Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Dissecting Behaviour-Oriented Concurrency

Author:
Pranav Bansal

Supervisor:
Prof. Philippa Gardner

Second Marker:
Dr. Nicolas Wu

Submitted in partial fulfillment of the requirements for the MSc degree in of Imperial College London

27th June 2023

Abstract

Reasoning about programs is hard. Reasoning about concurrent programs is harder.

Behaviour-Oriented Concurrency (BoC) is a novel programming paradigm that tackles the challenges of concurrent programming by emphasizing the creation and coordination of ordered atomic units of work. BoC provides guarantees like deadlock and data-race freedom, restricted determinism, and flexible coordination to ensure correct and efficient execution of concurrent programs while mitigating common concurrency issues.

Reasoning about BoC programs is crucial to ensure their correctness and reliability in the face of non-determinism and complex control flows. It enables the detection and prevention of concurrency-related bugs, enhancing software quality and robustness.

Compiling BoC programs to an intermediate language offers a valid approach for reasoning about program correctness. The intermediate language serves as a formal and abstract representation, facilitating the application of static analysis and formal verification techniques. By operating on this intermediate language, analysis tools can systematically reason about BoC program behaviour, detecting potential issues and preserving desired properties.

In this thesis, we introduce GIL+ as a target language for BoC programs. GIL+ incorporates the necessary constructs and semantics to capture the essence of BoC, facilitating the analysis and verification of BoC programs using formal methods. We provide a translation from BoC to GIL+ and establish its soundness.

Acknowledgements

I am deeply grateful to my supervisor, Prof. Philippa Gardner, for her invaluable feedback, guidance and enthusiasm throughout this endeavour. Your mentorship has given me insight into the world of academia, and I am thankful for that.

I would also like to thank Luke Cheeseman and Prof. Sophia Drossopoulou of Imperial College London and Matthew Parkinson of Microsoft Research for their invaluable insights into BoC.

I would like to thank my friends at Imperial College London for their camaraderie throughout this journey. Your friendship has brought joy, laughter, and a sense of belonging, making my time at Imperial truly memorable.

I am deeply grateful to my family, especially my parents and my sister Manya, for their unconditional love and unwavering faith in me. Your support and guidance have shaped the person I am today, and I cannot imagine my life without you.

Contents

List of Code Listings

List of Definitions

Chapter 1

Introduction

Programming languages such as Golang, Node.js and Elixir mark a shift towards the use of asynchronous concurrency in programming languages [1]. This recent affinity to asynchronous patterns can be attributed to the rise of multicore processors and the need to make efficient use of system resources. Asynchronous programming allows for better performance and scalability in systems by having multiple tasks be executed simultaneously without blocking the execution of other tasks. These advantages of asynchronous programs make them a fundamental building block for the cloud computing services offered by the likes of Amazon and Microsoft.

However, writing asynchronous programs comes with its challenges. The management of multiple concurrent tasks and the potential for race conditions make code structuring and reasoning about program correctness more complex. Unlike in synchronous programs, where the flow of control follows a predetermined order, asynchronous programs introduce non-determinism due to parallel execution and external factors, making the flow of control harder to predict.

To address the challenges of asynchronous programming, Project Verona[1] introduced Behaviour-Oriented Concurrency (BoC). BoC focuses on the asynchronous creation of ordered atomic units of work while providing guarantees such as *deadlock freedom*, *data-race freedom*, *restricted determinism* (a predetermined partial order of execution) and *flexible coordination* (access to shared resources such as memory). These guarantees aim to ensure the correct and efficient execution of concurrent programs, mitigating common concurrency issues.

Just like any other concurrency paradigm, proving total correctness of a BoC program is a non-trivial task. Traditional testing approaches fall short for code that utilizes asynchronous concurrency because such patterns add ambiguity to the control flow, introduce non-determinism and give rise to *heisenbugs*[2]. The following quote by Edsger W. Dijkstra perfectly sums up the limitations of traditional software testing for sequential and concurrent programs alike:

> *Testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence*

The question still remains, "How does one reason about correctness in a concurrent program?". Machine-aided software verification offers a promising solution to address the correctness challenges in concurrent programming. This approach, initially proposed by Alan Turing in 1949 [3], leverages formal methods and automated analysis techniques to prove the correctness of programs. Notable work by Hoare [4] and O'Hearn et al. [5, 6] laid the foundations for machine-aided verification, leading to the development of Gillian— a multi-language platform for the development of compositional symbolic analysis tools [7, 8, 9, 10].

Gillian enables the analysis and verification of programs written in a source language by translating them into a target language (TL) and performing analysis on the TL. The target language used in Gillian is called the Gillian Intermediate Language (GIL).

[1] a collaboration between Microsoft Research and Imperial College London (ICL)[2]
[2] a software bug that seems to disappear or alter its behaviour when one attempts to study it.

GIL serves as an intermediate representation that allows Gillian to support the analysis and verification of sequential programs written in languages such as C, JavaScript, and WISL (a simple while language). By translating programs into GIL, Gillian is able to provide a formal and unified framework for reasoning about program properties and conducting various analyses.

The use of a target language or intermediate language is a common approach in software verification tools. Tools like Facebook Infer[11], Verifast [12, 13] and Viper [14] also employ this approach, each with their own respective target languages. These target languages provide a formal and abstract representation of programs, enabling the application of static analysis techniques and formal verification methods. By utilizing a target language, these verification tools can leverage formal reasoning and analysis techniques to detect errors, prove program correctness, and perform various forms of program analysis.

Building on the success of Gillian in reasoning about program correctness using target languages, this Master's Thesis proposes a novel target language called GIL+ for the translation of BoC programs.

1.1 Contributions

In achieving our key objective of creating a new TL, we make the following contributions:

- **A formal model for MiniBoC**: In Section 4.2, we introduce small-step operational semantics for a simple language that is extended with Behaviour oriented Concurrency, MiniBoC.
- **A formal model for GIL+**: In Section 5.4, we introduce small-step operational semantics for our newly proposed Target Language, GIL+.
- **A sound translation from MiniBoC to GIL+**: In Section 5.5, we provide the translation from MiniBoC to GIL+. We also prove that is translation is sound in Chapter 6.
- **FlexibleBoC**: In Section 5.6.3, we propose an alternate BoC paradigm named FlexibleBoC that is well suited for resource contention and utilization.

1.2 Challenges

Throughout this project, we encountered several significant challenges that required careful consideration and problem-solving. Three primary challenges stood out:

- **Understanding BoC**: The Behaviour-Oriented Concurrency (BoC) paradigm presented a notable challenge due to its inherent complexities and the evolving nature of its formal model during the course of this thesis. BoC is not yet widely adopted, which made it necessary to delve deep into the existing literature and engage with the research community to grasp its intricacies. This involved understanding the theoretical foundations, design principles, and practical implications of BoC to ensure an accurate and comprehensive representation in the target language.
- **Adapting to Evolving Formal Model of BoC**: As BoC is still an emerging paradigm, its formal model underwent revisions and changes based on ongoing research during the course of this project.
- **Designing Sensible Small Step Operational Semantics**: One of the key contributions of this project was the creation of a new target language, GIL+, for translating MiniBoC programs. Designing the small-step operational semantics for GIL+ posed a significant challenge, as it required constructing a clear and intuitive set of rules that effectively captured the essential aspects of the BoC paradigm. This involved careful consideration of the ordering of operations, coordination mechanisms, and ensuring the semantics aligned with the intended behaviour of MiniBoC programs. Striking the right balance between simplicity, expressiveness, and capturing the essence of BoC was a non-trivial task that required iterative refinement and validation.

Chapter 2

Background

In this chapter, we delve into the foundational logics that serve as essential resources for creating a new Target Language. We begin by exploring Hoare Logic, a widely-used logic for program verification that allows for reasoning about pre- and post-conditions of program statements. Hoare Logic provides a framework for specifying and verifying program correctness, making it a valuable reference for analysing and understanding translated programs in the Target Language.

Next, we examine Separation Logic, a logic extension of Hoare Logic that enables reasoning about programs with dynamically allocated memory. Separation Logic introduces the notion of spatial separation to reason about the state of memory and the interactions between different program components. We then move on to Concurrent Separation Logic, which extends Separation Logic to handle concurrent programs. Concurrent Separation Logic provides mechanisms for reasoning about shared resources, synchronization, and thread interactions, making it a fundamental logic for reasoning about concurrent programs.

Furthermore, we provide an overview of Gillian and its associated intermediate language, GIL. Understanding the capabilities and features of Gillian and GIL gives additional context to the reader about the existing Gillian toolchain. Additionally, we provide an introduction to WISL, a simple while language.

2.1 Hoare Logic

In 1969, Sir Tony Hoare developed a formal system for reasoning about the correctness of computer programs aptly named *Hoare Logic*[1] (HL) [4]. HL is based on the use of first-order logic assertions that can be used to describe the state of a program. Using these assertions, one could define pre-conditions P and post-conditions, Q which represent the state of a program before and after some command $\mathtt{C}$ is executed. This program specification is denoted by a *Hoare Triple*:

$$\vdash \{P\}\ \mathtt{C}\ \{Q\}$$

The semantic interpretation of which is:

> Given a logical state satisfying pre-condition P, the program does not fault. If the program terminates, it results in a logical state satisfying post-condition Q. [16]

Using a set of rules for manipulating Hoare triples, HL can be used to prove program correctness as a safety property ("bad things don't happen" [17]). One such rule is the sequential composition rule, which highlights the composable nature of Hoare Triples. The rule is given as follows:

$$\frac{\vdash \{P\}\ \mathtt{C}_1\ \{Q'\} \qquad \vdash \{Q'\}\ \mathtt{C}_2\ \{Q\}}{\vdash \{P\}\ \mathtt{C}_1; \mathtt{C}_2\ \{Q\}}\ \text{SEQ}$$

Since HL is sound and complete, if HL is unable to prove program correctness, then the program is guaranteed to be incorrect.

[1] sometime's referred to as Floyd-Hoare logic due to significant contributions made by Robert W. Floyd's in [15]

2.2 Separation Logic

Although HL provides a robust framework for verifying program correctness, it can be challenging to use as the complexity of programs increases, especially when dealing with programs with memory allocated on the heap. As a solution to these shortcomings, Peter O'Hearn, John Reynolds and Hongseok Yang and came up with *Separation Logic* (SL) as an extension of Hoare Logic [5]. Hoare et al. identified that describing the complete state of the program becomes difficult when working with complex data structures. Usually, operations on a data structure involve mutating isolated parts of the structure while ignoring the rest. This ability to reason about the program heap *locally* rather than *globally*[2] is a key feature of SL.

Heap cells (Section 2.2.1) assertions, the spatial connective '$\star$' (Section 2.2.2) along with the frame rule (Section 2.2.3) allow for *local reasoning* allowing SL to scale while avoiding the unwieldy nature of HL.

2.2.1 Heap Cell Assertions

HL is based on first-order logic, which means that the reasoning does not scale well for heap-manipulating programs. SL introduced the following heap cell assertion to allow for reasoning about the heap:

$$x \mapsto y$$

This denotes a single cell in the heap at address[3] x with value y. SL also introduced an empty heap cell assertion representing an empty heap denoted by the following:

$$\texttt{emp}$$

2.2.2 The Separating Conjunction

SL also introduce a spatial connective, the *separating conjunction* ($\star$), which is a logical operator used to reason about the memory layout of a program. The assertion $P \star Q$, read "P and separately Q", represents a piece of program heap that can be divided into two disjoint *heaplets* (or *heap fragments*), with one heaplet satisfying P and the other Q. For e.g., Figure 2.1 shows two distinct heaps that reference each other and can be represented by the assertion $x \mapsto y \star y \mapsto x$.

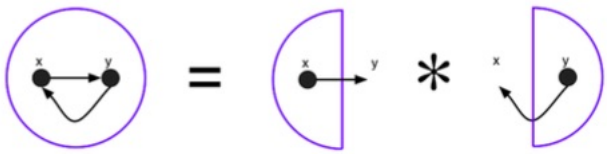

Figure 2.1: A visualization of the separating conjunction [18]

This separating conjunction can be used to define various different data structures, for e.g. one could define a *list* predicate like so:

$$list(x) \stackrel{\text{def}}{=} (x \doteq \texttt{null}) \vee (\exists v. x \mapsto v \star list(v))$$

This recursive-like structure when defining predicates in SL is quite common. The dot above the equals sign in the represents an empty heap and can be used with any another operator.

Some properties of the separating conjunction are as follows:

$$\begin{aligned}
\vdash P \star Q &\iff Q \star P && \text{(commutativity)} \\
\vdash P \star (Q \star R) &\iff (P \star Q) \star R && \text{(associativity)} \\
\vdash P \star \texttt{emp} &\iff P && \text{(identity)} \\
\vdash P_1 \wedge P_2 \star Q &\iff (P_1 \star Q) \wedge (P_2 \star Q) && \text{(distributivity over logical and)} \\
\vdash P_1 \vee P_2 \star Q &\iff (P_1 \star Q) \vee (P_2 \star Q) && \text{(distributivity over logical or)}
\end{aligned}$$

[2] the behaviour of a program is specified by the entirety of its state
[3] addresses in Sl are represented by natural numbers

Some additional notation in SL is defined for convenience:

$$x \mapsto - \stackrel{\text{def}}{=} \exists y.\ x \mapsto y$$

$$x \mapsto y_0, y_1, ..., y_n \stackrel{\text{def}}{=} x \mapsto y_0 \star x + 1 \mapsto y_1 \star ... \star x + n \mapsto y_n$$

2.2.3 Frame Rule

The *frame rule* is a crucial principle of SL that enables the temporary elimination of unnecessary logic while reasoning about a particular command, and is given as follows:

$$\frac{\vdash \{P\}\ \mathtt{C}\ \{Q\} \qquad \mathtt{mod(C)} \cap \mathtt{fv}(R) = \phi}{\vdash \{P \star R\}\ \mathtt{C}\ \{Q \star R\}}\ \text{FRAME}$$

where `mod(C)` is the set of modified variables for some command `C` and `fv(R)` represents the set of variables that free(i.e. not bound by a logical quantifier) in R.

The $\mathtt{mod(C)} \cap \mathtt{fv}(R) = \phi$ expression ensures that only logic reasoning about the heap mutated by the command is *framed off*.

2.3 Concurrent Separation Logic

2.3.1 Fundamental tenets

Reasoning about concurrent programs requires some basic terminology. This subsection goes through the basic principles that are a prerequisite to reasoning about concurrent programs using CSL.

- Parallel Process: To support reason about concurrent programs, the following notation was introduced:

  $$\mathtt{C}_1 \parallel \mathtt{C}_2$$

  which denotes the execution of two commands $\mathtt{C}_1$ and $\mathtt{C}_2$ programs in parallel by two distinct processes.

- Racy programs: For any programming model that allows concurrent processes (like threads), shared access to common state is possible. When this occurs in a program, the program is said to be *racy*. A more exact definition of racy is as follows:

  > A program is *racy* if two concurrent processes attempt to access the same portion of state at the same time. Otherwise, the program is *race-free* [19]

  Equation 2.1 shows an example of a racy program.

  $$x \coloneqq x + 1 \parallel x \coloneqq x - 1 \tag{2.1}$$

- Mutual Exclusion Group: A *mutual exclusion group* is a group of commands that are required to not overlap in their execution. A semaphore can be used to form mutual exclusion groups. A semaphore (s) is represented by a non-negative integer which has *wait*, $P(s)$, and *signal*, $V(s)$, operations. The wait operation decrements the value of the semaphore when it is greater than 0 and the signal increments the value.

- Daring programs: A program is *cautious* if, whenever concurrent processes access the same piece of state, they do so only within commands from the same mutual exclusion group. Otherwise, the program is *daring*. For e.g., the following program is daring as it accesses the same piece of state (heap address 10) in different mutual exclusion groups.

  $$\mathtt{semaphore}\ free \coloneqq 1, busy \coloneqq 0$$

P($free$)		P($busy$)
[10] $\coloneqq$ x	$\parallel$	y $\coloneqq$ [10]
V($busy$)		V($free$)

2.3.2 What is CSL?

Concurrent Separation Logic (CSL) was introduced by Peter O'Hearn as an extension of Separation Logic to reason about concurrent and parallel programs [19]. CSL states that if $\{P\}\ \mathtt{C}\ \{Q\}$ holds, then any execution of $\mathtt{C}$ starting from a state satisfying P will not result in a race condition or attempt to access a dangling pointer [20]. CSL makes use of two key concepts - *ownership* and *separation*

Ownership Hypothesis The ownership hypothesis states that each process has exclusive ownership of the memory it creates or modifies, and that the memory can be accessed by other processes only if it is explicitly shared. In languages like Rust [21] this concept of ownership is a part of the programming model, but that is not necessary for using CSL. The hypothesis is formally stated as follows:

> *A code fragment can only access those portions of the state that it owns*

Separation Hypothesis The separation hypothesis states that the memory of a program can be divided into disjoint regions, where each region corresponds to a heaplet which can be reasoned about independently of others. The hypothesis is formally stated as follows

> *At any time, the state can be partitioned into that owned by each process and each grouping of mutual exclusion*

It is important to note that the state partition caused by the separation property is not static and can change over time.

2.3.3 An intuitive understanding

This section aims to provide an intuitive understanding of CSL with the help of Listing 2.1.

$\{\mathtt{emp}\}$		$\{\mathtt{emp}\}$
P(free)		P(busy)
$\{10 \mapsto -\}$		$\{10 \mapsto -\}$
$[10] \coloneqq x$	$\parallel$	$y \coloneqq [10]$
$\{10 \mapsto -\}$		$\{10 \mapsto -\}$
V(busy)		V(free)
$\{\mathtt{emp}\}$		$\{\mathtt{emp}\}$

Listing 2.1: A Binary semaphore example

In the given code snippet, the heap address cell 10 is *attached* (intuitively) to semaphores. The assertions (in blue) in this code snippet represent the local state from the point of view of that process. The P(s) operations for a semaphore s transfers the ownership of heap address 10 to the process, and the V(s) takes back the ownership from the process. The ownership of the heap address is passed around between the semaphores and the processes. At any given time, this heap address is owned by either owned by exactly one of the processes or exactly one of the semaphores. The logical attaching of a resource to the semaphores and their ability to transfer ownership allows for concurrent reasoning of resources modularly.

The assertion annotations implicitly contain information about ownership (or permissions). An assertion P in a process implies that the process owns P, i.e. it has the right to dereference it at that point.

2.3.4 Disjoint Concurrency

The proof rule added by O'Hearn for disjoint concurrency is as follows:

$$\frac{\vdash \{P_1\}\ \mathtt{C}_1\ \{Q_1\} \qquad \vdash \{P_2\}\ \mathtt{C}_1\ \{Q_2\}}{\vdash \{P_1 \star P_2\}\ \mathtt{C}_1 \parallel \mathtt{C}_2\ \{Q_1 \star Q_2\}}\ \textsc{Disj-Par}$$

where $\mathtt{C}_1$ does not modify any free variables in P_1, Q_1 or $\mathtt{C}_1$ and conversely $\mathtt{C}_2$ does not modify any free variables in P_2, Q_2 or $\mathtt{C}_2$.

This rule can not be used to reason about racy programs, as there will always be information that won't be broken down into 2 disjoint heaps. Some intuition for this can be seen when reasoning about Equation 2.2 with pre-condition $\{10 \mapsto -\}$. To use the disjoint concurrency rule, $10 \mapsto -$ needs to be represented as $P_1 \star P_2$ which is never possible.

$$[10] \coloneqq 0 \parallel [10] \coloneqq 1 \tag{2.2}$$

2.3.5 Process Interaction

Listing 2.2 shows how a program is defined in CSL. A program consists of multiple resources (r_i is a variable list) and multiple processes ($\mathtt{C}_i$ is a command that will be run concurrently)

```
init
resource r_1, ..., r_m
C_1 || ... || C_2
```

Listing 2.2: A program in CSL

CSL uses *conditional critical regions* (CCRs) to allow reasoning about process interaction and the command for accessing them is as follows:

$$\texttt{with } r \texttt{ when } B \texttt{ do C endwith}$$

r is a resource, B is a boolean expression and `C` is a command. The `with` command represents a unit of mutual exclusion, and two `with` commands for the same resource can not run in parallel. `C` is executed if no other *region* for r is executing and if B is true. Variables and resources in CSL abide by the following rules:

(1) A variable belongs to at most one resource

(2) If a variable belongs to a resource r, then it cannot appear in a process except for in the critical region for r

(3) If a variable is mutated in one process, it cannot appear in another process unless it belongs to a resource.

These rules prevent variables from receiving interference from different processes. (1) and (2) imply that variables owned by resources can only be accessed from the critical region of a resource. Since regions are mutually exclusive, there is no interference here. Variables that are not owned by resources and are local to the thread can not be interfered with by another process due to (3).

A resource invariant formula is introduced for each resource (RI_r is the resource invariant for r). This resource invariant satisfies that any command changing a variable x ($x \coloneqq ...$) must be in a critical region for a resource. All resource invariants also have to be precise[4].

The inference rule in CSL is:

$$\frac{\{P\}\ \mathit{init}\ \{RI_{r_1} \star ... \star RI_{r_m} \star P\} \qquad \{P'\}\ \mathtt{C}_1 \parallel ... \parallel \mathtt{C}_n\ \{Q\}}{\{P\}\ \mathit{init}; \texttt{resource } r_1, ..., r_m; \mathtt{C}_1 \parallel ... \parallel \mathtt{C}_n\ \{RI_{r_1} \star ... \star RI_{r_m} \star Q\}}$$

The interference rule for parallel composition is:

$$\frac{\{P_1\}\ \mathtt{C}_1\ \{Q_1\} \qquad ... \qquad \{P_n\}\ \mathtt{C}_n\ \{Q_n\}}{\{P_1 \star ... \star P_n\}\ \mathtt{C}_1 \parallel ... \parallel \mathtt{C}_n\ \{Q_1 \star ... \star Q_n\}}$$

where no free variables in P_i or Q_i are changed in $\mathtt{C}_j$ when $j \neq i$. This rule is analogous to the disjoint concurrency rule introduced in Section 2.3.4.

The critical region rule in CSL is:

$$\frac{\{(P \star RI_r) \wedge B\}\ \mathtt{C}\ \{Q \star RI_r\}}{\{P\}\ \texttt{with } r \texttt{ when } B \texttt{ do C endwith}\ \{Q\}}$$

with the side condition that no other process modifies free variables in P or Q.

[4] an assertion P is precise if for all states (s, h) there is at most one heaplet $h' \subseteq h$ where $s, h \vDash P$

2.4 An overview of Gillian

Building upon the seminal work done by Hoare [4] and by O'Hearn et al. [5, 6] in the field of software verification, the Verified Software Research Group at ICL developed a symbolic analysis tool – Gillian [7, 8, 9, 10]. Gillian is a multi-language platform for symbolic analysis which supports three types of program analysis:

- *whole program symbolic testing* where users define a test suite of unit tests with symbolic inputs and outputs along with some constraints on them and Gillian attempts to find symbolic states that cause any of the test suites to fail
- *full verification* where users annotate function pre- and post-conditions and loop invariants. Gillian symbolically executes the program to ensure that the post-conditions for all the functions hold.
- *automatic compositional testing* where only provide the source code for a program. Gillian, using bi-abduction logic [23], deduces the SL assertion annotations and attempts to verify them. This is similar to Facebook's Infer tool [11, 24].

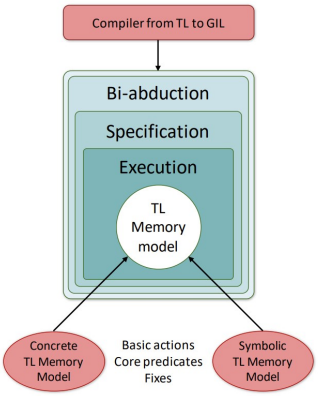

Figure 2.2: Overview of Gillian [22]

The key advantage that Gillian has over other tools is that it has a *parametric memory model*. The memory model is parametric with respect to a target language (TL). This allows Gillian to support a variety of languages with different memory models with the same backend. Given a compiler from TL to GIL (Gillian's intermediate representation) and the TL's memory model (defined in OCaml), Gillian can perform program analysis on programs written in the TL [8].

2.5 An overview of GIL

GIL is a simple goto language used as the intermediate representation in Gillian. GIL is parametric on a set of *memory actions*, $\mathcal{A}$, that is dependent on the TL. This set of memory actions describes the fundamental ways in which TL programs interact with memory.

2.5.1 Values

GIL values ($v \in \mathcal{V}$) are numbers, strings, booleans, uninterpreted symbols, types, procedure identifiers and lists of values. In GIL, uninterpreted symbols are primarily used to denote memory locations.

$$
\begin{array}{lll}
v \in \mathcal{V} \stackrel{\text{def}}{=} & & \\
& n \in \mathbb{R} & \text{(Numbers)} \\
& \mid s \in \mathcal{S} & \text{(Strings)} \\
& \mid b \in \mathcal{B} & \text{(Booleans)} \\
& \mid u \in \mathcal{U} & \text{(Uninterpreted Symbols)} \\
& \mid t \in \mathcal{T} & \text{(Types)} \\
& \mid p \in \mathcal{P} & \text{(Procedure Identifiers)} \\
& \mid \overline{v} \in \text{List}(\mathcal{V}) & \text{(Lists)}
\end{array}
$$

2.5.2 Expressions

There are two types of expressions in GIL – simple expressions ($e \in \mathcal{E}$) and symbolic expressions ($\hat{e} \in \hat{\mathcal{E}}$). GIL Expressions consist of values, program variables, unary operators and binary operators. GIL symbolic expressions are analogous.

$$
\begin{array}{l}
e \in \mathcal{E} \stackrel{\text{def}}{=} \\
\quad v \in \mathcal{V} \qquad \text{(Numbers)} \\
\quad \mid x \in \mathcal{X} \qquad \text{(Program Variables)} \\
\quad \mid \ominus e \qquad \text{(Unary Operator)} \\
\quad \mid e_1 \oplus e_2 \qquad \text{(Binary Operator)}
\end{array}
\qquad
\begin{array}{l}
\hat{e} \in \hat{\mathcal{E}} \stackrel{\text{def}}{=} \\
\quad \hat{v} \in \hat{\mathcal{V}} \qquad \text{(Numbers)} \\
\quad \mid \hat{x} \in \hat{\mathcal{X}} \qquad \text{(Program Variables)} \\
\quad \mid \ominus \hat{e} \qquad \text{(Unary Operator)} \\
\quad \mid \hat{e_1} \oplus \hat{e_2} \qquad \text{(Binary Operator)}
\end{array}
$$

2.5.3 Commands

GIL commands ($c \in C_{\mathcal{A}}$) consist of a variable assignment, conditional goto, function call, memory actions, allocation of uninterpreted/interpreted symbols, function return and error termination.

$$
\begin{array}{ll}
c \in \mathcal{C}_{\mathcal{A}} \stackrel{\text{def}}{=} & \\
\quad x \coloneqq e & \text{(Variable assignment)} \\
\quad \mid \text{goto } i & \text{(Unconditional goto)} \\
\quad \mid \text{ifgoto } e\ i\ j & \text{(Conditional goto)} \\
\quad \mid x \coloneqq e(e') & \text{(Procedure call)} \\
\quad \mid x \coloneqq \alpha(e) & \text{(Action execution)} \\
\quad \mid x \coloneqq \text{uSym/iSym}(e) & \text{(Uninterpreted/interpreted Symbol allocation)} \\
\quad \mid \text{return } e & \text{(Normal return)} \\
\quad \mid \text{fail } e & \text{(Error return)}
\end{array}
$$

2.5.4 Procedures

A GIL procedure ($f(\overline{x})\{\overline{c}\}$) consists of an identifier $f \in \mathcal{F}$, a list of parameters $\overline{x}$, and a body described by a list of commands $\overline{c}$. A set of GIL procedures with unique identifiers form a GIL program ($\mathtt{p}$).

$$proc \in \mathcal{P}roc_{\mathcal{A}} \stackrel{\text{def}}{=} f(\overline{x})\{\overline{c}\}$$

$$\mathtt{p} \in \mathcal{P}rog_{\mathcal{A}} : \mathcal{P} \rightharpoonup \mathcal{P}roc_{\mathcal{A}}$$

2.6 An overview of WISL

WISL is a simple while-based programming language used for research and educational purposes. It has a memory model composed of blocks, each of which is identified by a unique location. These blocks are described by:

- a list of contiguous memory cells that map numerical offsets to values stored in the heap
- a natural number bound that determines the size of the block

2.6.1 Informal Memory Model

WISL has the capability to track negative resources, which occur during allocation or deallocation. Negative information arises during allocation when a block with a specific size can not have offsets beyond that size (which is captured by the bound). During deallocation, the negative information that only complete blocks can be deleted is captured by the fact that a memory block can either map to a list of cells or a dedicated freed symbol.

The SL assertions language is extended with the following core predicates for WISL:

- bound(E_1, E_2) states that E_1 is a block pointer which points to a block of length E_2
- $E \mapsto \phi$ represents that E is a block pointer which, along with its entire block, has been freed

- $E \mapsto_b E_1, E_2$ which is just syntactic sugar for $E \mapsto E_1 \star E + 1 \mapsto E_2 \star \text{bound}(E, 2)$

2.6.2 Syntax

The WISL syntax is described as follows:

- **Assignment**: `x` $\coloneqq$ `E` assigns the expression `E` to variable `x` in the variable store.
- **Lookup**: `x` $\coloneqq$ `[E]` assigns the value present at heap address `E` to variable `x` in the variable store
- **Mutation**: `[E1]` $\coloneqq$ `E2` stores the value `E2` at heap address `E1`
- **Allocation**: `x` $\coloneqq$ `new(n)` allocates a new memory block consisting of `n` cells
- **Deallocation**: `free(E)` frees the memory block starting at address `E`
- **Conditional**: `if (B)` $\{$`C`$_1\}$ `else` $\{$`C`$_2\}$ executes `C`$_1$ if `B` is true otherwise executes `C`$_2$
- **No-op**: `skip` does nothing

Chapter 3

A New Concurrency Paradigm: Behaviour Oriented Concurrency

BoC is a concurrency paradigm that has stemmed from a collaboration between Microsoft Research and Imperial College London.

This chapter elucidates the fundamentals of concurrency and concurrency paradigms, subsequently delving into Behaviour Oriented Concurrency (BoC).

3.1 What is a Concurrency Paradigm?

Concurrency refers to the ability of a program to execute multiple tasks simultaneously, potentially overlapping or running in parallel. It represents a fundamental aspect of contemporary software development, facilitating the efficient utilization of resources and enhancing overall performance.

A concurrency paradigm denotes a specific approach or model employed in the design and implementation of concurrent programs. It encompasses a collection of principles, concepts, and constructs that empower developers to systematically reason about and manage concurrent execution. By adhering to a concurrency paradigm, developers can effectively address the challenges associated with concurrent programming and ensure the reliable and efficient operation of their software systems.

Concurrency paradigms encompass two fundamental aspects: **parallelism** and **coordination**. Parallelism refers to the ability of a program to execute multiple tasks simultaneously, leveraging the available computing resources for enhanced performance. It enables the distribution of workloads across different units of work, such as threads and processes, enabling efficient utilization of resources.

In contrast, coordination plays a pivotal role in managing the interaction and synchronization between concurrent tasks to ensure correct and predictable behaviour. It entails establishing mechanisms for data sharing, task synchronization, and communication. Effective coordination is essential for avoiding data races, which occur when multiple concurrent tasks access shared data simultaneously, potentially leading to inconsistent or erroneous results. By enforcing synchronization and orderly access to shared resources, coordination mechanisms mitigate the risk of data races and maintain data integrity.

Therefore, a concurrency paradigm must strike a balance between parallelism and coordination, harnessing the power of parallel execution while ensuring synchronization and coherence among concurrent tasks. By effectively managing both aspects, developers can design concurrent programs that are both efficient and reliable.

3.2 Triangle of Concurrency

Most concurrency paradigms make the trade-off between Safety, Scalability and allowing Concurrent Mutation [25]. Figure 3.1 shows the *triangle of concurrency* and where different concurrency paradigms lie on the triangle.

Most fundamental concurrency paradigms lack inherent memory safety, as they do not address the challenges posed by Concurrent Mutation, delegating this responsibility to the programmer. Ensuring safety in the presence of concurrent mutation becomes considerably more challenging and necessitates the introduction of some global synchronization mechanisms. Typically, this involves incorporating a garbage collector which adds a significant performance overhead, negatively impacting scalability. Recently, however, a shift has been observed in several concurrency paradigms, with the adoption of ownership-based approaches. These paradigms leverage ownership to provide both scalability and memory safety, marking a significant advancement in concurrent programming techniques.

The paradigm shift towards giving increased importance to scalability and memory safety in concurrency programming is strongly influenced by the emergence of cloud computing. As cloud-based systems and distributed computing platforms become prevalent, the need for concurrent programs that can scale efficiently and ensure memory safety becomes paramount.

3.3 Why do we need a new concurrency paradigm?

The emergence of languages like Rust has demonstrated that ownership-based approaches can gain popularity and receive positive feedback from developers [26]. Rust's ownership system provides memory safety guarantees and improves concurrency by enforcing strict ownership rules and eliminating data races. This success showcases the viability and effectiveness of ownership as a concurrency paradigm.

Another existing concurrency paradigm that incorporates elements of ownership is the actor model introduced by Agha [27]. Actors, as independent units of execution, encapsulate state and behaviour, resembling ownership over their internal resources. A key benefit of the actor model is that it combines coordination and parallelism constructs within a single framework, making it simpler for the developer to reason about programs. However, coordinating actions across multiple actors presents inherent complexities. It is commonly acknowledged that actors are not well-suited for performing atomic operations that involve multiple actors [28, 29].

Consider the scenario of implementing transactions across multiple tables, with each table represented by an actor. Coordinating these actors to ensure the atomicity, consistency, isolation, and durability (ACID) properties of transactions becomes rather challenging [30]. To achieve transactional coordination, various issues must be addressed. Ensuring all actors reach a consistent state requires careful orchestration and synchronization mechanisms. Coordinating the order of operations, handling concurrent updates or conflicts, and maintaining isolation boundaries are complex tasks. Additionally, failure handling and recovery pose further difficulties. When an actor repres-

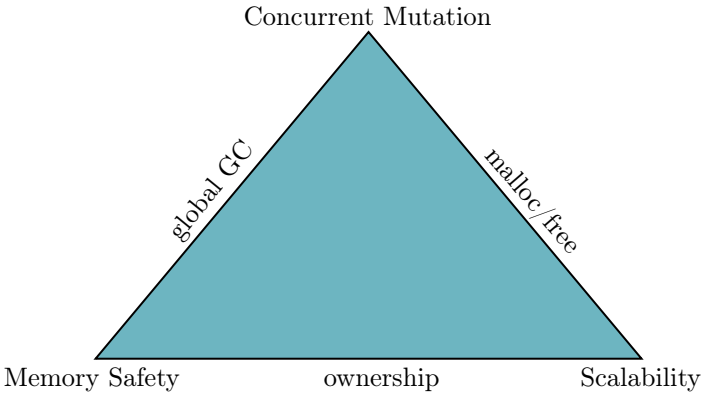

Figure 3.1: The triangle of concurrency [25]

enting a table fails during a transaction, ensuring proper rollback or recovery mechanisms across multiple actors becomes a non-trivial task. These complexities faced while synchronising actions across multiple actors in the actor model motivate the need for a new concurrency paradigm - Behaviour oriented Concurrency (BoC).

However, in BoC, the focus is not on enhancing Actors, but rather on revisiting the fundamental principles of Actors with the aim of unifying Actors and Transactions. BoC seeks to provide a unified framework that seamlessly integrates coordination and parallelism with transactional capabilities. This approach aims to simplify the coordination of concurrent activities, particularly in scenarios where transactional consistency and atomicity are crucial.

3.4 An overview of BoC

BoC is designed to be the primary concurrency feature of an underlying programming language. BoC enriches the underlying language with two core concepts: the concurrent owner or *cown* (pronounced as "cone"), and the *behaviour*. BoC is further explored in this section through a strongly typed pseudo-language with two key features:

- The `when` expression, which is used to *spawn* behaviours (c.f. Section 3.4.2).
- The `cown[T]` type, which represents contents of type `T` and has a constructor

3.4.1 Cowns

Cowns in BoC serve as unique entry points within the underlying language, providing a means to protect and encapsulate data. A cown ensures that the data it holds can only be accessed through its designated entry point, thereby establishing a clear boundary around the protected data in the program.

Listing 3.1 shows how a cown restricts access to its contents. The variable `hello_cown` is of type `cown[String]`, and hence the attempt to access the contents of `hello_cown`, in this case the string "hello", is invalid.

```
1 main() {
2     hello_str = "hello"
3     hello_str.append(" world!")  // valid
4
5     hello_cown = cown.create("hello")
6     hello_cown.append("Psych!") // invalid!!!
7 }
```

Listing 3.1: creating a cown to restrict access to data

The only way to access the contents of a cown is to spawn a behaviour requiring that cown.

3.4.2 Behaviours

Behaviours in BoC serve as the fundamental unit of concurrent execution. The `when` keyword along with a set of cowns and a closure is used to *spawn* a behaviour. Listing 3.2 shows a contrived hello world program in BoC. On Line 4, the contents of cown `msg_cown` are bound to `msg` i.e, the contents of the cown can be accessed and mutated via the `msg` variable.

```
1 main() {
2     msg_cown = cown.create("hello world!")
3
4     when(msg = msg_cown) { // Behaviour b
5         print(msg)
6     }
7 }
```

Listing 3.2: A contrived hello world program in BoC

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* (c.f. **Definition** 3.1) it have

run. In this case, the behaviour b can start running instantly as the `msg_cown` cown is available and there are no behaviours that *happen before* it.

Once the required cowns are available, they are acquired atomically. This atomic acquisition ensures exclusive access to the cowns and their associated data throughout the execution of the closure. A behaviour cannot acquire additional cowns during its execution or release the cowns before it has terminated. Upon termination of the closure, the behaviour terminates, and all its cowns become available.

3.4.3 Behaviour Ordering

The distinction between **spawning** and **running** of behaviours is crucial in BoC. When a behaviour is spawned using the `when` keyword, the spawning process is synchronous, meaning that it occurs immediately. Behaviours are spawned without delay, allowing for the concurrent execution of multiple behaviours. Contrarily, the running of behaviours is asynchronous and is subject to certain conditions. A behaviour can only start running when:

- all the required cowns it stated at spawn time are available
- all the behaviours that *happen before* it, have completed their execution

Definition 3.1 (BoC *happens before* ordering). A behaviour b will happen before another behaviour b' iff b and b' require overlapping sets of cowns, and b spawned before b'. It is often represented as $b \prec b'$, read b *happens before* b'

Listing 3.3 shows a contrived hello world program in BoC with two behaviours, namely, b_1 and b_2. Since the spawning of behaviours is synchronous, b_1 will spawn before b_2. Causally using **Definition** 3.1, one can say $b_1 \prec b_2$.

```
main() {
    msg_cown = cown.create("hello world!")

    when(msg = msg_cown) { // b1
        print("For the first time \n")
        print(msg)
    }

    when(msg = msg_cown) { // b2
        print("This is getting old now \n")
        print(msg)
    }
}
```

Listing 3.3: A contrived hello world in BoC with 2 behaviours

In Listing 3.4, there is no ordering between behaviours b_1 and b_2 as there is no overlap between their required cowns. Hence, The program is incorrect and either string, “hello” or “ world!” can be printed first.

```
main() {
    hello_cown = cown.create("hello")
    world_cown = cown.create(" world!")

    when(hello = hello_cown) { // b1
        print(hello)
    }

    when(world = world_cown) { // b2
        print(world)
    }
}
```

Listing 3.4: Incorrect hello world in BoC with 2 behaviours

3.4.4 Nesting Behaviours

Nesting behaviours can be useful to enforce *happens before* ordering in BoC programs. Consider Listing 3.5, in which the programmer is acquiring an account to perform some operations on it

and is maintaining a sequential log. We know that b_1 will spawn before b_2 due to the synchronous nature of spawns, which further implies $b_1 \prec b_2$. Since b_2 will always run after b_1, b_3 will always spawn after b_1 and hence one can say $b_1 \prec b_3$

```
main() {
    log_cown = cown.create(Log.create())
    acc_cown = cown.create(Account.create())

    when(log = log_cown) { // b1
        print("start log")
    }

    when(acc = acc_cown) { // b2
        when(log = log_cown) { // b3
            log.append("acquired account")
        }
        ...
    }
}
```

Listing 3.5: Creating a sequential log in BoC using nesting

However, there are certain caveats that arise when nesting behaviours. The contents of a cown associated with a behaviour, say b, can not be accessed by another behaviour inside b.

Let's say, in the previous example, the programmer to print a more descriptive message to the log and then updated b_2, depicted by Listing 3.6. Due to the atomic nature of the acquisition of cowns discussed in Section 3.4.2, behaviour b_3 does not have access to the contents of `hello_cown` and hence Line 3 is invalid.

```
when(acc = acc_cown) { // b2
    when(log = log_cown) { // b3
        log.append("acquired account belonging to", acc.owner()) //invalid!!!!
    }
    ...
}
```

Listing 3.6: Incorrectlty creating a descriptive log - caveats of nesting

The correct way to create a more descriptive log in b_2 is given by Listing 3.8.

```
when(acc = acc_cown) { // b2
    owner_acc = cown.create(acc.owner())
    when(log = log_cown, owner = owner_acc) { // b3
        log.append("acquired account belonging to", owner_acc)
    }
    ...
}
```

Listing 3.7: Correctly creating a descriptive log - caveats of nesting

```
x = cown(0)

when(ix = x) {
    ix = 1 // C1
}

when(ix = x) {
    ix = 2 // C2
}
```

Listing 3.8: Correctly creating a descriptive log - caveats of nesting

3.5 Benefits of using BoC

Reasoning about concurrent programs is particularly challenging due to the presence of parallelism, which requires coordination among concurrent tasks. BoC aims to simplify the developer's experience by providing `when` as a unified construct that combines coordination and parallelism. BoC's

unified construct provides a higher-level abstraction that encapsulates both coordination and parallelism, enabling developers to reason about and manage concurrency more easily. This unified construct in BoC offers several guarantees, ensuring desirable properties for BoC programs:

- **Data race freedom**: BoC guarantees that all programs written in BoC are free of data races. This eliminates the unpredictable and erroneous behaviour that can arise from concurrent accesses to shared data without proper synchronization.
- **Deadlock freedom**: BoC ensures that programs cannot indefinitely halt due to concurrent processes waiting on each other. This prevents scenarios where multiple processes are unable to proceed, leading to system-wide stalling or deadlock situations.
- **Flexible co-ordination**: Access to multiple independent resources is allowed. Like actors, BoC decomposes state into separate entities (cowns). However, actors also require the scope of its concurrent units (behaviours) to be confined to a single entity [Agha and Hewitt 1987; Agha 1985]. With BoC, this is not the case: a behaviour can have access to multiple entities (cowns) allowing for *flexible coordination* across behaviours. This enables developers to efficiently manage and coordinate concurrent activities that involve different resources.
- **Restricted determinism**: Through the *happens before* order, BoC provides an implicit order of fine-grained concurrency units, offering a level of determinism within the concurrent program. Since this *restricted determinism* comes from the source level semantics, the developer can reason about the program correctness and program performance with relative ease.

3.6 Current state of BoC

Currently, BoC has been defined using a formal model and has been demonstrated practically through a C++ library that functions as a runtime for BoC and a C# executable model [31, 32]. The formal model for BoC (c.f. Appendix A) is defined parametric to an underlying 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 in [33].

Chapter 4

Miniature Behaviour Oriented Concurrency: MiniBoC

This chapter presents MiniBoC, a simple language that has the BoC concurrency paradigms. A formal operational semantics for MiniBoC is given and the intricacies of BoC are further explored (through MiniBoC).

The development of BoC has undergone notable changes over the course of this project. The initial formal model, referred to as the *old* formal model or operational semantics, is presented in Appendix B. However, it has been recognized that the old semantics had a significant drawback related to the isolation constraints imposed on the underlying languages (c.f. Section B.2). These isolation constraints led to a considerable increase in the complexity of the BoC operational semantics.

The recognition of this drawback prompted the need for reevaluation and refinement of the BoC semantics. The revised and improved formal model, which is the focus of this project, will be referred to as the *new* formal model or operational semantics presented in Appendix A. As mentioned in Section 3.6, this model defers the isolation guarantees to the underlying language and hence does not provide a model for cowns.

MiniBoC, developed at the same time as the new BoC operational semantics, is building off the old operational semantics. It discards the concept of an underlying language and gives operational semantics for behaviours and cowns in a unified manner.

4.1 MiniBoC Syntax

The MiniBoC language is parametrised by the following sets: the set of *values*, `Val`, ranged over by $v, v_1, ...$ with $\mathbb{N} \cup \{\texttt{true}, \texttt{false}\} \subseteq \texttt{Val}$; the set of *variables*, `Var`: ranged over by $\texttt{x}, \texttt{y}, ...$; the set of *cown identifiers* (also referred to as cowns) `Cown`, ranged over by $\kappa, \kappa_1, ...$; the set of function names, `FName`: ranged over by $f, f_1, ...$; the set of behaviour names, `BName` ranged over by $b, b_1, b_2, ...$, with distinguished element $\mu \in \texttt{BName}$. μ represents the standard value for `BName`.

Definition 4.1 (MiniBoC Expressions). The set of *expressions*, `Exp` ranged over by $E, E_1, E_2, ...$, is given by:

$$E ::= v \mid x \mid E + E \mid E < E \mid ...$$

where $v \in \texttt{Val}, \texttt{x} \in \texttt{Var}$.

Notation. ... indicates that the expressions can be arbitrarily extended as required

Definition 4.2 (Program Variables for MiniBoC Expressions). The set of program variables for

a MiniBoC expression E, denoted by, $\texttt{pv}(E)$ is defined inductively:

$$\begin{aligned}
\texttt{pv}(v) &= \phi \\
\texttt{pv}(\texttt{x}) &= \{\texttt{x}\} \\
\texttt{pv}(E_1 + E_2) &= \texttt{pv}(E_1) \cup \texttt{pv}(E_2) \\
\texttt{pv}(E_1 < E_2) &= \texttt{pv}(E_1) \cup \texttt{pv}(E_2) \\
&\ldots
\end{aligned}$$

Definition 4.3 (MiniBoC Program Stores). The program stores of MiniBoC consists of the following:

$$\text{Variable Store} : \sigma \stackrel{\text{def}}{=} \texttt{Var} \to \texttt{Val} \mid \texttt{Cown}$$

σ represents the variable store in which a variable can either map to a value or a cown identifier. We say that the variable $\texttt{x}$ has value w in the store σ if $\sigma(\texttt{x}) = w$ where w is either a value v or a cown identifier κ.

$$\text{Cown Store} : \Delta \stackrel{\text{def}}{=} \texttt{Cown} \to \texttt{Val} \times \texttt{BName}$$

Δ represents the cown store in which a cown identifier maps to a value and a behaviour name. The value represents the *contents* of that cown. The behaviour name, b represents the most recently spawned behaviour that requires this cown to run.

We say that a cown has a cown identifier κ and contents v in cown store Δ if $\Delta(\kappa) = v$.

Definition 4.4 (MiniBoC Commands). The set of MiniBoC *commands*, $\texttt{Cmd}$, ranged over by, $\texttt{C}, \texttt{C}_1, ...$, is defined by:

$$\begin{aligned}
\texttt{C} ::= {} & \texttt{y} := E && \text{(assignment)} \\
\mid {} & \texttt{C}; \texttt{C} && \text{(sequential composition)} \\
\mid {} & \texttt{if } E \texttt{ then } \{\texttt{C}\} \texttt{ else } \{\texttt{C}\} && \text{(if)} \\
\mid {} & \texttt{z} := f(\vec{E}) && \text{(function call)} \\
\mid {} & \texttt{skip} && \text{(skip)} \\
\mid {} & \texttt{x} := \texttt{cown}(E) && \text{(cown creation)} \\
\mid {} & \texttt{when } (\vec{\texttt{ix}} = \vec{\texttt{x}}) \ \{\texttt{C}\} && \text{(when)}
\end{aligned}$$

where $\texttt{x}, \texttt{y}, \texttt{z} \in \texttt{Var}, \vec{\texttt{x}}, \vec{\texttt{ix}} \in \texttt{List(Var)}, E \in \texttt{Expr}, E \in \texttt{List(Expr)}, f \in \texttt{FName}$. MiniBoC has 2 BoC specific commands - cown creation and when, whereas the rest are standard.

The cown creation command is used to create a cown and assign the cown identifier to a variable. The `when` command is used to spawn a behaviour requiring some cowns while also binding the contents of those cowns to variables that are accessible in the body of the behaviour.

Notation. Generally, $\texttt{x}$ is used for variables that map to a cown identifier in the variable store, $\texttt{y}$ is used for variables that map to a value, and $\texttt{z}$ is used when the variable can map to either in the variable store.

Definition 4.5 (Program Variables for MiniBoC Commands). The set of *program variables* for MiniBoC commands is defined inductively on the structure of commands as follows:

$$\begin{aligned}
\texttt{pv}(\texttt{y} := E) &= \{\texttt{y}\} \cup \texttt{pv}(E) \\
\texttt{pv}(\texttt{C}_1; \texttt{C}_2) &= \texttt{pv}(\texttt{C}_1) \cup \texttt{pv}(\texttt{C}_2) \\
\texttt{pv}(\texttt{if } E \texttt{ then } \{\texttt{C}_1\} \texttt{ else } \{\texttt{C}_2\}) &= \texttt{pv}(E) \cup \texttt{pv}(\texttt{C}_1) \cup \texttt{pv}(\texttt{C}_2) \\
\texttt{pv}(\texttt{z} := f(\vec{E}_i|_{i=1}^{n})) &= \{\texttt{z}\} \cup \bigcup_{i=1}^{n} \texttt{pv}(E_i) \\
\texttt{pv}(\texttt{skip}) &= \phi \\
\texttt{pv}(\texttt{x} := \texttt{cown}(E)) &= \{\texttt{x}\} \cup \texttt{pv}(E) \\
\texttt{pv}(\texttt{when } (\vec{\texttt{ix}} = \vec{\texttt{x}}) \ \{\texttt{C}\}) &= \{\vec{\texttt{ix}}\} \cup \{\vec{\texttt{x}}\} \cup \texttt{pv}(\texttt{C})
\end{aligned}$$

Notation. The function call $\texttt{z} := f(\vec{E_i}|_{i=1}^{n})$ denotes a function f with n arguments $E_1, ...E_n$

Definition 4.6 (MiniBoC Variable Types). In MiniBoC, variables have one of 3 types

$$\texttt{T} \mid \texttt{cown[T]} \mid \texttt{content[T]}$$

A variable of type `T` refers to a variable that maps to a value of type `T` in the variable store. It is typically represented with $\texttt{y}, \texttt{y}', ...$

A variable of type `cown[T]`, sometimes referred to as a cown pointer, maps to a cown identifier in the variable store. The cown identifier in the cown store maps to a value of type `T`. Such a variable can only be created using the cown creation command. It is typically represented with $\texttt{x}, \texttt{x}', ...$

A variable of type `content[T]` refers to a variable that maps to a value of type `T` in the variable store. What differentiates this type from `T` is that variables of this type are bounded to the *contents* of a cown. Consequentially, such variables can only be created by the `when` command. They are typically represented by $\texttt{ix}, \texttt{iy}, ...$[1].

It is important to note that `T` is a *primitive* type and represents types such as integers and booleans.

4.1.1 When Closure Capture

Due to BoC's reliance on an underlying language, MiniBoC makes a choice with respect to the closure capture of the `when` command. BoC only asks for isolation from the underlying language, and hence a mutable reference of these variables can not be used in the body of a behaviour. In MiniBoC, a variable declared outside the closure of a behaviour can be used in the body of that behaviour if the variable is *copyable*. The value of the variable is copied into a variable of the same name in the body of the routine. Variables of type `content[T]` are not copyable. Intuitively this makes sense as one should not be able to read the contents of a cown without acquiring the cown. This intuition has been discussed as a caveat of nesting behaviours in Section 3.4.4.

Hence, in a well-formed BoC program, a variable of type `content[T]` declared outside the body of a behaviour will never be used inside the body of that behaviour.

Definition 4.7 (Copyable Program Variables for MiniBoC Commands). The set of *copyable program variables* for MiniBoC commands is defined inductively on the structure of commands as follows:

$$\begin{aligned} \texttt{cpv(when } (\vec{\texttt{ix}} = \vec{\texttt{x}})\ \{\texttt{C}\}) &= \texttt{pv(when } (\vec{\texttt{ix}} = \vec{\texttt{x}})\ \{\texttt{C}\}) \setminus \{\vec{\texttt{ix}}\} && \text{(when command)} \\ \texttt{cpv(C)} &= \texttt{pv(C)} && \text{(all other commands)} \end{aligned}$$

Definition 4.8 (MiniBoC Behaviour States). The behaviour state of MiniBoC consists of the following:

$$\text{Spawned Behaviours State} : S \stackrel{\text{def}}{=} \texttt{BName} \rightharpoonup \texttt{VarStore} \times \texttt{BName}^* \times [\texttt{Cown} \rightharpoonup \texttt{Var}] \times \texttt{Cmd}$$

The spawned behaviour state (S) consists of the *local behaviour state*, σ_b, the set of behaviours that happen before the spawned behaviour (c.f. **Definition** 3.1), $\{\vec{b}\}$, the *cown variable mapping* representing the variables bound to the contents of a cown, δ (or $[\vec{\kappa} \mapsto \vec{\texttt{ix}}]$), and the *behaviour body*, `C`.

$$\text{Running Behaviours State} : R \stackrel{\text{def}}{=} \texttt{BName} \rightharpoonup \texttt{VarStore} \times [\texttt{Cown} \rightharpoonup \texttt{Var}] \times \texttt{Cmd}$$

The running behaviour state (R) consists of the *local behaviour state*, σ_b, the *cown variable mapping* representing the variables bound to the contents of a cown, δ (or $[\vec{\kappa} \mapsto \vec{\texttt{ix}}]$), and the *behaviour body*, `C`.

The choice of having a cown routine mapping in the routine state is further justified in Section 4.1.2

Notation. we use T^* for sets and $List(T)$ for lists

[1] The prefix `i` is used for the word "inner" as the current refers to a value that exists *inside* a cown

Definition 4.9 (MiniBoC Function Contexts). A MiniBoC *function context*, `FContext` is defined as follows:

$$\text{Function Context} : \gamma \stackrel{\text{def}}{=} \texttt{FName} \to \texttt{List(Var)} \times \texttt{Cmd} \times \texttt{Exp}$$

A function context comprises *finite function parameters* given by a finite list of distinct variables, $\{\vec{\texttt{x}}\}$ and the *function body* given by the command $\texttt{C} \in \texttt{Cmd}$, and a *return expression* $E \in \texttt{Expr}$.

Notation. $f(\vec{x}) = \{\texttt{C}; \texttt{return } E\}$ is used to imply $\gamma(f) = (\{\vec{\texttt{x}}\}, \texttt{C}, E)$ for an implicit function context γ.

Definition 4.10 (MiniBoC Program). A MiniBoC program, represented by p, is given as follows:

$$\text{Program} : p \stackrel{\text{def}}{=} \texttt{FContext} \times \texttt{Cmd}$$

4.1.2 Subtlety of Cown Aliasing

Cown aliasing in BoC can be a subtle aspect, particularly when it occurs inside a behaviour. To explain this subtlety more clearly, we can turn to MiniBoC, where the distinction between a cown pointer and a cown identifier is more explicit. Consider the MiniBoC program provided in Listing 4.1, which demonstrates cown aliasing inside a behaviour.

```
1 x1 := cown(0)
2 x2 := cown(0)
3
4 when(ix1 = x1, ix2 = x2) {
5     x1 := x2
6     ix1 := 1
7 }
```

Listing 4.1: MiniBoC program with cown aliasing

In this program, we have two cown variables, x1 and x2, both initialized with the same value of 0. On Line 3, the variable store σ for this program is represented as $[\texttt{x1} \mapsto \kappa_1, \texttt{x2} \mapsto \kappa_2]$, where κ_1 and κ_2 are cown identifiers.

It is crucial to note that on Line 4, the variable `ix1` is bound to the cown identifier κ_1, not the variable containing the cown identifier `x1`. This distinction is essential to understand because the variables containing the cown identifiers, such as `x1` and `x2`, can be mutated at any point and, therefore, cannot be relied upon for maintaining the integrity of the program's logic.

This also justifies the presence of the cown routine mapping in the running and spawned behaviour states R and S.

4.2 MiniBoC Semantics

A formal description of the nature of MiniBoC is given using small-step operational semantics. The expressions are evaluated with respect to a variable store. In MiniBoC, only commands are side-effecting and expressions are not side-effecting.

Definition 4.11 (MiniBoC Expression Evaluation Function). Let $\sigma \in \texttt{VarStore}$ be a variable store, the *expression evaluation function*, $\mathcal{E}[\![-]\!]_\sigma \in \texttt{Exp} \rightharpoonup \texttt{Val}$ is defined inductively as follows:

$$\begin{aligned}
\mathcal{E}[\![v]\!]_\sigma &= v \\
\mathcal{E}[\![\texttt{x}]\!]_\sigma &= \sigma(\texttt{x}), && \texttt{where} \ \texttt{x} \in \mathit{dom}(s) \\
\mathcal{E}[\![E_1 + E_2]\!]_\sigma &= \mathcal{E}[\![E_1]\!]_\sigma + \mathcal{E}[\![E_2]\!]_\sigma, && \texttt{where} + : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\
\mathcal{E}[\![E_1 > E_2]\!]_\sigma &= \mathcal{E}[\![E_1]\!]_\sigma > \mathcal{E}[\![E_2]\!]_\sigma, && \texttt{where} > : \mathbb{N} \times \mathbb{N} \to \{\texttt{true}, \texttt{false}\} \\
&\ldots
\end{aligned}$$

Definition 4.12 (MiniBoC Operational Semantics). The small-step operational semantics for MiniBoC is described in this section using the judgements $\sigma, \Delta, S, R, \texttt{C} \xrightarrow{b}_\gamma \sigma', \Delta', S', R', \texttt{C}'$.

The b in $\xrightarrow{b}$ corresponds to the current running behaviour. When irrelevant to the rule, the current behaviour b is omitted from.

The behaviour name *main* is used for commands not running in a behaviour, i.e. the top level program commands. Thus, we can say that $b = main \vee b \in dom(R)$

- **Assignment**: Evaluate the expression E using variable store σ. If this evaluation is defined with value v, update the variable store accordingly $\sigma[\mathtt{y} \mapsto v]$. Otherwise, fault.

$$\frac{\mathcal{E}[[E]]_\sigma = v \qquad \sigma' = \sigma[\mathtt{y} \mapsto v]}{\sigma, \Delta, S, R, \mathtt{y} := E \rightarrow_\gamma \sigma', \Delta, S, R, \mathtt{skip}} \text{ Assn}$$

- **Sequential Composition**: Evaluate the command $\mathtt{C}_1$ with the current program state and then, and return the updated command and state sequentially composed with $\mathtt{C}_2$.

$$\frac{\sigma, \Delta, S, R, \mathtt{C}_1 \xrightarrow{b}_\gamma \sigma', \Delta', S', R, \mathtt{C}_1'}{\sigma, \Delta, S, R, \mathtt{C}_1; \mathtt{C}_2 \xrightarrow{b}_\gamma \sigma', \Delta', S', R, \mathtt{C}_1'; \mathtt{C}_2} \text{ Seq-Comp-Left}$$

If $\mathtt{C}_1$ is equivalent to `skip`, return $\mathtt{C}_2$.

$$\frac{}{\sigma, \Delta, S, R, \mathtt{skip}; \mathtt{C}_2 \rightarrow_\gamma \sigma, \Delta, S, R, \mathtt{C}_2} \text{ Seq-Comp-Skip}$$

- **If Condition**: Evaluate the expression E using the variable store σ. If E does not evaluate to a boolean, fault. If E is `true`, return $\mathtt{C}_1$ otherwise, if E is `false`, return $\mathtt{C}_2$.

$$\frac{\mathcal{E}[[E]]_\sigma = \mathtt{true}}{\sigma, \Delta, S, R, \mathtt{if}\ E\ \mathtt{then}\ \{\mathtt{C}_1\}\ \mathtt{else}\ \{\mathtt{C}_2\} \rightarrow_\gamma \sigma, \Delta, S, R, \mathtt{C}_1} \text{ If-True}$$

$$\frac{\mathcal{E}[[E]]_\sigma = \mathtt{false}}{\sigma, \Delta, S, R, \mathtt{if}\ E\ \mathtt{then}\ \{\mathtt{C}_1\}\ \mathtt{else}\ \{\mathtt{C}_2\} \rightarrow_\gamma \sigma, \Delta, S, R, \mathtt{C}_2} \text{ If-False}$$

- **Skip**: $\sigma, \Delta, S, R, \mathtt{skip}$ is the answer configuration and hence, no rule exists for it.

- **Function Call**: If $f \notin \gamma$, fault. Otherwise, $f(\vec{x})\{\mathtt{C}; \mathtt{return}\ E'\} \in \gamma$. Evaluate $\vec{E}$ using σ. If undefined, fault. If defined with values $\vec{w}$, create a store σ_f containing the function parameters, $\vec{x}$, initialised with their respective values, $\vec{w}$. Evaluate $\mathtt{C}$ using σ_f and the remaining program state. If this faults, fault. If it succeeds, evaluate E' using the new variable store σ_f'. If this is undefined, fault, otherwise, return the variable store $\sigma[\mathtt{y} \mapsto w']$ and the remainder of the updated program state.

$$\frac{\begin{array}{c} f(\vec{x})\{\mathtt{C}; \mathtt{return}\ E'\} \in \gamma \qquad \mathcal{E}[[\vec{E}]]_\sigma = \vec{w} \qquad \sigma_f = [\vec{x} \mapsto \vec{w}] \\ \sigma_f, \Delta, S, R, \mathtt{C} \rightarrow_\gamma^* \sigma_f', \Delta', S', R, \mathtt{skip} \qquad \mathcal{E}[[E']]_{\sigma_f'} = \vec{w'} \qquad \sigma' = \sigma[\mathtt{z} \mapsto w'] \end{array}}{\sigma, \Delta, S, R, \mathtt{z} := f(\vec{E}) \rightarrow_\gamma \sigma', \Delta', S', R, \mathtt{skip}} \text{ Func Call}$$

- **Cown Creation**: Evaluate the expression E using variable store σ. If this is defined, generate a new and unique cown identifier κ and update the variable store and cown store accordingly - $\sigma[\mathtt{x} \mapsto \kappa]$, $\Delta[\kappa \mapsto (v, \mu)]$. Otherwise, fault. Note that μ, the standard value for `BName` is used, as no behaviour requiring κ has spawned yet and hence no behaviour is waiting on κ.

$$\frac{\mathcal{E}[[E]]_\sigma = v \qquad fresh.\ \kappa \qquad \sigma' = \sigma[\mathtt{x} \mapsto \kappa] \qquad \Delta' = \Delta[\kappa \rightarrow (v, \mu)]}{\sigma, \Delta, S, R, \mathtt{x} := \mathtt{cown.create}(E) \rightarrow_\gamma \sigma', \Delta', S, R, \mathtt{skip}} \text{ CC}$$

- **Spawn Behaviour**: If $\vec{\mathtt{x}}$ map to cown identifiers $\vec{\kappa}$, get a new unique behaviour name, b otherwise fault. Evaluate $\vec{b}$ which represents the set of behaviour names that *happen before* the behaviour that is being spawned (i.e. b). Update the cown store as b is the most recently

spawned behaviour requiring cowns $\vec{\kappa}$. Add b to S with a copy of the global state σ, $\{\vec{b}\}$, the cown variable mappings $[\vec{\kappa} \mapsto \vec{\texttt{ix}}]$ and the behaviour body C.

$$\frac{\sigma(\vec{\texttt{x}}) = \vec{\kappa} \qquad fresh.\ b \qquad \Delta(\vec{\kappa}) = (\vec{v}_\kappa, \vec{b}_\kappa) \qquad \Delta' = \Delta[\vec{\kappa} \mapsto (\vec{v}_\kappa, b)] \qquad S' = S[b \mapsto (\sigma, \{\vec{b}_\kappa\}, [\vec{\kappa} \mapsto \vec{\texttt{ix}}], \texttt{C})]}{\sigma, \Delta, S, R, \texttt{when } (\vec{\texttt{ix}} = \vec{\texttt{x}}) \ \{\texttt{C}\} \rightarrow_\gamma \sigma, \Delta', S', R, \texttt{skip}} \text{ B-Spawn}$$

Notation. $[\vec{a} \mapsto \vec{b}]$ is shorthand for $[a_1 \mapsto b_1, ..., a_n \mapsto b_n]$ given that $\vec{a} = [a_1, ..., a_n] \wedge \vec{b} = [b_1, ..., b_n]$.

Notation. $[\vec{a} \mapsto b]$ is shorthand for $[a_1 \mapsto b, ..., a_n \mapsto b]$ given that $\vec{a} = [a_1, ..., a_n]$.

- **Run Behaviour**: For a spawned behaviour (i.e. $b \in dom(S)$), obtain the spawned behaviour state $(\sigma_b, \{\vec{b}\}, [\vec{\kappa} \mapsto \vec{\texttt{ix}}], \texttt{C}_b)$ from S. If all the behaviours in $\{\vec{b}\}$ have finished running (i.e. their behaviour body is `skip` or the behaviour is μ), remove b from S. Evaluate $\vec{\kappa}$ to $\vec{v}$ in the cown store, if this faults, fault, otherwise add b to R with updated state $\sigma_b[\vec{\texttt{ix}} \mapsto \vec{v}]$, the cown routine mapping and the behaviour body $\texttt{C}_b$.

$$\frac{b \in dom(S) \qquad S(b) = (\sigma_b, \{\vec{b}\}, [\vec{\kappa} \mapsto \vec{\texttt{ix}}], \texttt{C}_b) \qquad S' = S \setminus \{b\} \qquad \forall b \in \vec{b}.\ (b = \mu \vee R(b) = (-, -, \texttt{skip})) \qquad \Delta(\vec{\kappa}) = \vec{v} \qquad R' = R[b \mapsto (\sigma_b[\vec{\texttt{ix}} \mapsto \vec{v}], [\vec{\kappa} \mapsto \vec{\texttt{ix}}], \texttt{C}_b)]}{\sigma, \Delta, S, R, \texttt{C} \rightarrow_\gamma \sigma, \Delta, S', R', \texttt{C}} \text{ B-Run}$$

- **Progress Behaviour**: Any running behaviour (i.e. $b \in dom(R)$) can make progress at any given point. Obtain the running behaviour state, $(\sigma_b, [\vec{\kappa} \mapsto \vec{\texttt{ix}}], \texttt{C}_b)$, from R. Evaluate $\texttt{C}_b$ using the behaviour-local variable store and the remaining global state and update R accordingly

$$\frac{b \in dom(R) \qquad R(b) = (\sigma_b, \delta, \texttt{C}_b) \qquad \sigma_b, \Delta, S, R, \texttt{C}_b \xrightarrow{b}_\gamma \sigma_b', \Delta', S', R, \texttt{C}_b' \qquad R' = R[b \mapsto (\sigma_b', \delta, \texttt{C}_b')]}{\sigma, \Delta, S, R, \texttt{C} \xrightarrow{b'}_\gamma \sigma, \Delta', S', R', \texttt{C}} \text{ B-Progress}$$

- **Finish Behaviour**: This rule is just a special case of the previous rule. If the behaviour body $\texttt{C}_b$ evaluates to `skip`, it is vital that the cown contents are updated. This update of the cown contents has to happen now to prevent other behaviours from accessing outdated information.

$$\frac{b \in dom(R) \qquad R(b) = (\sigma_b, [\vec{\kappa} \mapsto \vec{\texttt{ix}}], \texttt{C}_b) \qquad \sigma_b, \Delta, S, R, \texttt{C}_b \rightarrow_\gamma \sigma_b', \Delta', S', R, \texttt{skip} \qquad R' = R[b \mapsto (\sigma_b', [\vec{\kappa} \mapsto \vec{\texttt{ix}}], \texttt{skip})] \qquad \sigma_b(\vec{\texttt{ix}}) = \vec{v} \qquad \Delta(\kappa) = (-, \vec{b}) \qquad \Delta'' = \Delta'[\vec{\kappa} \mapsto (\vec{v}, \vec{b})]}{\sigma, \Delta, S, R, \texttt{C} \rightarrow_\gamma \sigma, \Delta'', S', R', \texttt{C}} \text{ B-Finish}$$

4.3 Comparing MiniBoC with the new BoC operational semantics

Parallel to the development of MiniBoC, the new BoC operational semantics were introduced. These operational semantics, as depicted in Appendix A, take a different approach compared to the traditional language semantics.

Rather than defining a standalone programming language, BoC aims to be a concurrency paradigm that can be integrated into existing programming languages. As such, the operational semantics of BoC are designed to function more like a *library* of rules that can be added to an existing language.

When comparing the new operational semantics with the MiniBoC semantics, the most significant difference is the absence of the cown state from the former. Due to BoC's aim to be a library of rules, the cown state is deferred to the underlying language, which is captured by the MiniBoC semantics.

Another difference is the way both semantics approach spawned behaviours. MiniBoC has a Spawned Behaviour State, S, whereas the new semantics have a global queue-like structure for the pending behaviours, P.

4.4 Further Improvements

There are 3 main areas of improvements for MiniBoC:

- **Adding more commands**: By extending MiniBoC to include additional commands like loops (e.g., for and while loops), the language becomes more expressive and flexible.
- **Supporting objects in cowns**: Introducing support for objects within cowns expands the capabilities of MiniBoC. With object support, programmers can encapsulate data, providing a higher level of abstraction. One particularly intriguing capability that arises from object support in cowns is the ability to nest cowns. Nesting cowns allows for a hierarchical organization of shared data and behaviours, providing a structured approach to managing concurrent access. With nested cowns, developers can establish a multi-level structure where cowns can contain other cowns, forming a parent-child relationship.
- **Introducing different types of cowns**: In MiniBoC, cowns are used to protect data. More accurately, cowns are used to prevent two operations on data - read and write. An interesting opportunity to expand MiniBoC, as well as the broader BoC paradigm, is to introduce read cowns and write cowns that only protect the read and write operation on data stored inside a cown.

Chapter 5

GIL+

In this chapter, we introduce GIL+ as a target language for translating MiniBoC programs. GIL+ is designed to provide a low-level representation of MiniBoC programs, breaking them down into their most fundamental atomic actions. This approach is similar to how GIL (Gillian Intermediate Language) is used for existing languages. A small-step operational semantics for GIL+ is provided in the chapter.

By providing a formal and precise representation of MiniBoC programs, GIL+ facilitates analysis, reasoning, and verification of MiniBoC programs and the broader BoC paradigm. The evaluation of GIL+ sheds light on its effectiveness as a target language and its ability to capture the essential aspects of BoC and other adjacent concurrency paradigms.

In this chapter, we also evaluate the choices made by GIL+. Upon evaluation, we present an alternate BoC paradigm named FlexibleBoC.

5.1 GIL+ Objectives

Gillian, as discussed in Section 2.4, currently supports sequential programs through the Gillian Intermediate Language (GIL) outlined in Section 2.5. To extend Gillian's capabilities for reasoning about MiniBoC programs, an alternative intermediate language called GIL+ has been introduced.

While developing GIL+, its design scope was not limited solely to MiniBoC. The language aims to achieve the following objectives:

- Facilitate the analysis of MiniBoC: GIL+ provides features constructs that enable easier analysis of concurrent programs in the BoC paradigm. It offers support for fine-grained granular reasoning about MiniBoC.
- Flexibility to emulate other concurrency paradigms: GIL+ is designed to be versatile and adaptable, allowing it to emulate various concurrency paradigms beyond BoC. It provides mechanisms to model and reason about other concurrency models closely related to BoC, such as the actor model and other models such as async-await and Concurrent Separation Logic (CSL) as well. GIL+ aims to allow reasoning about concurrency paradigms that lie at the base of the triangle of concurrency (c.f. Figure 3.1).
- Exploring BoC variants: Although orthogonal to analysis, GIL+ serves as a platform for exploring different variants and extensions of the BoC paradigm. It allows researchers and developers to experiment with new ideas and variations within the BoC framework. By providing this flexibility, GIL+ supports the exploration and advancement of BoC as a concurrency paradigm.

5.2 An informal overview of GIL+

In GIL+, the introduction of *routines* provides named asynchronous units of work, serving as the fundamental building blocks of concurrency. Routines enable the programmer to organize and structure concurrent operations within the program.

Additionally, GIL+ recognizes the importance of cowns in BoC and other ownership-based concurrency paradigms. Consequently, GIL+ incorporates a version of cowns that requires the explicit creation of cown groups. The ownership of these cown groups is explicitly managed by routines, which are responsible for acquiring and releasing ownership as needed. Cown groups are only associated with routine names, and each routine can have its own cown group.

This explicit control over cown groups enhances the programmer's ability to manage ownership and ensures a clear delineation of responsibilities between routines and the associated cowns.By providing this level of control and clarity, GIL+ enhances the expressiveness and precision of BoC programming.

In GIL+, routines serve a similar purpose as behaviours in MiniBoC, acting as units of concurrency. However, one key distinction is that routines do not have a happens-before order imposed on them. Furthermore, GIL+ introduces a construct that enables a routine's execution to be dependent on the termination of another routine. This construct allows for synchronization between routines, where one routine can wait for the completion of another before proceeding.

5.3 GIL+ Syntax

The GIL+ language is parametrised by the following sets: the set of *values*, $\texttt{Val}$, ranged over by $v, v_1, ...$ with $\mathbb{N} \cup \{\texttt{true}, \texttt{false}\} \subseteq \texttt{Val}$; the set of *variables*, $\texttt{Var}$: ranged over by $\texttt{x}, \texttt{y}, ...$; the set of *cown identifiers* $\texttt{Cown}$, ranged over by $\kappa, \kappa_1, ...$; the set of function names, $\texttt{FName}$: ranged over by $f, f_1, ...$; the set of routine names, $\texttt{RName}$ ranged over by $\iota, \iota_1, \iota_2, ...$, with distinguished elements $\mu, \mathit{main} \in \texttt{RName}$. μ represents the standard value for $\texttt{RName}$ and *main* is a reserved routine name.

Definition 5.1 (GIL+ Expressions). The set of *expressions*, $\texttt{Exp}$ ranged over by $E, E_1, E_2, ...$, is given by:

$$E ::= v \mid \texttt{x} \mid E + E \mid E < E \mid ...$$

where $v \in \texttt{Val}, \texttt{x} \in \texttt{Var}$.

Definition 5.2 (Program Variables for GIL+ Expressions). The set of program variables for an expression E, denoted by, $\texttt{pv}(E)$ is defined inductively:

$$\begin{aligned} \texttt{pv}(v) &= \phi \\ \texttt{pv}(\texttt{x}) &= \{\texttt{x}\} \\ \texttt{pv}(E_1 + E_2) &= \texttt{pv}(E_1) \cup \texttt{pv}(E_2) \\ \texttt{pv}(E_1 < E_2) &= \texttt{pv}(E_1) \cup \texttt{pv}(E_2) \\ &... \end{aligned}$$

Definition 5.3 (GIL+ Program Stores). The program stores of GIL+ comprises the following:

$$\text{Variable Store} : \sigma \stackrel{\text{def}}{=} \texttt{Var} \rightarrow \texttt{Val} \mid \texttt{Cown} \mid \texttt{RName}^*$$

σ represents the variable store in which a variable can either map to a value, a cown identifier or a set of routine names. We say that the variable $\texttt{x}$ has value w in the store σ if $\sigma(\texttt{x}) = w$ where w is either a value v, a cown identifier κ or a set of routine names $\{\vec{\iota}\}$.

$$\text{Cown Store} : \Delta \stackrel{\text{def}}{=} \texttt{Cown} \rightarrow \texttt{Val}$$

Δ represents the cown store in which a cown identifier maps to a value. The value v represents the contents of that cown.

Definition 5.4 (GIL+ Commands). The set of GIL+ commands, $\texttt{Cmd}$, ranged over by, $\texttt{C}_1, \texttt{C}_2, ...$is defined as follows:

$$
\begin{array}{rll}
\texttt{Cmd},\ \texttt{C} ::= & \texttt{y} := E & \text{(assignment)} \\
| & \texttt{C}_1; \texttt{C}_2 & \text{(sequential composition)} \\
| & \texttt{if}\ E\ \texttt{then}\ \{\texttt{C}_1\}\ \texttt{else}\ \{\texttt{C}_2\} & \text{(if)} \\
| & \texttt{z} := f(\vec{E}) & \text{(function call)} \\
| & \texttt{skip} & \text{(skip)} \\
| & \texttt{x} := \texttt{cown}(E) & \text{(cown assignment)} \\
| & [\texttt{x}] := E & \text{(cown mutate)} \\
| & \texttt{y} := [\texttt{x}] & \text{(cown lookup)} \\
| & \texttt{wait(rs)} & \text{(wait)} \\
| & \texttt{rs} := \texttt{LAS}(\{\vec{\texttt{x}}\}, \iota) & \text{(lookup and set)} \\
| & \texttt{take} & \text{(take)} \\
| & \texttt{give} & \text{(give)} \\
| & \texttt{start}\ \iota\ \texttt{with}\ \vec{\texttt{z}}\ \{\texttt{C}\} & \text{(start routine)}
\end{array}
$$

where $\texttt{x}, \texttt{y}, \texttt{rs} \in \texttt{Var}, E \in \text{Expr}, \vec{E} \in \text{List(Expr)} \iota \in \texttt{RName}, f \in \texttt{FName}$. GIL+, similar to MiniBoC has some standard commands - assignment, sequential composition, if, function call and skip.

Notation. Generally, `x` is used for variables that map to a cown identifier in the variable store, `y` is used for variables that map to a value, `rs` is used for variables that map to a set of routine names, and `z` is used when the variable can map to any of them in the variable store.

5.3.1 Accessing Cowns in GIL+

In GIL+, similar to MiniBoC, several standard commands such as assignment, sequential composition, if statements, function calls, and skip are available. However, there is a distinction in how the contents of cowns are accessed and manipulated compared to MiniBoC.

In MiniBoC, behaviours bind the contents of a cown to a variable at the time of spawning. This allows for direct referencing and manipulation of the cown data using the assigned variable. However, in GIL+, this concept is not present. Instead, GIL+ employs the `[x]` syntax to represent the contents of a cown. This syntax enables accessing and modifying the data stored within a cown in GIL+ programs through the use of cown mutate and cown lookup commands.

The introduction of the `[x]` syntax in GIL+ serves a specific purpose - enabling direct assertions about cowns. In the BoC paradigm, cowns are updated indirectly through other variables, which can make it more challenging to make pre- and post-condition assertions that directly affect cowns. By using the `[x]` syntax, GIL+ aims to provide a means to express assertions and conditions explicitly targeting cown contents. Such assertions are pivotal for specification and verification of properties related to cowns.

5.3.2 Lookup and Set (LAS)

The `LAS` (Lookup and Set) command in GIL+ is utilized to create cown groups. The command $\texttt{rs} := \texttt{LAS}(\{\vec{\texttt{x}}\}, \iota)$ is used to form a cown group that can be subsequently taken by a specific routine. It then assimilates these routine names and updates the variable state accordingly. This ensures that the cown group is properly formed, and the variable `rs` contains the set of routine names to which the cowns were last added. It is worth noting that cown groups in GIL+ can overlap, meaning that a cown can be part of multiple cown groups simultaneously.

The introduction of the `LAS` command in GIL+ serves two main purposes. Firstly, it facilitates the creation of cown groups, allowing multiple cowns to be grouped together. Ownership of these groups is taken and given by the eponymous `take` and `give` commands. This enables routines to have explicit ownership over a set of related cowns. Secondly, the LAS command provides a dynamic lookup mechanism for routines that intend to take ownership of a cown group. This feature of `LAS` allows the programmer to make ordering guarantees about routine execution based

on information related to cowns. For e.g. by looking up the previous routine names associated with the cowns in the group, the `LAS` command allows a routine to determine which routines should have completed their execution before it can acquire ownership of the cown group. This dynamic lookup capability is vital for translating MiniBoC to GIL+.

5.3.3 Starting Routines in GIL+

The `start-with` command is used to spawn a new routine while capturing a set of variables ($\{\vec{\mathtt{z}}\}$). The variables in $\{\vec{\mathtt{z}}\}$ are copied into the body of the routine and can still be used throughout the program. However, the changes made to these variables within the body of the routine are only reflected locally. Any spawned routine can start running at any given time. Note that when the set of variables is empty, $\{\vec{\mathtt{z}}\} = \phi$, `start` ι $\{\mathtt{C}\}$ is used as syntactic sugar for `start` ι `with` ϕ $\{\mathtt{C}\}$.

The `wait` command is blocking and allows the routine calling this command to *wait* until a set of routines finish executing. A program variable is used to represent the set of routine names. This command is useful for synchronising action across multiple routines.

Definition 5.5 (Program Variables for GIL+ Commands). The set of program variables for GIL+ commands is defined inductively on the structure of commands as follows:

$$
\begin{aligned}
\mathtt{pv}(\mathtt{y} \coloneqq E) &= \{\mathtt{y}\} \cup \mathtt{pv}(E) \\
\mathtt{pv}(\mathtt{C}_1; \mathtt{C}_2) &= \mathtt{pv}(\mathtt{C}_1) \cup \mathtt{pv}(\mathtt{C}_2) \\
\mathtt{pv}(\mathtt{if}\ E\ \mathtt{then}\ \{\mathtt{C}_1\}\ \mathtt{else}\ \{\mathtt{C}_2\}) &= \mathtt{pv}(E) \cup \mathtt{pv}(\mathtt{C}_1) \cup \mathtt{pv}(\mathtt{C}_2) \\
\mathtt{pv}(\mathtt{z} \coloneqq f(\vec{E})) &= \{\mathtt{z}\} \cup \bigcup_{i=1}^{n} \mathtt{pv}(E_i) \\
\mathtt{pv}(\mathtt{skip}) &= \phi \\
\mathtt{pv}(\mathtt{x} \coloneqq \mathtt{cown}(E)) &= \{\mathtt{x}\} \cup \mathtt{pv}(E) \\
\mathtt{pv}([\mathtt{x}] \coloneqq E) &= \{\mathtt{x}\} \cup \mathtt{pv}(E) \\
\mathtt{pv}(\mathtt{y} \coloneqq [\mathtt{x}]) &= \{\mathtt{x}\} \cup \{\mathtt{y}\} \\
\mathtt{pv}(\mathtt{wait}(\mathtt{rs})) &= \{\mathtt{rs}\} \\
\mathtt{pv}(\mathtt{rs} \coloneqq \mathtt{LAS}(\{\vec{\mathtt{x}}\}, \iota)) &= \{\mathtt{rs}\} \cup \{\vec{\mathtt{x}}\} \\
\mathtt{pv}(\mathtt{take}) &= \phi \\
\mathtt{pv}(\mathtt{give}) &= \phi \\
\mathtt{pv}(\mathtt{start}\ \iota\ \mathtt{with}\ \vec{\mathtt{z}}\ \{\mathtt{C}\}) &= \{\vec{\mathtt{z}}\} \cup \mathtt{pv}(\mathtt{C}) \\
&\ldots
\end{aligned}
$$

Definition 5.6 (Routine Names for GIL+ commands). The set of routine names for variables is defined as follows:

$$
\begin{aligned}
\mathtt{rn}(\mathtt{start}\ \iota\ \mathtt{with}\ \vec{z}\ \{\mathtt{C}\}) &= \{\iota\} \cup \mathtt{rn}(\mathtt{C}) && \text{(for the start-wth command)} \\
\mathtt{rn}(\mathtt{C}) &= \phi && \text{(for all other commands)}
\end{aligned}
$$

Definition 5.7 (GIL+ Routine States). The routine state for GIL+ comprises:

$$\text{Spawned Routines} : S \stackrel{\text{def}}{=} \mathtt{RName} \rightharpoonup \mathtt{VarState} \times \mathtt{Cmd}$$

S represents the set of spawned routines. At any given time, any routine can start running. An arbitrary entry in S is shown - $S(r) = (\sigma_\iota, \mathtt{C})$ where ι is the routine name, σ_ι is state local to that routine and `C` is the body of the routine.

$$\text{Running Routines} : R \stackrel{\text{def}}{=} \mathtt{RName} \rightharpoonup \mathtt{VarState} \times \mathtt{Cown}^* \times \mathtt{Cmd}$$

R represents the set of running routines. These routines can make progress at any given time and interleave arbitrarily. An arbitrary entry in R is shown - $R(\iota) = (\sigma_\iota, \{\vec{\kappa}\}, \mathtt{C})$ where ι is the routine name, σ_ι is the variable store local to that routine, $\{\vec{\kappa}\}$ are the cowns owned by the routine ι and `C` is the body of the routine. Only the owner routine of a cown can mutate or lookup its contents.

Definition 5.8 (GIL+ Cown Group State). The cown group state in GIL+ comprises:

$$\text{Cown-Routine Mapping} : \Lambda \stackrel{\text{def}}{=} \texttt{Cown} \rightarrow \texttt{RName}$$

Λ represents a mapping from cown to routines. An arbitrary entry in Λ is shown - $\Lambda(\kappa) = \iota$ where κ is a cown identifier and ι is the routine name to whose group this cown was last added.

$$\text{Cown Groups} : \Gamma \stackrel{\text{def}}{=} \texttt{RName} \rightharpoonup \texttt{Cown}^*$$

Γ represents groups of cowns formed by the `LAS` command. An arbitrary entry in Γ is shown - $\Gamma(\iota) = \{\vec{\kappa}\}$ where ι is the routine name that has grouped the cowns $\vec{\kappa}$. ι can call `take` to take ownership of this group of cowns.

Definition 5.9 (GIL+ Function Contexts). A GIL+ *function context*, `FContext` is defined as follows:

$$\text{Function Context} : \gamma \stackrel{\text{def}}{=} \texttt{FName} \rightarrow \texttt{List(Var)} \times \texttt{Cmd} \times \texttt{Exp}$$

A function context comprises *finite function parameters* given by a finite list of distinct variables, $\{\vec{\texttt{x}}\}$ and the *function body* given by the command $\texttt{C} \in \texttt{Cmd}$, and a *return expression* $E \in \texttt{Expr}$.

Notation. $f(\vec{x}) = \{\texttt{C}; \texttt{return } E\}$ is used to imply $\gamma(f) = (\{\vec{\texttt{x}}\}, \texttt{C}, E)$ for an implicit function context γ.

Definition 5.10 (GIL+ Program). A GIL+ program, represented by p, is given as follows:

$$\text{Program} : p \stackrel{\text{def}}{=} \texttt{FContext} \times \texttt{Cmd}$$

5.4 GIL+ Semantics

A formal description of the nature of GIL+ is given using small-step operational semantics. The expressions are evaluated with respect to a variable store. Similar to MiniBoC, in GIL+, only commands are side-effecting and expressions are not side-effecting.

Definition 5.11 (GIL+ Expression Evaluation Function). Let $\sigma \in \texttt{VarStore}$ be a variable store, the *expression evaluation function*, $\mathcal{E}[\![-]\!]_\sigma \in \texttt{Exp} \rightharpoonup \texttt{Val}$ is defined inductively as follows:

$$\begin{aligned}
\mathcal{E}[\![v]\!]_\sigma &= v \\
\mathcal{E}[\![\texttt{x}]\!]_\sigma &= \sigma(\texttt{x}), && \texttt{where} \ \texttt{x} \in dom(s) \\
\mathcal{E}[\![E_1 + E_2]\!]_\sigma &= \mathcal{E}[\![E_1]\!]_\sigma + \mathcal{E}[\![E_2]\!]_\sigma, && \texttt{where} + : \mathbb{N} \times \mathbb{N} \rightarrow_\gamma \mathbb{N} \\
\mathcal{E}[\![E_1 > E_2]\!]_\sigma &= \mathcal{E}[\![E_1]\!]_\sigma > \mathcal{E}[\![E_2]\!]_\sigma, && \texttt{where} > : \mathbb{N} \times \mathbb{N} \rightarrow_\gamma \{\texttt{true}, \texttt{false}\} \\
&\ldots
\end{aligned}$$

Definition 5.12 (GIL+ Operational Semantics). The small-step operational semantics for GIL+ is described in this section using the judgements $\sigma, \Delta, \Lambda, \Gamma, S, R, \texttt{C} \stackrel{\iota}{\rightarrow}_\gamma \sigma', \Delta', \Lambda', \Gamma', S', R', \texttt{C}'$. ι represents the routine name in which the small step transition is occurring. When irrelevant to the rule, ι it is omitted. An important property of ι is that $\iota \in dom(R)$.

Note that *main* is a reserved routine name that is used as the routine name for transitions occurring at the top level of the program and is the only routine name that does not appear in R.

- **Assignment**: Evaluate the expression E using variable store σ. If this evaluation is defined with value v, update the variable store accordingly $\sigma[\texttt{y} \mapsto v]$. Otherwise, fault.

$$\frac{\mathcal{E}[\![E]\!]_\sigma = v \qquad \sigma' = \sigma[\texttt{y} \mapsto v]}{\sigma, \Delta, \Lambda, \Gamma, S, R, \texttt{y} := E \rightarrow_\gamma \sigma', \Delta, \Lambda, \Gamma, S, R, \texttt{skip}} \ \textsc{Assn}$$

- **Sequential Composition**: Evaluate the command $\texttt{C}_1$ with the current program state and then, and return, the updated command and state sequentially composed with $\texttt{C}_2$

$$\frac{\sigma, \Delta, \Lambda, \Gamma, S, R, \texttt{C}_1 \stackrel{\iota}{\rightarrow}_\gamma \sigma', \Delta', \Lambda', \Gamma', S', R', \texttt{C}_1'}{\sigma, \Delta, \Lambda, \Gamma, S, R, \texttt{C}_1; \texttt{C}_2 \stackrel{\iota}{\rightarrow}_\gamma \sigma', \Delta', \Lambda', \Gamma', S', R', \texttt{C}_1'; \texttt{C}_2} \ \textsc{Seq-Comp-Left}$$

$$\frac{}{\sigma, \Delta, \Lambda, \Gamma, S, R, \texttt{skip}; \texttt{C}_2 \xrightarrow{\iota}_\gamma \sigma', \Delta', \Lambda', \Gamma', S', R', \texttt{C}_2} \text{ Seq-Comp-Skip}$$

- **If Condition**: Evaluate the expression E using the variable store σ. If E does not evaluate to a boolean, fault. If E is `true`, return the program state along with $\texttt{C}_1$ otherwise, if E evaluates to `false`, return the program state along with $\texttt{C}_1$.

$$\frac{\mathcal{E}[[E]]_\sigma = \texttt{true} \qquad \sigma, \Delta, \Lambda, \Gamma, S, R, \texttt{C}_1 \rightarrow_\gamma \sigma', \Delta', \Lambda', \Gamma', S', R', \texttt{skip}}{\sigma, \Delta, \Lambda, \Gamma, S, R, \texttt{if } E \texttt{ then } \{\texttt{C}_1\} \texttt{ else } \{\texttt{C}_2\} \rightarrow_\gamma \sigma', \Delta', \Lambda', \Gamma', S', R', \texttt{skip}} \text{ If-True}$$

$$\frac{\mathcal{E}[[E]]_\sigma = \texttt{false} \qquad \sigma, \Delta, \Lambda, \Gamma, S, R, \texttt{C}_2 \rightarrow_\gamma \sigma', \Delta', \Lambda', \Gamma', S', R', \texttt{skip}}{\sigma, \Delta, \Lambda, \Gamma, S, R, \texttt{if } E \texttt{ then } \{\texttt{C}_1\} \texttt{ else } \{\texttt{C}_2\} \rightarrow_\gamma \sigma', \Delta', \Lambda', \Gamma', S', R', \texttt{skip}} \text{ If-False}$$

- **Skip**: $\sigma, \Delta, S, R, \texttt{skip}$ is the answer configuration and hence, no rule exists for it.
- **Function Call**: If $f \notin \gamma$, fault. Otherwise, $f(\vec{x})\{\texttt{C}; \texttt{return } E'\} \in \gamma$. Evaluate $\vec{E}$ using σ. If undefined, fault. If defined with values $\vec{w}$, create a store σ_f containing the function parameters, $\vec{x}$, initialised with their respective values, $\vec{w}$. Also rename all the routine names that are present in `C` using, *renameRoutines* defined as follows:

$$\textit{renameRoutine}(\texttt{C}) = \texttt{C}[\forall \iota \in \texttt{rn}(\texttt{C}).\ \iota/\iota_i]$$

where i corresponds to the number of times this function has been called. Evaluate $\textit{renameRoutines}(\texttt{C})$ using σ_f and the remaining program state. If this faults, fault. If it succeeds, evaluate E' using the new variable store σ'_f. If this is undefined, fault, otherwise, return the variable store $\sigma[\texttt{y} \mapsto w']$ and the remainder of the updated program state.

$$\frac{\begin{array}{c} f(\vec{x})\{\texttt{C}; \texttt{return } E'\} \in \gamma \qquad \mathcal{E}[[\vec{E}]]_\sigma = \vec{w} \qquad \sigma_f = [\vec{x} \mapsto \vec{w}] \\ \texttt{C}_f = \textit{renameRoutines}(\texttt{C}) \qquad \sigma_f, \Delta, \Lambda, \Gamma, S, R, \texttt{C} \xrightarrow{\iota}{}^*_\gamma \sigma'_f, \Delta', \Lambda', \Gamma', S', R', \texttt{skip} \\ \mathcal{E}[[E']]_{\sigma'_f} = \vec{w} \qquad \sigma' = \sigma[\texttt{z} \mapsto w] \end{array}}{\sigma, \Delta, \Lambda, \Gamma, S, R, \texttt{z} := f(\vec{E}) \xrightarrow{\iota}_\gamma \sigma', \Delta', \Lambda', \Gamma', S', R', \texttt{skip}} \text{ Func Call}$$

- **Cown Assignment**: Evaluate the expression E using variable store σ. If this evaluation is not defined fault, otherwise if defined with value v, with a fresh cown identifier κ, return new state $\sigma[x \mapsto \kappa]$ and new cown state $\Delta[\kappa \mapsto v]$. Additionally, update the cown-routine mapping with the new cown - $\Lambda[\kappa \mapsto \mu]$.

$$\frac{\mathcal{E}[[E]]_\sigma = v \qquad \textit{fresh. } \kappa \qquad \sigma' = \sigma[\texttt{x} \mapsto \kappa] \qquad \Delta' = \Delta[\kappa \mapsto v] \qquad \Lambda' = \Lambda[\kappa \mapsto \mu]}{\sigma, \Delta, \Lambda, \Gamma, S, R, \texttt{x} := \texttt{cown}(E) \rightarrow_\gamma \sigma', \Delta', \Lambda', \Gamma, S, R, \texttt{skip}} \text{ C-Asgn}$$

- **Cown Mutate**: If `x` maps to a cown identifier κ in the variable store, and this cown identifier is owned by the current running routine, evaluate E using σ and return the new cown state $\Delta[\kappa \mapsto v]$. If `x` does not map to a cown identifier, fault.

$$\frac{\sigma(\texttt{x}) = \kappa \qquad R(\iota) = (-, \{\vec{\kappa}\}, -) \qquad \kappa \in \{\vec{\kappa}\} \qquad \mathcal{E}[[E]]_\sigma = v \qquad \Delta' = \Delta[\kappa \mapsto v]}{\sigma, \Delta, \Lambda, \Gamma, S, R, [\texttt{x}] := E \xrightarrow{\iota} \sigma, \Delta', \Lambda, \Gamma, S, R, \texttt{skip}} \text{ C-Mutate}$$

- **Cown Lookup**: If `x` maps to a cown identifier κ in the variable store, such that $\Delta[\kappa \mapsto v]$ and this cown identifier is owned by the current running routine, return the newly updated state $\sigma[y \mapsto v]$. If `x` does not map to a cown identifier, fault.

$$\frac{\sigma(\texttt{x}) = \kappa \qquad R(\iota) = (-, \{\vec{\kappa}\}, -) \qquad \kappa \in \{\vec{\kappa}\} \qquad \Delta(\kappa) = v \qquad \sigma' = \sigma[\texttt{y} \mapsto v]}{\sigma, \Delta, \Lambda, \Gamma, S, R, \texttt{y} := [\texttt{x}] \xrightarrow{\iota} \sigma', \Delta, \Lambda, \Gamma, S, R, \texttt{skip}} \text{ C-Lookup}$$

- **Wait**: If `rs` does not map to a set of routine names in the variable store, fault. Otherwise, if `rs` maps to a set of routine names $\{\vec{\iota}\}$ and for each routine name, ι in $\{\vec{\iota}\}$, remove ι from the set if the associated routine has finished execution and update the state accordingly.

$$\frac{\sigma(\texttt{rs}) = \{\vec{\iota}\} \qquad \forall \iota \in \{\vec{\iota}\}.\ \iota = \mu \vee \iota \in dom(R) \wedge R(\iota) = (-, -, \texttt{skip})}{\sigma, \Delta, \Lambda, \Gamma, S, R, \texttt{wait(rs)} \rightarrow_\gamma \sigma, \Delta, \Lambda, \Gamma, S, R, \texttt{skip}}\ \textsc{Wait}$$

- **Lookup and Set**: For each `x` in $\{\vec{\texttt{x}}\}$, if it doesn't evaluate to a cown identifier κ in σ, fault. For each κ lookup routine name from the cown-routine mapping, i.e. $\Lambda(\kappa) = \iota'$ and update the cown-routine mapping $\Lambda[\kappa \mapsto \iota]$. Combine the values of ι' into one set to obtain $\{\vec{\iota}\}$ and return the updated state $\sigma[\texttt{rs} \mapsto \{\vec{\iota}\}]$. Also update the cown groups $\Gamma[\iota \mapsto \vec{\kappa}]$

$$\frac{\sigma(\vec{\texttt{x}}) = \vec{\kappa} \qquad \Gamma' = \Gamma[\iota \mapsto \{\vec{\kappa}\}] \qquad \Lambda(\vec{\kappa}) = \vec{\iota}_\kappa \qquad \Lambda' = \Lambda[\vec{\kappa} \mapsto \iota] \qquad \sigma' = \sigma[\texttt{rs} \mapsto \{\vec{\iota}_\kappa\}]}{\sigma, \Delta, \Lambda, \Gamma, S, R, \texttt{rs} := \texttt{LAS}(\vec{\texttt{x}}, \iota) \rightarrow_\gamma \sigma', \Delta, \Lambda', \Gamma', S, R, \texttt{skip}}\ \textsc{LAS}$$

- **Take Cowns**: Lookup the set of cowns $\{\vec{\kappa}\}$ for the current routine in the cown group state, if no group exists fault, otherwise, update the set of owned cowns in R for ι to contain $\{\vec{\kappa}\}$ if all the cowns are *available*. A cown is said to be available if it is not owned by any other routine. The availability predicate is defined as:

$$available(\kappa) \stackrel{\text{def}}{=} \forall \iota \in dom(R).\ R(\iota) = (-, \vec{\kappa}) \wedge \kappa \notin \vec{\kappa}$$

If the cowns aren't available, the command will not fault.

$$\frac{\begin{array}{c}\Gamma(\iota) = \{\vec{\kappa}\} \\ \forall \kappa \in \{\vec{\kappa}\}.\ available(\kappa) \qquad R(\iota) = (\sigma_\iota, \{\vec{\kappa'}\}, \texttt{C}_\iota) \qquad R' = R[\iota \mapsto (\sigma_\iota, \{\vec{\kappa'}\} \cup \{\vec{\kappa}\}, \texttt{C}_\iota)]\end{array}}{\sigma, \Delta, \Lambda, \Gamma, S, R, \texttt{take} \xrightarrow{\iota} \sigma, \Delta, \Lambda, \Gamma, S, R', \texttt{skip}}\ \textsc{Take}$$

- **Give Cowns**: For the current running routine, lose ownership of all the currently owned cowns by updating the set of owned cowns in R.

$$\frac{R(\iota) = (\sigma_\iota, \{\vec{\kappa}\}, \texttt{C}_\iota) \qquad R' = R[\iota \mapsto (\sigma_\iota, \phi, \texttt{C}_\iota)]}{\sigma, \Delta, \Lambda, \Gamma, S, R, \texttt{give} \xrightarrow{\iota} \sigma, \Delta, \Lambda, \Gamma, S, R', \texttt{skip}}\ \textsc{Give}$$

- **Spawn Routine**: If a routine with routine name ι already exists in S or (R), fault. Otherwise, if $\vec{\texttt{x}}$ evaluates to $\vec{w}$, update S with an entry for ι along with the appropriate state.

$$\frac{\iota \notin dom(S) \qquad \iota \notin dom(R) \qquad \sigma(\vec{\texttt{z}}) = \vec{w} \qquad \sigma_\iota = [\vec{\texttt{z}} \mapsto \vec{w}] \qquad S' = S[\iota \mapsto (\sigma_\iota, C)]}{\sigma, \Delta, \Lambda, \Gamma, S, R, \texttt{start}\ \iota\ \texttt{with}\ \vec{\texttt{z}}\ \{\texttt{C}\} \rightarrow_\gamma \sigma, \Delta, \Lambda, \Gamma, S', R, \texttt{skip}}\ \textsc{R-Spawn}$$

- **Run Routine**: Any routine in S can start running at any given time. A routine ι is removed from S and added to R with no owned cowns.

$$\frac{\iota \in dom(S) \qquad S(\iota) = (\sigma_\iota, \texttt{C}_\iota) \qquad R' = R[\iota \mapsto (\sigma_\iota, \phi, \texttt{C}_\iota)] \qquad S' = S \setminus \{\iota\}}{\sigma, \Delta, \Lambda, \Gamma, S, R, \texttt{C} \rightarrow_\gamma \sigma, \Delta, \Lambda, \Gamma, S', R', \texttt{C}}\ \textsc{R-Run}$$

- **Progress Routine**: Any routine in R can make progress at any given time. Take a small step for ι, after which R needs to be updated accordingly.

$$\frac{\begin{array}{c}\iota \in dom(R) \\ R(\iota) = (\sigma_\iota, \{\vec{\kappa}\}, \texttt{C}_\iota) \qquad \sigma_\iota, \Delta, \Lambda, \Gamma, S, R, \texttt{C}_\iota \xrightarrow{\iota} \sigma'_\iota, \Delta', \Lambda', \Gamma', S', R', \texttt{C}'_\iota \\ R'(\iota) = (-, \{\vec{\kappa'}\}, -) \qquad R'' = R'[\iota \mapsto (\sigma'_\iota, \{\vec{\kappa'}\}, \texttt{C}'_\iota)]\end{array}}{\sigma, \Delta, \Lambda, \Gamma, S, R, \texttt{C} \rightarrow_\gamma \sigma, \Delta', \Lambda', \Gamma', S', R'', \texttt{C}}\ \textsc{R-Progress}$$

5.5 Translating MiniBoC to GIL+

MiniBoC can be translated to GIL+ through the help of a translation function Θ.

5.5.1 MiniBoC to GIL+ Translation Function (Θ)

Θ takes in a MiniBoC command, $\texttt{C}_{mb}$, a set of routine names, R, and a set of program variable, V, that are copyable in MiniBoC. It outputs a GIL+ command, $\texttt{C}_{g+}$.

$$\Theta(\texttt{C}_{mb}, R, V) = \texttt{C}_{g+}$$

R represents the set of GIL+ routine names that have been used in the translation and is used to get fresh routine names. At the beginning of the translation, $R = \phi$.

V represents the set of MiniBoC program variables that are copyable. At the beginning of the translation, $V = \phi$.

Translating the assignment, if, function call, skip and the cown creation command is trivial, as Θ just acts as the identity function, as shown below:

$$\begin{aligned}
\Theta(\texttt{y} := E, R, V) &= \texttt{y} := E \\
\Theta(\texttt{if } E \texttt{ then } \{\texttt{C}_1\} \texttt{ else } \{\texttt{C}_2\}, R, V) &= \texttt{if } E \texttt{ then } \{\texttt{C}_1\} \texttt{ else } \{\texttt{C}_2\} \\
\Theta(\texttt{z} := f(\vec{E}), R, V) &= \texttt{z} := f(\vec{E}) \\
\Theta(\texttt{skip}, R, V) &= \texttt{skip} \\
\Theta(\texttt{x} := \texttt{cown}(E), R, V) &= \texttt{x} := \texttt{cown}(E)
\end{aligned}$$

When translating the sequential composition command, the set of used program identifiers, R has to be updated accordingly:

$$\Theta(\texttt{C}_1; \texttt{C}_2, R, V) = \Theta(\texttt{C}_1, R, V); \Theta(\texttt{C}_2, R \cup \texttt{rn}(\Theta(\texttt{C}_1, R, V)), V \cup \texttt{cpv}(\texttt{C}_1))$$

The output of $\Theta(\texttt{when } (\vec{\texttt{ix}} = \vec{\texttt{x}}) \{\texttt{C}\}, R, V)$ is shown in Listing 5.2 and a line-by-line breakdown is given below.

```
1 when(ix⃗ = x⃗) {
2     C
3 }
```

Listing 5.1: Output of $\Theta(\texttt{when } (\vec{\texttt{ix}} = \vec{\texttt{x}}) \{\texttt{C}\}, R, V)$

```
1 rs = LAS({x⃗},ι); // rs is local and ι is fresh
2 start ι with (y⃗,rs) { // y⃗ = V
3     wait(rs);
4     take;
5     ix⃗ := x⃗;
6     θ(C, ix⃗, R ∪ {ι}, V);
7     give;
8 }
```

Listing 5.2: Output of $\Theta(\texttt{when } (\vec{\texttt{ix}} = \vec{\texttt{x}}) \{\texttt{C}\}, R, V)$

On Line 1, the `LAS` command creates a cown group $\{\vec{\texttt{x}}\}$ for a routine ι and, looks up the routine names previously associated with the cowns in the group. These routine names are stored in a variable `rs`[1]. Note that ι is a *fresh identifier*, i.e. it does not appear in R.

On Line 2, a routine with the routine name ι is started. The routine captures a set of variables $\vec{\texttt{y}}$ and the variable `rs`. $\vec{\texttt{y}}$ represents all the program variables in the variable store that are copied over into the body of the behaviour and $\vec{\texttt{y}} = V$. The variable `rs` is used in the body of the routine and hence is captured as well.

On Line 3, the `wait` command waits on the set of routine names that were looked up in Line 1. This wait ensures that the GIL+ program respects the BoC happens before ordering.

[1] Using `rs` as a variable *local* adds an arbitrary constraint on the MiniBoC program being translated - `rs` can not be used a variable in the original MiniBoC program. However, one can comply with this trivially by using alpha substitution

On Line 4, the `take` command takes ownership of cowns that were grouped together as a result of the `LAS` command on Line 1. This ensures that all accesses to the contents of the cown in the routine body are allowed and valid.

In Section 4.1.2, we saw that $\vec{\texttt{ix}}$ variables in MiniBoC are bound to cown identifiers and not the variables containing the cown identifier. Line 5 preserves this property by creating a set of cowns, $\vec{\texttt{ix}}$, as aliases for the actual required cowns $\vec{\texttt{x}}$.

On Line 6, when translating the behaviour body `C` to in GIL+, it is necessary to account for the differences in accessing the contents of a cown compared to MiniBoC. Hence, an auxiliary translation function (θ) is used that replaces the occurrences of `ix` with `[ix]` (c.f. Section 5.5.3).

On Line 7, the `give` command, gives away the ownership of the cowns owned by ι at that point.

5.5.2 Preservation of BoC happens before ordering

This section informally shows how the GIL+ translation of a MiniBoC program preserves the BoC happens before ordering via an example. Consider Listing 5.3 with 2 behaviours, and its corresponding translation - Listing 5.4.

From the BoC happens before we know that $b_1 \prec b_2$.

```
1 x := cown(0);
2
3 when(ix = x) { // b1
4     ix := 1;
5 }
6
7 when(ix = x) { // b2
8     ix := 2;
9 }
```

Listing 5.3: MiniBoC Program with 2 behaviours

This ordering corresponds to `[x]` := 1 $\prec$ `[x]` := 2 in Listing 5.4.

Given that there is no defined ordering between routines in GIL+, it is possible for either routine to begin execution first. Assuming that routine ι_1 initiates execution and completes before ι_2 begins, the ordering is trivially preserved. However, even if ι_1 starts execution and ι_2 commences before ι_1 finishes, or if ι_2 starts before ι_1 does, the ordering remains intact. In both cases, we can establish that `rs1` := `LAS`$(\texttt{x}, \iota_1)$ $\prec$ `rs2` := `LAS`$(\texttt{x}, \iota_2)$. Consequently, we can infer that routine ι_1 will always be included in the set of routine names represented by `rs2`, denoted as $\iota_1 \in \sigma(\texttt{rs2})$. Thus, the `wait(rs2)` command guarantees that the statement `[x]` := 2 will execute only after ι_1 has completed execution. This, in turn, trivially implies the ordering `[x]` := 1 $\prec$ `[x]` := 2.

```
1  x := cown(E);
2  rs = LAS({x},ι1);
3  start ι1 with (x, rs) {
4      wait(rs);
5      take;
6      ix = x;
7      [ix] := 1;
8      give;
9  }
10
11 rs = LAS({x},ι2);
12 start ι2 with (x, rs) {
13     wait(rs);
14     take;
15     ix = x;
16     [ix] := 2;
17     give;
18 }
```

Listing 5.4: GIL+ Translation of Listing 5.3

An intriguing characteristic of this MiniBoC to GIL+ translation is that the `take` command, which has the potential to block, will never actually block due to the presence of the `LAS` and the `wait` commands.

5.5.3 Auxiliary Translation Function (θ)

In GIL+, due to the difference in accessing the contents of cowns compared to MiniBoC, an auxiliary translation function called θ is introduced. This function is used to translate the body of a behaviour in order to handle the accesses to cown contents.

The θ function takes four inputs: a command `C` that needs to be translated, a list of variables $\vec{\texttt{ix}}$ for which, a set of used routine names, R and a set of copyable program variable names V. The purpose of θ is to replace all occurrences of the cown variables $\vec{ix}$ with $[\vec{ix}]$. Upon taking these inputs, the function returns a translated GIL+ command.

This cown variable substitution, however, is not as simple as an alpha-substitution as in GIL+, `[ix]` is not a valid expression. To address this issue, a solution is to introduce a fresh variable to store the contents of cown `ix` in a previous command, and then utilize this fresh variable in the substitution.

Translating commands that do not have expressions, i.e. the skip and the sequential composition commands, is analogous to the main translation function:

$$\theta(\texttt{skip}, \vec{\texttt{ix}}, R, V) = \texttt{skip}$$
$$\theta(\texttt{C}_1; \texttt{C}_2, \vec{\texttt{ix}}, R, V) = \texttt{C}_1'; \theta(\texttt{C}_2, \vec{\texttt{ix}}, R \cup \texttt{rn}(\texttt{C}_1'), V \cup \texttt{cpv}(\texttt{C}_1)) \quad \text{where } \texttt{C}_1' = \theta(\texttt{C}_1, \vec{\texttt{ix}}, R, V)$$

For the `when` command, the main translation function can be used because we know that the contents of cown can not be accessed by a nested behaviour, as seen in Section 3.4.4.

$$\theta(\texttt{when } (\vec{\texttt{ix}}' = \vec{\texttt{x}}') \ \{\texttt{C}\}, \vec{\texttt{ix}}, R, V) = \Theta(\texttt{when } (\vec{\texttt{ix}}' = \vec{\texttt{x}}') \ \{\texttt{C}\}, R, V)$$

Commands with expressions may refer to one of the variables `ix` as an expression. Hence, for the translation of these commands, the contents of the cown `ix` are first stored in a set of fresh variables, say $\vec{\texttt{f}}$, and the all references to $\vec{\texttt{ix}}$ are replaced with $\vec{\texttt{f}}$.

Notation. $\vec{\texttt{f}} := [\vec{\texttt{ix}}];$ is used to represent $\texttt{f}_1 := [\texttt{ix}_1]; \texttt{f}_2 := [\texttt{ix}_2]; ...; \texttt{f}_n := [\texttt{ix}_n];$ where $\vec{\texttt{f}} = [\texttt{f}_1, ..., \texttt{f}_n]$ and $\vec{\texttt{ix}} = [\texttt{ix}_1, ..., \texttt{ix}_n]$

Notation. $E[\vec{\texttt{x}}/\vec{\texttt{y}}]$ means substitute all occurrences of variables $\texttt{x}_1, ...\texttt{x}_n$ with $\texttt{y}_1, ..., \texttt{y}_2$ respectively in the expression.

The auxiliary translation function, θ, for the cown creation and the if command is defined as follows:

$$\theta(\texttt{if } E \texttt{ then } \{\texttt{C}_1\} \texttt{ else } \{\texttt{C}_2\}, \vec{\texttt{x}}, \vec{\texttt{ix}}, R, V) = \vec{\texttt{f}} := [\vec{\texttt{ix}}]; \texttt{if } E[\vec{\texttt{ix}}/\vec{\texttt{f}}] \texttt{ then } \{\theta(\texttt{C}_1, \vec{\texttt{ix}}, R, V)\} \texttt{ else } \{\theta(\texttt{C}_2, \vec{\texttt{ix}}, R, V)\};$$
$$\theta(\texttt{x} := \texttt{cown}(E), \vec{\texttt{x}}, \vec{\texttt{ix}}, R, V) = \vec{\texttt{f}} := [\vec{\texttt{ix}}]; \texttt{x} := \texttt{cown}(E[\vec{\texttt{ix}}/\vec{\texttt{f}}]);$$

While translating the assignment and function call command, we consider two cases - the first, where the variable on the left-hand side in these commands is not in $\vec{\texttt{ix}}$ and the second, where the variable on the left-hand side is in $\vec{\texttt{ix}}$.

Consider the assignment command $\texttt{y} := E$. In the first case, i.e. $\texttt{y} \notin \vec{\texttt{ix}}$, we only need to define the set of fresh variables, as we have been doing so far. However, in the second case, when $\texttt{y} \notin \vec{\texttt{ix}}$, a cown mutation is required. The same logic is applied to the function call command.

The translation for the assignment and function call commands is given below:

$$\theta(\mathtt{y} := E, \vec{\mathtt{ix}}, R, V) = \vec{\mathtt{f}} := [\vec{\mathtt{ix}}]; \mathtt{y} := E[\vec{\mathtt{ix}}/\vec{\mathtt{f}}] \qquad \text{where } \mathtt{y} \notin \vec{\mathtt{ix}}$$
$$\theta(\mathtt{y} := E, \vec{\mathtt{ix}}, R, V) = \vec{\mathtt{f}} := [\vec{\mathtt{ix}}]; [\mathtt{y}] := E[\vec{\mathtt{ix}}/\vec{\mathtt{f}}] \qquad \text{where } \mathtt{y} \in \vec{\mathtt{ix}}$$

$$\theta(\mathtt{z} = f(\vec{E}), \vec{\mathtt{ix}}, R, V) = \vec{\mathtt{f}} := [\vec{\mathtt{ix}}]; \mathtt{z} := f(\overrightarrow{E[\vec{\mathtt{ix}}/\vec{\mathtt{f}}]}); \qquad \text{where } \mathtt{z} \notin \vec{\mathtt{ix}}$$
$$\theta(\mathtt{z} = f(\vec{E}), \vec{\mathtt{ix}}, R, V) = \vec{\mathtt{f}} := [\vec{\mathtt{ix}}]; [\mathtt{z}] := f(\overrightarrow{E[\vec{\mathtt{ix}}/\vec{\mathtt{f}}]}); \qquad \text{where } \mathtt{z} \in \vec{\mathtt{ix}}$$

5.6 Evaluating GIL+

In this section, we see how GIL+ achieves its aims that are laid out in Section 5.1

5.6.1 Analysis of MiniBoC

GIL+ facilitates the analysis of MiniBoC by breaking down BoC into smaller, fine-grained commands that are easier to write pre- and post-conditions about. For e.g. when reasoning about cowns in behaviours, one has to reason about them indirectly via the inner `ix` variables that are bound to the contents of the cown. However, in GIL+, one can directly reason about the cowns due to the `[x]` syntax.

5.6.2 Emulating other concurrency paradigms

In GIL+, it is also possible to emulate the concurrency paradigm of JavaScript's Async-await, which involves the usage of asynchronous functions and promises. Asynchronous functions can be invoked from GIL+ routines, and the promises associated with these asynchronous operations can be represented using cowns.

In JavaScript's async-await paradigm, asynchronous functions allow non-blocking execution, enabling the program to perform tasks concurrently. When an asynchronous function is called, it returns a promise, which represents the eventual completion or failure of the asynchronous operation. This promise can be used to handle the result of the asynchronous operation once it completes.

In GIL+, we can emulate this behaviour by associating promises with cowns. The cown can be used to represent the state of the asynchronous operation, similar to how promises represent the state in JavaScript. The GIL+ routine can interact with the cown, waiting for its completion or performing other operations based on the promise's state. The GIL+ `wait` command can also be used to model the await command from the async await paradigm.

Consider a contrived JavaScript asynchronous function `foo`, shown in Listing 5.5. The function is declared with the `async` keyword, indicating that it will return a promise [34].

```
1 async function foo() {
2   return 1;
3 }
4
5 foo().then((ret) => /* do domething with ret */) ; // 1
6 ret = await foo(); // 2
```

Listing 5.5: JavaScript Asynchronus Function

The code listing also demonstrates two different ways of calling this function:

1. In the first case, `foo()` is invoked, and the returned promise is chained with a `then` method to handle the resolved value. This allows for executing some code when the promise is fulfilled. In the example, the resolved value is captured in the `ret` parameter, which can then be used to perform further actions.
2. In the second case of calling the JavaScript asynchronous function foo, the await keyword is used to pause the execution of the surrounding async function until the promise returned by foo is fulfilled. The resolved value of the promise is then assigned to the variable ret.

When translating JavaScript asynchronous functions to GIL+, the functions are modified to return a cown, which enables the sharing of data across routines. This modification allows for seamless communication and data transfer between different parts of a GIL+ program.

The GIL+ translations of the function calls in Listing 5.5 is shown in Listing 5.6 and Listing 5.7. Let's examine each translation:

```
// 1
start τ {
    ret := foo();
    /* do domething with ret */
}
```

Listing 5.6: GIL+ Translation of `foo` function calls

The first translation is straightforward. The foo() function is called within a routine τ, and the returned value is assigned to the variable `ret`. The subsequent code can then operate on `ret` as needed.

```
// 2
ret = cown(0)
r := LAS({ret},τ)
start τ {
    take;
    [ret] := foo();
    /* do domething with ret */
    give;
}
wait(r)
_ := LAS({ret}, τc) // τc
take;
ret := [ret]
```

Listing 5.7: GIL+ Translation of `foo` function calls (continued)

In the 2nd translation, the routine τ, takes ownership of a `ret` cown and updates it with its contents to store the return value of `foo()`. The `wait(r)`, command ensures that the current routine waits until the promise is resolved. The $\_$ = `LAS`$(\{\texttt{ret}\}, \tau_c)$ command adds `ret` to the cown group for the current routine to allow subsequent access to the resolved value.

5.6.3 Exploring other BoC variants: FlexibleBoC

Upon identifying that the `take` command is non-blocking in the translation of a behaviour from MiniBoC to GIL+, we introduce FlexibleBoC. In FlexibleBoC, the MiniBoC `when` is translated to GIL+ without the `wait` command, as shown in Listing 5.8.

```
_ = LAS({x⃗},τ); // rs is no longer required
start τ with (x⃗,y⃗) {
    // wait(rs);
    take;
    ix⃗ := x⃗;
    θ(C, ix⃗, I ∪ {τ,rs});
    give;
}
```

Listing 5.8: Output of $\Theta($`when (`$\vec{\texttt{ix}}$ = $\vec{\texttt{x}}$`) {C}`$, I)$ in FlexibleBoC

As a result, in FlexibleBoC, there is no happens before ordering for behaviours. However, behaviours *fight* to gain exclusive access of cowns. Consider the following MiniBoC program given in Listing 5.9.

```
x1 := cown(0)
x2 := cown(0)
x3 := cown(0)

when(ix1 = x1, ix2 = x2) { ... } // b1
when(ix2 = x2, ix3 = x3) { ... } // b2
when(ix3 = x3, ix1 = x1) { ... } // b3
```

Listing 5.9: A MiniBoC program with 3 behaviours

According to normal BoC rules, in Listing 5.9, the $b1 \prec b2 \prec b3$ happens before ordering exists. However, in FlexibleBoC, no such ordering exists and there will be a fight over gaining exclusive ownership of cowns and any behaviour can start executing first.

In FlexibleBoC, behaviours contend for ownership of cowns, leading to non-deterministic execution sequences. This flexibility enables the modelling of scenarios where behaviours must compete for resources or synchronize their actions without imposing a strict ordering.

For instance, in the dining philosophers problem, philosophers may contend for exclusive ownership of forks (cowns) to perform their eating actions. The absence of a fixed happens-before ordering in FlexibleBoC allows different philosophers to start eating or release their forks in any order, mimicking the inherent concurrency and unpredictability of the problem.

FlexibleBoC, with its non-deterministic execution and absence of a fixed happens-before ordering, can find utility in various real-life scenarios where concurrent and flexible behaviour is required, such as:

- Ticket Booking System: In a ticket booking system, multiple users may concurrently attempt to book tickets for the same event or seat. With FlexibleBoC, the system can model the contention for ticket availability, allowing any user to start the booking process first. This flexibility accurately represents real-world scenarios where users compete for limited resources and enables the system to handle concurrent ticket bookings effectively.
- Concurrent File Access: In a file system, multiple processes or threads may need to access and modify files concurrently. FlexibleBoC can be employed to model the concurrent access and contention for file resources. With no fixed ordering, different processes can access and modify files as per availability, allowing for efficient utilization of resources and avoiding unnecessary delays
- Collaborative Document Editing: In collaborative document editing tools, multiple users may simultaneously edit the same document. FlexibleBoC can model the concurrent editing process, where different users contend for exclusive ownership of document sections. This flexibility allows users to start editing their sections in any order, enabling real-time collaboration and efficient synchronization of document changes.

These examples highlight the versatility of FlexibleBoC in capturing the complexities of concurrent and flexible behaviour in real-life systems. By embracing non-deterministic execution and relaxed ordering constraints, FlexibleBoC enables more accurate modelling and efficient handling of concurrent scenarios. This versatility of FlexibleBoC in turn shows how GIL+ can be used to identify other BoC variants or BoC adjacent concurrency paradigms.

Chapter 6

Soundness

In this chapter, our goal is to establish the soundness of the MiniBoC to GIL+ translation. Soundness means demonstrating that each step in the MiniBoC semantics corresponds to a corresponding step in the GIL+ semantics, ensuring that the translated programs preserve the intended behaviour.

By establishing the soundness of the MiniBoC to GIL+ translation, we can confidently rely on the GIL+ semantics to reason about the behaviour and properties of programs originally written in MiniBoC. This soundness property enhances the trustworthiness and reliability of the translated programs, enabling their effective use in practical applications.

6.1 Behaviour Name to Routine Name Bijective Function (b2r)

To aid with our soundness proof, we use a bijective function $b2r \stackrel{\text{def}}{=} \texttt{BName} \times \texttt{RName}$ that takes in a behaviour name and returns a unique routine name, i.e. $b2r(b) = \iota$ where b is a behaviour name and ι is the routine name returned.

6.2 Storing extra information in the MiniBoC Operational Semantics

To aid with the soundness proof, we modify the running routine state in the MiniBoC operational semantics as follows:

$$\text{Running Behaviours State} : R \stackrel{\text{def}}{=} \texttt{BName} \rightharpoonup \texttt{VarStore} \times \texttt{BName}^* \times [\texttt{Cown} \rightharpoonup \texttt{Var}] \times \texttt{Cmd}$$

The updated running behaviour state (R) consists of the local behaviour state, σ_b, the set of behaviours that had to *happen before* this behaviour, the *cown variable mapping* representing the variables bound to the contents of a cown, δ (or $[\vec{\kappa} \mapsto \vec{\texttt{ix}}]$), and the *behaviour body*, $\texttt{C}$.

This modification of R stores information about the BoC happens-before ordering for the running routines as well. This modification mainly affects the run routine rule. The updated run routine rule is described as follows, with the updated part highlighted in red:

$$\frac{\begin{array}{c} b \in dom(S) \\ S(b) = (\sigma_b, \{\vec{b}\}, [\vec{\kappa} \mapsto \vec{\texttt{ix}}], \texttt{C}_b) \qquad S' = S \setminus \{b\} \qquad \forall b \in \vec{b}.\ R(b) = (-, -, \texttt{skip}) \\ \Delta(\vec{\kappa}) = \vec{v} \qquad R' = R[b \mapsto (\sigma_b[\vec{\texttt{ix}} \mapsto \vec{v}], \{\vec{b}\}, [\vec{\kappa} \mapsto \vec{\texttt{ix}}], \texttt{C}_b)] \end{array}}{\sigma, \Delta, S, R, \texttt{C} \rightarrow_\gamma \sigma, \Delta, S', R', \texttt{C}} \text{ B-Run}$$

This modification is truly trivial, as this additional piece of state is not used by any of the other rules and is never mutated again. Its sole purpose is to facilitate the soundness proof without impacting the behaviour or functionality of the MiniBoC operational semantic rules.

Other rules that rely on state information from R are trivially updated to ignore this extra state information, i.e. if $R(b) = (\sigma_b, \delta_b, \mathtt{C}_b)$ in the unaltered operational semantics, it is treated as $R(b) = (\sigma_b, -, \delta_b, \mathtt{C}_b)$

6.3 MiniBoC Well-Formed Configuration

The well-formedness conditions for a MiniBoC configuration are as follows:

- A behaviour can not be in the running state and spawned state at the same time - $dom(R) \cap dom(S) = \phi$
- All the cowns acquired by the running behaviours are disjoint from each other, i.e. - $\forall b \in dom(R).\ (R(b) = (-, -, [\vec{\kappa} \mapsto -], -) \implies \forall b' \in dom(R) \setminus \{b\}.R(b') = (-, -, [\vec{\kappa}' \mapsto -], -)) \wedge \vec{\kappa} \cap \vec{\kappa}' = \phi$

Note that these well-formedness conditions aren't exhaustive, only the conditions relevant to the proof are listed.

6.4 State Translation

The translation of the BoC state $(b, \gamma_{mb}) \vdash \sigma_{mb}, \Delta_{mb}, S_{mb}, R_{mb}, \mathtt{C}_{mb}$, where b is the current behaviour and γ_{mb} is the function context, gives us $(\iota, \gamma_{g+}) \vdash \sigma_{g+}, \Delta_{g+}, \Lambda_{g+}, \Gamma_{g+}, S_{g+}, R_{g+}, \mathtt{C}_{g+}$ where ι is the current routine and γ_{g+} is the function context.

- $\sigma_{g+} = \sigma_{mb}$ if C_{mb} is not the `when` command. If $\mathtt{C}_{mb} = \mathtt{when}\ (\vec{\mathtt{ix}} = \vec{\mathtt{x}})\ \{\mathtt{C}\}$ is the when command, then $\sigma_{g+} = \sigma_{mb}[\mathtt{rs} \mapsto \{\vec{\iota}\}]$ where $\sigma_{mb}(\vec{\mathtt{x}}) = \vec{\kappa}$, $\Delta_{mb}(\vec{\kappa}) = (-, \vec{b})$ and $b2r(\vec{b}) = \vec{\iota}$
- $\Delta_{g+} = [\vec{\kappa} \mapsto \vec{v}]$ if $\Delta_{mb} = [\vec{\kappa} \mapsto (\vec{v}, -)]$.
- $\Lambda_{g+} = [\vec{\kappa} \mapsto b2r(\vec{b})]$ if $\Delta_{mb} = [\vec{\kappa} \mapsto (-, \vec{b})]$.
- $\Gamma_{g+} = [b2r(\vec{b}_S) \mapsto \mathrm{dom}(\vec{\delta}_S), b2r(\vec{b}_R) \mapsto \mathrm{dom}(\vec{\delta}_R)]$ if $S_{mb} = [\vec{b}_S \mapsto (-, -, \vec{\delta}_S, -)]$ and $R_{mb} = [\vec{b}_R \mapsto (-, \vec{\delta}_R, -)]$
- S_{g+}. $\forall b \in dom(S_{mb}).\ S_{mb}(b) = (\sigma_b, \{\vec{b}\}, [\vec{\kappa} \mapsto \vec{\mathtt{ix}}], \mathtt{C}_b)$, we can add an entry to S_{g+} as follows:

  $$S_{g+}(b2r(b)) = (\sigma_b[\mathtt{rs} \mapsto \{b2r(\vec{b})\}], \mathtt{wait(rs)}; \mathtt{take}; \vec{\mathtt{ix}} = \vec{\mathtt{x}}; \theta(\mathtt{C}_b, \vec{\mathtt{ix}}, R, V); \mathtt{give};)$$

  where $\sigma_b(\vec{\mathtt{x}}) = \vec{\kappa}$, $R = b2r(dom(R_{mb})) \cup b2r(dom(S_{mb}))$ and $V = \mathrm{dom}(\sigma_{mb})$
- R_{g+}. $\forall b \in dom(R_{mb}).\ R_{mb}(b) = (\sigma_b, \{\vec{b}\}, [\vec{\kappa} \mapsto \vec{\mathtt{ix}}], \mathtt{C}_b)$, we can add an entry to S_{g+} as follows:

  $$R_{g+}(b2r(b)) = (\sigma_b[\mathtt{rs} \mapsto \{b2r(\vec{b})\}][\vec{ix} \mapsto \vec{\kappa}], \vec{\kappa}, \theta(\mathtt{C}_b, \vec{\mathtt{ix}}, R, V); \mathtt{give};)$$

  where $R = b2r(dom(R_{mb})) \cup b2r(dom(S_{mb}))$ and $V = \mathrm{dom}(\sigma_{mb})$
- $\iota = b2r(b)$
- γ_{g+} is just γ_{mb} but with all the commands and the expressions of the functions translated GIL+ commands and expressions using the Θ translation function.

When relying upon this translation in a proof, this translation is referred to as MGT (MiniBoC to GIL+ Translation).

6.5 Proof Statement

For some well-formed MiniBoC configuration $\sigma_{mb}, \Delta_{mb}, S_{mb}, R_{mb}, \mathtt{C}_{mb}$ and some well-formed GIL+ configuration, $\sigma_{g+}, \Delta_{g+}, \Lambda_{g+}, \Gamma_{g+}, S_{g+}, R_{g+}, \mathtt{C}_{g+}$ we have to prove the following:

$$\sigma_{mb}, \Delta_{mb}, S_{mb}, R_{mb}, \mathtt{C}_{mb} \xrightarrow{b}_{\gamma} \sigma'_{mb}, \Delta'_{mb}, S'_{mb}, R'_{mb}, \mathtt{C}'_{mb}$$

$$\implies$$

$$\sigma_{g+}, \Delta_{g+}, \Lambda_{g+}, \Gamma_{g+}, S_{g+}, R_{g+}, \mathtt{C}_{g+} \xrightarrow{\iota}{}^{*}_{\gamma} \sigma'_{g+}, \Delta'_{g+}, \Lambda'_{g+}, \Gamma'_{g+}, S'_{g+}, R'_{g+}, \mathtt{C}'_{g+}$$

where:

- translating $\sigma_{mb}, \Delta_{mb}, S_{mb}, R_{mb}, \mathtt{C}_{mb}$ gives us $\sigma_{g+}, \Delta_{g+}, \Lambda_{g+}, \Gamma_{g+}, S_{g+}, R_{g+}, \mathtt{C}_{g+}$
- translating $\sigma'_{mb}, \Delta'_{mb}, S'_{mb}, R'_{mb}, \mathtt{C}'_{mb}$ gives us $\sigma'_{g+}, \Delta'_{g+}, \Lambda'_{g+}, \Gamma'_{g+}, S'_{g+}, R'_{g+}, \mathtt{C}'_{g+}$
- $b2r(b) = \iota$.
- $\mathtt{C}_{mb}\Theta(\mathtt{C}, R, V)$ where $R = dom(S_{g+}) \cup dom(R_{g+})$ and $V = dom(\sigma_{mb})$

6.6 Proof

Soundness of this translation is proved via case analysis on the MiniBoC operational semantic rules. In each case, we assume the antecedent of our proof statement and prove the consequent of the proof statement. [1]

In the context of our soundness proof, it is important to note that the assignment, sequential composition, if condition, function call, cown creation, and progress behaviour rules in MiniBoC have corresponding rules in GIL+. As a result, establishing the soundness of these cases is relatively straightforward. In this section, we will provide a proof for the assignment and sequential composition cases, while noting that the remaining cases follow similar reasoning The Spawn Behaviour and the Run Behaviour are non-trivial, as they rely heavily on the MiniBoC to GIL+ translation being correct semantically.

Regarding the Finish Behaviour rule, it can be considered a special case of the progress behaviour rule, as it represents the completion of a behaviour. The additional consequences of the Finish Behaviour rule follow from the translation of states and the auxiliary translation function. By ensuring that the translated GIL+ commands and states accurately reflect the MiniBoC behaviour, the proof for the Finish Behaviour rule can be derived from the progress behaviour rule.

6.6.1 Assignment

We assume the antecedent for this case, i.e. a successful execution of the MiniBoC assignment rule for the current behaviour b.

$$\frac{\mathcal{E}[[E]]_{\sigma_{mb}} = v \qquad \sigma'_{mb} = \sigma_{mb}[\mathtt{y} \mapsto v]}{\sigma_{mb}, \Delta_{mb}, S_{mb}, R_{mb}, \mathtt{y} := E \xrightarrow{b}_{\gamma} \sigma'_{mb}, \Delta_{mb}, S_{mb}, R_{mb}, \mathtt{skip}} \tag{6.1}$$

Proof

(1)	$\sigma_{mb} = [\vec{\mathtt{z}} \mapsto \vec{w}]$	Premise
(2)	$\sigma_{g+} = [\vec{\mathtt{z}} \mapsto \vec{w}][\mathtt{rs} \mapsto -]$	(MGT, 1)
(3)	$\sigma_{mb} = \sigma_{g+}$	(1, 2)
(4)	$\mathcal{E}[[E]]_{\sigma_{mb}} = v$	from Equation 6.1
(5)	$\mathcal{E}[[E]]_{\sigma_{g+}} = v$	(3, 4)
(6)	$\sigma'_{mb} = \sigma_{mb}[\mathtt{y} \mapsto v]$	from Equation 6.1
(7)	$\sigma'_{mb} = [\vec{\mathtt{z}} \mapsto \vec{w}][\mathtt{rs} \mapsto -][\mathtt{y} \mapsto v]$	(1, 6)
(8)	$\sigma'_{g+} = [\vec{\mathtt{z}} \mapsto \vec{w}][\mathtt{rs} \mapsto -][\mathtt{y} \mapsto v]$	(MGT, 7)
(9)	$\sigma'_{g+} = \sigma_{g+}[\mathtt{y} \mapsto v]$	(2, 7)

Using (5) and (9) we can prove the consequent:

$$\sigma_{g+}, \Delta_{g+}, \Lambda_{g+}, \Gamma_{g+}, S_{g+}, R_{g+}, \mathtt{y} := E \rightarrow_{\gamma} \sigma'_{g+}, \Delta_{g+}, \Lambda_{g+}, \Gamma_{g+}, S_{g+}, R_{g+}, \mathtt{skip}$$

[1] In the logical statement $A \implies B$, A and B are typically referred to as the "antecedent" and the "consequent"

Hence, proving soundness for this case.

6.6.2 Spawn Behaviour

We assume the antecedent for this case, i.e. a successful execution of the MiniBoC spawn behaviour rule for the current behaviour b_c as.

$$\frac{\sigma_{mb}(\vec{\mathtt{x}}) = \vec{\kappa} \qquad fresh.\ b \qquad \Delta_{mb}(\vec{\kappa}) = (\vec{v}_\kappa, \vec{b}_\kappa) \qquad \Delta'_{mb} = \Delta_{mb}[\vec{\kappa} \mapsto (\vec{v}_\kappa, b_c)] \qquad S'_{mb} = S_{mb}[b \mapsto (\sigma_{mb}, \{\vec{b}_\kappa\}, [\vec{\kappa} \mapsto \vec{\mathtt{ix}}], \mathtt{C})]}{\sigma_{mb}, \Delta_{mb}, S_{mb}, R_{mb}, \mathtt{when}\ (\vec{\mathtt{ix}} = \vec{\mathtt{x}})\ \{\mathtt{C}\} \xrightarrow{b_c}_{\gamma} \sigma_{mb}, \Delta'_{mb}, S'_{mb}, R_{mb}, \mathtt{skip}} \tag{6.2}$$

We know that when $\mathtt{C}_{mb} = \mathtt{when}\ (\vec{\mathtt{ix}} = \vec{\mathtt{x}})\ \{\mathtt{C}\}$, then:

$$\mathtt{C}_{g+} = \Theta(\mathtt{when}\ (\vec{\mathtt{ix}} = \vec{\mathtt{x}})\ \{\mathtt{C}\}, R, V) = \mathtt{rs} := \mathtt{LAS}(\vec{\mathtt{x}}, \iota); \mathtt{start}\ \iota\ \mathtt{with}\ \vec{\mathtt{y}}, \mathtt{rs}\ \{\mathtt{C}'\}$$

where $C' = \mathtt{wait(rs)}; \mathtt{take}, \vec{\mathtt{ix}} := \vec{\mathtt{x}}; \theta(\mathtt{C}, R \cup \{\iota\}); \mathtt{give};$, $V = dom(\sigma_{mb})$ and $R = b2r(dom(S)) \cup b2r(dom(R))$. Because of the way the translation function Θ is defined, we can say that $\vec{\mathtt{y}} = dom(\sigma_{mb})$.

Proof: $b = main$

(1)	$b2r(b_c) = \iota_c$	Premise
(2)	$b2r(b) = \iota$	Premise
(3)	$\sigma_{mb} = [\vec{\mathtt{z}} \mapsto \vec{w}]$	Premise
(4)	$S_{mb} = [\vec{b}_S \mapsto (-, -, \vec{\delta}_S, -)]$	Premise
(5)	$R_{mb} = [\vec{b}_R \mapsto (-, -, \vec{\delta}_R, -)]$	Premise
(6)	$\Delta_{mb} = [\vec{\kappa}_\Delta \mapsto (\vec{v}_\Delta, \vec{b}_\Delta)]$	Premise
(7)	$\vec{\mathtt{y}} = dom(\sigma_{mb})$	Premise
(8)	$\sigma_{mb}(\vec{x}) = \vec{\kappa}$	from Equation 6.2
(9)	$\Delta_{mb}(\vec{\kappa}) = (\vec{v}_\kappa, \vec{b}_\kappa)$	from Equation 6.2
(10)	$\sigma_{g+} = [\vec{\mathtt{z}} \mapsto \vec{w}][\mathtt{rs} \mapsto \{\vec{b}_\kappa\}]$	(MGT, 3, 9, 10)
(11)	$\sigma_{mb} \subseteq \sigma_{g+}$	(3, 10)
(12)	$\sigma_{mb}(\vec{\mathtt{x}}) = \vec{\kappa}$	from Equation 6.2
(13)	$\sigma_{g+}(\vec{\mathtt{x}}) = \vec{\kappa}$	(11, 12)
(14)	$\Gamma_{g+} = [b2r(\vec{b}_S) \mapsto dom(\vec{\delta}_S), b2r(\vec{b}_R) \mapsto dom(\vec{\delta}_R)]$	(MGT, 4, 5)
(15)	$S'_{mb} = S_{mb}[b \mapsto (\sigma_{mb}, \{\vec{b}_\kappa\}, [\vec{\kappa} \mapsto \vec{\mathtt{ix}}], \mathtt{C})]$	from Equation 6.2
(16)	$R'_{mb} = R_{mb}$	from Equation 6.2
(17)	$\Gamma'_{g+} = [b2r(\vec{b}_S) \mapsto dom(\vec{\delta}_S), b2r(\vec{b}_R) \mapsto dom(\vec{\delta}_R)][b2r(b) \mapsto \vec{\kappa}]$	(MGT, 15, 14)
(18)	$\Lambda_{g+} = [\vec{\kappa}_\Delta \mapsto b2r(\vec{b}_\Delta)]$	(MGT, 5)
(19)	$\vec{\kappa} \subseteq \vec{\kappa}_\Delta$	(6, 9)
(20)	$\vec{b}_\kappa \subseteq \vec{b}_\Delta$	(6, 9)
(21)	$\Lambda_{g+}(\vec{\kappa}) = b2r(\vec{b}_\kappa)$	(19, 19, 20)
(22)	$\Delta'_{mb} = \Delta_{mb}[\vec{\kappa} \mapsto (\vec{v}_\kappa, b_c)]$	from Equation 6.2

(23) $\Lambda'_{g+} = [\vec{\kappa}_\Delta \mapsto b2r(\vec{b_\Delta})][\vec{\kappa} \mapsto b2r(b_c)]$ (MGT, 22)

(24) $\Lambda'_{g+} = \Lambda_{g+}[\vec{\kappa} \mapsto b2r(b_c)]$ (18, 23)

(25) $\Lambda'_{g+} = \Lambda_{g+}[\vec{\kappa} \mapsto \iota_c]$ (1, 22)

(26) $\sigma'_{mb} = \sigma_{mb}$ from Equation 6.2

(27) $\sigma'_{g+} = \sigma_{g+}$ (26)

(28) $\sigma'_{g+} = [\vec{\mathtt{z}} \mapsto \vec{w}][\mathtt{rs} \mapsto \{\vec{b}_\kappa\}]$ (10, 26)

(29) $\sigma'_{g+} = [\vec{\mathtt{z}} \mapsto \vec{w}][\mathtt{rs} \mapsto \{\vec{b}_\kappa\}][\mathtt{rs} \mapsto \{\vec{b}_\kappa\}]$ (28)

(30) $\sigma'_{g+} = \sigma_{g+}[\mathtt{rs} \mapsto \{\vec{b}_\kappa\}]$ (10, 29)

(31) $\sigma'_{g+} = \sigma_{g+}$ (10, 28)

(32) $fresh.b$ from Equation 6.2

(33) $b \notin dom(S_{mb})$ (32)

(34) $b \notin dom(R_{mb})$ (32)

(35) $\iota \notin dom(S_{g+})$ (2, 32, 33)

(36) $\iota \notin dom(R_{g+})$ (2, 32, 34)

(37) $\vec{\mathtt{y}} = \vec{\mathtt{z}}$ (2, 6)

(38) $\sigma_{g+}(\vec{\mathtt{y}}, \mathtt{rs}) = \vec{w}, \{\vec{b}_\kappa\}$ (10, 37)

(39) $S'_{g+} = S_{g+}[\iota \mapsto (\sigma_{mb}[\mathtt{rs} \mapsto \{b2r(\vec{b}_\kappa)\}], \mathtt{C}')]$ (MGT, 2, 8, 15)
where $\mathtt{C}' = \mathtt{wait(rs)}; \mathtt{take}, \vec{\mathtt{ix}} := \vec{\mathtt{x}}; \theta(\mathtt{C}, R, V); \mathtt{give};$
and $R = dom(R_{g+}) \cup dom(S_{g+}), V = dom(\sigma_{mb})$

(40) $\sigma_{g+} = \sigma_{mb}[\mathtt{rs} \mapsto \{\vec{b}_\kappa\}]$ (3, 10)

(41) $S'_{g+} = S_{g+}[\iota \mapsto (\sigma_{g+}, \mathtt{C}')]$ (39, 30)
where $\mathtt{C}' = \mathtt{wait(rs)}; \mathtt{take}, \vec{\mathtt{ix}} := \vec{\mathtt{x}}; \theta(\mathtt{C}, R, V); \mathtt{give};$
and $R = dom(R_{g+}) \cup dom(S_{g+}), V = dom(\sigma_{mb})$

This allows us to prove the consequent of the proof statement as follows:

$$\sigma_{g+}, \Delta_{g+}, \Lambda_{g+}, \Gamma_{g+}, S_{g+}, R_{g+}, \mathtt{rs} := \mathtt{LAS}(\vec{\mathtt{x}}, \iota); \mathtt{start}\ \iota\ \mathtt{with}\ \vec{\mathtt{y}}, \mathtt{rs}\ \{\mathtt{C}'\}$$

$$\xrightarrow{\iota_C}_\gamma \sigma'_{g+}, \Delta_{g+}, \Lambda'_{g+}, \Gamma'_{g+}, S_{g+}, R_{g+}, \mathtt{skip}; \mathtt{start}\ \iota\ \mathtt{with}\ \vec{\mathtt{y}}, \mathtt{rs}\ \{\mathtt{C}'\}$$

(Using SEQ-COMP-LEFT and (13, 17, 21, 25, 30))

$$\xrightarrow{\iota_C}_\gamma \sigma'_{g+}, \Delta_{g+}, \Lambda'_{g+}, \Gamma'_{g+}, S_{g+}, R_{g+}, \mathtt{start}\ \iota\ \mathtt{with}\ \vec{\mathtt{y}}, \mathtt{rs}\ \{\mathtt{C}'\}$$ (Using SEQ-COMP-SKIP)

$$\xrightarrow{\iota_C}_\gamma \sigma_{g+}, \Delta_{g+}, \Lambda'_{g+}, \Gamma'_{g+}, S_{g+}, R_{g+}, \mathtt{start}\ \iota\ \mathtt{with}\ \vec{\mathtt{y}}, \mathtt{rs}\ \{\mathtt{C}'\}$$ (Using (31))

$$\xrightarrow{\iota_C}_\gamma \sigma'_{g+}, \Delta_{g+}, \Lambda'_{g+}, \Gamma_{g+}, S_{g+}, R_{g+}, \mathtt{skip}$$ (Using R-SPAWN and (35, 36, 38, 41))

Hence, proving soundness for this case.

6.6.3 Run Behaviour

We assume the antecedent for this case, i.e. a successful execution of the MiniBoC Run Behaviour rule in the current behaviour b_c.

$$\frac{\begin{array}{c} b \in dom(S_{mb}) \qquad S_{mb}(b) = (\sigma_b, \{\vec{b}\}, [\vec{\kappa} \mapsto \vec{\mathtt{ix}}], \mathtt{C}_b) \\ S'_{mb} = S_{mb} \setminus \{b\} \qquad \forall b \in \vec{b}.\ (b = \mu \vee R_{mb}(b) = (-,-,\mathtt{skip})) \\ \Delta(\vec{\kappa}) = \vec{v} \qquad R'_{mb} = R_{mb}[b \mapsto (\sigma_b[\vec{\mathtt{ix}} \mapsto \vec{v}], \{\vec{b}\}, [\vec{\kappa} \mapsto \vec{\mathtt{ix}}], \mathtt{C}_b)] \end{array}}{\sigma_{mb}, \Delta_{mb}, S_{mb}, R_{mb}, \mathtt{C}_{mb} \xrightarrow{b_c}_{\gamma} \sigma_{mb}, \Delta_{mb}, S'_{mb}, R'_{mb}, \mathtt{C}_{mb}}\ \text{B-Run} \tag{6.3}$$

Proof

(1)	$b2r(b_c) = \imath_c$	Premise
(2)	$b2r(b) = \imath$	Premise
(3)	$\sigma_{mb} = [\vec{\mathtt{z}} \mapsto \vec{w}]$	Premise
(4)	$\sigma_{mb}(\vec{\mathtt{x}}) = \vec{\kappa}$	Premise
(5)	$\Delta_{mb} = [\vec{\kappa}_\Delta \mapsto (\vec{v}_\Delta, \vec{b}_\Delta)]$	Premise
(6)	$b \in dom(S_{mb})$	from Equation 6.3
(7)	$b2r(b) \in b2r(dom(S_{mb}))$	(6)
(8)	$dom(S_{g+}) = b2r(S_{mb})$	MGT
(9)	$\imath \in dom(S_{g+})$	(2, 7, 8)
(10)	$S_{mb}(b) = (\sigma_b, \{\vec{b}\}, [\vec{\kappa} \mapsto \vec{\mathtt{ix}}], \mathtt{C}_b)$	from Equation 6.3
(11)	$S_{g+}(b2r(b)) = (\sigma_b[\mathtt{rs} \mapsto \{b2r(\vec{b})\}], \mathtt{C}_\imath)$ where $\mathtt{C}_\imath = \mathtt{wait(rs)}; \mathtt{take}, \vec{\mathtt{ix}} := \vec{\mathtt{x}}; \theta(\mathtt{C}_b, R, V); \mathtt{give};$ and $R = dom(R_{g+}) \cup dom(S_{g+}), V = dom(\sigma_{mb})$	(MGT, 4, 10)
(12)	$S_{g+}(\imath) = (\sigma_\imath, \mathtt{C}_\imath)$ where $\sigma_\imath = \sigma_b[\mathtt{rs} \mapsto \{b2r(\vec{b})\}]$	(2, 11)
(13)	$S'_{mb} = S_{mb} \setminus \{b\}$	from Equation 6.3
(14)	$S'_{g+} = S_{g+} \setminus \{b2r(b)\}$	MGT, 13
(15)	$S'_{g+} = S_{g+} \setminus \{\imath\}$	2, 14
(16)	$R''_{g+} = R_{g+}[\imath \mapsto (\sigma_\imath, \phi, \mathtt{C}_\imath)]$	Premise
(17)	$\imath \in R''_{g+}$	(16)
(18)	$R''_{g+}(\imath) = (\sigma_\imath, \phi, \mathtt{C}_\imath)$	(16)
(19)	$\sigma_\imath(\mathtt{rs}) = \{b2r(\vec{b})\}$	(12)
(20)	$\forall b \in \vec{b}.\ (b = \mu \vee R_{mb}(b) = (-,-,\mathtt{skip}))$	from Equation 6.3
(21)	$\forall \imath' \in b2r(\vec{b}).\ \imath' = \mu \vee R_{mb}(\imath') = (-,-,-,\mathtt{skip})$	(MGT, 20)
[2]		
(22)	$\sigma_\imath, \Delta_{g+}, \Lambda_{g+}, \Gamma_{g+}, S'_{g+}, R''_{g+}, \mathtt{C}_\imath \xrightarrow{\imath}_{\gamma}$ $\sigma_\imath, \Delta_{g+}, \Lambda_{g+}, \Gamma_{g+}, S'_{g+}, R''_{g+}, \mathtt{C}'_\imath$ where $\mathtt{C}'_\imath = \mathtt{take}; \vec{\mathtt{ix}} := \vec{\mathtt{x}}; \theta(\mathtt{C}_b, R, V); \mathtt{give};$	Using WAIT and (19, 21)

[2] (21) should be `skip;give` according to the translation, however, it can trivially be reduced to `skip` using the SEQ-COMP-SKIP and GIVE rules to reach (21)

(23) $\Gamma_{g+}(b2r(b)) = \vec{\kappa}$ (MGT, 10)

(24) $\sigma_\iota, \Delta_{g+}, \Lambda_{g+}, \Gamma_{g+}, S_{g+}, R''_{g+}[\iota \mapsto (\sigma_\iota, \phi, \mathtt{C}'_\iota)], \mathtt{C}'_\iota \xrightarrow{\iota}_\gamma \sigma_\iota, \Delta_{g+}, \Lambda_{g+}, \Gamma_{g+}, S_{g+}, R''_{g+}[\iota \mapsto (\sigma_\iota, \{\vec{\kappa}\}, \mathtt{C}'_\iota)], \mathtt{C}''_\iota$ where $\mathtt{C}''_\iota = \vec{\mathtt{ix}} := \vec{\mathtt{x}}; \theta(\mathtt{C}_b, R, V); \mathtt{give};$ Using TAKE and (23)

[3]

(25) $R'''_{g+} = R''_{g+}[\iota \mapsto (\sigma_\iota, \{\vec{\kappa}\}, \mathtt{C}''_\iota)]$ Premise

(26) $\iota \in dom(R'''_{g+})$ (25)

(27) $R'''_{g+}(\iota) = (\sigma_\iota, \{\vec{\kappa}\}, \mathtt{C}''_\iota)$ (25)

(28) $\mathcal{E}[[\vec{\mathtt{x}}]]_{\sigma_{mb}} = \vec{\kappa}$ (4)

(29) $\sigma_{mb} \subseteq \sigma_{g+}$ (MGT)

(30) $\mathcal{E}[[\vec{\mathtt{x}}]]_{\sigma_{g+}} = \vec{\kappa}$ (28, 29)

(31) $\sigma_\iota, \Delta_{g+}, \Lambda_{g+}, \Gamma_{g+}, S_{g+}, R'''_{g+}, \mathtt{C}''_\iota \xrightarrow{\iota}_\gamma \sigma_\iota[\vec{\mathtt{ix}} \mapsto \vec{\kappa}], \Delta_{g+}, \Lambda_{g+}, \Gamma_{g+}, S_{g+}, R'''_{g+}, \theta(\mathtt{C}_b, R, V); \mathtt{give};$ Using Assgn and (30)

(32) $R'_{mb} = R_{mb}[b \mapsto (\sigma_b[\vec{\mathtt{ix}} \mapsto \vec{v}], \{\vec{b}\}, [\vec{\kappa} \mapsto \vec{\mathtt{ix}}], \mathtt{C}_b)]$ from Equation 6.3

(33) $R'_{g+} = R_{g+}[b2r(b) \mapsto (\sigma_b[\vec{\mathtt{ix}} \mapsto \vec{v}][\mathtt{rs} \mapsto \{b2r(\vec{b})\}][\vec{\mathtt{ix}} \mapsto \vec{\kappa}], \{\vec{\kappa}\}, \theta(\mathtt{C}_b, R, V); \mathtt{give};)]$ (MGT, 32)

(34) $R'_{g+} = R_{g+}[\iota \mapsto (\sigma_b[\vec{\mathtt{ix}} \mapsto \vec{v}][\mathtt{rs} \mapsto \{b2r(\vec{b})\}][\vec{\mathtt{ix}} \mapsto \vec{\kappa}], \{\vec{\kappa}\}, \theta(\mathtt{C}_b, R, V); \mathtt{give};)]$ (2, 33)

(35) $R'_{g+} = R_{g+}[\iota \mapsto (\sigma_\iota[\vec{\mathtt{ix}} \mapsto \vec{\kappa}], \{\vec{\kappa}\}, \theta(\mathtt{C}_b, R, V); \mathtt{give};)]$ (12, 34)

(36) $R'''_{g+}[\iota \mapsto (\sigma_\iota[\vec{\mathtt{ix}} \mapsto \vec{\kappa}], \{\vec{\kappa}\}, \theta(\mathtt{C}_b, R, V); \mathtt{give};)] = R_{g+}[\iota \mapsto (\sigma_\iota[\vec{\mathtt{ix}} \mapsto \vec{\kappa}], \{\vec{\kappa}\}, \theta(\mathtt{C}_b, R, V); \mathtt{give};)]$ (16, 25)

(37) $R'_{g+} = R'''_{g+}[\iota \mapsto (\sigma_\iota[\vec{\mathtt{ix}} \mapsto \vec{\kappa}], \{\vec{\kappa}\}, \theta(\mathtt{C}_b, R, V); \mathtt{give};)]$ (34, 36)

This allows us to prove the consequent of the proof statement as follows:

$\sigma_{g+}, \Delta_{g+}, \Lambda_{g+}, \Gamma_{g+}, S_{g+}, R_{g+}, \mathtt{C}_{g+}$

$\xrightarrow{\iota C}_\gamma \sigma_{g+}, \Delta_{g+}, \Lambda_{g+}, \Gamma_{g+}, S'_{g+}, R''_{g+}, \mathtt{C}_{g+}$ (Using R-Run and (9, 12, 15, 16))

$\xrightarrow{\iota C}_\gamma \sigma_{g+}, \Delta_{g+}, \Lambda_{g+}, \Gamma_{g+}, S'_{g+}, R''_{g+}[\iota \mapsto (\sigma_\iota, \phi, \mathtt{C}'_\iota)], \mathtt{C}_{g+}$ (Using R-Progress and (17, 18, 22))

$\xrightarrow{\iota C}_\gamma \sigma_{g+}, \Delta_{g+}, \Lambda_{g+}, \Gamma_{g+}, S'_{g+}, R''_{g+}[\iota \mapsto (\sigma_\iota, \{\vec{\kappa}\}, \mathtt{C}''_\iota)], \mathtt{C}_{g+}$ (Using R-Progress and (24))

$\xrightarrow{\iota C}_\gamma \sigma_{g+}, \Delta_{g+}, \Lambda_{g+}, \Gamma_{g+}, S'_{g+}, R'''_{g+}, \mathtt{C}_{g+}$ (Using (25))

$\xrightarrow{\iota C}_\gamma \sigma_{g+}, \Delta_{g+}, \Lambda_{g+}, \Gamma_{g+}, S'_{g+}, R'''_{g+}[\iota \mapsto (\sigma_\iota, \{\vec{\kappa}\}, \theta(\mathtt{C}_b, R, V); \mathtt{give};)], \mathtt{C}_{g+}$ (Using R-Progress and (25, 26, 31))

$\xrightarrow{\iota C}_\gamma \sigma_{g+}, \Delta_{g+}, \Lambda_{g+}, \Gamma_{g+}, S'_{g+}, R'_{g+}, \mathtt{C}_{g+}$ (Using (37))

Hence, proving soundness for this case

6.6.4 Sequential Composition Left

We assume the antecedent for this case, i.e. a successful execution of the MiniBoC Sequential Composition Left rule in the current behaviour b.

[3] the TAKE rule in (24) requires an extra condition, $available(\vec{\kappa})$ which comes from the well-formedness condition of MiniBoC semantics

$$\frac{\sigma_{mb}, \Delta_{mb}, S_{mb}, R_{mb}, \mathtt{C}_{mb1} \xrightarrow{b}_{\gamma} \sigma'_{mb}, \Delta'_{mb}, S'_{mb}, R_{mb}, \mathtt{C}_{mb1\prime}}{\sigma_{mb}, \Delta_{mb}, S_{mb}, R_{mb}, \mathtt{C}_{mb1}; \mathtt{C}_{mb2} \xrightarrow{b}_{\gamma} \sigma'_{mb}, \Delta'_{mb}, S'_{mb}, R_{mb}, \mathtt{C}'_{mb1}; \mathtt{C}_{mb2}} \text{ SEQ-COMP-LEFT}$$

We use induction to prove this case, and our inductive hypothesis is given below:

$$\sigma_{mb}, \Delta_{mb}, S_{mb}, R_{mb}, \mathtt{C}_{mb1} \xrightarrow{b}_{\gamma} \sigma'_{mb}, \Delta'_{mb}, S'_{mb}, R_{mb}, \mathtt{C}'_{mb1}$$
$$\Longrightarrow$$
$$\sigma_{g+}, \Delta_{g+}, \Lambda_{g+}, \Gamma_{g+}, S_{g+}, R_{g+}, \mathtt{C}_{g+1} \xrightarrow{\iota}{}^{*}_{\gamma} \sigma'_{g+}, \Lambda'_{g+}, \Gamma'_{g+}, S'_{g+}, R_{g+}, \mathtt{C}'_{g+1} \quad \text{(IH)}$$

where $b2r(b) = \iota$

Proof

(1) $\sigma_{mb}, \Delta_{mb}, S_{mb}, R_{mb}, \mathtt{C}_{mb1} \xrightarrow{b}_{\gamma} \sigma'_{mb}, \Delta'_{mb}, S'_{mb}, R_{mb}, \mathtt{C}_{mb1\prime}$ from Section 6.6.4

(2) $\sigma_{g+}, \Delta_{g+}, \Lambda_{g+}, \Gamma_{g+}, S_{g+}, R_{g+}, \mathtt{C}_{g+1} \xrightarrow{\iota}{}^{*}_{\gamma} \sigma'_{g+}, \Lambda'_{g+}, \Gamma'_{g+}, S'_{g+}, R_{g+}, \mathtt{C}'_{g+1}$ (IH)

We know that for some $\sigma''_{g+}, \Lambda''_{g+}, \Gamma''_{g+}, \Lambda''_{g+}, \Gamma''_{g+}, S''_{g+}, R_{g+}, \mathtt{C}''_{g+1}$ such that

(3) $\sigma_{g+}, \Delta_{g+}, \Lambda_{g+}, \Gamma_{g+}, S_{g+}, R_{g+}, \mathtt{C}_{g+1} \xrightarrow{\iota}_{\gamma} \sigma''_{g+}, \Lambda''_{g+}, \Gamma''_{g+}, \Lambda''_{g+}, \Gamma''_{g+}, S''_{g+}, R_{g+}, \mathtt{C}''_{g+1}$

(4) $\sigma''_{g+}, \Lambda''_{g+}, \Gamma''_{g+}, \Lambda''_{g+}, \Gamma''_{g+}, S''_{g+}, R_{g+}, \mathtt{C}''_{g+1} \xrightarrow{\iota}{}^{*}_{\gamma} \sigma'_{g+}, \Lambda'_{g+}, \Gamma'_{g+}, S'_{g+}, R_{g+}, \mathtt{C}'_{g+1}$

(5) $\sigma_{g+}, \Delta_{g+}, \Lambda_{g+}, \Gamma_{g+}, S_{g+}, R_{g+}, \mathtt{C}_{g+1}; \mathtt{C}_{g+2} \xrightarrow{\iota}_{\gamma} \sigma''_{g+}, \Lambda''_{g+}, \Gamma''_{g+}, \Lambda''_{g+}, \Gamma''_{g+}, S''_{g+}, R_{g+}, \mathtt{C}''_{g+1}; \mathtt{C}_{g+2}$ Using SEQ-COMP-LEFT and (3)

Repeating steps shown in (3), (4), (5) for the entirety of $\xrightarrow{\iota}{}^{*}_{\gamma}$, we can show that:

$$\sigma_{g+}, \Delta_{g+}, \Lambda_{g+}, \Gamma_{g+}, S_{g+}, R_{g+}, \mathtt{C}_{g+1}; \mathtt{C}_{g+2} \xrightarrow{\iota}{}^{*}_{\gamma} \sigma'_{g+}, \Lambda'_{g+}, \Gamma'_{g+}, \Lambda'_{g+}, \Gamma'_{g+}, S'_{g+}, R_{g+}, \mathtt{C}'_{g+1}; \mathtt{C}_{g+2}$$

Hence, proving soundness for this case.

Chapter 7

Evaluation

This chapter merely summarises the evaluation work done in the previous chapters:

- In Section 4.4, we evaluate MiniBoC and list further improvements for MiniBoC. We also suggest extending the existing BoC paradigm with cowns that not only protect data but protect specific operations on data such as read and write.
- In Section 5.6, we evaluate how well GIL+ achieves its objectives. The choices made in GIL+ have also been justified when formally introducing the syntax of GIL+.

Chapter 8

Conclusion

The primary objective of this thesis was to introduce GIL+ as a target language for BoC programs to enable reasoning about their correctness through various program analysis techniques such as machine-aided software verification. However, during the design process, we expanded the scope of GIL+ to achieve additional aims. These aims were outlined in Section 5.1, and they encompassed not only supporting the translation of BoC programs, but also providing a framework for reasoning about other concurrency paradigms. For instance, we demonstrated how GIL+ can be used to reason about JavaScript's async-await paradigm in Section 5.6.2.

Furthermore, we extended the BoC concurrency paradigm by presenting a small-step operational semantics for a language that incorporates BoC concepts. This extension allows us to explore and analyse the behaviour of programs that leverage the unique features offered by BoC. Additionally, we introduced a variant of BoC called FlexibleBoC, which may be better suited for programs that model resource contention.

By achieving these objectives and expanding the capabilities of GIL+, this thesis contributes to the field of concurrency analysis and provides a solid foundation for reasoning about BoC programs.

8.1 Future Work

This project opens up several intriguing avenues for future exploration and research, some of which have been discussed further below.

8.1.1 Read-Write Cowns

Cowns are quintessential in BoC and in their current form they protect data. More precisely, they protect 2 operations from occurring on data - reads and writes. However, an intriguing concept is the introduction of read-write cowns, which specifically prevent write operations on data while allowing other operations, such as reads, to proceed. This introduces a more fine-grained level of control and flexibility in the usage of cowns.

By introducing read-write cowns, developers can have more control over the concurrency of their programs. This allows for scenarios where multiple concurrent read operations can be performed on shared data while still ensuring exclusive access for write operations. This finer-grained protection can lead to improved performance and concurrency in certain situations.

8.1.2 Program Analysis

With the establishment of GIL+ as a target language for MiniBoC programs, a range of program analysis techniques can be introduced to reason about the correctness and properties of these programs. Symbolic execution, a powerful analysis method, can be applied to explore different program paths and generate test cases. Additionally, full verification techniques can be employed

to formally prove the correctness of MiniBoC programs, ensuring that they adhere to the desired specifications and properties.

By leveraging GIL+ as the foundation for program analysis, developers and researchers can gain deeper insights into the behaviour and characteristics of MiniBoC programs, leading to improved code quality, reliability, and overall understanding of Behaviour oriented Concurrency.

8.2 Ethical Considerations

Gillian's nature is to verify a program's correctness, a natural consequence of Gillian is the discovery of bugs in applications. Gillian has been effective in finding bugs in production-grade software. Gillian led to the discovery of 5 bugs in total [9], two of which were security vulnerabilities in C and JavaScript. Since, Gillian is open-source and available to the public [35]. This proves that Gillian and machine-aided software verification is an effective bug-finding technique.

Thus, It is possible that a *bad actor* can implement GIL+ and use it to find zero-day bugs in software using BoC. However, to successfully misuse GIL+, this bad actor will need to:

- Have a deep understanding of Gillian and GIL+
- Have a deep understanding of BoC
- Have an understanding of the OSS they are trying to exploit
- Find bugs that raise security concerns

Even though the odds of a malicious actor possessing the knowledge to successfully exploit bugs are infinitesimal, a simple counter-measure is to use Gillian to detect and resolve these bugs beforehand.

References

[1] Kerstiens C. Concurrency is not parallelism; 2020. https://blog.heroku.com/concurrency_is_not_parallelism [Accessed Feb 2023].

[2] Project Verona; 2023. https://www.microsoft.com/en-us/research/project/project-verona/ [Accessed May 2023].

[3] Turing A. Checking a Large Routine. National Cataloguing Unit for the Archives of Contemporary; 1949. Available from: https://turingarchive.kings.cam.ac.uk/publications-lectures-and-talks-amtb/amt-b-8.

[4] Hoare CAR. An Axiomatic Basis for Computer Programming. Commun ACM. 1969 oct;12(10):576–580. Available from: https://doi.org/10.1145/363235.363259.

[5] O'Hearn P, Reynolds J, Yang H. Local Reasoning about Programs that Alter Data Structures. In: Fribourg L, editor. Computer Science Logic. Berlin, Heidelberg: Springer Berlin Heidelberg; 2001. p. 1-19.

[6] Reynolds JC. Separation logic: a logic for shared mutable data structures. In: Proceedings 17th Annual IEEE Symposium on Logic in Computer Science; 2002. p. 55-74.

[7] Verified Software Group. Imperial College London;. https://vtss.doc.ic.ac.uk/research/gillian.html[AccessedJan2023] [Accessed Feb 2023].

[8] Fragoso Santos J, Maksimovic P, Ayoun S, Gardner P. Gillian, Part I: a Multi-language Platform for Symbolic Execution. In: Donaldson AF, Torlak E, editors. Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. ACM; 2020. p. 927-42. Available from: https://doi.org/10.1145/3385412.3386014.

[9] Maksimovic P, Ayoun S, Santos JF, Gardner P. Gillian, Part II: Real-World Verification for JavaScript and C. In: Silva A, Leino KRM, editors. Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II. vol. 12760 of Lecture Notes in Computer Science. Springer; 2021. p. 827-50. Available from: https://doi.org/10.1007/978-3-030-81688-9_38.

[10] Santos JF, Maksimović P, Ayoun SE, Gardner P. Gillian: Compositional Symbolic Execution for All; 2020.

[11] Calcagno C, Distefano D. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In: Bobaru M, Havelund K, Holzmann GJ, Joshi R, editors. NASA Formal Methods: Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18–20, 2011, Proceedings. vol. 6617 of Lecture Notes in Computer Science. Springer; 2011. p. 459-65.

[12] Jacobs B, Smans J, Philippaerts P, Vogels F, Penninckx W, Piessens F. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In: Proceedings of the Third International Conference on NASA Formal Methods. NFM'11. Berlin, Heidelberg: Springer-Verlag; 2011. p. 41–55.

[13] Bart Jacobs, Frank Piessens, et al . VeriFast source code;. https://github.com/verifast/verifast [Accessed Feb 2023].

[14] Muller P, Schwerhoff M, Summers AJ. Viper: A Verification Infrastructure for Permission-Based Reasoning. In: Jobstmann B, Leino KRM, editors. Verification, Model Checking, and Abstract Interpretation. Berlin, Heidelberg: Springer Berlin Heidelberg; 2016. p. 41-62.

[15] Floyd RW. In: Colburn TR, Fetzer JH, Rankin TL, editors. Assigning Meanings to Programs. Dordrecht: Springer Netherlands; 1993. p. 65-81. Available from: https://doi.org/10.1007/978-94-011-1793-7_4.

[16] Gardner P. Lecture slides for Scalable Software Verification. Imperial College London;. Available from: https://vtss.doc.ic.ac.uk/teaching/separationlogic.html.

[17] Lamport L. Proving the Correctness of Multiprocess Programs. IEEE Transactions on Software Engineering. 1977;SE-3(2):125-43.

[18] Separation logic and bi-abduction: Infer. Meta;. https://fbinfer.com/docs/separation-logic-and-bi-abduction/ [Accessed Feb 2023].

[19] OHearn PW. Resources, Concurrency, and Local Reasoning. Theor Comput Sci. 2007 apr;375(1–3):271–307. Available from: https://doi.org/10.1016/j.tcs.2006.12.035.

[20] Brookes S. A Semantics for Concurrent Separation Logic. In: Gardner P, Yoshida N, editors. CONCUR 2004 - Concurrency Theory. Berlin, Heidelberg: Springer Berlin Heidelberg; 2004. p. 16-34.

[21] The Rust Foundation. The Rust programming language;. https://www.rust-lang.org [Accessed May 2023].

[22] Gardner P. Gillian: a Multi-language Platform for Compositional Symbolic Analysis; 2021. https://www.college-de-france.fr/agenda/seminaire/logiques-de-programmes-quand-la-machine-raisonne-sur-ses-logiciels/gillian-multi-language-platform-for-compositional-symbolic-analysis [Accessed May 2023].

[23] Calcagno C, Distefano D, O'Hearn P, Yang H. Compositional Shape Analysis by Means of Bi-Abduction. In: Shao Z, Pierce BC, editors. POPL '09: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21–23, 2009. ACM; 2009. p. 289-300.

[24] The Infer Team. About Infer; 2017. https://fbinfer.com/docs/about-Infer [Accessed Jan 2023].

[25] Parkinson M. When Concurrency Strikes. Imperial College London; 2023. Talk.

[26] Overflow S. Stack Overflow Developer Survey; 2022. https://survey.stackoverflow.co/2022/#technology-most-loved-dreaded-and-wanted [Accessed May 2023].

[27] Agha GA. Actors: A Model of Concurrent Computation in Distributed Systems (Parallel Processing Semantics Open Programming Languages Artificial Intelligence). University of Michigan; 1985.

[28] Kraft P, Kazhamiaka F, Bailis P, Zaharia M. {Data-Parallel} Actors: A Programming Model for Scalable Query Serving Systems. In: 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22); 2022. p. 1059-74.

[29] Sang B, Petri G, Ardekani MS, Ravi S, Eugster P. Programming Scalable Cloud Services with AEON. CoRR. 2019;abs/1912.03506. Available from: http://arxiv.org/abs/1912.03506.

[30] Bernstein P. Actor-Oriented Database Systems. In: Proceedings of the 2018 IEEE 34th International Conference on Data Engineering. IEEE Computer Society; 2018. p. 13-4. Available from: https://www.microsoft.com/en-us/research/publication/actor-oriented-database-systems/.

[31] Cheeseman L. When Concurrency Strikes. Pre-print, under submission at OOPSLA'23.

[32] Microsoft. Verona C++ Runtime GitHub repository; 2022. https://github.com/microsoft/verona-rt [Accessed Jan 2023].

[33] Clarke D, Wrigstad T. External Uniqueness Is Unique Enough. In: Cardelli L, editor. ECOOP 2003 – Object-Oriented Programming. Berlin, Heidelberg: Springer Berlin Heidelberg; 2003. p. 176-200.

[34] JavaScript Documentation: async Function;. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function [Accessed June 2023].

[35] GillianPlatform. Gillian GitHub repository; 2023. https://github.com/GillianPlatform [Accessed Jan 2023].

Appendix A

"New" BoC Operational Semantics

This appendix contains an excerpt from the original Behaviour-Oriented Concurrency paper that has been submitted for OOPSLA'23.

Definition A.1 (Simple underlying programming language). A tuple $(\mathit{Context}, \mathit{Heap}, \hookrightarrow, \mathit{finished})$ is an *underlying programming language* if all of the points that follow hold. We use the identifiers $E, E', \ldots$ to range over elements of $\mathit{Context}$, and $h, h' \ldots$ for Heap.

(1) The execution relation $\hookrightarrow$ has signature $\hookrightarrow \subseteq (\mathit{Context} \times \mathit{Heap}) \to (\mathit{Context} \times \mathit{Heap})$.

(2) The set $\mathit{finished} \subseteq \mathit{Context}$ describes terminal contexts.

In **Definition** A.2 we show how we can extend any underlying language to obtain a BoC language. The extension enriches the underlying language so that programmers can *create* cowns and *spawn* new behaviours; also, the extension introduces concurrency to the language by enabling *running* multiple behaviours at a time.

Definition A.2 (BoC extensions). We define the Behaviour-Oriented Concurrency (BoC) extension for an underlying language, $(\mathit{Context}_u, \mathit{Heap}_u, \hookrightarrow, \mathit{finished}_u)$, and obtain a BoC tuple $(\mathit{Context}, \mathit{Heap}, \hookrightarrow, \mathit{finished}, \rightsquigarrow, \mathit{Tag})$:

(1) We require the execution relation to be extended to accommodate $\hookrightarrow_{\texttt{when}\ (\overline{\kappa})\ \{E\}}$ where $\hookrightarrow_{\texttt{when}\ (\overline{\kappa})\ \{E\}} \subseteq \mathit{Context} \to \mathit{Context}$

(2) Configurations are tuples, $\mathit{Conf} = \mathit{PendingBehaviours} \times \mathit{RunningBehaviours}$ where

$$P \in \mathit{PendingBehaviours} = (\mathit{Tag}^* \times \mathit{Context})^*$$
$$R \in \mathit{RunningBehaviours} = \mathcal{P}(\mathit{Tag}^* \times \mathit{Context})$$

(3) The execution relation $\rightsquigarrow \subseteq (\mathit{Conf} \times \mathit{Heap}) \to (\mathit{Conf} \times \mathit{Heap})$ is defined in Figure A.1.

[STEP]
$$\frac{E, h \hookrightarrow E', h'}{R \uplus (\overline{\kappa}, E), P, h \rightsquigarrow R \uplus (\overline{\kappa}, E'), P, h'}$$

[SPAWN]
$$\frac{E \hookrightarrow_{\texttt{when}\ (\overline{\kappa'})\ \{E''\}} E'}{R \uplus (\overline{\kappa}, E), P, h \rightsquigarrow R \uplus (\overline{\kappa}, E'), P : (\overline{\kappa'}, E''), h}$$

[RUN]
$$\frac{\left(\bigcup_{(\overline{\kappa'}, \_) \in (P' \cup R)} \overline{\kappa'}\right) \cap \overline{\kappa} = \varnothing}{R, P' : (\overline{\kappa}, E) : P'', h \rightsquigarrow R \uplus (\overline{\kappa}, E), P' : P'', h}$$

[END]
$$\frac{\mathit{finished}(E)}{R \uplus (\overline{\kappa}, E), P, h \rightsquigarrow R, P, h}$$

Figure A.1: Semantics for BoC

We will now discuss each of the rules in Figure B.2.

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 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.

RUN describes running a behaviour. A behaviour can be run 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.

END describes terminating a behaviour. This step requires that some running behaviour has reached some terminal state, it 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 RUN 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 run first.

These semantics provide a means to schedule behaviours such that no running behaviours are granted access to the same cowns at once. This is not in itself enough to isolate behaviours: we also need the *contents* of these cowns and the states of the behaviours to not overlap. Thus, we require the underlying language to provide a mechanism for memory isolation. These semantics permit, and in fact we strongly recommend, such a mechanism to ensure behaviour isolation.

Assume the underlying language has a mechanism for isolation, such as a type system. 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 [**?**]. 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 we 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, we expect that the type system ensures that the data protected (owned by) the cowns being released ($\overline{\kappa}$) are disjoint.

Assuming this provisioned isolation, we can claim that behaviours are atomic operations. A behaviour will acquire its cowns, execute with access to isolated access to its cowns until completion and then release the cowns. There is no way one behaviour can observe another partially executed behaviour.

Ensuring deadlock freedom

BoC is deadlock-free by construction, as the semantics in Figure A.1 can always reduce unless all the behaviours have finished. To show this, we assume the underlying semantics cannot get stuck, that is:

$$\forall E.\ [\ (\forall h.\ \exists E', h'.\ E, h \hookrightarrow E', h')\ \ \vee\ \ (\exists E', \kappa, E''.\ E \hookrightarrow_{\texttt{when}\ (\overline{\kappa})\ \{E''\}} E')\ \ \vee\ \ \mathit{finished}(E)\ ] \quad \text{(A.1)}$$

This may seem like too strong an assumption, as we 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*. We can also satisfy this by defining well-formed configurations and proving preservation of such over the execution relation, but this is more involved than we require here.

We proceed by case analysis on R being empty. If it is not empty, then we can apply the assumption (A.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 set using RUN. 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.

Appendix B

"Old" BoC Operational Semantics

This appendix consists of a draft of the old operational semantics.

B.1 Extending a language with BoC

Definition B.1 (Simple underlying programming language). A tuple ($Expr$, Val, Var, $Context$, Tag, $\texttt{error}$, $\hookrightarrow$) is an *underlying programming language* if all of the points that follow hold. We use the identifiers $e, e', \ldots$ range over elements of $Expr$, $v, v', \ldots$ for Val, $x, x', y, y', \ldots$ for Var, $\texttt{E}$ for $Context$, $\sigma, \sigma', \ldots$ for $Stack$, and $\kappa, \kappa', \ldots$ for Tag.

(1) $Expr$ and $Context$ are inductively defined sets where $Val \subseteq Expr$ and $Var \subseteq Expr$ and $Tag \cup \{\texttt{unit}\} \subseteq Val$. Each context, $\texttt{E}$, has a single hole (written $\bullet$) that can be replaced with some e (written $\texttt{E}[e]$) to create an element of $Expr$.

(2) The execution relation $\hookrightarrow$ has signature $\hookrightarrow \subseteq (Expr \times Stack) \to ((Expr \times Stack) \uplus \texttt{error})$ where $Stack = (Var \to Val)$. We require that the following property must hold:

(a) If $e_1, \sigma_1 \hookrightarrow e_2, \sigma_2$ then $visible(e_2, \sigma_2) \subseteq visible(e_1, \sigma_1)$ where $visible$ as defined in Definition B.2.

Definition B.2 (Visible). The function $visible$ with the signature $visible : (Expr \times Stack) \to \mathcal{P}(Tag)$ finds the elements of Tag visible from an underlying program configuration.

(1) $visible(\kappa, \varnothing) = \{\kappa\}$

(2) $visible(v, \varnothing) = \varnothing$ *where* $v \notin Tag$

(3) $visible(e, \sigma) = \bigcup_{v \in \mathrm{rng}(\sigma)} visible(v, \varnothing) \cup visible(e, \varnothing)$

(4) $visible(\texttt{E}[e], \sigma) = visible(\texttt{E}[\texttt{unit}], \sigma) \cup visible(e, \sigma)$

In Definition B.6 we show how we can extend any underlying language to obtain a BoC language. The extension enriches the underlying language so that programmers can *create* cowns and *spawn* new behaviours; also, the extension introduces concurrency to the language by enabling the *dispatch* and running of multiple behaviours at a time.

Definition B.3 (BoC extensions). We define the Behaviour-Oriented Concurrency (BoC) extension for an underlying language, ($Expr_u$, Val_u, $Context_u$, Tag_u, $\texttt{error}_u$, $\hookrightarrow$), and obtain an BoC tuple ($Expr$, Val, $Context$, Tag, $\texttt{error}$, $\hookrightarrow$, $\rightsquigarrow$, $ErrorConf$):

(1) $Val = Val_u$

(2) $Expr$ extends the inductive definition of $Expr_u$ with the two expressions:
$e \in Expr ::= \ldots$ as in $Expr_u \ldots \mid \texttt{cown}\ e \mid \texttt{when}\ (\overline{x = e})\ [\,\overline{y = e}\,]\ \{e\}$
We require $\overline{x = e}$, the sequence of required cowns in $\texttt{when}\ (\overline{x = e})\ [\,\overline{y = e}\,]\ \{e\}$ to be non-empty.

(3) *Context* extends the inductive definition of $Context_u$ so that:

$$\begin{aligned}\mathtt{E} \in \mathit{Context} ::= {} & \ldots \text{ as in } \mathit{Context}_u \ldots \\ \mid {} & \mathtt{cown}\ \bullet \\ \mid {} & \mathtt{when}\ (\overline{x = \kappa}, x = \bullet, \overline{x = e})\ [\,\overline{y = e}\,]\ \{e\} \\ \mid {} & \mathtt{when}\ (\overline{x = \kappa})\ [\,\overline{y = v}, y = \bullet, \overline{y = e}\,]\ \{e\}\end{aligned}$$

(4) *visible* is extended for *Expr* as follows:

$$\mathit{visible}(e, \sigma) = \begin{cases} \mathit{visible}(e', \sigma) & \text{if } e = \mathtt{cown}\ e' \\ \mathit{visible}(e_1, \sigma) \cup \mathit{visible}(e_2, \sigma) \cup \mathit{visible}(e_3, \varnothing) & \text{if } e = \mathtt{when}\ (\overline{x = e_1})\ [\,\overline{y = e_2}\,]\ \{e_3\} \\ \ldots & \textit{as in Definition B.2} \end{cases}$$

(5) *BehaviourId* is an enumerable set used for behaviour identifiers, ranged over by $\beta, \beta', \ldots$

(6) Behaviours are tuples, $\mathit{Behaviour} = (\mathcal{P}(\mathit{Var} \times \mathit{Tag}) \times \mathit{Expr} \times \mathit{Stack})$

(7) Configurations are tuples, $\mathit{Conf} = \mathit{AvailableCowns} \times \mathit{PendingBehaviours} \times \mathit{RunningBehaviours}$ where

$$\begin{aligned} A \in \mathit{AvailableCowns} &= \mathit{Tag} \to \mathit{Val} \\ P \in \mathit{PendingBehaviours} &= \mathit{BehaviourId} \to \mathit{Behaviour} \\ R \in \mathit{RunningBehaviours} &= \mathit{BehaviourId} \to \mathit{Behaviour} \end{aligned}$$

(8) *ErrorConf* represents configurations where some static or dynamic error occurs.

(9) The execution relation $\rightsquigarrow \subseteq \mathit{Conf} \to (\mathit{Conf} \uplus \mathit{ErrorConf})$ is defined in Figure B.2

[OS-STEP]

$$\frac{R(\beta) = (\overline{(x, \kappa)}, e, \sigma) \qquad e, \sigma \hookrightarrow e', \sigma'}{A, P, R \rightsquigarrow A, P, R[\beta \mapsto (\overline{(x, \kappa)}, e', \sigma')]}$$

[OS-CREATE]

$$\frac{R(\beta) = (\overline{(x, \kappa)}, \mathtt{E}[\mathtt{cown}\ v], \sigma) \qquad \kappa \notin \mathit{AllCownIds}(A, P, R)}{A, P, R \rightsquigarrow A[\kappa \mapsto v], P, R[\beta \mapsto (\overline{(x, \kappa)}, \mathtt{E}[\kappa], \sigma)]}$$

[OS-SPAWN]

$$\frac{\begin{array}{l} R(\beta) = (\overline{(x, \kappa)}, \mathtt{E}[\mathtt{when}\ (\overline{x' = \kappa'})\ [\,\overline{y = v}\,]\ \{e\}], \sigma) \\ \forall i, j \in 1 \ldots n. (i \neq j \implies \kappa'_i \neq \kappa'_j) \\ \beta' \notin \mathit{AllBehavIds}(A, P, R) \end{array}}{A, P, R \rightsquigarrow A, P[\beta' \mapsto (\overline{(x', \kappa')}, e, \overline{(y \mapsto v)})], R[\beta \mapsto (\overline{(x, \kappa)}, \mathtt{E}[\mathtt{unit}], \sigma)]}$$

[OS-DISPATCH]

$$\frac{P(\beta) = (\overline{(x, \kappa)}, e, \sigma) \qquad \sigma' = \sigma[x_1 \mapsto A(\kappa_1), \ldots, x_n \mapsto A(\kappa_n)]}{A, P, R \rightsquigarrow A \setminus \overline{\kappa}, P \setminus \beta, R[\beta \mapsto (\overline{(x, \kappa)}, e, \sigma')]}$$

[OS-END]

$$\frac{R(\beta) = (\overline{(x, \kappa)}, v, \sigma)}{A, P, R \rightsquigarrow A[\kappa_1 \mapsto \sigma(x_1), \ldots, \kappa_n \mapsto \sigma(x_n)], P, R \setminus \beta}$$

Figure B.1: BoC Semantics

We will now discuss each of the rules in Figure B.2.

OS-STEP describes a step where a running behaviour is able to make a step in the underlying language; this updates the expression and stack pair in the behaviour.

OS-CREATE describes the creation of a new cown. A new cown is created with a value, provided by the behaviour, and added to the set of available cowns. Other behaviours will be able to request access to this cown in the future through the new cown identifier that was provided.

OS-SPAWN describes spawning of a new behaviour. The behaviour requests access to a number of cowns $\overline{\kappa}$, provides captured variables that will construct the behaviour's starting stack and an expression that will be executed $\overline{x}$. This new behaviour is added to the set of pending behaviours and will wait until its cowns are available. There is no value outcome of creating this behaviour, and so the creating behaviour is provided with `unit`.

OS-DISPATCH describes dispatching a behaviour. Once the cowns requires by some behaviour are available, the behaviour has the potential to start running. Dispatching the behaviour makes all the required cowns unavailable to other behaviours; furthermore, the initial stack of the behaviour is enriched to provided access to the contents of the required cowns and the behaviour is set to running.

OS-END describes terminating a behaviour. This step requires that some running behaviour has reached some terminal value. The behaviour's acquired cowns are made available again; but, now, their contents reflect the value that can be found in the behaviour's stack.

In Definition B.9 we define what it means for a configuration to be well-formed. This requires the shorthands from Definition B.4.

Definition B.4 (Auxiliary Functions). We define the auxiliary functions

$$\begin{aligned} \downarrow_{cowns} &: \mathit{Behaviour} \to \mathcal{P}(\mathit{Tag}) & \mathit{AvailableCownIds} &: \mathit{Conf} \to \mathcal{P}(\mathit{Tag}) \\ \mathit{PendingBehavIds} &: \mathit{Conf} \to \mathcal{P}(\mathit{BehaviourId}) & \mathit{RunningBehavIds} &: \mathit{Conf} \to \mathcal{P}(\mathit{BehaviourId}) \\ \mathit{AllCownIds} &: \mathit{Conf} \to \mathcal{P}(\mathit{Tag}) & \mathit{AllBehavIds} &: \mathit{Conf} \to \mathcal{P}(\mathit{BehaviourId}) \end{aligned}$$

as follows:

$$\begin{aligned} ((\_, \overline{\kappa}), \_, \_)\downarrow_{cowns} &= \overline{\kappa} & \mathit{AvailableCownIds}(A, P, R) &= \mathrm{dom}(A) \\ \mathit{PendingBehavIds}(A, P, R) &= \mathrm{dom}(P) & \mathit{RunningBehavIds}(A, P, R) &= \mathrm{dom}(R) \\ \mathit{AllBehavIds}(A, P, R) &= \mathrm{dom}(P) \cup \mathrm{dom}(R) \end{aligned}$$

$$\mathit{AllCownIds}(A, P, R) = \bigcup_{\beta \in \mathrm{dom}(R)} (R(\beta)\downarrow_{cowns}) \cup \mathrm{dom}(A)$$

Definition B.5 (Well-Formed Configuration). A configuriaton is well-formed, $WF \subseteq (\mathit{AvailableCowns} \times \mathit{PendingBehaviours} \times \mathit{RunningBehaviours})$, $WF(A, P, R)$iff all of the following hold:

(1) $\mathit{PendingBehavIds}(A, P, R) \cap \mathit{RunningBehavIds}(A, P, R) = \varnothing$
(2) If $\beta_1 \neq \beta_2$ and $\{\beta_1, \beta_2\} \subseteq \mathit{RunningBehavIds}(A, P, R)$ then $R(\beta_1)\downarrow_{cowns} \cap R(\beta_2)\downarrow_{cowns} = \varnothing$
(3) If $\beta \in \mathit{RunningBehavIds}(A, P, R)$ then $R(\beta)\downarrow_{cowns} \cap \mathit{AvailableCownIds}(A, P, R) = \varnothing$
(4) If $\beta \in \mathit{PendingBehavIds}(A, P, R)$ and $\kappa \in P(\beta)\downarrow_{cowns}$:
- $\kappa \in \mathit{AvailableCownIds}(A, P, R)$, or
- there exists $\beta' \in \mathrm{dom}(R)$ such that $\kappa \in R(\beta')\downarrow_{cowns}$

(5) If $t \in (\mathrm{rng}(P) \cup \mathrm{rng}(R))$ and $t\downarrow_{cowns} = \overline{\kappa}$ and $i \neq j$ then $\kappa_i \neq \kappa_j$
(6) $\bigcup_{v \in \mathrm{rng}(A)} \mathit{visible}(v, \varnothing) \cup \bigcup_{(\_, e, \sigma) \in \mathrm{rng}(P) \cup \mathrm{rng}(R)} \mathit{visible}(e, \sigma) \subseteq \mathit{AllCownIds}(A, R)$

where all free variables are universally quantified.

Item 1 ensures that the pending and running behaviours have disjoint identifiers. Item 2 and Item 3 ensure that a cown acquired by a running behaviour is neither acquired by another running behaviour, nor is it available. Item 5 ensures pending behaviours only wait on cowns that exist. Item 4 ensures that no behaviour uses the same cown via different variables; we make this restriction to ensure cown contents are unambiguous when a behaviour end. Item 7 ensures that any cown that is visible, through a cown's contents or behaviour's stack, also exists as a known cown in the configuration.

B.2 Isolation Guarantees for Data-Race Freedom

Our semantics do not prevent any data-races. However, we can guarantee against data-races by with a number of isolation requirements in the underlying language.

We assume an underlying programming language as follows:

Definition B.6 (Underlying Language). A tuple (*Expr*, *Val*, *Addr*, *Context*, *Heap*, *Stack*, *Permission*, `error`, $\hookrightarrow$, $*$, `#`, $\_ \vDash \_ \sim \_$, $\_ \vDash \_ \diamond$, $\Downarrow$, *visible*) is an *underlying programming language*, if

1. The identifiers $e, e', \ldots$ range over elements of *Expr*, $v, v', \ldots$ for *Val*, $\mathtt{E}$ for *Context*, $\chi, \chi', \ldots$ for *Heap*, $\sigma, \sigma', \ldots$ for *Stack*, $\pi, \pi', \ldots$ for *Permission* and $\kappa, \kappa', \ldots$ for *Tag*.
2. *Expr* and *Context* are inductively defined sets, and $Val \subseteq Expr$, and $Tag \subseteq Val$, and $\mathtt{unit} \in Val$. We require the implicit application operator $\_ : Context \times Expr \to Expr$ where $\mathtt{E}[e]$ represents the application $\_\ \mathtt{E}\ e$.
3. $Var \to Val \in Stack$
4. An underlying configuration is $UnderlyingConf = (Expr \times Heap \times Stack \times Permission)$
5. The function *visible* with the signature $visible : UnderlyingConf \to \mathcal{P}(Tag)$ intuitively generates elements of *Tag* visible from an underlying program state.
   (a) $visible(\kappa, \chi, \varnothing, \pi) = \{\kappa\}$
   (b) $visible(\mathtt{unit}, \chi, \varnothing, \pi) = \varnothing$
   (c) $visible(e, \chi, \sigma, \pi) = \bigcup_{v \in \mathrm{rng}(\sigma)} visible(v, \chi, \varnothing, \pi) \cup visible(e, \chi, \varnothing, \pi)$
   (d) $visible(\mathtt{E}[e], \chi, \sigma, \pi) = visible(\mathtt{E}[\mathtt{unit}], \chi, \sigma, \pi) \cup visible(e, \chi, \sigma, \pi)$
6. The judgement $\chi \vDash \pi \diamond$ is defined. Intuitively, this judges that an element of *Permission* is well-formed w.r.t to a heap.
7. The judgement $\# \subset Permission \times Permission$ and an associative and commutative function $*$ with signature $* \subseteq (Permission \times Permission) \to Permission$ are defined.
   (a) $\pi_1 * \pi_2$ is defined iff $\pi_1 \# \pi_2$
   (b) An element $\epsilon \in Permission$ such that for all $\pi \in Permission$ it holds that $\pi * \epsilon = \pi$
   (c) If $(\pi_1 * \pi_2) * \pi_3$ is defined then $\pi_1 * \pi_3$ and $\pi_2 * \pi_3$ are defined.
   (d) If $\pi = \pi_1 * \pi_2$ then $visible(e, \chi, \sigma, \pi_1) \cup visible(e, \chi, \sigma, \pi_2) \subseteq visible(e, \chi, \sigma, \pi)$
   (e) If $\pi = \pi_1 * \pi_2$ then $\chi \vDash \pi \diamond$ iff $\chi \vDash \pi_1 \diamond$ and $\chi \vDash \pi_2 \diamond$
8. The function $\Downarrow \subseteq (Heap \times Val) \to Permission$ is defined. This function generates a protected set, intuitively those that should be protected by a value/capability.
9. The judgement $\pi \vDash e, \chi, \sigma \diamond$ is defined
   (a) It holds that $\chi \Downarrow v \vDash v, \chi, \varnothing \diamond$
   (b) If $\pi_1 \vDash e, \chi, \sigma \diamond$ and $\pi = \pi_1 * \pi_2$ then $visible(e, \chi, \sigma, \pi) = visible(e, \chi, \sigma, \pi_1)$
10. The judgement $\pi \vDash \chi_1 \sim \chi_2$ is defined. Intuitively this judges that an element of *Permission* considers two heaps to simulate one another.
    (a) If $\pi \vDash \chi_1 \sim \chi_2$ and $\pi = \pi_1 * \pi_2$ then $\pi_1 \vDash \chi_1 \sim \chi_2$ and $\pi_2 \vDash \chi_1 \sim \chi_2$.
    (b) If $\pi \vDash \chi_1 \sim \chi_2$ and $\chi_1 \vDash \pi \diamond$ then $\chi_2 \vDash \pi \diamond$
    (c) If $\pi \vDash \chi_1 \sim \chi_2$ and $\pi = (\chi_1 \Downarrow v)$ then $\pi = (\chi_2 \Downarrow v)$.
    (d) If $\pi \vDash \chi_1 \sim \chi_2$ then $visible(e, \chi_1, \sigma, \pi) = visible(e, \chi_2, \sigma, \pi)$
    (e) If $\pi \vDash \chi_1 \sim \chi_2$ and $\pi \vDash e, \chi_1, \sigma \diamond$ then $\pi \vDash e, \chi_2, \sigma \diamond$
11. The execution relation $\hookrightarrow$ has signature $\hookrightarrow \subseteq UnderlyingConf \to (UnderlyingConf \uplus \mathtt{error})$
    (a) If $e_1 \notin Val$ and $e_1, \chi_1, \sigma_1, \pi_1 \not\hookrightarrow \mathtt{error}$ then there exists e_2, χ_2, σ_2, π_2 such that $e_1, \chi_1, \sigma_1, \pi_1 \hookrightarrow e_2, \chi_2, \sigma_2, \pi_2$
    (b) If $e_1, \chi_1, \sigma_1, \pi_1 \hookrightarrow e_2, \chi_2, \sigma_2, \pi_2$ then $visible(e_2, \chi_2, \sigma_2, \pi_2) \subseteq visible(e_1, \chi_1, \sigma_1, \pi_1)$
    (c) If $e_1, \chi_1, \sigma_1, \pi_1 \hookrightarrow e_2, \chi_2, \sigma_2, \pi_2$ and $\chi_1 \vDash \pi_1 \diamond$ then $\chi_2 \vDash \pi_2 \diamond$.

(d) If $e_1, \chi_1, \sigma_1, \pi_1 \hookrightarrow e_2, \chi_2, \sigma_2, \pi_2$ and $\pi_1 \vDash e_1, \chi_1, \sigma_1 \diamond$ then $\pi_2 \vDash e_2, \chi_2, \sigma_2 \diamond$

(e) If $e_1, \chi_1, \sigma_1, \pi_1 \hookrightarrow e_2, \chi_2, \sigma_2, \pi_2$ and $\chi \vDash \pi_1 \diamond$ and $\pi_1 * \pi_3$ is defined and $\chi_1 \vDash \pi_3 \diamond$ then $\pi_2 * \pi_3$ is defined and $\pi_3 \vDash \chi_1 \sim \chi_2$

(f) If $e_1, \chi_1, \sigma_1, \pi_1 \hookrightarrow e_2, \chi_2, \sigma_2, \pi_2$ and $e_3, \chi_2, \sigma_3, \pi_3 \hookrightarrow e_4, \chi_3, \sigma_4, \pi_4$ and $\chi_1 \vDash \pi_1 \diamond$ and $\chi_1 \vDash \pi_3 \diamond$ and $\pi_1 * \pi_3$ is defined then there exists χ_4 such that $e_3, \chi_1, \sigma_3, \pi_3 \hookrightarrow e_4, \chi_4, \sigma_4, \pi_4$ and $e_1, \chi_4, \sigma_1, \pi_1 \hookrightarrow e_2, \chi_3, \sigma_2, \pi_2$

Definition B.7 (BoC extensions). We define the Behaviour-Oriented Concurrency (BoC) extension for a well-behaved language, $(\mathit{Expr}_u, \mathit{Val}_u, \mathit{Context}_u, \mathit{Heap}_u, \mathit{Stack}_u, \Downarrow, \hookrightarrow_u)$, as follows:

1. $\mathit{Val} = \mathit{Val}_u$
2. *Expr* extends the inductive definition of Expr_u, so that $\mathit{Val} \subseteq \mathit{Expr}$, and `cown` e and `when` $(\overline{x = e})$ $[\overline{y = e}]$ $\{e\}$ are expressions.

   Intuitively, $e \in \mathit{Expr} ::= \ldots$ as in $\mathit{Expr}_u \ldots \mid$ `cown` $e \mid$ `when` $(\overline{x = e})$ $[\overline{y = e}]$ $\{e\}$

   We require $\overline{x = e}$, the sequence of required cowns in `when` $(\overline{x = e})$ $[\overline{y = e}]$ $\{e\}$ to be non-empty.
3. *Context* extends the inductive definition of $\mathit{Context}_u$ so that:

$$\begin{aligned} \mathtt{E} \in \mathit{Context} ::= {} & \ldots \text{ as in } \mathit{Context}_u \ldots \\ \mid {} & \texttt{cown} \bullet \\ \mid {} & \texttt{when } (\overline{x = \kappa}, x = \bullet, \overline{x = e}) \; [\overline{y = e}] \; \{e\} \\ \mid {} & \texttt{when } (\overline{x = \kappa}) \; [\overline{y = v}, y = \bullet, \overline{y = e}] \; \{e\} \end{aligned}$$

4. The relation $\hookrightarrow$ with signature $\hookrightarrow \subseteq (\mathit{Expr} \times \mathit{Heap}_u \times \mathit{Stack}_u \times \mathit{Permission}_u) \to (\mathit{Expr} \times \mathit{Heap}_u \times \mathit{Stack}_u \times \mathit{Permission}_u) \uplus \mathtt{error}_u$ is defined as in $\hookrightarrow_u$
5. The judgement $\pi \vDash e, \chi, \sigma \diamond$ is extended for *Expr* as follows:

$$\begin{aligned} \pi \vDash \texttt{cown } e, \chi, \sigma \diamond \quad &\iff \quad \pi \vDash e, \chi, \sigma \diamond \\ \pi \vDash \texttt{when } (\overline{x = e_1}) \; [\overline{y = e_2}] \; \{e_3\}, \chi, \sigma \diamond \quad &\iff \quad \forall e \in \overline{e_1} \cup \overline{e_2}. (\pi \vDash e, \chi, \sigma \diamond) \wedge \epsilon \vDash e_3, \chi, \varnothing \diamond \end{aligned}$$

6. *visible* is extended for *Expr* as follows:

$$\mathit{visible}(e, \chi, \sigma, \pi) = \begin{cases} \ldots & \text{as in visible} \\ \mathit{visible}(e', \chi, \sigma, \pi) & \text{if } e = \texttt{cown } e' \\ \bigcup_{e' \in \overline{e_1} \cup \overline{e_2}} (\mathit{visible}(e', \chi, \sigma, \pi)) \cup \mathit{visible}(e_3, \chi, \varnothing, \epsilon) & \text{if } e = \texttt{when } (\overline{x = e_1}) \; [\overline{y = e_2}] \; \{e_3\} \end{cases}$$

7. *BehaviourId* is an enumerable set used for behaviour identifiers, ranged over by $\beta, \beta', \ldots$
8. Configurations are tuples $\mathit{Conf} = \mathit{Heap} \times \mathit{AvailableCowns} \times \mathit{PendingBehaviours} \times \mathit{RunningBehaviours}$ where

$$\begin{aligned} \mathit{PendingBehaviour} &= (\mathcal{P}(\mathit{Var} \times \mathit{Tag}) \times \mathit{Expr} \times \mathit{Stack}) \\ \mathit{RunningBehaviour} &= (\mathcal{P}(\mathit{Var} \times \mathit{Tag}) \times \mathit{Expr} \times \mathit{Stack} \times \mathit{Protected}) \end{aligned}$$

$$\begin{aligned} A \in \mathit{AvailableCowns} &= \mathit{Tag} \to \mathit{Val} \\ P \in \mathit{PendingBehaviours} &= \mathit{BehaviourId} \to \mathit{PendingBehaviour} \\ R \in \mathit{RunningBehaviours} &= \mathit{BehaviourId} \to \mathit{RunningBehaviour} \end{aligned}$$

9. *ErrorConf* represents a state where some static or dynamic error occurs.
10. The execution relation $\rightsquigarrow \subseteq \mathit{Conf} \to (\mathit{Conf} \uplus \mathit{ErrorConf})$ is defined in Figure B.2.

As have extended the definition of *Expr*, *Context*, $\_ \vDash \_, \_, \_ \diamond$ and *visible* we also require that underlying properties, that involve these components, still hold. We need to show the properties of the underlying still hold with the extended Expr and Context. This involves the wf conf, visible and exec

Definition B.8 (Auxiliary Functions). We define the auxiliary functions $\downarrow_{cowns} : (PendingBehaviour \cup RunningBehaviour) \to \mathcal{P}(Tag)$ as follows:

$$((\_, \overline{\kappa}), \_, \_, \_)\downarrow_{cowns} = \overline{\kappa}((\_, \overline{\kappa}), \_, \_)\downarrow_{cowns} = \overline{\kappa}$$

$perms : Heap \times (Val \cup PendingBehaviour \cup RunningBehaviour) \to Permission$ as follows:

$$perms(\chi, t) = \begin{cases} \chi \Downarrow v, & \text{if } t = v \\ *_{v \in \mathrm{rng}(\sigma)}(\chi \Downarrow v), & \text{if } t = (\_, \_, \sigma) \\ \pi, & \text{if } t = (\_, \_, \_, \pi) \end{cases}$$

$vis : Heap \times (Val \cup PendingBehaviour \cup RunningBehaviour) \to \mathcal{P}(Tag)$ as follows:

$$vis(\chi, t) = \begin{cases} visible(v, \chi, \varnothing, perms(\chi, v)), & \text{if } t = v \\ visible(e, \chi, \sigma, perms(\chi, t)), & \text{if } t = (\_, e, \sigma) \\ visible(e, \chi, \sigma, \pi), & \text{if } t = (\_, e, \sigma, \pi) \end{cases}$$

$AllCownIds : AvailableCowns \times RunningBehaviours \to \mathcal{P}(Tag)$ as follows:

$$AllCownIds(A, R) = \bigcup_{\beta \in \mathrm{dom}(R)} (R(\beta)\downarrow_{cowns}) \cup \mathrm{dom}(A)$$

$behavs : PendingBehaviours \times RunningBehaviours \to \mathcal{P}(BehaviourId)$ as follows:

$$behavs(P, R) = \mathrm{dom}(P) \cup \mathrm{dom}(R)$$

Definition B.9 (Well-Formed Configuration). We define the well-formed judgement $WF \subseteq (Heap \times AvailableCowns \times PendingBehaviours \times RunningBehaviours)$, where free variables are implicitly universally quantified, as $WF(A, P, R)$ iff:

1. $\mathrm{dom}(P) \cap \mathrm{dom}(R) = \varnothing$
2. If $\beta_1 \neq \beta_2$ and $\{\beta_1, \beta_2\} \subseteq \mathrm{dom}(R)$ then $R(\beta_1)\downarrow_{cowns} \cap R(\beta_2)\downarrow_{cowns} = \varnothing$
3. If $\beta \in \mathrm{dom}(R)$ then $R(\beta)\downarrow_{cowns} \cap \mathrm{dom}(A) = \varnothing$
4. If $\beta \in \mathrm{dom}(P)$ and $P(\beta)\downarrow_{cowns} = \overline{\kappa}$ and $i \neq j$ then $\kappa_i \neq \kappa_j$
5. If $\beta \in \mathrm{dom}(P)$ and $P(\beta)\downarrow_{cowns} = \overline{\kappa}$ then for all $\kappa \in \overline{\kappa}$:
   - $\kappa \in \mathrm{dom}(A)$, or
   - there exists $\beta' \in \mathrm{dom}(R)$ such that $\kappa \in R(\beta')\downarrow_{cowns}$
6. $\chi \vDash *_{t \in (\mathrm{rng}(A) \cup \mathrm{rng}(P) \cup \mathrm{rng}(R))} perms(\chi, t) \diamond$
7. $\bigcup_{t \in (\mathrm{rng}(A) \cup \mathrm{rng}(P) \cup \mathrm{rng}(R))} vis(\chi, t) \subseteq AllCownIds(A, R)$
8. If $\beta \in \mathrm{dom}(R) \cup \mathrm{dom}(P)$ and:
   - $R(\beta) = (\_, e, \sigma, \pi)$ or
   - $P(\beta) = (\_, e, \sigma)$ and $\pi = perms(\chi, P(\beta))$

   then $\pi \vDash e, \chi, \sigma \diamond$

B.3 Concurrency Semantics with Isolation Guarantees

[OS-STEP]

$$\frac{R(\beta) = (\overline{(x, \kappa)}, e, \sigma, \pi) \quad e, \chi, \sigma, \pi \hookrightarrow e', \chi', \sigma', \pi'}{\chi, A, P, R \rightsquigarrow \chi', A, P, R[\beta \mapsto (\overline{(x, \kappa)}, e', \sigma', \pi')]}$$

[OS-CREATE]

$$\frac{R(\beta) = (\overline{(x, \kappa)}, \mathtt{E[cown}\ v\mathtt{]}, \sigma, (\chi \Downarrow v) * \pi) \quad \kappa \notin \mathit{cowns}(A, R) \quad \pi \vDash \mathtt{E}[\kappa], \chi, \sigma \diamond}{\chi, A, P, R \rightsquigarrow \chi, A[\kappa \mapsto v], P, R[\beta \mapsto (\overline{(x, \kappa)}, \mathtt{E}[\kappa], \sigma, \pi)]}$$

[OS-SPAWN]

$$\frac{\begin{array}{l} R(\beta) = (\overline{(x, \kappa)}, \mathtt{E[when}\ (\overline{x' = \kappa'})\ [\overline{y = v}\,]\ \{e\}\mathtt{]}, \sigma, *_{v \in \overline{v}}(\chi \Downarrow v) * \pi) \\ \forall i, j \in 1 \ldots n. (i \neq j \implies \kappa'_i \neq \kappa'_j) \\ \beta' \notin \mathit{behavs}(P, R) \\ *_{v \in \overline{v}}(\chi \Downarrow v) \vDash e, \chi, \overline{(y \mapsto v)} \diamond \\ \pi \vDash \mathtt{E[unit]}, \chi, \sigma \diamond \end{array}}{\chi, A, P, R \rightsquigarrow \chi, A, P[\beta' \mapsto (\overline{(x', \kappa')}, e, \overline{(y \mapsto v)})], R[\beta \mapsto (\overline{(x, \kappa)}, \mathtt{E[unit]}, \sigma, \pi)]}$$

[OS-DISPATCH]

$$\frac{\begin{array}{l} P(\beta) = (\overline{(x, \kappa)}, e, \sigma) \\ \overline{\kappa} \subseteq \mathrm{dom}(A) \\ \sigma' = \sigma[x_1 \mapsto A(\kappa_1), \ldots, x_n \mapsto A(\kappa_n)] \\ \pi = *_{v \in \mathrm{rng}(\sigma')}(\chi \Downarrow v) \\ \pi \vDash e, \chi, \sigma' \diamond \end{array}}{\chi, A, P, R \rightsquigarrow \chi, A \setminus \overline{\kappa}, P \setminus \beta, R[\beta \mapsto (\overline{(x, \kappa)}, e, \sigma', \pi)]}$$

[OS-END]

$$\frac{R(\beta) = (\overline{(x, \kappa)}, v, \sigma, *_{x \in \overline{x}}(\chi \Downarrow \sigma(x)) * \pi)}{\chi, A, P, R \rightsquigarrow \chi, A[\kappa_1 \mapsto \sigma(x_1), \ldots, \kappa_n \mapsto \sigma(x_n)], P, R \setminus \beta}$$

Figure B.2: BoC Semantics