take $n = k$ arbitrarily and assume holds:

$$\begin{aligned} & R_1, P_1, h_1 \text{ is well-formed} \\ \wedge\ & R_1, P_1, h_1 \underset{\beta/\sim\beta}{\overset{k}{\rightsquigarrow}} R_2, P_2, h_2 \\ \Longrightarrow\ & \exists R_3, P_3, h_3, n_1, n_2.[R_1, P_1, h_1 \underset{\beta}{\overset{n_1}{\rightsquigarrow}} R_3, P_3, h_3 \underset{\sim\beta}{\overset{n_2}{\rightsquigarrow}} R_2, P_2, h_2 \\ & \wedge\ n = n_1 + n_2] \end{aligned}$$

Show the lemma holds for $n = k + 1$.

We can restructure

$$R_1, P_1, h_1 \underset{\beta/\sim\beta}{\overset{k+1}{\rightsquigarrow}} R_2, P_2, h_2$$

to

$$R_1, P_1, h_1 \underset{\beta/\sim\beta}{\overset{k}{\rightsquigarrow}} R_4, P_4, h_4 \underset{\beta/\sim\beta}{\rightsquigarrow} R_2, P_2, h_2$$

And we know from assumption 2 that:

$$\begin{aligned} & \exists R_5, P_5, h_5, m_1, m_2.[R_1, P_1, h_1 \underset{\beta}{\overset{m_1}{\rightsquigarrow}} R_5, P_5, h_5 \underset{\sim\beta}{\overset{m_2}{\rightsquigarrow}} R_4, P_4, h_4 \\ & \wedge\ n = m_1 + m_2] \end{aligned}$$

So we restructure the previous steps as follows:

$$R_1, P_1, h_1 \underset{\beta}{\overset{m_1}{\rightsquigarrow}} R_5, P_5, h_5 \underset{\sim\beta}{\overset{m_2}{\rightsquigarrow}} R_4, P_4, h_4 \underset{\beta/\sim\beta}{\rightsquigarrow} R_2, P_2, h_2$$

Now we have two subcases to investigate.

Subcase 4.1 ($R_4, P_4, h_4 \rightsquigarrow_{\overline{\kappa}} R_2, P_2, h_2$). We know by Lemma 6 that:

$$\exists R_6, P_6, h_6.R_5, P_5, h_5 \underset{\beta}{\rightsquigarrow} R_6, P_6, h_6 \underset{\sim\beta}{\overset{m_2}{\rightsquigarrow}} R_2, P_2, h_2$$

So we choose $R_3, P_3, h_3 = R_6, P_6, h_6$, $n_1 = m_1 + 1$ and $n_2 = m_2$ and we know:

$$R_1, P_1, h_1 \underset{\beta}{\overset{n_1}{\rightsquigarrow}} R_6, P_6, h_6 \underset{\sim\beta}{\overset{n_2}{\rightsquigarrow}} R_2, P_2, h_2$$

So we're done.

Subcase 4.2 ($R_4, P_4, h_4 \rightsquigarrow_{\sim\beta} R_2, P_2, h_2$). In this subcase we choose $R_3, P_3, h_3 = R_5, P_5, h_5, n_1 = m_1$ and $n_2 = m_2 + 1$ and we're done.

The lemma holds for $n + 1$ and so we know the lemma holds.

Lemma proved by induction. □

B.0.5 Proving $\lambda_{\textsf{when}}$ satisfies the BoC interface properties

Progress

λ_{when} satisfies Property 1. This is satisfied by the combination of STUCK and the definition of *finished*.

Any configuration that can no longer make progress, without resulting in an *error*, and isn't *finished* will evaluate to *error*, which will then be *finished*. □

Running Isolation

Lemma 8.

$$\begin{aligned}\forall h_1, h_2, \alpha_1, \alpha_2, V, \overline{\kappa_{1\ldots n}} \in Tag, t_{1\ldots n} \in Term.[V = \divideontimes_{i \in 1\ldots n}(\overline{\kappa_i}, (\overline{\kappa_i}, t_i), r) \\ \wedge\ h_1, \alpha_1 \vDash V \\ \wedge\ h_2, \alpha_2 \vDash V \\ \implies \alpha_1 = \alpha_2]\end{aligned}$$

Proof of Lemma 8. We can see this by inspection of the definition of λ_{when} compose in Definition 21 and λ_{when} well-formed (Definition 20).

Whenever, the well-formed judgement is satisfied, it must be the union of the cowns in the views. □

λ_{when} satisfies Property 2.

Assumption 1.

$$(\overline{\kappa_1}, t_1), h_1 \hookrightarrow (\overline{\kappa_2}, t_2), h_2 \wedge (\exists \alpha. h_1, \alpha \vDash (\overline{\kappa_1}, (\overline{\kappa_1}, t_1), r) * V)$$

We need to show there exists some α such that $h_2, \alpha \vDash (\overline{\kappa_2}, (\overline{\kappa_2}, t_2), r) * V$.
We know $\overline{\kappa_1} = \overline{\kappa_2}$.
We know:

$$\exists \alpha, \alpha_1, \alpha_2. [h_1, \alpha_1 \vDash (\overline{\kappa_1}, (\overline{\kappa_1}, t_1), r) \wedge h_1, \alpha_2 \vDash V \wedge \alpha = \alpha_1 * \alpha_2]$$

by Definition 12.

We choose the same α, α_1, α_2 as before such that:

$$h_2, \alpha_1 \vDash (\overline{\kappa_1}, (\overline{\kappa_1}, t_2), r) \wedge h_2, \alpha_2 \vDash V \wedge \alpha = \alpha_1 * \alpha_2$$

As $\alpha_1 = \overline{\kappa_1}$ by definition of λ_{when} well-formed (Definition 20).
We can see $dom(h_1) \subseteq dom(h_2)$ by inspection of the evaluation relation.
Thus $h_2, \alpha_1 \vDash (\overline{\kappa_1}, (\overline{\kappa_1}, t_2), r)$.
And also $h_2, \alpha_2 \vDash V$ as $dom(h_1) \subseteq dom(h_2)$ and Lemma 8.
So we have found an α such that:

$$h_2, \alpha \vDash (\overline{\kappa_2}, (\overline{\kappa_2}, t_2), r) * V$$

Thus the property is satisfied. □

Spawn Isolation

λ_{when} satisfies Property 3.

Assumption 1.

$$(\overline{\kappa}, t) \hookrightarrow_{\mathsf{when}\ (\overline{\kappa'})\ \{(\overline{\kappa'}, t'')\}} (\overline{\kappa}, t') \wedge (\exists \alpha. h, \alpha \vDash (\overline{\kappa}, (\overline{\kappa}, t), r) * V)$$

We must show there exists α such that:

$$h, \alpha \vDash (\overline{\kappa}, (\overline{\kappa}, t'), r) * V * (\overline{\kappa'}, (\overline{\kappa'}, t''), p)$$

We know there exists α, α_1, α_2 such that:

$$\begin{aligned} h, \alpha_1 &\vDash (\overline{\kappa}, (\overline{\kappa}, t), r) \\ h, \alpha_2 &\vDash V \\ \alpha &= \alpha_1 * \alpha_2 \end{aligned}$$

We choose $\alpha_3 = \varnothing$ and show there exists α' such that:

$$\begin{aligned} \alpha' &= \alpha_1 * \alpha_2 * m3 \\ h, \alpha_3 &\vDash (\overline{\kappa'}, (\overline{\kappa'}, t''), p) \\ h, \alpha' &\vDash (\overline{\kappa}, (\overline{\kappa}, t'), r) * V * (\overline{\kappa'}, (\overline{\kappa'}, t''), p) \end{aligned}$$

We know $\alpha_3 = \varnothing$ from Definition 20, we also know $cowns(t'', h) \subseteq cowns(t, h)$ by observation of the rules of $\lambda_{\texttt{when}}$, and we $cowns(t, h) \subseteq dom(h)$ by assumption 1. So we know $h, \alpha_3 \vDash (\overline{\kappa'}, (\overline{\kappa'}, t''), p)$

Trivially $\alpha' = \alpha_1 * \alpha_2 * \alpha_3 = \alpha_1 * \alpha_2 * \varnothing = \alpha_1 * \alpha_2 = \alpha$.

And we know that $h, \alpha \vDash (\overline{\kappa}, (\overline{\kappa}, t'), r) * V$ where $h, \alpha_1 \vDash (\overline{\kappa}, (\overline{\kappa}, t'), r)$ by much the same reasoning as for Appendix B.0.5. The $cowns(t', h) \subseteq cowns(t, h)$ and we know $cowns(t, h) \subseteq dom(h)$

Thus we have found such an α and so the property is satisfied. □

Start Isolation

Proof.

Assumption 1.

$$(\exists \alpha. h, \alpha \vDash (\overline{\kappa}, (\overline{\kappa}, t), p) * V) \wedge (\overline{\kappa} \# \bigcirc_{(\overline{\kappa'}, \_, r) \in V} \overline{\kappa'})$$

We need to show:

$$\exists \alpha. h, \alpha \vDash (\kappa, (\overline{\kappa}, t), r) * V$$

We choose $\alpha' = \alpha * \overline{\kappa}$ which we know is defined as $\overline{\kappa} \# \bigcirc_{(\overline{\kappa'}, \_, r) \in V} \overline{\kappa'}$.

We know $h, \overline{\kappa} \vDash (\kappa, (\overline{\kappa}, t), r)$, as we know $h, \alpha \vDash (\kappa, (\overline{\kappa}, t), p)$ from assumption 1 so we know $cowns(t, h) \subseteq dom(h) \wedge \overline{\kappa} \subseteq dom(h)$.

Thus we have found such an α and so the property is satisfied. □

Data-race freedom

Proof.

Assumption 1.

$$\begin{aligned} &(\exists \alpha. h_1, \alpha \vDash (\overline{\kappa_1}, (\overline{\kappa_1}, t_1), r) * (\overline{\kappa_2}, (\overline{\kappa_2}, t_2), r)) \\ &\wedge (\overline{\kappa_1}, t_1), h_1 \hookrightarrow (\overline{\kappa_1}, t_1'), h_2 \\ &\wedge (\overline{\kappa_2}, t_2), h_2 \hookrightarrow (\overline{\kappa_2}, t_2'), h_3 \end{aligned}$$

We need to show:

$$\exists h_4. ((\overline{\kappa_2}, t_2), h_1 \hookrightarrow (\overline{\kappa_2}, t_2'), h_4 \wedge (\overline{\kappa_1}, t_1), h_4 \hookrightarrow (\overline{\kappa_1}, t_1'), h_3)$$

Consider that this proof would require a analysis of interchanging all pairs of rules for $\lambda_{\texttt{when}}$ which, with 9 rules, is 81 pairs. We can cut this down but considering only the interesting pairs of rules, which are

the rules which read/write the heap. In these cases we choose $h_4 = h_1 = h_2 = h_3$ and the relation will trivially hold as it is provided by assumption 1.

Explicitly, we do not have to worry about the COWN/WHEN pair. The WHEN does not have a premise, and we have avoided this through the definition of well-formed which prevents behaviours naming cowns which do not exist but may be created in the future.

The HOLE rules are covered by inspecting the base rules, thus if the base rules can interchange, the so can the HOLE rules.

This brings us down to considering only 3 rules, COWN, DEREF, and ASSIGN, which is 9 cases.

Case 1 (First COWN).

Subcase 1.1 (Second COWN). We know $dom(h_2) \backslash dom(h_1) = \{\kappa\}$ and $dom(h_3) \backslash dom(h_2) = \{\kappa'\}$ by inspection of COWN.

Thus we choose $h_4 = h_1[\kappa \mapsto h_2(\kappa)]$ and we can demonstrate $(\overline{\kappa_2}, t_2), h_1 \hookrightarrow (\overline{\kappa_2}, t_2'), h_4 \wedge (\overline{\kappa_1}, t_1), h_4 \hookrightarrow (\overline{\kappa_1}, t_1'), h_3$.

Subcase 1.2 (Second by DEREF). We know $dom(h_2) \backslash dom(h_1) = \{\kappa\}$ (the fresh cown).

So $t_2 = !\kappa'$

We know also that $cowns(t_2, h_1) \subseteq dom(h_1)$ by Definition 21.

Thus, $\kappa' \neq \kappa$.

We choose $h_4 = h_1$ and we can demonstrate $(\overline{\kappa_2}, t_2), h_1 \hookrightarrow (\overline{\kappa_2}, t_2'), h_4 \wedge (\overline{\kappa_1}, t_1), h_4 \hookrightarrow (\overline{\kappa_1}, t_1'), h_3$.

Subcase 1.3 (Second by ASSIGN). This follows similarly to DEREF subcase.

We know there exists κ, v such that $t_2 = \kappa := v$.

Thus we choose $h_4 = h_1[\kappa \mapsto v]$.

we can demonstrate $(\overline{\kappa_2}, t_2), h_1 \hookrightarrow (\overline{\kappa_2}, t_2'), h_4 \wedge (\overline{\kappa_1}, t_1), h_4 \hookrightarrow (\overline{\kappa_1}, t_1'), h_3$.

Case 2 (First by DEREF).

Subcase 2.1 (Second by COWN). The dereference cannot access the cown created as it has not yet been created in h_1.

The cown must also be fresh in h_2 so cannot be the cown dereferenced.

So we choose $h_4 = h_3$ and we can demonstrate $(\overline{\kappa_2}, t_2), h_1 \hookrightarrow (\overline{\kappa_2}, t_2'), h_4 \wedge (\overline{\kappa_1}, t_1), h_4 \hookrightarrow (\overline{\kappa_1}, t_1'), h_3$.

Subcase 2.2 (Second by DEREF). The heap does not change in either step so we choose $h_4 = h_1$ and we can demonstrate $(\overline{\kappa_2}, t_2), h_1 \hookrightarrow (\overline{\kappa_2}, t_2'), h_4 \wedge (\overline{\kappa_1}, t_1), h_4 \hookrightarrow (\overline{\kappa_1}, t_1'), h_3$.

Subcase 2.3 (Second by ASSIGN). We know that $\overline{\kappa_1} \# \overline{\kappa_2}$ by Definition 21, and so by analysis of the step the assign cannot write to the cown read by the dereference (they must both belong to their cown set).

Thus we choose $h_4 = h_3$ and we can demonstrate $(\overline{\kappa_2}, t_2), h_1 \hookrightarrow (\overline{\kappa_2}, t_2'), h_4 \wedge (\overline{\kappa_1}, t_1), h_4 \hookrightarrow (\overline{\kappa_1}, t_1'), h_3$.

Case 3 (First by ASSIGN).

Subcase 3.1 (Second by COWN). The cown must be created fresh in the second step and so we know $dom(h_3) \backslash dom(h_2) = \kappa$.

We choose $h_4 = h_1[\kappa \mapsto h_3(\kappa)]$ and we can demonstrate $(\overline{\kappa_2}, t_2), h_1 \hookrightarrow (\overline{\kappa_2}, t_2'), h_4 \wedge (\overline{\kappa_1}, t_1), h_4 \hookrightarrow (\overline{\kappa_1}, t_1'), h_3$.

Subcase 3.2 (Second by DEREF). We know that $\overline{\kappa_1} \# \overline{\kappa_2}$ by Definition 21, and so by analysis of the step the assign cannot write to the cown read by the dereference (they must both belong to their cown set).

Thus we choose $h_4 = h_1$ and we can demonstrate $(\overline{\kappa_2}, t_2), h_1 \hookrightarrow (\overline{\kappa_2}, t_2'), h_4 \wedge (\overline{\kappa_1}, t_1), h_4 \hookrightarrow (\overline{\kappa_1}, t_1'), h_3$.

Subcase 3.3 (Second by ASSIGN). We know that $\overline{\kappa_1} \# \overline{\kappa_2}$ by Definition 21, and so by analysis of the step the assigns cannot write to the same cown (they must both belong to their cown set).

We know $t_2 = \kappa := v$ and so we choose $h_4 = h_1[\kappa \mapsto v]$ and we can demonstrate $(\overline{\kappa_2}, t_2), h_1 \hookrightarrow (\overline{\kappa_2}, t_2'), h_4 \wedge (\overline{\kappa_1}, t_1), h_4 \hookrightarrow (\overline{\kappa_1}, t_1'), h_3$.

We have argued that the property is satisfied for all cases and so the property holds. □

Aside, if $\overline{\kappa} = \varnothing$ for a behaviour, in $\lambda_{\mathtt{when}}$, then the composition will defined for any behaviour that also uses $\overline{\kappa'} = \varnothing$, this includes the behaviour composing with itself. In the above this is not a problem, we are not truly swapping the order of steps, but saying we can find some heap which satisfies the property, and a behaviour with $\overline{\kappa} = \varnothing$ cannot read/write an existing heap location, so it is straightforward. It is an important detail in Lemma 5 that we identify the behaviours by identifier, otherwise we would have to otherwise preclude behaviours with $\overline{\kappa} = \varnothing$ trying to swap local steps.

Progress

Proof. Take $\overline{\kappa}, t, h$ arbitrarily.
If $\exists t', h'$ such that $t' \neq \mathit{error}$ and $(\overline{\kappa}, t), h \hookrightarrow (\overline{\kappa}, t'), h'$ then we're done.
If $\mathit{finished}(\overline{\kappa}, t)$ then we're done.
If $\exists \overline{\kappa'}, t'$ such that $(\overline{\kappa}, t) \hookrightarrow_{\mathtt{when}\ (\overline{\kappa'})\ \{(\overline{\kappa'},\ t')\}} (\overline{\kappa}, \mathit{unit})$ then we're done.
Otherwise we know that $(\overline{\kappa}, t), h \hookrightarrow (\overline{\kappa}, \mathit{error}), h$ by Definition 18. □

Appendix C

Join calculus in Boc

Listing C.1: Join Calculus in Verona

```
// Copyright Microsoft and Project Verona Contributors.
// SPDX-License-Identifier: MIT
#include <memory>
#include <debug/harness.h>
#include <cpp/when.h>

#include <optional>
#include <iostream>

using namespace verona::cpp;

using namespace std;

namespace Joins {

  struct Observer {
    virtual void notify() = 0;

    virtual ~Observer() {};
  };

  template<typename T>
  struct Channel {
    /* Channel has:
        - a queue of data to be read
        - a list of observers to notify whenever there is data
        Observers subscribe to the channel and whenever
        there is data they will be notified
    */
    queue<unique_ptr<T>> data;
    // polymorphic cown_ptr
    vector<cown_ptr<unique_ptr<Observer>>> observers;

    static void notify_all(acquired_cown<Channel<T>>& channel) {
      for (auto& observer : channel->observers) {
        when(observer) << [c=channel.cown()] (acquired_cown<unique_ptr<Observer>>
            ↪observer) {
```

```
37          assert(c);
38          (*observer)->notify();
39        };
40      }
41    }
42
43    static void write(acquired_cown<Channel<T>>& channel, unique_ptr<T> value) {
44      channel->data.push(move(value));
45      Channel<T>::notify_all(channel);
46    }
47
48    static void write(cown_ptr<Channel<T>> channel, unique_ptr<T> value) {
49      when(channel) << [value=move(value)] (acquired_cown<Channel<T>> channel) mutable
            ↪ {
50        Channel<T>::write(channel, move(value));
51      };
52    }
53
54    static unique_ptr<T> read(acquired_cown<Channel<T>>& channel) {
55      if (channel->data.size() > 0) {
56        unique_ptr<T> front = move(channel->data.front());
57        channel->data.pop();
58        if (channel->data.size() > 0)
59          Channel<T>::notify_all(channel);
60        return front;
61      }
62      return nullptr;
63    }
64
65    bool has_data() {
66      return data.size() != 0;
67    }
68
69    static void subscribe(acquired_cown<Channel<T>>& channel, cown_ptr<unique_ptr<
          ↪Observer>> observer) {
70      channel->observers.push_back(observer);
71      if (channel->data.size() > 0) {
72        when(move(observer)) << [c=channel.cown()] (acquired_cown<unique_ptr<Observer>>
              ↪ observer) {
73          assert(c);
74          (*observer)->notify();
75        };
76      }
77    }
78  };
79
80  template <typename T>
81  static unique_ptr<T> read(acquired_cown<Channel<T>>& channel) {
82    return Channel<T>::read(channel);
83  }
84
85  template <typename T>
86  static void write(acquired_cown<Channel<T>>& channel, unique_ptr<T> value) {
87    Channel<T>::write(channel, move(value));
88  }
89
90  template <typename T>
```

```
91  static void write(cown_ptr<Channel<T>> channel, unique_ptr<T> value) {
92    Channel<T>::write(move(channel), move(value));
93  }
94
95  template <typename T>
96  static void subscribe(acquired_cown<Channel<T>>& channel, cown_ptr<unique_ptr<
        ↪Observer>> observer) {
97    Channel<T>::subscribe(channel, move(observer));
98  }
99
100 template<typename S, typename R>
101 struct Message {
102   /*
103     A Message is a pair of:
104       - Data to place on a channel
105       - A callback to reply to (this is like a synchronous join)
106     Either of these can be empty
107
108     Note: ideally we could subclass this and make the message
109     types nicer but polymorphism in cown_ptr<> is fiddly
110   */
111   optional<unique_ptr<S>> data;
112
113   using F = function<void(unique_ptr<R>)>;
114   optional<F> reply;
115
116   Message(unique_ptr<S> data): data(move(data)), reply(nullopt) {};
117   Message(unique_ptr<S> data, F reply): data(move(data)), reply(forward<F>(reply))
        ↪{};
118   Message(F reply): data(nullopt), reply(forward<F>(reply)) {};
119 };
120
121 /* Data and Reply type aliases for convience */
122 template<typename S>
123 using DataMessage = Message<S, nullopt_t>;
124
125 template<typename R>
126 using ReplyMessage = Message<nullopt_t, R>;
127
128 /*
129   Pattern<N> consists of N channels
130   There two important functions for these structs:
131     - And : This continues building a pattern with one more channel
132     - Do : Registers a pattern as an observer on the channels,
133               this takes a callback to run when there is data on
134               all the channels
135
136   The internal Pattern is the observer. When notified
137   the Pattern checks all channels to which it is subscribed
138   and if there is data on all of them, reads the data and
139   executes the callback.
140 */
141 template<typename ...Args>
142 struct Pattern {
143
144   tuple<cown_ptr<Channel<Args>>...> channels;
145
```

```
146     Pattern(cown_ptr<Channel<Args>>... channels): channels(channels...) {}
147
148     template<typename M>
149     Pattern<Args..., M> And(cown_ptr<Channel<M>> channel) {
150       return apply([channel=move(channel)](auto &&...args) mutable {
151         return Pattern<Args..., M>(args..., move(channel));
152       }, move(channels));
153     }
154
155     using F = function<void(unique_ptr<Args>...)>;
156
157     struct P : public Observer {
158       tuple<typename cown_ptr<Channel<Args>>::weak...> channels;
159       F f;
160
161       P(F f, cown_ptr<Channel<Args>>... channels): f(forward<F>(f)), channels(channels
              ↪.get_weak()...) {}
162
163       void notify() {
164         /* promote all channels to strong refs */
165         tuple<cown_ptr<Channel<Args>>...> cs = apply([](auto &&...args) mutable {
166           return make_tuple<cown_ptr<Channel<Args>>...>(move(args.promote())...);
167         }, channels);
168
169         /* If any cown can't be promoted it is because the channel has been deallocated
170             thus the pattern can no longer match */
171         if(apply([](auto &&...args) mutable { return (!args || ...) ; }, cs))
172           return;
173
174         /* Otherwise, attempt to read a value from all channels and call the pattern callback */
175         apply([f=f](auto &&... args) mutable {
176           when(args...) << [f=forward<F>(f)](acquired_cown<Channel<Args>>... channels)
                  ↪mutable {
177             if ((!channels->has_data() || ...))
178               return;
179             f(read(channels)...);
180           };
181         }, move(cs));
182       }
183     };
184
185     void Do(F run) {
186       auto pattern = make_cown<unique_ptr<Observer>>(
187         apply([run=forward<F>(run)](auto &&...args) mutable {
188           return make_unique<P>(forward<F>(run), args...);
189         }, channels));
190
191         apply([pattern=move(pattern)](auto &&...args) mutable {
192           when(args...) << [pattern=move(pattern)](acquired_cown<Channel<Args>>...
                  ↪channels) mutable {
193             (subscribe(channels, pattern), ...);
194           };
195         }, move(channels));
196       }
197     };
198
199     /*
```

```
200       Join starts of the construction of a pattern
201     */
202     namespace Join {
203       template<typename M>
204       static Pattern<M> When(cown_ptr<Channel<M>> channel) {
205         return Pattern<M>(channel);
206       }
207     };
208
209     void run() {
210       /*
211         Builds a put and get channel:
212           - put channel contains messages with data
213           - get channel contains messages with replies
214       */
215       auto put = make_cown<Channel<DataMessage<int>>>();
216       auto get = make_cown<Channel<ReplyMessage<int>>>();
217
218       write(put, make_unique<DataMessage<int>>(make_unique<int>(20)));
219
220       write(put, make_unique<DataMessage<int>>(make_unique<int>(51)));
221
222       /* send a repliable message on get */
223       write(get, make_unique<ReplyMessage<int>>([](unique_ptr<int> msg) mutable {
224          cout << *msg << " -- 1" << endl;
225       }));
226
227       /* create a pattern so if there is a message on put then print it */
228       Join::When(put).Do([](unique_ptr<DataMessage<int>> msg) {
229         cout << **(msg->data) << " -- 0" << endl;
230       });
231
232       /* create a pattern so that if there is a message on put and get then reply to get */
233       Join::When(put).And(get).Do([](unique_ptr<DataMessage<int>> put, unique_ptr<
              ↪ReplyMessage<int>> get) {
234         (*(get->reply))(move(*(put->data)));
235       });
236
237       write(get, make_unique<ReplyMessage<int>>([](unique_ptr<int> msg) mutable {
238          cout << *msg << " -- 2" << endl;
239       }));
240
241       write(put, make_unique<DataMessage<int>>(make_unique<int>(409)));
242     }
243   }
244
245   int main(int argc, char** argv)
246   {
247     SystematicTestHarness harness(argc, argv);
248     harness.run(Joins::run);
249   }
```

Appendix D

Bank Account using two-phase commit

```
use "promises"

actor Account
  var balance: U64

  // apply is
  // - either None for no pending transaction
  // - a function for a pending transaction
  var _apply: ({(Account)} | None)

  new create(opening: U64) =>
    balance = opening
    _apply = None

  // prepare the account for a deposit event
  // - respond with agree if the account isn't already part of some transaction
  be deposit(amount: U64, response: Promise[Bool]) =>
    match _apply
    | let _: None =>
      _apply = ({(account: Account) => account.balance = account.balance + amount })
      response(true)
    | let _: {(Account)} => response(false)
    end

  // prepare the account for a withdraw event
  // - respond with agree if the account isn't already part of some transaction
  // and the account has a large enough balance
  be withdraw(amount: U64, response: Promise[Bool]) =>
    match _apply
    | let _: None =>
      _apply = ({(account: Account) => account.balance = account.balance - amount })
      response(amount < balance)
    | let _: {(Account)} => response(false)
    end

  // clear the pending event
  be abort() =>
    _apply = None

```

```
  // commit the pending event
  be commit() =>
    match _apply
    | let fn: ({(Account)}) => fn(this)
    end

actor TransactionCoordinator
  be transfer(acc1: Account, acc2: Account, amount: U64, success: Promise[Bool]) =>
    let p1 = Promise[Bool]
    let p2 = Promise[Bool]
    acc1.withdraw(amount, p1)
    acc2.deposit(amount, p2)

    Promises[Bool].join([p1; p2].values()).next[None]({(agree: Array[Bool]) =>
      for commit in agree.values() do
        if not commit then
          acc1.abort()
          acc2.abort()
          success(false)
          return
        end
      end
      acc1.commit()
      acc2.commit()
      success(true)
    })

actor Main
  new create(env: Env) =>
    var acc1 = Account(500)
    var acc2 = Account(300)

    var result = Promise[Bool]
    TransactionCoordinator.transfer(acc1, acc2, 50, result)
    result.next[None]({(success: Bool) =>
      env.out.print("transfer: " + success.string())
    })
```

Appendix E

ReasonableBanking

```
use "cli"
use "collections"
use "time"
use "../../util"

class iso Banking is AsyncActorBenchmark
  var _accounts: U64
  var _transactions: U64

  new iso create(accounts: U64, transactions: U64) =>
    _accounts = accounts
    _transactions = transactions

  fun box apply(c: AsyncBenchmarkCompletion, last: Bool) =>
    Coordinator(c, _accounts, _transactions)

  fun tag name(): String => "Banking 2PC"

actor Coordinator
  let _accounts: Array[Account]
  let _bench: AsyncBenchmarkCompletion
  var _count: U64
  let _tellers: U64

  new create(c: AsyncBenchmarkCompletion, accounts: U64, transactions: U64) =>
    _bench = c
    _accounts = Array[Account]
    _count = 0

    let initial = F64.max_value() / ( accounts * transactions ).f64()
    for i in Range[U64](0, accounts) do
      _accounts.push(Account(i, initial))
    end

    _tellers = 2
    for t in Range[U64](0, _tellers) do
      let accs: Array[Account] iso = Array[Account]
      for a in _accounts.values() do
        accs.push(a)
```

```
      end
      Teller(this, initial, consume accs, transactions)
    end

  be done() =>
    if (_count = _count + 1) == (_tellers - 1) then
      _bench.complete()
    end

actor Manager
  var _commit: Bool
  var _waiting : U8
  let _teller: Teller
  let _accounts: Array[Account]

  new create(teller: Teller, waiting: U8) =>
    _commit = true
    _waiting = waiting
    _accounts = Array[Account]
    _teller = teller

  fun ref _decide(account: Account, commit: Bool) =>
    _accounts.push(account)
    if (_waiting = _waiting - 1) == 1 then
      if _commit and commit then
        for a in _accounts.values() do
          a.commit()
        end
      else
        for a in _accounts.values() do
          a.abort()
        end
      end
      _teller.completed()
    else
      _commit = _commit and commit
    end

  be yes(account: Account) => _decide(account where commit = true)

  be no(account: Account) => _decide(account where commit = false)

actor Teller
  let _coordinator: Coordinator
  let _initial_balance: F64
  let _transactions: U64

  var _spawned: U64

  let _random: SimpleRand

  var _completed: U64
  var _accounts: Array[Account]

  var _acquired: (None | U64)
  var _pending: (None | (U64, U64))

```

```
97  new create(coordinator: Coordinator, initial_balance: F64, accounts: Array[Account]
        ↪ iso, transactions: U64) =>
98    _coordinator = coordinator
99    _initial_balance = initial_balance
100   _transactions = transactions
101   _spawned = 0
102   _random = SimpleRand(123456)
103   _completed = 0
104   _accounts = consume accounts
105
106   _acquired = None
107   _pending = None
108
109   _next_transaction()
110
111 be _next_transaction() =>
112   match _pending
113     | let _: None =>
114       if _spawned < _transactions then
115         let source = _random.nextInt(where max = _accounts.size().u32() - 1).u64()
116         var dest = _random.nextInt(where max = _accounts.size().u32() - 1).u64()
117
118         while source == dest do
119           dest = _random.nextInt(where max = _accounts.size().u32() - 1).u64()
120         end
121
122         try
123           _accounts(source.min(dest).usize())?.acquire(this)
124         end
125         _pending = (source, dest)
126       end
127     | (let source: U64, let dest: U64) =>
128       try
129         _accounts(source.min(dest).usize())?.acquire(this)
130       end
131   end
132
133 fun ref _reply(index: U64) =>
134    match _acquired
135      | let _: None =>
136        match _pending
137          | (let source: U64, let dest: U64) =>
138            _acquired = index
139            try
140              _accounts(source.max(dest).usize())?.acquire(this)
141            end
142        end
143      | let acc: U64 => true
144        match _pending
145          | (let source: U64, let dest: U64) =>
146
147            let manager = Manager(this, 2)
148            let amount = _random.nextDouble() * 1000
149            try
150              _accounts(source.usize())?.credit(amount, manager)
151              _accounts(dest.usize())?.debit(amount, manager)
152            end
```

```
            _spawned = _spawned + 1

            _acquired = None
            _pending = None
            _next_transaction()
          end
        end

  be yes(index: U64) => _reply(index)

  be completed() =>
    _completed = _completed + 1

    if _completed == _transactions then
      _coordinator.done()
    end

class DebitMessage
  let amount: F64
  let manager: Manager

  new create(amount': F64, manager': Manager) =>
    amount = amount'
    manager = manager'

class CreditMessage
  let amount: F64
  let manager: Manager

  new create(amount': F64, manager': Manager) =>
    amount = amount'
    manager = manager'

type StashToken is (DebitMessage | CreditMessage)

actor Account
  let index: U64
  var balance: F64

  var rollback: F64

  var stash: Array[StashToken]
  var stash_mode: Bool

  var acquire_stash: Array[Teller]
  var acquired: Bool

  new create(index': U64, balance': F64) =>
    index = index'
    balance = balance'

    stash = Array[StashToken]
    stash_mode = false

    acquire_stash = Array[Teller]
    acquired = false

```

```
    rollback = 0

  be unstash() =>
    try
      match stash.shift()?
        | let m: CreditMessage => _credit(m.amount, m.manager)
        | let m: DebitMessage => _debit(m.amount, m.manager)
      end
    else
      stash_mode = false
    end

  be acquire(teller: Teller) =>
    if acquired then
      acquire_stash.push(teller)
    else
      acquired = true
      teller.yes(index)
    end

  fun ref _release() =>
    acquired = false
    try
      match acquire_stash.shift()?
        | let t: Teller => t.yes(index)
      end
    end

  be commit() =>
    rollback = 0
    unstash()

  be abort() =>
    balance = balance - rollback
    rollback = 0
    unstash()

  fun ref _debit(amount: F64, manager: Manager) =>
    if balance >= amount then
      balance = balance - amount
      rollback = -amount
      manager.yes(this)
    else
      rollback = 0
      manager.no(this)
    end

  fun ref _credit(amount: F64, manager: Manager) =>
    balance = balance + amount
    rollback = amount
    manager.yes(this)

  be debit(amount: F64, manager: Manager) =>
    if not stash_mode then
      stash_mode = true
      _debit(amount, manager)
    else
```

```
      stash.push(DebitMessage(amount, manager))
    end
    _release()

  be credit(amount: F64, manager: Manager) =>
    if not stash_mode then
      stash_mode = true
      _credit(amount, manager)
    else
      stash.push(CreditMessage(amount, manager))
    end
    _release()
```