Memory safety for synchronous reactive programming

Darine Rammal

UNIVERSITÉ D'ORLÉANS

ÉCOLE DOCTORALE MATHÉMATIQUES, INFORMATIQUE, PHYSIQUE THÉORIQUE ET INGÉNIERIE DES SYSTÈMES

Laboratoire d'Informatique Fondamentale d'Orléans

THÈSE présentée par :

Darine RAMMAL

soutenue le : **19 Janvier 2024**

pour obtenir le grade de : **Docteur de l'Université d'Orléans**

Discipline/ Spécialité : **Informatique**

Memory Safety for Synchronous Reactive Programming

THÈSE dirigée par :

M. Pierre RÉTY	Maître de Conférences HDR, Université d'Orléans

RAPPORTEURS :

M. Fabrice MOURLIN	Maître de Conférences HDR, Université Paris-Est Créteil
M. Julien SIGNOLES	Directeur de Recherche, CEA Paris Saclay

JURY :

M. Abdelmalek BENZEKRI	Professeur, Président du jury, Université Paul Sabatier
Mme Wadoud BOUSDIRA	Maître de Conférences, Université d'Orléans (Co-Encadrante)
M. Frédéric GAVA	Professeur, Université Paris-Est Créteil
M. Frédéric LOULERGUE	Professeur, Université d'Orléans
M. Fabrice MOURLIN	Maître de Conférences HDR, Université Paris-Est Créteil
M. Julien SIGNOLES	Directeur de Recherche, CEA Paris Saclay

Membre invité :

M. Frédéric DABROWSKI	Maître de Conférences, Université d'Orléans (Co-Encadrant)

Dédicace

je dédie cet ouvrage
À ma mère et à mon père, qui m'ont accompagnée et encouragée tout au long de cette thèse. Ils ont su rapprocher les distances, même si des milliers de kilomètres nous séparent.
À mes soeurs, beaux-frères, et à tous ceux qui ont partagé avec moi chaque moment d'émotion durant la réalisation de ce travail. Leur soutien et leur confiance en moi ont été inestimables.
À mon tendre, qui m'a soutenu depuis le tout début pour que je puisse réaliser mon rêve.
À tous ceux que j'aime.

Remerciements

Je commence par exprimer ma profonde gratitude envers mon Dieu qui m'a soutenu quand j'en avais le plus besoin. Grâce à Lui, j'ai pu devenir la personne que je suis aujourd'hui. Je Le remercie sincèrement, car je n'ai jamais fait face à quoi que ce soit sans Le sentir présent avant, pendant et après chaque étape de ma vie.

Je souhaite adresser mes remerciements les plus chaleureux à mon directeur de thèse, Pierre Réty, ainsi qu'à mes co-encadrants, Wadoud Bousdira et Frédéric Dabrowski. Vous m'avez honoré en acceptant de superviser ma thèse, et je vous suis infiniment reconnaissante pour votre encadrement, vos encouragements et votre soutien. Votre présence dans ma vie académique et personnelle au cours de ces dernières années a été inestimable, et je ne peux pas exprimer assez combien je vous suis reconnaissante. Je nourris l'espoir sincère de m'approcher un jour de votre stature en tant que chercheurs, mais aussi en tant que personnes exemplaires sur le plan moral et humain.

J'adresse également mes remerciements aux membres du jury pour l'honneur qu'ils ont fait en acceptant de participer à cet événement important de ma vie académique. Je tiens à adresser mes remerciements particuliers à M. Signoles et à M. Mourlin pour avoir rapporté ma thèse. Cela a été un immense plaisir de bénéficier de leurs conseils avisés.

Je saisis cette occasion pour remercier chaleureusement toute l'équipe du LIFO pour leur accueil chaleureux et leur efficacité, qui ont rendu ces années de recherche agréables et enrichissantes. Un remerciement particulier à mes collègues de bureau Thi Bich Hanh Diep et Guillaume Cleuziou pour leurs échanges et le partage de leurs expériences.

Je ne peux pas passer sous silence ma reconnaissance envers l'équipe LMV à laquelle j'ai appartenu. Chacun des membres a contribué à créer une atmosphère familiale au-delà du simple cadre professionnel, et je chéris les bons moments passés ensemble, notamment lors de nos journées d'équipe.

Je tiens à exprimer ma profonde gratitude envers ma famille, mes parents, Mohammad et Insaf, mes sœurs, Sanaa, Hadia et Farah, ainsi que mes beaux-frères Ali, Hussein et Mohammad. Votre soutien inconditionnel et votre amour ont été essentiels tout au long de ces années loin de mon pays natal. Votre présence à mes côtés a été un pilier essentiel de ma vie, et je vous remercie du fond du cœur. Ce travail de thèse témoigne de l'estime et de l'amour que j'ai pour vous.

À toi, mon fiancé, Karim, je suis extrêmement reconnaissante pour ta présence et tes encouragements tout au long de ce parcours. Ta patience, ton écoute et ton soutien pendant la rédaction de ce manuscrit ont été précieux pour moi.

Un remerciement spécial à mes amis pour leur soutien inébranlable et leur confiance en moi. J'apprécie également énormément le soutien et la bonne humeur de mes collègues doctorants du LIFO, notamment lors de nos moments de joie partagés et pendant les pauses café. Vous avez rendu cette aventure académique encore plus riche.

Finalement, je tiens à remercier du fond du cœur toutes les personnes qui ont contribué, de près ou de loin, à la réalisation de cette thèse. Vos encouragements, vos conseils et votre présence ont été essentiels à mon succès, et je vous en suis infiniment reconnaissante.

Contents

List of Figures

Listings

Chapter 1

Introduction

Confidence isn't optimism or pessimism, and it's not a character attribute. It's the expectation of a positive outcome

– Rosabeth Moss Kanter

In the fast paced world of technology, where confidence is a crucial factor, the reliability of operating systems becomes paramount for ensuring the optimal performance of computer systems. Operating system refers to the programs and code that operate at a low-level and provide the basic functions of a computer, including device drivers and utility software. Nevertheless, a minor error or bug in an operating system can have major consequences, such as failures, data leakage, safety violations, etc. However, given that the C language combines the features of both high-level and low-level languages, it is commonly used for low-level programming. Consequently, several low-level operating system components are written in C or C++ (e.g. Linux kernel, FreeBSD, and MacOS's kernel). Yet, a critical concern arises: Pertaining to the responsibility of managing memory safety of these software. C, being a language that does not automatically handle memory management, places this responsibility squarely on the programmer's shoulders. Relying solely on the programmer becomes a matter of confidence. Even experienced C programmers may unintentionally introduce errors that compromise memory safety, leading to significant issues.

The majority of software vulnerabilities can be traced back to programming errors concerning memory management [89, 49]. These errors encompass memory leaks, use after free, uninitialized memory, and double free errors [31, 40, 86]. To elaborate, consider a C program with two pointers pointing to the same memory location. If the first pointer modifies the memory's value, it silently alters the value pointed to by the second pointer as well. This can lead to critical consequences. Additionally, Windows, a widely-used operating system, has suffered damages partly due to being primarily written in two unsafe programming languages, namely "C and C++". Even a minor mistake in the code of such foundational systems can render them vulnerable to attackers, leaving them exposed. Consequently, these errors can result in severe and intrusive effects, including privilege escalation. In fact, ap-

proximately 70% of security vulnerabilities in software products are attributed to memory safety violations, specifically errors with C/C++ pointers [92].

A significant endeavor has been underway to address memory errors in the C language, particularly concerning memory safety. However, the concept of safety extends beyond memory and type safety; it also involves safeguarding programming languages' abstractions, such as controlling aliasing. To tackle this challenge, various approaches have been explored. High-level programming languages like Java and SML employ a garbage collector, an automatic mechanism for managing memory allocation and deallocation in programs [50, 98]. While effective, this solution can lead to the accumulation of unused data, potentially slowing down the program. The garbage collector periodically scans the program's memory at runtime to identify and reclaim unused data, freeing up space for future allocations. Over three decades of research, alternative solutions have been proposed, including linear types, ownership types [33, 71, 58], and region-based memory management [95, 41, 37, 88], which incorporate concepts like borrowing and unique pointers [30]. The effort [30] presents a static method for ensuring data race freedom by enforcing at most one mutable reference to an existing object, mitigating issues related to aliased and mutable memory. Moreover, [95] explores the combination of regions and linear types. Among these solutions, region-based memory management has been a prominent focus in research. For instance, ML KIT [93], Cyclone [37], and real-time Java memory management in "Real-Time" systems [79, 57] draw from region-based approaches. In the pursuit of pragmatism, the development team behind Rust [84] has succeeded in combining the efficiency of high-level programming languages with the memory safety found in high-level languages.

Sponsored by Mozilla and collaboratively developed by a broad and varied community of contributors, Rust is a statically-typed programming language designed for performance, reliability and safety. As a new language, Rust is designed to mitigate the wave of security vulnerabilities, namely safe concurrency and memory management without the need for garbage collection. Rust's type system effectively limits aliasing possibilities, ensuring that accessible memory addresses are safely deallocated when a variable's scope is exited. However, strictly adhering to this type system can be overly restrictive. Here, Rust's pragmatism comes into play, allowing developers to create libraries where the type system can be disabled using the `"unsafe"` keyword. Features like `Rc` and `Mutexes` rely on this mechanism. `Rc`s are read-only references with safety based on reference counting, allowing flexibility in breaking lexical scope constraints. Nonetheless, libraries utilizing the `unsafe` feature must adhere to specific invariants to maintain memory safety [52]. In Rust, a multithreading library, referred to as `"std::thread::spawn"` in Rust terminology, enables concurrent programming using system threads. While the concurrent programming model is widely used, it can be too resource-intensive for certain systems, such as low-resource embedded systems. Several attempts have been made to explore the Rust type system, including Patina [82], one of the initial efforts, Oxide [97], and FR [73], which focused on formally modeling the Rust type system and proving their type and borrowing safety. They concentrated on the safe aspect of the Rust language. Meanwhile, Rustbelt [52] offers formal research that centers on the `unsafe` aspect of Rust.

Until now, we have explored memory issues in software in general. Let us now narrow our focus and delve into reactive systems, which are computer systems designed to respond

in real time to events or stimuli from their environment. Reactive systems can adopt various concurrency models, and some leverage cooperative threads to achieve their goals (e.g., the Akka framework for building concurrent and distributed systems in Java and Scala). Cooperative threads, implemented in user space, present a viable alternative. However, like any concurrency model, they come with their own set of trade-offs and require careful attention to ensure effective and efficient implementation. Several synchronous languages have been proposed for implementing reactive systems, including well-known examples like Esterel [22], Lustre [43], and Signal [60]. These languages have demonstrated their effectiveness in designing and verifying reactive systems. The design of these languages enables the use of formal methods to ensure strong reliability properties. However, achieving these guarantees often comes at the expense of reduced expressiveness, similar to automata. With the proliferation of IoT systems relying on microcontrollers and microprocessors rather than integrated circuits, there is a growing demand for more expressive programming languages that can maintain strong reliability properties. Some proposals have emerged to extend the model of synchronous reactive programming languages to general-purpose programming languages. Two notable examples are Reactive-C [26, 29] and ReactiveML [63] languages. However, when it comes to safety-critical or resource-constrained systems, these proposals encounter a familiar challenge. On one hand, Fairthreads, being a C library, places the burden of memory allocation on users, introducing the risk of dangling pointer errors or double-free errors. On the other hand, ReactiveML, as an extension of Ocaml [100], relies on a garbage collector for memory management, introducing execution overhead. However, as mentioned previously, garbage collection is not a suitable alternative for embedded systems as it causes breaks at runtime to deallocate unused memory. Ultimately, this comparison provided a good challenge for this thesis. Specifically, focusing on the Rust language, which is one of the most reliable languages nowadays to meet the requirements of reactive systems. Thus, the main objective is to have a synchronous reactive language dedicated to embedded systems, which is between Reactive-C and Reactive-ML such as to refrain from using the garbage collector and to provide a type safety without performance overhead.

In Synchronous Programming Languages, a collection of behaviors is executed in successive rounds called instants as shown in Figure 1.1. During each instant, behaviors react to signals, which may have values, produced either by the environment or by the behaviors themselves. An instant terminates when all reactions are completed, producing an output in response to the environment. At the start of each instant, the state of signals is cleared, and they receive new inputs from the environment. In synchronous languages like Esterel, reactions to the absence of a signal can occur instantaneously. However, such reactions can result in non-causal behaviors, which are detected and rejected through static checks. Similarly, the compiler ensures that programs are reactive or productive. However, these checks heavily rely on strict language restrictions, such as the absence of dynamic data and static scheduling. Fairthreads builds upon the SL language [29] and proposes a different approach by delaying reactions to the absence of a signal until the next instant. This approach avoids causality issues and allows for dynamic data and thread spawning. Solutions based on formal methods have been developed to guarantee productivity [6, 7, 5, 4]. However, these solutions assume the absence of memory errors.

In this thesis, we revisit Fairthreads by introducing a Rust-like type system in an experimental language called MSSL (Memory Safe Synchronous Language). MSSL is a reactive

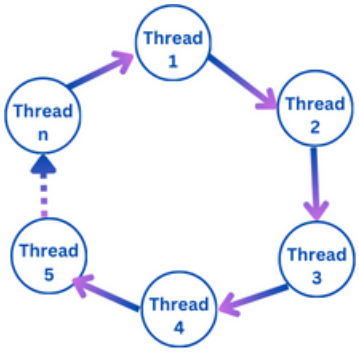

Figure 1.1: Round-Robin Scheduling

programming language based on cooperative threads and synchronous execution. In this programming model, threads run in turn. The change of control is done on demand through a yield command. A notion of signal allows threads to synchronise. Owing to the notion of logical time, called instants, threads progress synchronously with a coherent view of the state of signals (present or absent) at each instant. Due to the use of cooperative scheduling, all threads execute simultaneously. Consequently, there is no necessity to safeguard shared data among these threads using locking primitives. Nonetheless, even if the execution of threads is cooperative, the Rust type system requires adaptations to allow data sharing between threads since it introduces a new form of aliasing that must be controlled. Reference counting, as proposed by Rust, appears to be an interesting solution since it allows to get rid of lexical scope constraints. This is necessary to authorize the sharing of data between different threads. This solution is however limited by the fact that these references are read-only. Removing this constraint would break the memory safety properties at the thread level. We therefore propose to combine the reference counting approach with the aliasing constraints of standard Rust references. This new type of smart pointers, which is the main contribution of this thesis, is named `Trc` (Thread Reference Counting), which supports cooperative multithreading. Similar to Rust's `Rc` pointers, `Trc` pointers utilise a reference counting mechanism for their management. Intuitively, `Trc` pointers encapsulate shared data, akin to Rust's `Mutexes`. Unlike `Mutexes`, the data contained within `Trc` pointers does not require unpacking each time it is accessed. This feature effectively eliminates the intricate deadlock issues associated with `Mutexes`. One key distinction between `Trc` pointers and Rust's `Rc` lies in the mutability of their content. Our type system prohibits (1) aliasing of a `Trc` by single threads and (2) holding a reference to the content of a `Trc` at cooperation points. In both cases pointer invalidation could occur.

Overview and contributions

This thesis has led to the development of MSSL as a synchronous reactive language, inheriting the safety aspects of the Rust language. Our efforts have resulted in five main contributions: (1) we shed the light on FR_{FT}, a cooperative kernel of MSSL that lacks synchronization operations, and it extends FR [73]. (2) We introduce the new design of the smart pointer, `Trc`. In addition, as in Rust, we provide copy and move semantics, including mutable and immutable references, re-borrowing, lifetime, dynamic allocation via `Box`

that allows partial moves. Consequently, we supply a type system able of capturing all the aforementioned aspects. (3) Next, we provide the type, borrow and concurrency safety theorem results for FR_{FT}, based on the approach of [99]. This presented a challenge as we needed to argue that a program allowing data sharing between well-typed threads would execute safely without encountering issues. Additionally, we relied on the concept of logical instant, indicating that the execution of a safe program could be an infinite sequence of instants while preserving memory safety between threads. This ensures that when a thread executes, it maintains its type by preventing any dangling reference. Moreover, its execution preserves the type of other threads without creating undesired data in their memories, while adhering to the rules of `Trc`, lifetimes, and borrowing. (4) We provide the complete MSSL syntax, along with several reactive and non-reactive extensions to address any missing elements. We enrich MSSL with the notion of signals. Signals are powerful, flexible, and have compelling communication capabilities. At this point, threads can create signals, emit signals, etc. Besides, we outline how method invocation can be implemented. (5) Finally, we present an implementation of the full MSSL language in Java using the ANTLR framework [11], with a focus on `Trc` and borrows, while also incorporating other MSSL extensions. This implementation is based on the semantic and type rules outlined in this thesis, providing a comprehensive understanding of how to verify the soundness of our rules in relation to the actual implementation.

`Trc` vs Safe Rust

Continuing with the exploration of one of the key contributions of MSSL, let us now compare this novel alternative with the existing Rust constructs to emphasize the innovation brought by `Trc`. We will examine three examples written in Rust, each of which has a limitation. However, we will also present an example written in MSSL, leveraging the `Trc` extension, and highlighting its advantages. Listing 1.1 showcases a rejected example in Rust using the `Box` smart pointer, as follows:

```
1  fn createVec(){
2      let mut x = Box::new(0);
3      // Thread
4      // Transfert Ownership
5      thread::spawn(move||modifyVec(x));
6      *x=1; // x is moved!
7      }
8  fn modifyVec(x:Box<i32>){
9      *x=2; }
10 }
```

Listing 1.1: Rust with `Box`

In Listing 1.1, we introduce two functions: `createVec` and `modifyVec`. At line 2, we declare a smart pointer, `Box`, initialized to 0, and bind it to `x`. Subsequently, at line 5, we spawn a new thread using "`thread::spawn`" following Rust's syntax. This new thread will execute the

"modifyVec" function, taking the variable x as an argument. Consequently, the new thread has exclusively taken ownership of x, rendering it inaccessible to the current thread. This explains the error displayed on line 6. The characteristics of the Box pointer are as follows:

- Box is a *smart unique* pointer that is used to allocate memory on the heap
- *Weak point*: it **transfers** the ownership of that memory to the new thread

In Listing 1.2, we present the same example as in Listing 1.1 using the smart pointer Rc provided by Rust. This example is also rejected by the Rust type system as follows:

```
1 fn createVec(){
2     let mut x = Rc::new(0);
3     // Thread
4     // Multiple Ownership
5     thread::spawn(move||modifyVec(x.clone()));
6     *x=1; //Error: x is read-only!
7     }
8 fn modifyVec(x:Rc<i32>){
9     *x=2; }
```

Listing 1.2: Rust with Rc

In Listing 1.2, on line 2, we declare a smart pointer, Rc, initialized to 0, and associate it with x. Unlike Box in the previous listing, Rc is a reference-counted pointer characterized as read-only. Additionally, multiple instances of the same Rc can be created using the "clone" keyword in Rust syntax, incrementing the reference count by 1 for each instance. Consequently, on line 5, instead of transferring full ownership of x, we provide the new thread with a reference copy of x. This implies that both threads share the same read-only data. Hence, an error occurs when attempting to modify the contents of x on line 6. Furthermore, another error is displayed on line 5 because, according to Rust's type system, Rc is not considered safe for concurrent use between threads. Another proposed solution is demonstrated in Listing 1.3. The characteristics of the Rc pointer are as follows:

- Rc is an *immutable* counting reference. It allows multiple ownership of the same data created by Rc :: clone()
- *Weak point*: Rc is **not safe** to communicate with threads

Rust provides another type of reference counting pointer called Arc, which stands for Atomic Reference Counting. Arc is a solution specifically designed for safely sharing data between threads in a concurrent environment. Listing 1.3 displays the same example as in Listings 1.1 and 1.2, but this time using the smart pointer Arc. Unfortunately, this example is also rejected by the Rust type system, as depicted below:

```
1 fn createVec(){
2     let mut x = Arc::new(Mutex::new(0));
```

```
3     // Thread
4     // Multiple Ownership
5     thread::spawn(move||modifyVec(x.clone()));
6     // Mutate the value
7     *x.lock().unwrap()=1;
8     }
9  fn modifyVec(x:Arc<Mutex<i32>>){
10    *x.lock().unwrap()=2;
11    }
```

Listing 1.3: Rust with `Arc`

In Listing 1.3, accepted by the Rust type system, on line 2, we declare a smart pointer, `Arc`, initialized to 0, and bind it to `x`. Similar to `Rc` in the previous listing, `Arc` is a reference-counting pointer characterized as read-only, but it can be safely shared between threads. To modify the contents of `Arc`, which is immutable but has multiple aliases, Rust employs a model called inner mutability, akin to `Mutex`, which safeguards data from unsafe access. Consequently, no errors are displayed on line 7 when modifying the contents of `x`. The characteristics of the `Arc` pointer are as follows:

- `Arc` is an *atomically reference counted*. As `Rc`, it is *immutable* but it is **thread-safe**.
- *Weak point*: To mutate the `Arc`'s content, a *mutex* is required! **However, in cooperative mode, no mutex is needed**

Now let's present the same example as the previous listings, using the MSSL syntax. This example uses the `Trc` extension provided by MSSL as follows:

```
1  fn createVec(){
2     // Shared value is encapsulated in a Trc
3     let mut x=trc(0);
4     // Thread
5     // Multiple Ownership
6     spawn(modifyVec(x.clone));
7     // Mutate the value
8     *x=1;
9     }
10 fn modifyVec(x:Trc<i32>){
11    *x=2; }
```

Listing 1.4: MSSL with `Trc`

Listing 1.4 is written in MSSL, where on line 3, we declare a smart pointer, `Trc`, initialized to 0, and link it to `x`. Similar to `Rc` and `Arc`, on line 6, we provide a copy of `Trc`'s reference to the new thread. However, unlike `Arc`, on line 8, we modify the contents of `x` without requiring a `mutex`. Finally, the characteristics of the `Trc` pointer are as follows:

- `Trc` is a *smart pointer*. As `Rc`, it allows *multiple ownership* of the same data and as `Arc`, it is **safe-thread**
- *strong point*: `Trc` combines data *sharing* and *mutability* between threads without requiring the use of **locking primitives**

To conclude, after comparing the constructs offered by the Rust language with the new `Trc` extension provided by MSSL, we reach to the following conclusions:

- `Trc` has multiple ownership (shared data between threads) compared to `Box`.
- `Trc` is thread-safe, unlike `Rc`.
- `Trc` allows mutable references, unlike `Rc` and `Arc`.
- `Trc` does not require a mutex for data mutation, unlike `Arc`.

Outline

This thesis is structured into the following chapters:

Chapter 2:

Chapter 2 is divided into two sections. The first section provides an overview of reactive systems and synchronous languages. In this part, we delve into the characteristics of three prominent synchronous languages: Lustre [72], Signal [60], and Esterel [23]. We thoroughly examine the essential concepts and clock operators for each language, with a particular focus on addressing the causality problem for the Esterel language. Furthermore, we discuss the requirements of these languages to ensure the safe functioning of IoT systems and real-time systems. In the second section, we delve into the Rust language, placing emphasis on its type system, dynamic allocation, and memory management features.

Chapter 3:

In Chapter 3, we introduce FR_{FT} as a subset of MSSL, encompassing the cooperative kernel of MSSL. FR_{FT} serves as a formal model of the core Rust language, extended with a novel smart pointer and two multi-threading extensions. This chapter primarily focuses on exploring the characteristics of the Rust type system, including ownership, borrowing (and re-borrowing), and lifetimes. These aspects include copy semantics, move semantics, and partial semantics. Subsequently, we delve into the central contributions of FR_{FT}, particularly `Trc`. Building upon the concepts introduced in FR [73], we formalize the semantics of this language in the context of operational semantics. This semantic model incorporates three new extensions: the first extension facilitates data sharing between threads without the need for locks, the second extension enables the creation of cooperative threads, and the last extension provides explicit cooperation between threads until the end of an instant.

Chapter 4:

Chapter 4 presents the FR_{FT} type system and delves into its ability to preserve the properties of the new `Trc` extension between threads, while ensuring memory safety during cooperation. In this chapter, we provide a comprehensive proof of soundness for FR_{FT}, leveraging the syntactic approach introduced by Wright and Felleisen [99]. Specifically, we demonstrate that a well-typed expression in FR_{FT} is inherently either a value or a reducible expression, relying on the lemma of Step Progress 4.9. Furthermore, we establish the lemma of Step Preservation 4.10, which confirms that when a semantic rule is applied to a well-typed expression, the resulting evaluation step remains well-typed with the same type, all while preserving the borrowing and `Trc` invariants. Moreover, this lemma ensures that other threads also maintain their well-typed property. Building upon the Step Progress and Step Preservation lemmas, we derive two additional lemmas: the Slice Progress lemma 4.14, which affirms that a well-typed thread can be executed more than once, and the Slice Preservation lemma 4.15, which guarantees that a well-typed thread remains well-typed even after multiple executions, with other threads also maintaining their well-typed status. Continuing from the Slice Progress and Slice Preservation lemmas, we deduce the Instant Progress lemma 4.16, asserting that a well-typed program can perform an instant. Additionally, the Instant Preservation lemma 4.17 claims that a well-typed program remains well-typed after making an instant. Finally, chapter 4 concludes with our theorem of type, borrow, and concurrency safety. This essential theorem establishes that if we execute a program from an initial state, it is capable of performing one or more instants while adhering to type safety, borrowing safety, and concurrency safety principles in the context of FR_{FT}.

Chapter 5:

In chapter 5, we present the MSSL language as a whole, encompassing various extensions. We extend FR_{FT} to incorporate the invocation of methods outside the `spawn` expression, and we elaborate on how the MSSL type system effectively supports and handles this extension. Furthermore, we enhance the reactive aspect of FR_{FT} by introducing the concept of a signal. This enriches MSSL with multiple reactive extensions, such as signal creation, signal emission, and signal awaiting, among others. These extensions highlight the system's reactivity and underscore the significance of signals, showcasing the inherent flexibility present in the core of MSSL.

Chapter 6:

Chapter 6 showcases an implementation of the complete MSSL language that combines the practical aspects of Rust, specifically FR [73], with a new smart pointer called `Trc` and reactive constructs. This implementation is carried out in Java and is publicly available on GitHub. As previously mentioned, MSSL aims to enhance the Fairthreads programming model. To achieve this goal, we conduct a thorough inspection of references and data shared between threads in an MSSL program. Subsequently, we facilitate an automatic translation to Reactive-C. By utilizing this implementation, we ensure high local memory safety for

each thread and maintain robust global memory safety between threads, leveraging the characteristics of Rust. Additionally, we provide examples of Fairthreads and demonstrate how MSSL programs are translated into Fairthreads. Finally, we summarize our findings and conclusions in Chapter 7.

Chapter 2

State of the Art

Education is the most powerful weapon which you can use to change the world

– Nelson Mandela

In this chapter, we start with an introduction to synchronous languages, where detailed outlines of Esterel, Lustre, and Signal are provided. Following that, we explore the syntax and type system of the Rust language. Lastly, the chapter concludes with a discussion on the diverse efforts undertaken to formalize the semantics of Rust and showcase its safety and reliability.

Dans ce chapitre, nous commençons par une introduction aux langages synchrones, où des contours détaillés d'Esterel, Lustre et Signal sont fournis. Ensuite, nous explorons la syntaxe et le système de types du language Rust. Enfin, le chapitre se conclut par une discussion sur les divers efforts entrepris pour formaliser la sémantique de Rust et démontrer sa sécurité et sa fiabilité.

1 Background and Overview

Reactive programming [76] has experienced a significant surge in popularity from the 1980s to the present day, making it a well-suited paradigm for developing interactive and event-driven applications. The ever-growing number of connected objects in the embedded systems industry has led to the continuous emergence of new techniques and opportunities. Reactive programming simplifies the development of such applications by providing abstractions to express time-varying values. These real-time systems, known as reactive systems, operate in constant interaction with the environment and are bound by strict non-functional constraints, such as limited time and memory resources. Often, these systems are safety-critical, where even a minor data leak at the memory level can lead to unforeseen failures, posing risks to the mission, the system, or its surroundings. Most real-time reactive systems comprise components that evolve simultaneously and communicate with each other. Ensuring reliability is crucial in such systems. To address the requirements of reactive systems and implement their fundamental concepts, the first three synchronous languages were developed: (1) Esterel [28] is an imperative and control-flow language, (2) Lustre [43] is a declarative and data-flow language, and (3) Signal [60] is also a declarative and data-flow language. Additionally, other synchronous languages have been proposed, including SL [29], Reactive-C [26, 29], Lucid [32, 80] and ReactiveML [63].

However, the majority of these languages, like Esterel, are primarily implemented using host languages, particularly the C language. Real-time reactive systems, which require "soft real-time" constraints, demand safe programming. One notable concern [31, 39] is the careless use of mutable pointers, leading to aliasing issues. For instance, in a C program with two pointers pointing to the same memory location, any modification performed by the first pointer will silently affect the value pointed to by the second pointer. Unfortunately, the C compiler is not equipped to handle data races, dangling pointers, or prevent use-after-free scenarios reliably. In response to this problem, various efforts have been made to mitigate memory errors in the C language, specifically focusing on aliasing control. A garbage collector is often employed to manage active objects that are no longer in use and are considered garbage. Additionally, the relationship between dangling pointers and the garbage collector is defined as follows: the last reference to an object is destroyed with its garbage [50]. In the context of a language called Cyclone (resembling C but with improved safety), [37] concentrates on region-based memory management. Each object in Cyclone belongs to a specific region, and heap regions can be garbage collected. [37] introduces a static typing discipline to prevent dangling pointer dereferences and space leaks. Their approach ensures that each pointer is bound to a unique region, thereby determining the validity of the data being pointed to.

Rust, a statically typed programming language, focuses on three main objectives: speed, safety and concurrency. It is sponsored by Mozilla and developed collaboratively by a diverse community of contributors in an open-access manner. Its standout feature is the ownership and borrowing system, which sets it apart from other languages. This system ensures both high performance and memory safety, effectively avoiding the issues often encountered in languages like C. The type system in Rust guarantees that pointers used in a program are always valid, a marked departure from the C language's behavior. This

is achieved through the concept of unique ownership, which controls aliasing. In Rust, there are two types of pointers: mutable pointers (`&mut` T), which can mutate data and are unique, and immutable or shared pointers ($\&\ T$), which are only allowed to read data and can be duplicated. Furthermore, each pointer in Rust is bound to a lifetime, defining its validity throughout the program. These rules, along with others, play a crucial role in ensuring memory safety. To enforce these rules and guarantee memory safety, Rust employs static analysis known as "borrow checking". This analysis verifies the adherence to the ownership and borrowing system, preventing potential memory-related issues.

This chapter begins by providing an overview of synchronous languages, offering detailed outlines of Esterel, Lustre, and Signal. It then delves into the syntax and type system of Rust. Finally, the chapter concludes by discussing the various efforts made to formalize the semantics of Rust and demonstrate its safety and reliability.

2 Reactive System

The term "reactive system" was introduced by D. Harel and A. Pnueli [47]. It pertains to information systems that engage in dynamic interactions with their environment, responding to external stimuli at a rate determined by the environment [45]. Reactive systems react swiftly and consistently, capable of bringing about changes in their surroundings. This is crucial in domains like embedded applications, transportation, mobile systems, human-machine interfaces, and real-time operating systems. Several reactive programming frameworks, such as ReactiveX, the Akka toolkit for distributed systems, and the Reactive Manifesto with principles for building reactive systems, have been developed. These systems employ a combination of asynchronous and synchronous approaches to design and verify their behavior. The synchronous approach, assuming instantaneous reactions with zero processing time, is a more recent development and offers several advantages.

3 Synchronous Languages

The synchronous paradigm emerged in the 1980s, coinciding with the rapid growth of real-time embedded systems in various domains like surveillance systems, multimedia, and factories. These systems are often critical, necessitating safe and efficient programming methods. To address these requirements, synchronous languages [17] were developed. The synchronous approach caters to the needs of real-time systems programming by providing elegant, simple, and expressive mathematical foundations, enabling efficient formal analyses and property verification. In the synchronous approach, real-time considerations are abstracted, and reasoning is done on a logical time scale. This logical time is structured as a sequence of instants, where each instant represents a system reaction, transforming incoming information (input signals) into outgoing information (output signals). Signals play a central role in this reaction calculation and serve as a means of communication among different components of a synchronous program. Dedicated research conducted by various teams, including experts in automation and computer science, has led to the creation of specialized languages such as Esterel [28], an imperative event-driven and control-flow

oriented language; Lustre [43], a declarative event-driven and data-flow oriented language; and Signal [60], a declarative, sample-driven, and data-flow oriented language. Additionally, other formalisms and languages like SyncCharts [9, 85], SL [29], Scade [34], Argos [64], Reactive-C [27], Lucid Synchrone [32], ReactiveML [63], Zelus [24], and SCCharts [46] have incorporated further ideas and advancements. In the remainder of this chapter, we will provide a brief overview of the features of the first three synchronous languages.

3.1 Lustre

Overview

Lustre [43] is a programming language for synchronous data-flow systems developed in 1984, specifically designed for reactive systems that interact with their environments in real-time. Its inception was inspired by the concurrency model proposed by Gilles Kahn in the 1970s [54], which was originally intended for describing signal processing systems. Lucid Synchrone [32], another data-flow language, is based on Lustre's synchronous model. It was introduced to address the need for extending Lustre with higher-level mechanisms. The primary aim of Lucid Synchrone is to be easily understandable and accessible to control engineers who are familiar with various systems of equations, such as differential equations, finite difference equations, Boolean equations, etc. These engineers are also accustomed to working with synchronous data-flow formalisms like block diagrams, analogue networks, and logic circuits. Furthermore, the data-flow model used in the Lustre language has influenced the development of the industrial language Scade [34], which supports a graphical version of this language.

Main Concept

A Lustre program consists of a series of modules called nodes, each comprising a set of equations that determine the values of program variables at each time step. These equations are commonly treated as "time invariants" [44]. For instance, if we have the equation $X = (X_1, X_2, ..., X_n)$ at time n (logical time), the value of the variable X would be X_n. In Basic Lustre, each variable or expression is referred to as a flow, which represents an infinite sequence of values. As these flows are described by equations and not assignments, their order is not significant. Lustre is generally categorized as a mono-clock program, meaning that all processes within the system share the same logical time scale. Although some processes may be deactivated at specific instants using clocks, they execute at the same periodic real-time rate when active. The clock keyword plays a crucial role in Lustre and is closely related to flows. It is represented as a Boolean that determines when a flow is active or inactive (i.e., the activation conditions). For example, in Lustre, $z = x + y$, the flow z is defined by the expression $x + y$, indicating that the arithmetic operation extends to flows. Thus, at any instant t, $z_t = x_t + y_t$.

Clock Operators

Lustre not only supports data operators but also temporal operators, encompassing various operations that define sequential and dependent functions. One essential temporal operator is the "`pre`" (short for "previous") operator, acting as memory or delay, allowing Lustre to retain values from previous cycles. For instance, if we have the flow $X = (x_0, x_1, ..., x_n)$, then `pre`(X) corresponds to the flow (`nil`, x_0, ..., x_{n-1}), where "`nil`" denotes an undefined value ("uninitialized", similar to imperative languages). To ensure safety, the Lustre compiler guarantees that data operators cannot be applied to `nil`.

To establish initial values, Lustre introduced the "`->`" ("followed by") operator. For flows $X = (x_0, x_1, ..., x_n)$ and $Y = (y_0, y_1, ..., y_n)$ of the same type, "X -> Y" results in $(x_0, y_1, ..., y_n)$. The first argument of the resulting flow remains equal to X, while the rest adopts the values of Y in subsequent instants, effectively concealing the "`nil`" value introduced by the `pre` operator.

In a Lustre program, cycles are employed to compute the value of each variable, leading to cyclic behavior. However, variables can be defined conditionally, enabling their computation only under specific conditions. Two temporal operators facilitate this: (1) "X = E `when` B", where E is an expression and B is a Boolean, produces a sequence of values of E when the Boolean B is true (i.e., X is computed on clock B). The time of X is then the sequence of cycles where B is true. (2) The "`current`" operator allows us to define an expression E of clock B, and "`current`(E)" yields the same clock as the Boolean B. For each cycle of this clock, the computed value is the value taken by E during the last cycle where B was true.

Importantly, Lustre programs cannot contain syntactically cyclic definitions. An acyclic equation has a unique solution, ensuring a unique order of static dependence between flows (topological sorting). For example, *"x = x and `pre(x)`"* is cyclic, but *"x = y and `pre(x)`"* is acyclic. Having cyclic equations can lead to causality problems, making it crucial for control engineers to avoid zero delay loops in their systems to guarantee functional behavior, as specified in Esterel for reactive and deterministic programs to be logically correct and constructive. Here is a Lustre example illustrating the equations provided below:

$$\begin{cases} Y_1 = \dfrac{X_1}{2} \\ Y_n = \dfrac{X_n + X_{n-1}}{2} \end{cases}$$

```
node filter (i:  real) returns (o:  real);
   let
      o = (i + 0 -> pre(i))/2;
   tel
```

Table 2.1: Example in Lustre with corresponding equations

3.2 SIGNAL

Overview

SIGNAL [60, 19, 8] is a declarative synchronous language specifically designed for specifying real-time systems [18, 19]. Similar to Lustre, SIGNAL is a data-flow language that handles infinite sequences of typed values, known as signals instead of flows. However, SIGNAL differs from Lustre in its extensive utilisation of the concept of absence of signals. In SIGNAL, clocks can be explicitly defined as first-class objects, allowing for greater flexibility. Additionally, SIGNAL permits the interconnection of clocks within the same program. For example, the expression "z = x `default` y" denotes that the clock of z is the union of the clocks of x and y. This feature makes SIGNAL a multi-clock language, while Lustre, in comparison, is a single-clock language where all clocks are sub-clocks of the program's base clock.

Main Concept

In SIGNAL, the fundamental building blocks are known as *signals*, which are represented by ordered sequences of values of the same type. These values are implicitly indexed on a discrete logical time, where the logical time is considered a partially ordered set [20]. This logical time determines the clock of a *signal*, defining the set of instants when the *signal* is present.

Within SIGNAL, there are specific *signals* known as `pure` *signals*, which solely represent their presence and have an event type. When two *signals* share the same clock, they are termed synchronous *signals*. A SIGNAL program comprises equations that define local and output *signals* as block diagrams. Each *signal* is associated with a clock, and its value, whenever present, must be well-defined. The sum of the clocks of the *signals* involved in a program forms its activation clock.

Interconnections between *signals* in SIGNAL are established using operators, enabling the creation of new *signals* and more. The clock model is integral to SIGNAL's system of equations. Therefore, the SIGNAL compiler ensures that the clocks of *signals* are well-defined, consistent, and free from circular references. This verification process, known as clock calculus, synthesizes the order of calculation to enhance the program's execution efficiency.

Signal Operators

The SIGNAL core language consists of a concise set of primitive constructs (operators) that each defines its own meaning and imposes constraints on clocks and evaluation order, i.e., causality:

- z := x `op` y. Similar to Lustre [43], these operators are mono-clock. For example, an operation like "z := x+y" requires z, x, and y to have the same clock, meaning they

need to be present simultaneously. These operations extend instantaneous relations to relations dealing with flows, where $\forall\ t, z_t = x_t + y_t$.

- *y:= x$1.* This operator is specific to the SIGNAL language and enables access to past values of a signal. Similar to the previous operator, it requires the input signal and output signal to be present simultaneously. Consequently, the output signal y is delayed by one step (*$1*) with respect to the input signal x. Additionally, for initialisation, $y := x\$\ \texttt{init}\ x_0$ is introduced, where x_0 denotes a constant.

- $z := x$ `when` b (b is boolean). This is a multi-clock operator that results in an event-based downsampling of signals. Specifically, the output signal z is equal to signal x when both x and b are true. Otherwise, z is not generated. The example below illustrates the behavior of the `when` construction. It is important to note that we use "$\perp$" to indicate the absence of data:

$$\begin{aligned}
&\texttt{b}: \texttt{ff}\ \ \texttt{tt}\ \ \perp\ \ \perp\ \ \texttt{tt}\ \ \texttt{ff}\ \ \perp\ \ \texttt{ff}\\
&\texttt{x}: \texttt{x}_1\ \ \texttt{x}_2\ \ \texttt{x}_3\ \ \perp\ \ \texttt{x}_4\ \ \perp\ \ \texttt{x}_5\ \ \perp\\
&\texttt{z}: \perp\ \ \texttt{x}_2\ \ \perp\ \ \perp\ \ \texttt{x}_4\ \ \perp\ \ \perp\ \ \perp
\end{aligned}$$

- $z := x$ `default` y. This is a multi-clock operator that enables a deterministic merging of two signals. The output signal z is the combination of the input signals x and y. In essence, this operation is deterministic, with priority typically given to signal x. If x is present, z will take its value; otherwise, if y is present, z will adopt the value of y. Conversely, if neither x nor y is present, z will be absent. The example below illustrates the behavior of the merge construction:

$$\begin{aligned}
&\texttt{x}: \texttt{x}_1\ \ \perp\ \ \perp\ \ \texttt{x}_2\ \ \texttt{x}_3\ \ \texttt{x}_4\ \ \perp\ \ \texttt{x}_5\\
&\texttt{y}: \perp\ \ \texttt{y}_1\ \ \perp\ \ \texttt{y}_2\ \ \perp\ \ \texttt{y}_3\ \ \perp\ \ \texttt{y}_4\\
&\texttt{z}: \texttt{x}_1\ \ \texttt{y}_1\ \ \perp\ \ \texttt{x}_2\ \ \texttt{x}_3\ \ \texttt{x}_4\ \ \perp\ \ \texttt{x}_5
\end{aligned}$$

- $p \mid q$. This is a communication operator that allows two programs, p and q, to communicate by sharing the same signals. For example: $z := x + y \mid x := z\ \$\ \texttt{init}\ a$. Here, *'a'* represents a constant value. The result is as follows:

$$\begin{aligned}
&z_t = x_t + y_t \ \text{ and}\\
&x_t = z_{t-1} \ \text{ such that } x_1 = a \text{ and } \texttt{t} \geqslant 1
\end{aligned}$$

  As a result: $z_t = z_{t-1} + y_t$ and $z_0 = a$.

3.3 Esterel

Overview

Esterel [28, 23, 21], conceived by Gerard Berry in the early 1980s, is an imperative synchronous language that embraces an event-driven style through control-flow description.

Its primary purpose is to program real-time systems, particularly embedded systems. These systems operate in response to the dynamic evolution of external processes (e.g., their environment) with which they interact, requiring precise control over their behavior. Such systems are subject to temporal constraints [42], making Esterel suitable for real-time process controllers, GSM terminals, smart cards, human-machine interfaces, and more.

By contrast with Lustre [43] and SIGNAL [60] languages, which adopt a data-flow approach [54] using systems of equations to manipulate data flows, an Esterel program consists of multiple concurrent processes communicating via broadcast signals and organized into modules. SyncCharts [9], a graphical formalism, is employed for the Esterel synchronous language, and determinism plays a crucial role in ensuring the control and correctness of reactive systems. Additionally, it is noteworthy that a commercial version of SyncCharts, known as Safe State Machine [10], is also available.

Main Concept

In Esterel, *signals* serve as the fundamental means of communication. Each *signal* is characterised by a name, a type and a state (either present or absent). At the control level, *signals* enable conditional activation of specific parts of the program, known as *pure signals*. These *signals* play a role in controlling the program flow, such as using "`present then else`" constructions. Additionally, at the data level, *signals* act as a mechanism for sharing valued variables between concurrent processes. These *signals* are referred to as *valuated signals*, and they allow sharing information via `emit` operations. Through instantaneous broadcasts, all processes within the *signals*' visibility area can access their state, promoting deterministic behavior. Moreover, in line with the synchronous approach, a *signal* can only have one state at a time during an instant, and it is re-evaluated at each instant, reflecting its event-driven nature.

The Esterel language offers a range of high-level control structures that empower programmers to implement synchronous reactive systems. Here is a sample list of instructions used to define the control flow based on *signal* operations:

- `emit` S. This instruction is instantaneous and emits the signal S, terminating immediately after emission.
- `pause`. This instruction suspends the execution until the next instant.
- `await` S. This instruction suspends the execution until the next occurrence of the signal S. If the signal S is already present in the current instant, "`await immediate S`" terminates immediately.
- `present` S `then` p `else` q `end`. This instructions checks the presence of the signal S and executes the instruction p if S is present. Otherwise, it executes the instruction q.
- `signal` S `in ... end`. This instruction declares a local signal S, starting at "`in`" and terminating at "`end`".

- `suspend` p `when` S. In this case, if the signal S is present, instruction p remains in its state. If S is absent, p is executed for the current instant.
- `abort` p `when` S. The statements of the instruction p are killed as soon as the signal S occurs (i.e., strong preemption). If p terminates, the block terminates as well.
- [p || q]. This instruction executes instructions p and q in parallel. The parallel construction completes only when both p and q have terminated. For example, [`await` S_1 `||` `await` S_2]; emit S_3 indicates that the S_3 signal is emitted when both S_1 and S_2 have been emitted.
- [p ; q]. This instruction executes instruction q immediately after instruction p terminates.

Constructive causality analysis

The Esterel language possesses a notable characteristic known as the "coherence law", wherein it can react to the absence of a *signal* in each reaction, ensuring that each *signal* undergoes a single evaluation in each instant. This implies that if a *signal* is absent, emissions cannot be executed in the current reaction. However, in Esterel, this feature becomes complex due to the possibility of having several potential emitters for the same signal. Below, we provide two examples of Esterel programs to illustrate this behavior. In the first example, we have the relation:

```
present s1 else emit s2 end
```
(2.1)

Under the synchronous assumption, the state of *signal* `s2` is determined simultaneously with the state of *signal* `s1`. This instantaneous feedback, characteristic of synchronous formalisms, can lead to issues. For instance, the second example:

```
present s else emit s end
```
(2.2)

Example 2.2, though syntactically correct, does not hold a meaningful interpretation. Here, the instantaneous reaction to the absence of *signal* s results in its immediate emission (with the `else emit s end` branch). As per the law of *signal* coherence, if s is a local *signal*, it should not be emitted elsewhere. Consequently, two possible executions arise for this program: (1) assuming that s is present, then s is not emitted (a contradiction); (2) assuming that s is absent, then s is emitted (another contradiction). The presence or absence of a *signal* at a given instant leads to an incoherent or, more accurately, causally incorrect program. Similarly, it introduces the possibility of obtaining incorrect programs through the parallel composition of correct modules. For example, the below program is also non-deterministic:

```
signal s1, s2 in
  present s1 then emit s2 end present
  present s2 then emit s1 end present
end signal
```
(2.3)

Over the years, numerous solutions have been proposed to address the issue mentioned earlier in Esterel, known as "causality cycles." The objective is to accommodate the widest

range of programs that exhibit "reasonable" behavior. For instance, in a study by [23], a program is deemed logically correct if, for any stage of its execution (i.e., for any sequence of inputs), there exists only one evaluation of its signals that adheres to the *signal* consistency law. This ensures a deterministic execution without any possibility of deadlock. Another approach, known as the constructive semantics [21], draws inspiration from digital circuits and three-valued logic. It avoids making assumptions about the presence state of signals. Instead, this approach employs pre-determined information and performs a symbolic execution to ascertain which signals can no longer be transmitted. Such a program is referred to as constructive, where the behavior is determined systematically based on symbolic execution.

The Esterel "causality cycles" problem has been investigated and as a result the proposal of several synchronous languages was born. One such language is SL (Synchronous Language) [29], which is based on Esterel's semantics but introduces a new approach to solve the issue. In SL, a *signal* is considered absent under the condition of a construction (`if/else`), which delays the evaluation of the else branch to the next instant. This approach ensures that a *signal* cannot be simultaneously present and absent, avoiding conflicting states. Subsequent research has led to the development of several synchronous programming paradigms based on the SL language, such as SML (SugarCubes [25]) [81], ReactiveML [63], Scheme [62], and Reactive-C [26]. The latter Reactive-C, proposed by Boussinot, aimed to improve the C language by incorporating reactive systems similar to Esterel, while avoiding the "causality cycles" problem. Similar to SL, Reactive-C resolves the issue by delaying the time following the reaction to the absence of a *signal*. This is achieved by introducing new instances of reactive procedures, which are utilised dynamically in concurrent execution. Reactive-C introduces a concept called "Fairthreads", which sets it apart from other formalisms relying on standard parallel operators adapted to instants. Fairthreads demonstrate efficiency by sharing a memory between threads without overhead. Additionally, they can be executed cooperatively when linked to a scheduler or preemptively (unlinked threads). These threads can also dynamically migrate from one scheduler to another within the same application. In Fairthreads, an application consists of multiple logical clocks instead of a single clock, as seen in other formalisms like Lustre. Each clock is associated with a scheduler controlling the fair threads cooperatively linked to it, allowing simultaneous use of clocks. The program in Listing 2.1 demonstrates the use of the Fairthreads library through C programming.

```
#include "fthread.h"
#include "stdio.h"

void r (void *id)
{
   while (1) {
      fprintf (stderr,"Reactive ");
      ft_thread_cooperate ();
   }
}

```

```
12 void p (void *id)
13 {
14    while (1) {
15       fprintf (stderr,"Programming!\n");
16       ft_thread_cooperate ();
17    }
18 }
19
20 int main(void)
21 {
22   ft_scheduler_t sched = ft_scheduler_create ();
23   ft_thread_create (sched,r,NULL,NULL);
24   ft_thread_create (sched,p,NULL,NULL);
25   ft_scheduler_start (sched);
26   ft_exit ();
27   return 0;
28 }
```

Listing 2.1: Print Reactive Programming with Fairthreads

Listing 2.1 presents a C program using the Fairthreads library. Starting with the main function, from line 20 to line 28, one scheduler and two fair threads are created using the `ft_scheduler_create` and `ft_thread_create` functions. These fair threads are linked to the same scheduler based on the parameters of the `ft_thread_create` function, and each executes the r and p functions respectively. Subsequently, once the fair threads and the scheduler have been successfully created, the scheduler starts using the `ft_scheduler_start` function. On line 26, we call the `ft_exit` function to prevent the whole program from immediately terminating. This program is designed to print "Reactive programming!" at regular intervals. This rhythm is maintained by using the `ft_cooperate` function, invoked on lines 8 and 16. In accordance with the Fairthreads library, `ft_cooperate` serves as a way of explicitly relinquishing control to the scheduler operating within it. Note that an error will be generated if the fair thread is not linked to a scheduler. In the following Listing, we demonstrate the synchronization of fair threads using events as follows:

```
1  #include "fthread.h"
2  #include <stdio.h>
3  #include <unistd.h>
4
5  ft_event_t e1, e2;
6
7  void b1 (void *args)
8  {
9    //broadcast e1
10   ft_thread_generate (e1);
11   //wait e2
```

```
12   ft_thread_await (e2);
13   fprintf (stdout,"receive e2\n");
14 }
15
16 void b2 (void *args)
17 {
18   //wait e1
19   ft_thread_await (e1);
20    fprintf (stdout,"receive e1\n");
21   //broadcast e2
22   ft_thread_generate (e2);
23 }
24
25 /************************************/
26 int main(void)
27 {
28    int c, *cell = &c;
29    ft_thread_t th1, th2;
30    ft_scheduler_t sched = ft_scheduler_create ();
31
32    //create events
33    e1 = ft_event_create (sched);
34    e2 = ft_event_create (sched);
35
36    th1 = ft_thread_create (sched,b1,NULL,NULL);
37    th2 = ft_thread_create (sched,b2,NULL,NULL);
38    ft_scheduler_start (sched);
39    pthread_join (ft_pthread (th1),(void**)&cell);
40    pthread_join (ft_pthread (th2),(void**)&cell);
41    fprintf (stdout,"exit\n");
42    exit (0);
43 }
```

Listing 2.2: Fairthreads with events

In Listing 2.2, similar to Listing 2.1, one scheduler is created using the `ft_scheduler_create` function, along with two threads created via the `ft_thread_create` function, each executing functions b1 and b2 respectively. However, these two threads communicate through events. Consequently, two events are created using the `ft_event_create` function and linked to e1 and e2 respectively. Subsequently, within the body of the b1 function, the fair thread promptly generates the e1 event in the scheduler it belongs to via the `ft_thread_generate` function. Through this action, e1 will be broadcasted to all fair threads associated with the same scheduler. The fair thread then implicitly waits for the e2 event, which hasn't been generated yet, using the `ft_thread_await` function. The same

process applies to the b2 function. Finally, the `pthread_join` function suspends the execution of the thread call until the fair thread provided as a parameter terminates. Note that in Chapter 6, we use examples to demonstrate how we translate the MSSL program into Fairthreads.

While the development of reliable and safe systems is critical, many embedded systems continue to be implemented in efficient yet "insecure" languages, like C. Such languages lack memory safety and expose systems to time-dependent errors that go unchecked by the compilers. To achieve a safe Esterel-like language and avoid unknown vulnerabilities, the need to replace C with a more reliable language arises. Some languages, like Reactive-ML, implement garbage collectors for automatic storage management (as an extension of Ocaml [100]). However, garbage collection introduces significant overhead and is not suitable for real-time embedded systems. To address this challenge, Rust is introduced as a language that combines ownership and region types to manage memory without garbage collection. Rust ensures safe use of pointers at compile time, providing strong safety for embedded software. The rest of this chapter will go through into Rust's type system and its key characteristics.

4 An Introduction to Rust

In 2006, Mozilla introduced Rust, a modern, secure, concurrent, and convenient systems programming language. Unlike languages like C, Python, or Java, Rust ensures zero-cost memory safety without incurring runtime performance penalties. Rust manages memory internally, avoiding direct memory management functions like calloc or malloc, thereby preventing the memory-related errors commonly associated with the C languages [31, 74, 39, 40, 13].

While Rust supports both functional and imperative paradigms, it is renowned for its performance, concurrency, and memory safety. Released as Rust version 1.0 in 2015, it has undergone continuous development since then. The distinctive feature that sets Rust apart from other languages lies in its "type system," which plays a pivotal role in addressing memory trust issues. Rust's type system is built on two key concepts: Ownership and Borrowing. These concepts concede developers control over the lifetime of data and the ability to manage mutations, ensuring safe memory usage.

The foundation of Rust's type system is influenced by the works of [33, 71, 58], focusing on linear types and ownership, and [41, 37], emphasizing region-based memory management. In this section, we present a subset of Rust, thoroughly examining Rust source code through various examples. We will introduce the most important syntax constructs and key concepts of Rust's type system, which will be essential for the subsequent parts of this thesis. For a comprehensive introduction to Rust, we recommend The Rust Programming Language [84], an introductory book authored by the Rust developers.

4.1 Syntax of Rust

Rust is primarily considered an imperative language. Specifically, local bindings are created using the "`let`" keyword, and by default, they are immutable. To obtain mutable bindings, the "`mut`" keyword is employed. Additionally, these local variables are stored in the stack by default:

```
//creates an integer on the stack and binds it to x;
//by default, x is declared as immutable.
//this means that it cannot modify its content;
let x: i32 = 1;
//error: cannot assign twice to immutable variable;
x = 5;
//creates an integer on the stack and binds it to y;
//y is declared as mutable;
//then the content of y can be modified;
let mut y: i32 = 1;
//ok!
y = 5;
```

Listing 2.3: Creation of new Bindings

Listing 2.3 demonstrates the creation of two local variables: `x`, which is immutable by default (i.e., cannot be modified), and `y`, which is mutable using the `mut` keyword (i.e., can be modified). Similar to C/C++, Rust includes concepts of integers, unsigned integers, and floats. However, the syntax for these types is different. In Rust, signed integers are denoted by **`i`** and unsigned integers by **`u`**, and these types are explicitly sized with 8, 16, 32, or 64. For instance, **`u8`** represents an 8-bit unsigned integer, and **`i16`** represents a 16-bit signed integer. Rust also provides two forms of floating-point size: **`f32`** and **`f64`**.

Besides local bindings, Rust incorporates several familiar elements found in other programming languages. These include `structs, traits` (comparable to Java's interfaces), `enumerations, lambda functions,` and `modules`.

```
// struct
struct Circle {
    a: i32,
    b: i32,
}
//enums
enum Option<T> {
    None,
    Some(T),
}
```

Listing 2.4: Destructuring Structures and Enumerations in Rust

In Listing 2.4, we define a type called "Circle", which consists of two integer fields: a and b. In Rust, product types (i.e., structs) resemble C/C++ structs, where each struct contains a set of fields that can be accessed using the dot operator. Furthermore, Rust uses enumerations (enums) to define sum types, and they are declared using the "enum" keyword. An enum can have one or more variants, and each variant has a constructor that takes a fixed number of arguments with specified types. In Listing 2.4, we showcase an enum named "Option<T>", where "T" represents a generic type parameter, making it suitable for any type. This enum defines two variants: "Some" and "None". Also, in Rust, an optional value can either have a specific value represented by "Some" or no value at all represented by "None" (equivalent to NULL in some other languages). In addition, Rust provides additional pre-defined enums, such as "Result", which is often used to represent the outcome of an operation. The "Result" enum has two variants: "Success", which denotes a successful result, and "Failure", which indicates an error message.

When it comes to functions, Rust utilises the "fn" keyword for both definition and declaration. Similar to other programming languages, Rust functions can take one or more arguments and can have a return type specified using the "return" keyword, although it is not mandatory. Below is an example showcasing the declaration of a function in Rust, as seen in Listing 2.5:

```
//declaring a function without arguments and return value;
fn hello() { println!("Hello␣World!"); }
//declaring a function with an argument and a return value;
fn foo(mut x: i32) -> i32 {
    return x;
}
```

Listing 2.5: Declaration of a function in Rust

Regarding control flow structures, Rust employs a syntax similar to Java and C for constructs like "if," "while", "loop", and "for". Additionally, akin to the switch statement in C++, Java, and JavaScript, Rust offers a "matching" expression, which is simpler and more practical compared to intricate if/else constructions. For instance, consider the following example:

```
let x = Some(5);
     //pattern matching to test for the Value of x Being 0 or not;
     match x {
         Some(0) => println!("Ok!"), //result of the pattern;
         // result of this pattern is y = 5
         Some(y) => println!("Default␣case,␣x␣=␣{:?}", y),
         // the default option;
         _ => println!("Error!"),
    }
```

Listing 2.6: Pattern Matching in Rust

Listing 2.6 showcases several syntax differences, where we utilize "⇒" to transition from

the corresponding value to the expression that should be executed. Furthermore, the last match ("arm") serves as an optional default or wildcard case. Apart from matching strings, integers, and other data types, we can also match variants of `enums, structs, tuples,` and `references`. Referring back to Listing 2.4, where the Option `enum` was introduced, it is commonly used to check if an element is "`None`" before performing operations on it, essentially testing whether `T` is `None` (similar to `NULL`) or not. However, when dealing with a value of type `Option<T>`, a question arises: how to extract the value of type `T` from a `Some` variant in order to use it? In such cases, the match expression proves useful as it automatically handles the internal `T` value if we have a `Some(T)` variant.

In conclusion, we have provided an overview of a subset of Rust's syntax. Moving forward, we will now delve into the feature set of the Rust type system.

4.2 Affine Types and Ownership

After addressing the memory safety issue, particularly aliasing, which affects the confidence level [31, 74, 39, 40], Rust distinguishes itself from most other programming languages by its specific memory model: "unique ownership". This distinctive feature grants Rust an efficient memory safety. The literature suggests that Rust's unique ownership bears similarities to affine systems [94], where each variable is limited to being used at most once [75]. In the case of Rust's affine types, they ensure that each value possesses a single unique owner. Unlike languages like C, Rust manages aliasing through move semantics and employs affine types to prevent it. The following example demonstrates how Rust effectively handles "unique ownership":

```
1
2  fn hello(mut str3: String){
3     //ensuring Safe String mutation: exclusivity of access by "str3"
4     str3.push_str("␣World!");
5     println!(str3);
6  }
7
8  fn main(){
9  //dynamic memory allocation for Strings on the heap;
10 let mut str1: String = String::from("Hello");
11 //transfer ownership of "str1" to "str2"
12 let mut str2 = str1;
13 //str1 is now statically inaccessible, and logically uninitialized
14
15 //transfer ownership of "str2" to "hello" function
16 hello(str2);
17 //str2 is now statically inaccessible, and logically uninitialized
18 }
```

Listing 2.7: Rust's Approach to Mutability and Move Semantics

As previously mentioned and evident in Listing 2.7, Rust's ownership system encompasses two crucial aspects: (1) control over where and when data exists, and (2) control over where and when mutation is allowed to occur. In this specific example, local variables are immutable by default, and the `mut` keyword grants the owner the ability to modify their values. In the listing, a new string is created and bound to `str1` on line 10, thereby establishing `str1` as the owner of the newly created string. The use of the `mut` keyword indicates that the value can be read from and written to by `str1`. On line 12, the ownership of `str1` is moved to the new variable `str2` (directly moving the value out of the identifier). Consequently, `str1` relinquishes the ownership of its contents, and now `str2` becomes the new owner of the value. As a result, `str1` is no longer accessible; this is the essence of the "move semantics" mechanism. This behavior applies even when moving variables as parameter values in a function (e.g., line 17).

The concept of unique ownership in Rust ensures that any value in a Rust program has only one owner, and this unique ownership is maintained through Rust's move semantics. In simple terms, when a variable holds a value in memory and is assigned to another variable or passed to a function, the ownership of the value is transferred to the new location. Consequently, the previous location loses access to the value, while the new location gains the ability to read and modify it. At any given time, only one location in memory has access to the contents of a value, granting the owner exclusive access control (preventing aliasing).

In addition to move semantics, Rust's type system also provides an alternative for unique ownership called "copy semantics". For example, primitive types like integer and Boolean have copy semantics. When an integer value (e.g., i32) is assigned to another variable, a deep copy of the contents is created, and the original value remains valid. The Rust type system infers affine types to protect other types (e.g., structures, enumerations, vectors, strings, etc). These types can implement copy semantics if their contents also support copy semantics recursively. Otherwise, they default to move semantics. To achieve copying behavior for data structures, Rust offers the `Clone` trait feature, where all types used in the structure definition must also implement `Clone`.

```
1  //declaring a Circle struct;
2  struct Circle{
3      name: String
4      a: i32,
5      b: i32,
6  }
7
8  fn main(){
9      //instantiates a Circle struct and assigns it to "circle";
10     //circle owns their content
11     let mut circle = Circle{
12                          name: String::new("origin"),
13                          a: 0,
14                          b: 0};
15     //the field "circle.name" is moved to "point";
```

```
16      let mut point = circle.name;
17      //error: read a partial move location;
18      println!("circle␣:{:?}", circle);
19  }
```

Listing 2.8: Partial Moves in Rust

Rust not only supports move semantics but also offers "partial move semantics", allowing us to move specific parts of a data structure. In such scenarios, the data is treated as partially moved. Referring to the aforementioned Listing 2.8, on line 16, we observe a field (`name`) being moved from `circle` to `point` (i.e., the type of `name` has move semantics). After this line, the Rust type system considers the `circle` type as "partially moved" because it contains a field that has been moved. Consequently, although the other fields of `circle` remain accessible, the entire instance of `circle` cannot be used. This is why line 18 results in a compile-time error.

4.3 Memory Management

In the context of memory management, Rust uses the special concept of *ownership*, which is defined by a set of rules. As mentioned earlier, each value in Rust must have a variable as its owner, meaning that at any given time, there can only be one owner per variable. When the owner goes out of scope, its destructor is automatically called, freeing the associated resources and releasing the memory used by values on the stack or the heap (values created on the heap will be discussed later). This approach ensures that memory is not released manually, and there is no need to worry about memory leaks, as is often the case in languages like C. For instance, in Listing 2.7, the value of `"str3"` is automatically removed as soon as the execution of the method is finished. Similarly, in Listing 2.8, when the owner `"point"` goes out of scope as shown on line 19, its value is automatically deallocated. Unlike languages with garbage collectors, Rust does not rely on such mechanisms. Instead, ownership rules are checked and enforced by the borrow checker, and the destructors of each variable in a program are automatically called at runtime. This design choice eliminates the overhead or pause time caused by a garbage collector and, in turn, achieves robust memory optimization [50].

4.4 Rust's Borrow Checker

According to "The Rust Programming Language" book [84], Rust's ownership model operates as an intermediary model that meticulously monitors the usage of data within the program. When dealing with a particular variable, programmers have three options: (1) moving the variable (data) itself, (2) making a copy of it, or (3) temporarily borrowing it while keeping track of its ownership (refer to section 4.5). To manage and enforce these ownership rules, Rust employs a crucial mechanism known as the "borrow checker", which possesses a special *flow-sensitive* nature. By analyzing the data traces, the borrow checker ensures the enforcement of unique ownership and verifies borrowed references, as explained in

Section 4.5. Consequently, Rust achieves automatic insertion and release of memory allocations, ensuring memory safety without the need for a garbage collector.

4.5 Regions and Borrowing in Rust

Rust's first feature in its type system tackled the problem of aliasing by introducing default move semantics. However, constantly moving values on every access is not practical. To maintain practicality while addressing the issue, Rust's type system provides a second feature: "borrowing". Although borrowing is more restrictive than unique ownership, it is a crucial aspect when sharing data.

Rust's primary goal is to prevent memory errors, especially those arising from references, such as use-after-free and dangling pointers seen in the C language. While Java's garbage collection resolves such issues related to memory allocation, Rust has introduced its own solution based on regions, drawing inspiration from languages like the Cyclone programming language [95, 88, 37, 41, 37]. Tofte and Talpin [93] introduced the concept of regions, where each reference is associated with a region of the program. Within a region, a reference remains valid, and the compiler ensures that the reference does not outlive its associated region, guaranteeing valid values at every usage. This reference verification is performed statically at compile time by Rust's borrow checker, eliminating the need for a garbage collector.

In Rust, regions correspond to lexical scopes (pairs of matching braces) and are known as "lifetimes". There are two implementations for lifetimes: the original implementation based on lexical scopes (lexical lifetimes) and the newer one called non-lexical lifetimes [67], which provides more flexibility and accepts a wider range of programs. While lexical lifetimes are straightforward and have been implemented since Rust version 1.0, they are sufficient for most needs.

Rust introduces a distinction between two kinds of references: "shared references" (immutable references) and "mutable references" (unique references). To retain Rust's main objective, mutable references impose certain restrictions on their usage. Shared references borrow data immutably for the lifetime of the reference, allowing only read access but no modifications. The borrow checker permits multiple immutable references to the same data since reading a value does not lead to modifications [101]. Additionally, the owner of the data is not allowed to modify it as long as it is borrowed as immutable (during the lifetime of the references). Consequently, the borrow checker enforces that only read access to the data is allowed. In Listing 2.9, we will explore an example of immutable references:

```
struct Circle{
    a: i32,
    b: i32,
}

fn main(){
    //creating a struct Circle and bind it to the variable "circle";
```

```
    //circle: exclusively owning its contents
    let mut circle = Circle{
                        a: 0,
                        b: 0};
    //c1 borrows the circle.a field immutably;
    //circle cannot modify the content of a
    //as long as c1 exists;
    let mut c1 = &circle.a;
    //c2 borrows the circle.a field immutably; ok!
    let mut c2 = &circle.a;

}
```

Listing 2.9: Immutable Borrowing in Rust

Mutable references in Rust, unlike immutable references, have the ability to modify the content they point to. However, to adhere to the principle of unique ownership (where only one variable can modify data at a time) and avoid concurrent and potentially conflicting updates, the Rust type system imposes restrictions on mutable references. Specifically, only one mutable reference to the same data is allowed at any given time. Moreover, during this period, the data owner is not permitted to read or write the data. From a deeper perspective, creating a mutable reference can be seen as a temporary transfer of ownership from the data's owner to the mutable reference. Consequently, when the reference's lifetime expires, the owner regains "read/write" access to its data. The example provided in Listing 2.10 illustrates the use of mutable borrowings.

```
struct Circle{
    a: i32,
    b: i32,
}
fn main(){
     //creating a struct Circle and bind it to the variable "circle";
    //circle: exclusively owning its contents
    let mut circle = Circle{
                        a: 0,
                        b: 0};
    //c1 borrows the circle.a field mutably;
    //circle cannot use a
    let mut c1 = &mut circle.a;
    //error: circle.a is also borrowed as mutable;
    println!("cricle: {:?}", circle.a);

    println!("cricle: {:?}", c1);
}
```

Listing 2.10: Mutable Borrowing in Rust

In Listing 2.10, on line 13, creating a mutable reference for the fields of the struct "`circle`" results in the ownership of the value of "`circle.a`" being transferred to the reference "`c1`" as long as it remains active (i.e., in scope). Consequently, `circle.a` cannot be used in any way while `c1` is still in scope. As a result, a compile-time error occurs on line 15.

Another limitation imposed by the borrow checker is preventing data from being borrowed simultaneously as both mutable and immutable. This restriction is illustrated in the provided Listing 2.11:

```
1  struct Circle{
2      a: i32,
3      b: i32,
4  }
5
6  fn main(){
7      //creating a struct Circle and bind it to the variable "circle";
8      //circle: exclusively owning its contents
9      let mut circle = Circle{
10                          a: 0,
11                          b: 0};
12      //c1 borrows the circle.a field immutably;
13      let mut c1 = &circle.a;
14      //c2 borrows the circle.a field mutably;
15      //error:cannot borrow circle.a as mutable
16      //because it is also borrowed as immutable;
17      let mut c2 = &mut circle.a;
18      println!("cricle:␣{:?}", *c1);
19 }
```

Listing 2.11: Special Scenarios in Rust's Borrow Checker

In scenarios where (1) multiple mutable references or (2) a mutable and an immutable reference exist at the same memory location simultaneously, unexpected errors like data races may occur. For instance, if a value is shared between two threads, one reading the value from a shared reference while the other is modifying it via a mutable reference, it could lead to spurious data inconsistencies. Borrowing consistently imposes restrictions on the owner of the value, whether it is mutable or immutable. If the value is borrowed immutably, the owner will also have immutable access to it. However, the value cannot be written or moved while the borrowing is in effect. On the other hand, if the value is mutably borrowed, the owner cannot read or write until the reference is dropped. These restrictions play a crucial role in maintaining memory consistency and avoiding potential data hazards.

4.6 Dynamic Allocation in Rust

By default, all values in Rust are allocated on the stack, provided the compiler can determine their size. However, in cases where the size of the value cannot be determined at compile time, Rust requires us to allocate the values on the heap. This dynamic allocation can be achieved through various methods such as using `Vec<T>`, `String`, `Box<T>`, and more. For instance, `Vec<T>` represents a resizable array type, different from the fixed-size arrays in C (e.g., `array[length]`), where the size must be specified at creation. Since `Vec<T>` has an arbitrary size, it cannot be stored in the stack and must be dynamically allocated.

Rust shines over other languages due to its unique ability to dynamically allocate memory without requiring manual deallocation as in C or a garbage collector as in Java. This feature ensures that the responsibility of deallocating data allocated on the heap is not on the programmer. In Rust, memory deallocation is triggered automatically when, for example, the owner of a `Box<T>` goes out of scope, freeing the memory on the heap by invoking its destructor. This thesis focuses specifically on the `Box<T>` type, which creates a unique reference to a value of type `T` allocated on the heap, offering automatic memory management and deallocation without extra manual intervention.

4.7 Exploring Concepts of Lifetimes in Rust

As previously detailed, every variable in Rust, whether it is a reference or not, has a lifetime. Generally, the lifetime of each owner is the scope in which it is created. However, according to [67], the lifetime of a reference begins when the reference is created and ends when the reference is last used. Each reference in Rust is annotated with a lifetime, providing the compiler with information on how long a reference can be safely used.

The initial implementation of lifetimes in Rust focused on lexical scope, where the lifetime of each variable, reference, or not, typically lasts until the end of the block containing it. In contrast, the newer implementation, known as Non-Lexical Lifetimes [65] (or Polonius), aims to be as short as possible and more accurate. Lifetimes are computed to form the minimal set of program points required to satisfy all constraints imposed by the borrow checker. Consequently, the borrow checker can determine that borrows have shorter lifetimes than the containing scope. As an example, following the Non-Lexical-Lifetime (NLL) concept, in Listing 2.10, the lifetime of "`c1`" starts upon its creation (line 13) and concludes when "`c1`" is last used (line 17). However, if we consider the same example and eliminate line 17, the program executes without encountering any errors. For the purpose of this thesis, we have followed the concept of the first implementation, lexical lifetimes, as it is a more straightforward approach to develop and aligns with our intended objectives.

4.7.1 The Significance of Rust Lifetimes

Regarding the concept of the Rust ownership model, specifically "borrowing", the borrow checker plays a crucial role in managing memory allocation and deallocation. It ensures that references do not point to memory that has been released by checking lifetimes at

compile time. As a result, borrowers must ensure that valid references remain valid every time they are used. Paying attention to lifetimes becomes particularly important when returning references from functions and when creating data structures with references.

In essence, lifetimes serve as a mechanism to inform the borrow checker about the validity of a reference. Based on the information provided by this mechanism, the borrow checker enforces the lifetime rules, ensuring that the reference remains accessible for at least the desired duration. This process guarantees that references are handled appropriately and safely throughout the program's execution.

4.7.2 Generic Lifetime Parameters in Rust

The lifetime notation in Rust reveals that all references come with attached lifetime parameters as part of their type. This feature becomes particularly convenient for complex scenarios where the borrow checker cannot infer the validity of borrows, such as in functions. To address such cases, Rust introduces lifetime parameters, also known as generic lifetimes, which are typically denoted using simple lowercase letters, starting with `'a, 'b`, and so on.

```
fn foo<'a, 'b>(x: &'a i32, y: &'b i32) -> &'b i32{ return y;}

fn longest<'a>(x: &'a i32, y: &'a i32) -> &'a i32{
    if x > y {
        x
    } else {
        y
    }
}
```

Listing 2.12: Function Declarations with Lifetime Parameters in Rust

In Listing 2.12, we define two functions: `foo` and `longest`. The `foo` function takes two references to integers as parameters and returns one reference. In the function signature, we will notice new arguments denoted as `'a` and `'b`, which are lifetime parameters used to bound the lifetimes of the input and output.

When dealing with the `foo` function, the Rust type system binds the second argument "`y`" to the output reference during compile time. It enforces the rule that a reference cannot outlive the object it refers to, ensuring that the reference is created after the creation of the object and destroyed before the object itself is destroyed. The Rust compiler considers the return type of the function to be a *subtype* of `y`, meaning it outlives itself (i.e., reflexivity). Technically, a lifetime is not a type that can be constructed as an instance, unlike regular types such as `i32, Box, Vec,` etc. However, in the context of a function signature, lifetime parameters are used as type parameters. For example, the type of `y` is not just a reference to an integer; it also includes its lifetime. Moreover, we cannot assume that the return type is a subtype of "`x`", as the Rust compiler lacks information about whether a lifetime

`'a` is a subtype of a lifetime `'b` (i.e., whether `'a` completely encloses lifetime `'a: 'b`). Consequently, the function body cannot return `x` using the same return signature.

Moving on to the "`longest`" function in Listing 2.12, we observe that its return signature is a subtype of both `x` and `y`. As a result, the Rust compiler infers that the return type is the union of the types of `x` and `y`. However, for memory safety, the lifetime of the return type is determined to be the minimum lifetime of `x` and `y` during the function invocation, adhering to the concept of minimizing the cases to ensure memory safety.

4.7.3 Lifetime Elision in Rust

In Rust's type system, we are required to explicitly introduce lifetime annotations for references in function arguments or data structures when the lifetime of reference parameters and return types cannot be inferred by the compiler. However, to reduce the burden of annotations, Rust allows for the elision of lifetime parameters in certain common scenarios. For example, when a function does not return a reference or when there is only one reference input parameter, etc.

This process of minimizing annotations is governed by "lifetime elision rules," which are built into the Rust compiler. These rules facilitate the automatic omission of certain lifetime annotations, simplifying code without sacrificing safety. However, it is essential to note that the elision rules do not provide complete inference. If the rules are applied deterministically, but ambiguity remains regarding the lifetime of references, the compiler will be unable to determine the lifetime of the remaining references, leading to an error.

```
1
2  fn foo(x: &i32, y: u32){...}                    //lifetime elided
3  fn foo<'a>(x: &'a i32, y: u32){...}            // expanded
4
5  fn bar(x: &i32, y: u32)->&i32{...}             //lifetime elided
6  fn bar<'a>(x: &'a i32, y: u32)->&'a i32{...} // expanded
7
8  fn foobar(&mut self) -> &mut T {...}            // elided
9  fn foobar<'a>(&'a mut self) -> &'a mut T{...}   // expanded
10
11
12 fn f_foo(x: &i32, y: &i32){...}             // illegal
13 fn b_bar(x: &i32, y: &i32)->&i32 {...}     // illegal
14 fn f_b_foo(x: i32, y: i32)->&i32 {...}    // illegal
```

Listing 2.13: Function Declarations with Elided Lifetime Parameters in Rust

In the absence of explicit annotations, the Rust compiler applies three rules to determine reference lifetimes:

1. Each parameter that is a reference gets its own lifetime parameter. For instance, in

Listing 2.13, the "`foo`" function, which has one reference parameter, obtains a lifetime parameter.

2. When there is only one elided lifetime in the input, it is associated with all elided lifetimes in the output. In the "`bar`" function shown above, which has a single reference parameter, the return type lifetime takes the same lifetime as the input.

3. If there are multiple elided lifetimes in the input, but one of them is associated with "`self`" or "`mut self`", it is then associated with all elided lifetimes in the output. This rule is illustrated in the "`foobar`" function in the given Listing.

In cases where these rules do not apply or ambiguity persists, explicit lifetime annotations are necessary.

4.8 State Sharing in Rust

To prevent memory issues, particularly data races, Rust's ownership and borrowing concept disallows aliasing and mutation simultaneously. This is crucial when multiple threads access the same memory location in an unsynchronized manner, with one thread reading and another writing. Rust offers various concurrency mechanisms for implementing multiple threads, maximizing performance, and ensuring robustness with fewer errors. Consequently, Rust is inherently thread-safe by default.

To achieve controlled aliasing and mutation, Rust provides a synchronization mechanism called a `mutex`. A `mutex` allows interior mutability in a thread-safe manner, enabling the sharing of a value among multiple threads. When a thread takes ownership of a `mutex`, it can safely use the data within it. After completing its tasks, the thread releases the lock, preventing other threads from accessing the data simultaneously. The use of `mutexes` is a fundamental requirement in Rust's type system to ensure a thread-safe environment and prevent data races.

Another feature of Rust's type system is the provision of shared ownership between multiple threads to optimize memory usage. The smart pointer `Rc<T>` allows shared ownership of an immutable value of type `T` allocated on the heap, functioning similarly to a `Box`. The `clone` function creates a new instance of `Rc`, pointing to the same allocation on the heap and increasing the number of references. When the last `Rc` pointer in a given allocation is destroyed, the value stored in that allocation is dropped.

However, the approach taken by `Rc` in handling references lacks concurrency primitives when the counter is increased or decreased, making it non-thread-safe. To address this, Rust offers another type of reference counting called `Arc<T>`. Similar to `Rc`, `Arc` provides shared ownership between multiple parts of a read-only program, but the key distinction lies in being thread-safe due to its "atomic reference counting" nature. `Arc` utilizes atomic operations, ensuring that changes to the counter cannot be interrupted by another thread.

To share ownership in a mutable manner, Rust's type system combines `Arc` and `mutex`. The combination of `Arc` and `mutex` allows multiple threads to safely share and modify data.

The provided example in Listing 2.14 demonstrates how two threads share the same value, with one thread incrementing it and the other displaying the value, all while maintaining a thread-safe environment without data races.

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn increment_count(mut counter : Arc<Mutex<i32>>){
    let handle = thread::spawn(move || {
        //acquire a lock on the shared state in this thread
        let mut num = counter.lock().unwrap();
        // mutate the shared state
        *num += 1;
        });
}
fn print_count(mut counter : Arc<Mutex<i32>>){
    let handle = thread::spawn(move || {
        //acquire a lock on the shared state
        println!("Counter: {:?}", *counter.lock().unwrap());
        });
}
fn main() {
    // create a mutex to shared state
    // Wrap it in an Arc object to share safely
    let counter = Arc::new(Mutex::new(0));
    // create an atomic copy of the shared state
    increment_count(counter.clone());
    // create another atomic copy of the shared state
    print_count(counter.clone());

    handle.join().unwrap();
     //acquire a lock on the shared state and
     //print it in the main thread
    println!("Result: {}", *counter.lock().unwrap());
}
```

Listing 2.14: Concurrent Ownership with Multiple Threads in Rust

In Listing 2.14, we create a value '0' of type `i32` and enclose it first in a `mutex` to enable mutability for sharing. Subsequently, we wrap the `mutex` object in an `Arc` to ensure atomicity across threads. We then call two functions, '`increment_count`' and '`print_count`'. These functions spawn two child threads, each taking a clone of the counter (i.e., a copy of the pointer that refers to the same piece of data in the heap). Consequently, both threads will use this shared state, with one thread incrementing the value to '1', and the other printing it. After completing the usage of the '`join()`' function, we print the final value of the

shared state in the main thread. It is important to note that to use the features shown in the above Listing, they must be imported from the standard library, as they are libraries and not native language features. In conclusion, this Listing illustrates Rust's approach to limiting concurrency features to ensure safety and proper thread management.

4.9 The Unsafe Features of Rust Language

Rust is divided into two languages: Safe Rust and Unsafe Rust. This thesis is centered around Safe Rust. However, Unsafe Rust [52, 69] is not a safe language; it is simply a superset of Safe Rust. Similar to Safe Rust, Unsafe Rust includes features such as lifetimes, unique ownership, and other powerful constructs that contribute to building robust programs. However, unlike Safe Rust, Unsafe Rust does not provide compile-time memory safety guarantees, although it still maintains some level of safety.

With Unsafe Rust, we have the flexibility to perform unsafe behaviors that are forbidden in Safe Rust. For example, in a C program, we can read uninitialized or `NULL` values, but with Unsafe Rust, such errors are avoided. While this allows us to work with unverified data, it also comes with the risk of problems like index exceeding the table size. Unsafe Rust is essential for providing various operating API systems and enabling certain low-level operations, such as implementing low-level data structures like linked lists or dereferencing raw pointers, interacting with foreign function interfaces (FFIs), etc. Many standard Rust libraries include pieces of unsafe operations, but Rust developers ensure that this unsafe code is properly encapsulated within the APIs. Sometimes, even for experienced programmers, it becomes necessary to break the constraints imposed by the Rust type system and express safe programs that go beyond static checking. In such cases, the '`unsafe`' keyword is used explicitly in the program, signaling to the Rust compiler to bypass some of its standard safety checks. However, it then becomes the responsibility of the programmer to ensure that Rust's safety guarantees are preserved within the unsafe code. Just like in C, extra caution and careful handling of the code are required in such situations.

4.10 A Survey of Rust Semantics

Thus far, we have introduced the Rust language. However, to thoroughly analyze the behavior of a Rust program, it becomes essential to grasp the underlying semantics and type system of Rust. This understanding is particularly significant as the main objective of this thesis is to develop a language with a type system inspired by Rust's (enabling precise control over aliasing and mutation). In this section, we explore existing research that delves into the Rust type system. Several endeavors have been made in the literature to formalize the semantics of Rust, proposing various models for its type system and establishing proofs of memory safety and soundness.

Metal. Rusty Types developed by [16] and inspired from the Rust programming language, its goal is to achieve reliable memory management by avoiding errors commonly encountered in C/C++ languages. The `Metal` language, a practical language, is presented as a result of the influence of Rust's ownership and borrowing concepts. `Metal` models

ownership and reference invariants as *capabilities*, where each variable possesses indirect *capabilities* on memory locations, based on [35]. These capabilities are represented as $\mathcal{K}$ where $\mathcal{K}$ `::= move | read | write`. By incorporating this information, the behavior of each variable is governed when the *capability* changes, enabling flow-sensitive move type judgments and ensuring memory safety.

Patina. Patina, a significant development by Reed [82], represents the first formal semantics for the Rust Type System. The primary focus of this work lies in providing progress and preservation proofs for memory safety, unique pointers, and borrowed references within Rust. Central to Patina's objectives is the establishment of a clear memory model to ensure Rust's memory safety properties. Patina also describes how the borrow checker is able to determine where data needs to be initialized and where it needs to be freed. It achieves this by defining three layers: inner, intermediate, and outer. The inner layer is responsible for enforcing safety constraints on references, preventing occurrences of dangling pointers. The intermediate layer ensures that variables consistently adhere to initialisation or borrowing restrictions. Finally, the outer layer handles the safety of variable declarations and guarantees reliable memory deallocation. Moreover, like `Box` in Rust, Patina defines a *unique pointer* using a specific syntax (e.g. `x :~int`), incorporating move semantics to prevent duplicate frees in memory. To further ensure successful memory deallocation without leaving inappropriate data behind, Patina introduces a "*shallow free*" statement. This statement enforces static checks on constraints, preventing the deallocation of a single pointer that is still borrowed. One notable contribution of Patina is the introduction of the "*shadow heap*" concept, which captures information regarding the initialisation status of memory locations. This work laid the foundation for subsequent research endeavors focused on formalizing Rust's ownership and borrowing systems.

Oxide. Oxide, a recent formally-defined programming language developed by Weiss et al. [97], closely resembles a version of the Rust source code. While Rust lacks a formal specification for memory layout, especially concerning its unsafe code base, Oxide primarily focuses on formalizing the property system, particularly the safe aspects of Rust. Libraries implemented with unsafe code are disregarded in Oxide's scope. The main objective of the work is to model ownership and borrowing from Rust's type system. One of the key motivations behind Oxide's development is the adoption of Non-Lexical-Lifetime (NLL) from the second version of Rust [67]. This feature alters the lifetime of references, making it different from the blocks in which they are created. Instead, the lifetime of a reference starts upon creation and ends when it is most recently used. As a result of this unique aspect of Rust, based on regions, Oxide has introduced a new contribution regarding regions to track aliasing. Oxide proposes a fresh perspective on lifetimes, treating them as an approximation of "reference provenances" through *region-based alias management*, represented as sets of locations. Similar to Rust, Oxide enforces the ownership of use-once variables through the application of move and copy semantics. Additionally, Oxide adopts the concept of aliasing, restricted to references, by introducing shared and unique references with loans [1]. A unique reference corresponds to zero shared references, while multiple shared references are allowed with zero unique references. However, this approach also means that during type checking, the borrower checker will reject inappropriate alias models.

In Oxide, each reference in a program is linked to a *region*, which represents a collection

of *loans*. A loan designates the state that arises when a reference is created. The borrow checker in Oxide utilizes this *region* information to analyze if the creation of a reference complies with the borrowing restrictions. To demonstrate this approach, let's consider the following example:

```
1  struct Circle(u32,u32);
2  // explicitly define a region
3  //'a->{}, 'b->{}
4  letrgn<'a,'b>{
5      let c = Circle(0,0);
6      // create a unique reference with a region " 'a "
7      let x = &'a uniq c;
8      // create a shared reference with a region " 'b "
9      let y = &'b shrd c; // Error
10     *x.0 = 1; // a unique loan is live
11 }
```

Listing 2.15: Shared and Unique References in Oxide

In Listing 2.15, we introduce a type called "`Circle`", which consists of a pair of `u32` values (similar to Rust). On line 5, we create a new `Circle(0,0)` instance and assign it to the variable `c`. Following the ownership concept in Rust, we establish that the variable `c` is the sole owner of the created value.

Next, on line 7, we create a unique reference to `c`. In Oxide, when borrowing is initiated, it includes an annotation for its associated region (already declared on line 3). Moving on to line 9, we attempt to create a new shared reference to `c`. Similar to Rust's Non-Lexical-Lifetime (NLL) concept, it is not allowed to have a shared reference while the mutable reference still exists. In accordance with Oxide's approach, since we have a unique (mutable) loan for `c`, attempting to create a shared reference on line 9 results in an error. In Oxide, the loan consists of a place and an ownership qualifier. The references forms (i.e., `shrd` and `uniq`) serve as ownership qualifiers. For instance, on line 7, when `x` is created, the region associated with it, `'a`, is mapped to the loan $\{^{\texttt{uniq}}\texttt{c}\}$. Then, on line 9, when `y` is introduced, the region related to it, `'b`, is mapped to the loan $\{^{\texttt{shrd}}\texttt{c}\}$. By examining the set of loans, the borrow checker in Oxide effectively detects any violation of uniqueness for the unique reference.

The borrow checker in Oxide effectively enforces the reference invariant and is capable of detecting whether memory parts are moved or not. To represent this information in the type of a variable, Oxide marks the entire type with a dagger symbol (e.g., `c : Circle`† indicates that the location is dead), which differs from the approach used in Patina [82]. Additionally, Oxide provides a comprehensive set of inference rules and syntax-like safety proofs using progress and preservation [99]. However, Oxide lacks a clear modeling of value allocation in the heap. The judgments used in the paper do not explicitly explain how memory allocated to the heap is modeled. The main focus of the examples in Oxide is on their novel contribution of "*region-based alias management*" and how the borrow checker effectively identifies and handles violations of the appropriation constraints using loans.

Lightweight. Pearce [73] introduced FR, a lightweight formal programming language that represents a subset of Rust, encompassing the safe part and including explicit modeling of boxes (i.e., heap-allocated memory). Unlike Oxide, FR does not support Non-Lexical Lifetimes [67]. This work draws inspiration from Featherweight Java [48] to achieve a relatively lightweight formalization of Rust. FR closely resembles Rust 1.0, which introduced lifetimes based on the lexical structure of programs (i.e., lexical scope). Compared to Rust, FR closely aligns with Rust's syntax and supports copy and move semantics, partial moves (applicable to tuples), the ability to define mutable and immutable references, and explicit lifetime annotations (e.g., `'a`) included in the reference signature, specifically for function cases. For other cases, such as blocks, all variables declared within a block, including references, automatically inherit their lifetimes. Similar to Oxide, FR utilises partial types (denoted as $\lfloor T \rfloor$, where T is a type in FR) and partial values (denoted as $\mathtt{v}^{\perp}$, where `v` is a value in FR) in its dynamic syntax. These partial types and values allow FR to statically capture valuable information about whether a location has been moved or not, helping identify whether a location is currently undefined. In FR, a reference type can be either a shared reference type or a mutable reference type. For instance, "`&x`" denotes a shared reference type, which makes it easy to identify that `x` is borrowed as immutable in the environment. Moreover, FR allows a reference type in the program to refer to multiple locations. For example, "`&mut x,y`" is a reference type indicating that the reference refers to both `x` and `y`. Although this shape might be meaningless in physical terms or not yet possible for the type system, this information plays a crucial role in enforcing reference safety. The following example illustrates the application of this idea in FR:

```
1 {
2     let mut x = 0;
3     let mut y = 1;
4     let mut z = &x;
5     if (x==y){
6         z = &y;
7     }
8     else {}
9 }
```

Listing 2.16: Control Flow in an FR Program

In FR, variable declarations correspond to mutable locations, and the language emphasizes mutability to differentiate between the two types of references. In the provided example, we create two variables, `x` and `y`, on lines 2 and 3, respectively. Then, on line 4, we create a shared reference to `x` and bind it to `z`. The control flow on line 5 checks `if x == y`; however, during type checking, the type system lacks information about this condition. Consequently, it cannot determine which branch should be executed at runtime. To address this, the type system assumes the union of the two branches, resulting in the merge of the two environments of the `if` and `else` statements. This merge denotes the coherent combination of types presented in these environments (i.e., $T_3 = T_1 \sqcup T_2$ where $T_1 \sqsubseteq T_3$ and $T_2 \sqsubseteq T_3$). As a result, after line 8, the type of `z` refers not only to `x` (`&x`) but also to `y`, as dictated by the if branch. Thus, the updated type of `z` becomes `&x,y`.

As previously mentioned, FR's approach is based on the lexical structure of the program. This means that every variable declared within a block assumes the lifetime of the outer block. For instance, in Listing 2.17, the variables `x, y,` and `z` have the lifetime of the outer block (note that the assignment of lifetimes to blocks is designed to reflect their relative nesting, e.g., {let mut x = 0; { let mut y = 1;}^{m}}^{l} such that `l` $\succeq$ `m`). Consequently, at the end of a block, all declared variables are automatically deallocated, a behavior enforced by the type system while handling blocks.

In addition to Oxide [97], FR also provides a clear approach for modeling heap memory allocation by explicitly introducing the `box` expression. Similar to Rust, FR treats the `box` as a unique pointer (or owning reference) that allows memory allocation on the heap (i.e., the heap location has only a single owner by enforcing the ownership transfer). On the other hand, borrowed references (shared/mutable references) are not responsible for memory deallocation. Instead, owning references are responsible for recursively abandoning the slot they refer to when they are dropped. It is crucial to consider this difference in the typing semantics when determining when dropping is appropriate. To highlight these essential features, consider the following example:

```
1 {
2     let mut x = box(box(0));
3     {
4         let mut y  = &*x;
5     }// the end of the lifetime of y
6 }// the end of the lifetime of x
```

Listing 2.17: Smart Pointers in FR Program

In the above example, within the outer block, we create a `box` to `box` containing the value `0` and bind it to `x`. Inside the inner block, we create a shared reference to the contents of `x` (`*x`). The lifetime of `y` terminates at the end of the inner block, and since `y` has a borrowed reference value, this means that it is not responsible for the deallocation of the value it refers to. Otherwise, at the end of the outer block or when `x`'s lifetime ends, all the values that `x` refers to will be deallocated recursively (on line 6). FR also extends its syntax to include tuples, thus transforming the strictly linear form (e.g. as x in Listing 2.17) and turning it into a tree form (e.g. `let mut x = (box(0), box(0))`). Additionally, FR introduces a valuable mechanism for *lifting* types from typing environments to signatures: $\Gamma_1 \vdash (\overline{S}) \rightarrow (S) \Leftarrow (\overline{T}) \rightarrow (T) \dashv \Gamma_2$. This mechanism enables the computation of return types while adhering to the constraints of references and their lifetimes, following the subtyping relation introduced by Rust and its side effects. This mechanism ensures type compatibility within the function declaration and invocation.

```
1 fn foo(mut x: &'r mut &'q int, mut y: &'q int) {....}
2 {
3     let mut a = 0;
4     let mut b = 1;
5     let mut c = &a;
6     foo(&mut c, &b);
7     // additional code that uses c
```

```
8 }
```

Listing 2.18: Addressing Side Effects through Function Invocations in FR

In Listing 2.18 and after the execution of the `foo` function on line 6, the type of `c` must be upgraded from `&a` to `&a,b`, as anticipated by the lifting mechanism. The type system has no knowledge of what may occur after the execution of `foo`, so it examines the signatures of the function's parameters. In this case, `x` is a mutable reference that can change its content, and `y` is a shared reference. Since the content of `x` is a shared reference with the same lifetime and type as `y`, we can claim that `'q int` is a subtype of `'q int`. This implies that the type system assumes that, perhaps in the body of `foo`, we have the statement: `*x = y`. The result of the mechanism is: $\Gamma_2[\texttt{a} \mapsto \langle \texttt{int} \rangle^1, \texttt{b} \mapsto \langle \texttt{int} \rangle^1, \texttt{c} \mapsto \langle \&\texttt{a}, \texttt{b} \rangle^1]$. Finally, this work provides a soundness proof for checking whether the semantic and typing rules can preserve type safety and reference invariants using progress and preservation [99]. In conclusion, we have expanded upon this research and introduced a new extension to FR called FR_{FT}. Consequently, the detailed aspect of FR in this section is not exhaustive, as we intend to provide a comprehensive explanation in Chapters 3 and 4 while presenting FR_{FT}.

RustBelt. RustBelt, developed by Jung et al. [52], is a formal semantic model of Rust that aims to verify the safety and soundness of programs. It primarily focuses on the unsafe part of Rust, as many standard Rust libraries include appropriately "encapsulated" unsafe operations. The goal is to prove that any safe Rust program using such unsafe libraries remains safe for memory and threads. To achieve this, RustBelt introduces a core Rust language with a type system called λ_{Rust}. In comparison with FR [73] and Oxide [97], λ_{Rust} is considerably more closely related to MIR [66] than to surface Rust. MIR is the mid-level intermediate representation of the Rust compiler on which proper borrowing usage is checked. On the other hand, λ_{Rust} incorporates the essence of Rust such as: borrowing, lifetimes, and the inclusion of lifetimes. This work has also been implemented in Coq where it gives a proof by outlining the verification conditions that an *unsafe* feature must be satisfied to be considered a safe extension of the language.

The initiative was to prove that a program in Rust is safe even if it contains unsafe code. While FR and Oxide use the standard *"progress and preservation"* technique introduced by Wright and Felleisen [48] to prove type safety, RustBelt uses the Coq proof assistant [91] to formally prove the safety of each (new) library that uses unsafe Rust features since the former technique does not apply to languages where there is an open interaction between the safe and unsafe parts of Rust. Subsequently, λ_{Rust} semantics takes a semantic approach to type safety [3, 2, 70]. Additionally, this effort also verified several standard Rust libraries that use unsafe code as: `Arc, Rc, Cell, RefCell, Mutex, RwLock, mem::swap, thread::spawn,` etc. Beyond the RustBelt project [52] , this endeavor has inspired other works in the Rust ecosystem. For example, GhostCell [101] is a safe extension of Rust, providing a *zero-cost* abstraction for thread-safe interior mutability (akin to `Mutex, RwLock,` etc.). The focus of this effort is to provide a new design for *interior mutability* that is improved over existing ones. For example, wrapping each list of structs with a `Mutex` or `Rwlock` by sharing it across threads (to keep the mutation safe) leads to unnecessary overhead for the program because it binds *permissions to data.* Inspired by this challenge, this effort avoids this overhead by *separating permissions from data* with a *single permission.* Moreover, various other efforts [90, 53, 51, 61] have emerged, building on the Rust

discipline and aiming to verify safe system programming in Rust.

KRust [96] and K-Rust [55]. They are two distinct works that followed RustBelt, despite their similar names. The primary objective behind both endeavors was to create executable formal semantics for the Rust language using the K-framework [83]. However, they differ in scope and completeness. K-Rust encompasses all the safe libraries of Rust and fully incorporates the entire Rust type system. On the other hand, KRust focuses on a realistic subset of the Rust language, making it more limited in its coverage. Finally, to advance the task of program verification in Rust, researchers have proposed several approaches that harness Rust's type discipline to enable deductive proof, as demonstrated in works such as [12, 38, 68, 59].

Chapter 3

FR_{FT} Language and its Semantics

Insanity: doing the same thing over and over again and expecting different results

– Albert Einstein

This chapter focuses on introducing the main objective of MSSL in a reactive synchronous context. Additionally, we present FR_{FT}'s syntax, which serves as the cooperative kernel of MSSL and represent a formal version of the Rust language subset with two extensions for multi-threading. FR_{FT} introduces a novel abstraction called `Trc`, which presents a challenge in ensuring the safety of shared memory among multiple threads. This challenge is addressed by combining the reference counting approach with Rust's standard reference aliasing constraints. Subsequently, we examine the features of `Trc` and introduce the operational semantics of the FR_{FT} language. Finally, we describe the cooperative operational semantics of FR_{FT}.

Ce chapitre met l'accent sur l'introduction de l'objectif principal de MSSL dans un contexte réactif synchrone. De plus, nous présentons la syntaxe de FR_{FT}, qui sert de noyau coopératif de MSSL et représente une version formelle du sous-ensemble du langage Rust avec deux extensions pour le multi-threading. FR_{FT} introduit une nouvelle abstraction appelée `Trc`, qui pose un défi pour garantir la sécurité de la mémoire partagée entre plusieurs threads. Ce défi est résolu en combinant l'approche de comptage de références avec les contraintes standard de référencement aliasing de Rust. Ensuite, nous examinons les caractéristiques de `Trc` et introduisons la sémantique opérationnelle du langage FR_{FT}. Enfin, nous décrivons la sémantique opérationnelle coopérative de FR_{FT}.

1 MSSL, Formally

In this chapter, we present MSSL, a memory-safe synchronous reactive language that focuses on enhancing the Fairthreads programming model by providing memory safety inspired by the Rust language. Similar to Fairthreads, MSSL utilizes cooperative execution of threads via a round-robin scheduler, accompanied by signals for self-synchronization among threads. The cooperative scheduling eliminates the need for protecting shared data with locking primitives. Threads in MSSL are automatically linked to the scheduler and executed cooperatively at the same rate. As depicted in Figure 3.1, the threads reside within a synchronous area connected to a single scheduler. Each thread is executed one at a time, while the others wait in a queue, following a cyclic pattern. When a thread finishes its execution for the current instant, the scheduler passes the control to the next thread in the queue within the same instant. This cycle of execution repeats until all threads are completely executed. This programming model offers a high degree of predictability and determinism in program execution, making it particularly valuable in safety-critical applications like avionics, automotive systems, medical devices, etc. The scheduler defines instants shared by all linked threads based on the concept of logical instant, facilitating automatic synchronization between threads at the end of each instant. The sequence of instants is represented as a sequence of discrete time steps or cycles. Essentially, a thread executes until its next cooperation point, at which it relinquishes control to the scheduler. The cycle time is determined when all threads have completed their execution for the currently running cycle.

1.1 Synchronized Areas in MSSL

In MSSL, signals are a way of communicating between threads. They offer a powerful and flexible means of communication throughout the program. Signals can be broadcasted, received, and emitted simultaneously, ensuring instantaneous reactions. Additionally, signals can be either absent or present at any instant. Intuitively, a thread can create a signal with an absent state by default or emit a signal to make it present. The responsibility of broadcasting the emitted signals to all other threads lies with the scheduler. In an MSSL program, every thread executes for an instant until its next cooperation point. This leads to two ways of defining cooperation points for threads: (1) the explicit way, wherein the thread executes the cooperation expression (further elaborated later), and (2) the implicit way, where the thread waits for a signal that has not yet been emitted. In both scenarios, the thread yields control to the scheduler. It is important to highlight that in the second scenario, the thread can regain control in the next cycle at the same instant if the awaited signal is emitted later, even by another thread.

Signal. The concept of signals in MSSL shares similarities with events found in synchronous reactive languages. Moreover, MSSL is distinguished by its memory management of signals. In MSSL, signals are implemented using reference counting, following the default copy semantics as in Rust. As a result, threads synchronize with each other using signals, and each thread can possess a copy of any signal. Due to the reference counting mechanism, when the counter falls to 1 at the end of the scope, the signal's destructor

is automatically called, indicating that it is no longer referenced by any other part of the program. Note that, in MSSL syntax, we have separated signals from standard values.

Instant. In MSSL, a program is defined as a set of threads (T) and is composed of a sequence of *instants*. At the start of each instant, all executable threads belong to T. As each thread completes its execution for the current round, it is removed from T and resumes execution in the subsequent round (cycle). Consequently, an instant is deemed complete only when all threads in the set T have finished executing for that instant. During an instant, threads may be either suspended (cooperating), awaiting a signal that has not been emitted, or have finished executing. However, if a signal is emitted within an instant, the instant is not finished, and the threads will be executed in another round. This initiates a new round of execution.

Inter-Instant. MSSL incorporates a form of *weak watchdog* instruction, which we will explore in chapter 5 (referred to as weak preemption [29]). This instruction consists of a signal and a body. Irrespective of whether the specified signal is present, the body of this instruction is always executed. However, at the end of the current instant, if the body has not been entirely reduced, we then check for the presence of the signal. If the signal is emitted, the body of the *weak preemption* instruction is killed. To facilitate this behavior, MSSL introduces a new phase known as the "*inter-instant*" phase, specifically dedicated to watching. During this phase, the non-terminating body is terminated in each thread.

In the upcoming sections, we dig into FR_{FT}, a cooperative kernel of MSSL that operates without signals. Our proposal extends FR [73] by introducing a novel type of smart pointers known as `Trc` (Thread Reference Counting), which enables cooperative multi-threading. Despite this extension, FR_{FT} retains key features from Rust's ownership discipline, including ownership, borrowing, re-borrowing, lifetimes, and lifetime inclusion. In contrast to the full syntax of MSSL, FR_{FT} as a language is restricted by the "`cooperate`" command, a special form of synchronization. To enhance our comprehension, we initially omit signal-based synchronization and focus on demonstrating the soundness of the FR_{FT} type system through our contribution, the safety of shared data between threads. The limitation of thread cooperation until the end of the instant through the "`cooperate`" command blurs the distinction between an instant and a round. In this context, an instant concludes when all threads have finished their execution, either entirely or for the current instant. In the ensuing sections, we present FR_{FT}, starting by highlighting its novelty, followed by introducing its syntax, and finally demonstrating its operational semantics.

1.2 FR_{FT}'s Latest Breakthrough: the `Trc`

In comparison to FR [73], FR_{FT} introduces two key aspects: (1) it incorporates reactivity into the language by providing reactive multi-threading instructions, which include threads spawning and explicit cooperation (see Figure 3.5). (2) One of our primary contributions is the introduction of a novel type of smart pointer called `Trc` (Thread reference counting), specifically designed for inter-thread communication. To facilitate this, any shared data between threads must be encapsulated in a `Trc`. This concept emerged from the combination of the reference counting approach (as it untie from the lexical scope constraints,

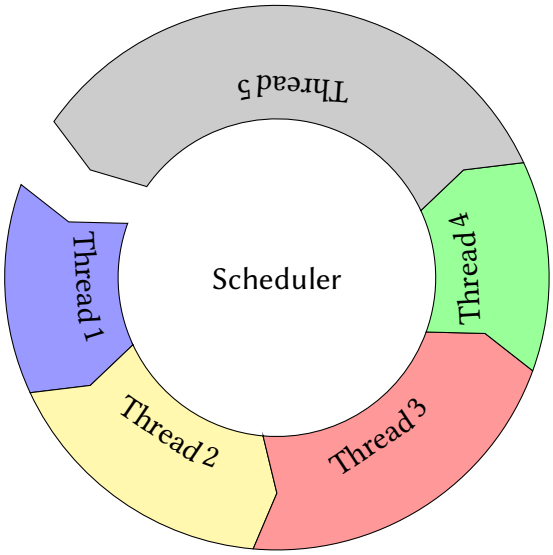

Figure 3.1: Synchronized area

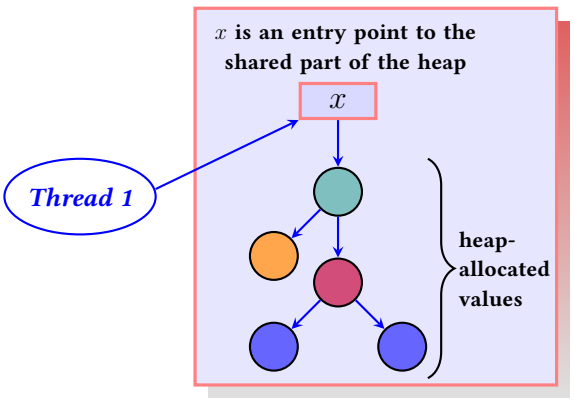

Figure 3.2: An active `Trc`

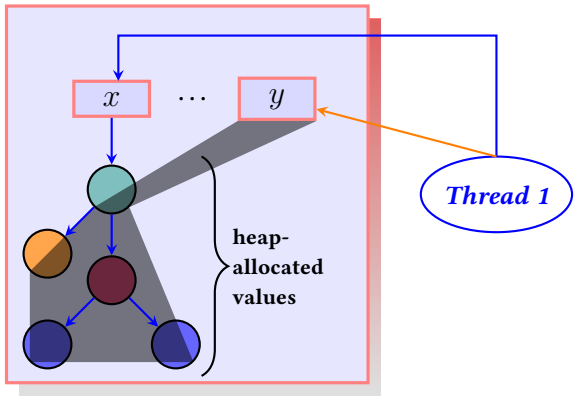

Figure 3.3: An active and an inactive `Trc`

which are necessary for sharing data between different threads) and the standard Rust reference aliasing constraints (ownership). Consequently, the FR_{FT} type system enforces this property, protecting the shared data from any concurrent corruption. Designing this new kind of smart pointer presented the challenge of combining sharing and mutability without necessitating a locking discipline (as in Rust). Specifically, `Trc` offers three main features: (1) enabling sharing among threads, (2) ensuring *uniqueness* of `Trc` per thread, and (3) ensuring that a thread cannot possess references to shared data during cooperation. Finally, `Trc` pointers can be categorized into two types: (1) active `Trc` (Figure 3.2) and (2) inactive `Trc` (Figure 3.3).

Features of a `Trc` Smart Pointer

In Figure 3.2, the active `Trc`, *x*, is dedicated to *Thread 1*. It allocates memory on the heap, and *x* serves as the entry point to the shared part of the heap, allowing read and write operations

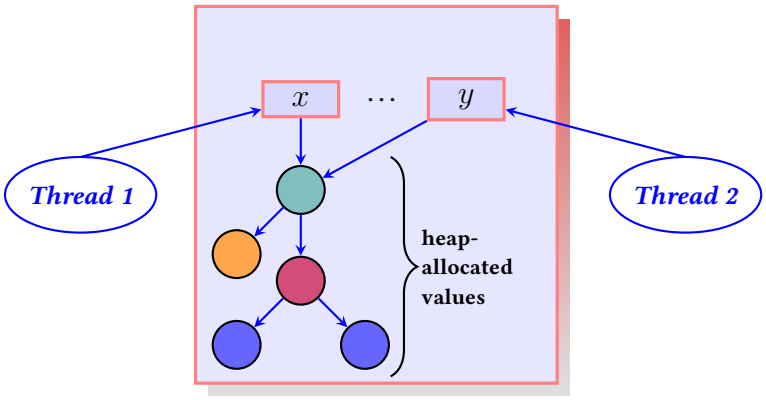

Figure 3.4: Ensuring Memory Safety Across Threads

on it. The active `Trc` provides access to the shared data. Note that data associated with a `Trc` is automatically allocated in the heap. On the other hand, Figure 3.3 illustrates the inactive `Trc`, represented by y. The inactive `Trc` is a copy of the active `Trc` x but with a restriction that it cannot access the shared data in the heap, shown by the shaded area. This restriction ensures the uniqueness of mutability, as imposed by the Rust type system. Both x and y share the same data and are associated with the same thread (e.g., *Thread 1*). However, *Thread 1* can access the heap data over x, but not over y, following the "*use_once*" variables approach [14, 15]. An inactive `Trc` is intended to be communicated to other threads. When it is communicated to another thread, the inactive `Trc` becomes active, meaning that only an active `Trc` can access the shared data. This transformation is demonstrated in Figure 3.4, which shows that when *Thread 2* receives y (considered inactive in *Thread 1*'s environment), it automatically becomes active in *Thread 2*'s environment.

Nonetheless, many features in FR_{FT} closely resemble those in FR. FR_{FT}, like FR, is an imperative language that embraces copy and move semantics, along with support for mutable and immutable borrowing, partial moves, and reference lifetimes essential in function signatures. The lifetimes in FR_{FT} are primarily determined by the lexical structure of programs, i.e., the lexical scope. However, the recent versions of Rust have introduced the concept of Non-Lexical Lifetimes (NLL) [67].

Prior to introducing the syntax of FR_{FT}, we will provide an example written in FR_{FT} as follows:

$$\texttt{fn}\ bar(\texttt{mut}\ x : \blacklozenge\texttt{int})\ \{{}^{*}x = 1;\ \texttt{cooperate};\ {}^{*}x = 2\}^{\texttt{m}}$$

$$\{\texttt{let mut}\ x = \texttt{trc}(0);\ \texttt{spawn}(bar(x.clone));\ \texttt{let mut}\ y = x.clone;\ \texttt{spawn}(bar(y))\}^{\texttt{l}}$$

This example showcases the dynamic syntax of FR_{FT}, with a "bar" function that takes a parameter x of type `Trc` active. The body of the bar function is a block denoted by a lifetime "`m`", containing a sequence of expressions separated by semicolons (i.e. $\overline{e}$ where e represents an expression). Subsequently, we encounter a block associated with the lifetime "`l`", representing the main program. Similarly structured, this block includes a sequence of expressions:

- `let mut` x `= trc`(0): this expression allocates a new active `Trc` in the heap and associates it with variable x.

- `spawn`($bar(x.clone)$): this expression spawns a new thread to execute the bar function, passing as a parameter an inactive `Trc` by copying the active `Trc` x via "$clone$".
- `let mut` y = $x.clone$: the active `Trc` is copied, and the copy is linked to the variable y. Consequently, y represents an inactive `Trc`.
- `spawn`($bar(y)$): another thread is created to execute the bar function, using the inactive `Trc` y as a parameter.

Using the `Trc` extension, the two threads share the same memory region. Then we can represent this example as follows (Listing **??**), demonstrating the *source-level* expressions that a user could write:

```
fn bar(mut x: trc<int>) {
    *x = 1;
    cooperate;
    *x = 2;
}

// the main of the program
{
 let mut x =  trc(0);
 // T1
 spawn(bar(x.clone)); //OK
 let mut y = x.clone;
 // T2
 spawn(bar(y)); //OK
}
```

Listing 3.1: Example FR_{FT} at source level

1.3 Syntax of FR_{FT}

In this section, we introduce FR_{FT}, a subset of the Rust language featuring a novel pointer type called `Trc`. We will start by presenting its syntax and then proceed to examine its operational semantics. Figure 3.5 introduces the FR_{FT} syntax, which is organised into five main groups: (1) values associated with each memory location (respectively partial values). (2) Types related to variables (respectively partial types). (3) Lvals representing the names of memory locations. (4) Actual expressions in the language. Finally, (5) Signature functions. The final sentence of Figure 3.5 showcases the program layout using the FR_{FT} syntax. Now, let us demonstrate the key features of FR_{FT}:

Values. A value in FR_{FT} can take different forms, including a special constant ϵ, an integer n, a Boolean (`true/false`) or a memory location. A value ϵ is produced by an expression that finishes, such as the reduction of an assignment expression. According to

Values	v	::=	$\epsilon \mid n \mid \texttt{true} \mid \texttt{false} \mid \ell_a^{\blacksquare} \mid \ell_a^{\blacklozenge} \mid \ell_a^{\diamond} \mid \ell_a^{\circ} \mid \ell_{\texttt{m::}x}^{\circ}$
Partial Values	$v^{\perp}$	::=	$v \mid \perp$
Types	τ	::=	$\epsilon \mid \texttt{int} \mid \texttt{bool} \mid \texttt{\&mut}\ \overline{\omega} \mid \texttt{\&}\ \overline{\omega} \mid \diamond\overline{\omega} \mid \blacklozenge\tau \mid \blacksquare\tau$
Partial Types	$\tilde{\tau}$	::=	$\tau \mid \blacksquare\tilde{\tau} \mid \lfloor\tau\rfloor$
LVals	ω	::=	$x \mid *\omega$
Expressions	e	::=	$v \mid \omega \mid \hat{\omega} \mid \overline{e} \mid \{e\}^{1} \mid \texttt{let mut}\ x = e \mid \texttt{box}(e) \mid \texttt{\&[mut]}\ \omega \mid \omega = e$
			$\mid \texttt{trc}(e) \mid \omega.clone \mid e_1 \oplus e_2 \mid e_1 \otimes e_2 \mid \texttt{spawn}(f(\overline{e})) \mid \texttt{cooperate}$
Functions	f	::=	$\texttt{fn}\ f(\overline{\texttt{mut}\ x : \texttt{S}})\{\overline{e}\}^{1}$
Signatures	S	::=	$\epsilon \mid \texttt{int} \mid \texttt{bool} \mid \blacksquare\texttt{S} \mid \blacklozenge\texttt{S}$
Programs	p	::=	$f\ p \mid \{e\}^{1}$

Figure 3.5: Syntax of FR_{FT}

Figure 3.5, we distinguish five kinds of memory locations: (1) a value $\ell_a^{\blacksquare}$ denotes an owning reference as a result of the reduction of box(e) expression, (2) a value $\ell_a^{\blacklozenge}$ denotes an active Trc, which is the result of the reduction of trc(e), (3) a value $\ell_a^{\diamond}$ denotes an inactive Trc obtained through the reduction of $\omega.clone$. Similar to $\ell_a^{\blacksquare}$, we have $\ell_a^{\blacklozenge}$ and $\ell_a^{\diamond}$ that also denote an owning reference. (4) A value ℓ_a° denotes a borrowed reference from a location in the heap, and a value $\ell_{\texttt{m::}x}^{\circ}$ denotes a borrowed reference from the location of a variable x with a lifetime m. Note that, ℓ_a is a location that is not bound to any variable contrary to $\ell_{\texttt{m::}x}$. To differentiate the aforementioned, we have added the subscript a. We use the form ℓ to refer to $\ell_{\texttt{m::}x}$ or ℓ_a (e.g. ℓ° can be either $\ell_{\texttt{m::}x}^{\circ}$ or ℓ_a°). The key distinction between owning and borrowed references is that *owning* references are responsible for recursively dropping their values whereas *borrowed* references do not.

Partial values. extend values with a special constant ($\perp$) denoting a moved value, signifying the need to prevent any reading or writing to an inaccessible memory location after the move operation. This behavior models the move semantics in FR_{FT}.

Types. Types include primitive types ϵ, int and bool, reference types (&mut $\overline{\omega}$ and & $\overline{\omega}$) and box types ($\blacksquare\tau$), similar to their counterpart in FR [73]. In the context of types, &mut $\overline{\omega}$ denotes a mutable reference to the location held of the lvals $\overline{\omega}$, where $\overline{\omega}$ denotes a list of lvals (e.g. & x, y, z), while & $\overline{\omega}$ denotes an immutable reference. Additionally, borrowing types can reference multiple locations, which is crucial for capturing type information and enforcing borrow invariance in FR_{FT}.
The type $\blacksquare\tau$ denotes a heap-allocated value of type τ. New types are represented as $\diamond\overline{\omega}$ and $\blacklozenge\tau$. A value of type $\blacklozenge\tau$ signifies an active Trc, while $\diamond\overline{\omega}$ denotes an inactive Trc. Similarly to references, the type of an inactive Trc may refer to several paths (e.g. $\diamond x, y$).

Partial types. We use the notation $\lfloor\tau\rfloor$ to represent partial types, indicating that one or more components of the type are currently undefined. This notation is used to signify a value that might contain moved locations, which is relevant for typing partial values. Furthermore, a type of the form $\blacksquare\tilde{\tau}$ means that the content of the box type is moved (e.g. $\{\texttt{let mut}\ x = \texttt{box}(0);\ \texttt{let mut}\ y = *x\}^{1}$ where, in this case, the type of y changes from

▪int to ▪⌊int⌋). Note that, according to Figure 3.5, a Trc type cannot have the same previous case since it is not possible to move out of a Trc (explained later).

Expressions. Expressions e in FR_{FT} are numerous, but largely standard and some of these are cooperative constructions. We reuse the definition of the expressions of FR, augmented by the following ones: (1) trc(e) allocates a new active Trc initialized with the value denoted by e, (2) $\omega.clone$ returns an inactive copy of the active Trc denoted by ω, (3) spawn($f(\overline{e})$) runs $f(\overline{v})$ as a new thread where $\overline{v}$ are the values denoted by $\overline{e}$, (4) cooperate yields the control to other threads. Other expressions are identical to those in FR. In more details, $\{e\}^{\mathtt{l}}$ denotes a block, the scope of which is expressed by the lifetime l and $\overline{e}$ is a sequence of expressions separated by semicolons (e.g. " $\{\mathtt{let\ mut}\ x\ \mathtt{=}\ 0;\ x\ \mathtt{=}\ 1\}^{\mathtt{l}}$ "). The lifetimes in FR_{FT} form a partial ordering (l ≥ m stands for m is inside l and l ≥ l is always valid) that reflects the nesting property. For example " $\{\mathtt{let\ mut}\ x\ \mathtt{=}\ 0;\ \{\mathtt{let\ mut}\ y\ \mathtt{=}\ \mathtt{box}(x);\}^{\mathtt{m}}\ x\ \mathtt{=}\ 2\}^{\mathtt{l}}$ ": x is declared in the external block where its lifetime is l while y is declared in the internal block where its lifetime is m. Following this example, we can deduce that the lifetime of y is smaller than that of x according to the relation l ≥ m. This means that y must be removed from the memory before x. The expression $\hat{\omega}$ denotes a non destructive read of the value held at ω; otherwise ω specifies a destructive reading of the value held at ω. The box(e) construct allocates a new Box in the heap memory, initialized with the value denoted by e. The rest of our expressions are standards. They include sequencing $\overline{e}$, assignment w = e, and arithmetic and Boolean expressions (respectively $e_1 \oplus e_2$ and $e_1 \otimes e_2$) according to Figure 3.5.

Functions and Signatures. In FR_{FT}, functions are declared using the fn keyword, similar to Rust. However, when it comes to the function call inside the spawn expression, functions are declared without specifying a return type. The shapes of the function signatures are illustrated in Figure 3.5. It's important to emphasize that these signatures cannot have the form $\&'$l[mut] S as references are not allowed when invoking spawn($f(\overline{v})$) due to their limited lifetime. Similarly, the signatures cannot be ◇S since the inactive Trc becomes active by applying the *activate* function (as defined in section 3.6) at spawn time. Lastly, when declaring functions, parameters are defined using the mut keyword to maintain consistency with the syntax of variable declarations, where all variables are mutable according to Figure 3.5.

The mut keyword in Rust serves two distinct purposes. However, to simplify our semantics, we explicitly introduce mut in the variable declaration syntax, emphasizing its essential roles, namely, as a mutable reference and as a shared reference. Consequently, each variable in FR_{FT} is linked to a corresponding lifetime, which means that a variable in FR_{FT} corresponds to a mutable location that can be updated as the program proceeds and whose lifetime is bound to the block that encloses it. At this point, FR_{FT} manages memory without relying on garbage collection mechanisms, but instead, it centers around the lifetime of each variable. As execution proceeds within a block, a new mutable location is generated for every declared variable, which is then automatically deallocated when the block ends. Finally, FR_{FT} enforces the rule of no "variable shadowing", in other words a variable can only be instantiated once.

An insight into typing. The missing signature of the form ◇S is due to the activation of an inactive Trc at spawn time. Therefore, when typing the spawn expression, a verification

is performed to ensure that two active Trc's in the signatures do not belong to the same shared data. This is necessary to guarantee the uniqueness of the active Trc in the thread environment as explained in Listing 3.3 below.

1.4 FR_{FT} Restrictions

Our motivation for developing our approach lies in addressing mutable data sharing in the context of cooperative threading. To achieve this objective, we introduce a crucial technical contribution named Trc. Below, we provide three user-level examples written in FR_{FT}, each describing a specific behavior of Trc.

```
1 {
2  let mut x =  trc(0);
3  let mut y = x.clone;
4  *x = 1; // OK: mutation is done!
5  *y = 2;// ERROR: cannot assign to data in an ‘inactive Trc‘
6 }
```

Listing 3.2: Inaccessible Data with an inactive Trc

In this example, we begin by creating a new active Trc and assigning it to the variable x, rendering x the owner of the new value inside the Trc. Then, on line 3, we explicitly create an inactive Trc by copying the active Trc using the clone expression and binding it to y. As mentioned earlier, the key distinction between an active and an inactive Trc lies in their roles regarding shared data on the heap. An active Trc serves as the entry point for shared data, while an inactive Trc cannot read or write its contents. Consequently, when attempting to modify the data via y on line 5, we encounter an error because y is an inactive Trc, and it lacks the permission to read or write its contents. On the other hand, the mutation is successfully performed via an active Trc on line 4. Note that if we present Listing 3.2 using the syntax shown in Figure 3.5, it would appear as follows:

$$\{\texttt{let mut } x = \texttt{trc}(0);\ \texttt{let mut } y = x.clone;\ {}^*x = 1;\ {}^*y = 2\}^1$$

```
1 fn bar(mut x: trc<int>, mut y: trc<int>) {
2     // additional code that uses x and y
3 }
4
5 {
6  let mut x =  trc(0);
7  //ERROR: No more than one inactive trc is allowed
8  spawn(bar(x.clone, x.clone));
9 }
```

Listing 3.3: Strengthening the *uniqueness* of Trc

In Listing 3.3, similar to the previous example, we initialise `x` as an active `Trc`. However, instead of creating an inactive `Trc`, on line 8, we create two separate inactive `Trcs` and pass them as arguments to the function called `bar`. Specifically, using the `spawn` expression on line 8, we create a new thread that will execute the `bar` function. When this new thread takes control, the previous inactive `Trcs` become active. Since these `Trcs` are copies of the same source, it leads to a violation of the *uniqueness* of `Trc` ownership. Due to this violation, we encounter an error on line 8 to enforce the *uniqueness* ownership of `Trcs`. As with the previous Listing, Listing 3.3 will be represented as follows using the syntax presented in Figure 3.5:

$$\texttt{fn}\ bar(\texttt{mut}\ x : \blacklozenge\texttt{int},\ \texttt{mut}\ y : \blacklozenge\texttt{int})\{\ldots\}^{\texttt{m}}$$
$$\{\texttt{let mut}\ x = \texttt{trc}(0);\ \texttt{spawn}(bar(x.clone, x.clone))\}^{\texttt{l}}$$

```
1  fn foo(mut x: trc<int>) {
2      // additional code that uses x
3  }
4
5  // T1 ( the main of the program)
6  {
7   let mut x =  trc(0);
8   // T2
9   spawn(foo(x.clone)); //OK
10  // T3
11  spawn(foo(x.clone)); //OK
12 }
```

Listing 3.4: Sharing Data Across Multiple Threads

In Listing 3.4, we replace the `bar` function with the `foo` function. On lines 9 and 11, we explicitly create two threads using the `spawn` expression, and for each function, we create a copy of `x`, which is an active `Trc`. Subsequently, these two threads are meant to execute the same function and share the same data in the heap. The FR_{FT} type system accepts this example since the threading is cooperative, and it is safe for each thread to have access to the same mutable data. This safety is ensured as only one thread is active at a time, and there is no need for mutexes to manage the accessibility to the shared data, as described in Figure 3.6. This prevents the potential deadlock problems that may arise when utilizing mutexes.

2 Operational Semantics of FR_{FT}

The operational semantics of FR_{FT} is established through a set of small-step rules presented in this section. These rules introduce various general forms of reduction, as outlined below: ***Expression***. Firstly, a state in FR_{FT} is represented as T, S where T is a set of threads, and S is a program store that maps locations to the partial values associated with a lifetime `m`, i.e. $\langle v^{\perp} \rangle^{\texttt{m}}$. The domain of S is a mapping from *Locations*×*Lifetimes* → *Partial Values*×*Lifetimes* where *Locations* and *Lifetimes* are sets of locations and lifetimes, respectively. Pertaining to the locations, two forms are used: (1) $\ell_{\texttt{m}::x}$ a location that is bound to the variable x with

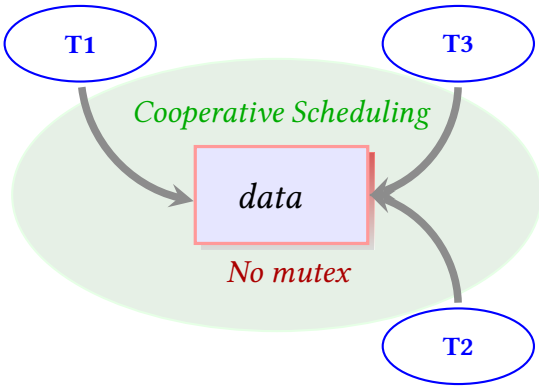

Figure 3.6: Accessing Data through Cooperative Threads

a lifetime $\mathtt{m}$, (2) ℓ_a a location that is not bound to any variable. Each thread is identified as a pair $(t, \{\overline{e}\}^{\mathtt{l}})$ where t is the thread's name and $\{\overline{e}\}^{\mathtt{l}}$ is the code which is currently executed by t under the lifetime $\mathtt{l}$. In a FR_{FT} program, threads are grouped in a set bound to be scheduled in a round-robin scheduler. Reduction rules have the form $\langle T, S \xrightarrow{T_0}_i T', S' \rangle^{\mathtt{l}}$ where T_0 is a, possibly empty, set of threads spawned during the reduction. The index i, ranging over 0 and 1, denotes either a computation termination (0) or a cooperation (1). These rules rely on an auxiliary kind of rules having the form $\langle S \triangleright e \xrightarrow{T_0}_i S' \triangleright e' \rangle^{\mathtt{l}}$ denoting local computation of threads, where the state has the form $S \triangleright e$. A global environment $\mathcal{D}$ represents a declaration context consisting of a series of high-level function definitions. $\mathcal{D}$ is added when a FR_{FT} program is reduced to store the declared functions. Then, for all f such that $f ::= \mathtt{fn}\ f(\overline{\mathtt{mut}\ x : \mathtt{S}})\{\overline{e}\}^{\mathtt{l}}$, we have $\mathcal{D}[f \mapsto \lambda(\overline{x})\{\overline{e}\}^{\mathtt{m}}]$. To simplify the reduction rules, $\mathcal{D}$ is excluded as it remains unchanged during expression execution. Consider the following reduction block: $\langle S \triangleright \{\mathtt{let\ mut}\ x = \mathtt{trc}(0)\}^{\mathtt{l}} \rightarrow_0 \{\ell_a \mapsto \langle 0 \rangle^{\mathtt{l}}\} \triangleright \{\mathtt{let\ mut}\ x = \ell_a^{\blacklozenge}\}\rangle^{\mathtt{l}}$ is a simple reduction that creates a location ℓ_a in S by reducing the $\mathtt{trc}(0)$ expression where the counter is initialized to 1. Continuing the execution, the reduction proceeds to $\langle \{\ell_a \mapsto \langle 0 \rangle^{\mathtt{l}}\} \triangleright \{\mathtt{let\ mut}\ x = \ell_a^{\blacklozenge}\}^{\mathtt{l}} \rightarrow_0 \{\ell_{\mathtt{l}::x} \mapsto \langle \ell_a^{\blacklozenge} \rangle^{\mathtt{l}}, \ell_a \mapsto \langle 0 \rangle^{\mathtt{l}}\} \triangleright \epsilon \rangle^{\mathtt{l}}$. Here, a variable x is created which has a location $\ell_{\mathtt{l}::x}$ allocated in lifetime $\mathtt{l}$ and actually has the value $\ell_a^{\blacklozenge}$, referring to the previously created location ℓ_a. Finally, the expression terminates immediately (represented by ϵ).

Thread. Secondly, to execute a thread for more than one step, the reduction rule is denoted as : $\langle S \triangleright e \overset{T}{\Longrightarrow} S' \triangleright e' \rangle^{\mathtt{l}}$.

Instant. Thirdly, at the beginning of each instant, all threads are assumed to be in a set T. When a thread executes, it is removed from T and resumed in the next instant. When all threads in T have been executed, it indicates that the current instant is finished, denoted as follows: $T, S \Longrightarrow T', S'$. Lastly, a sequence of instants can be represented by the rule: $T, S \Longrightarrow^* T', S'$ corresponding to the reflexive transitive closure of $\Longrightarrow$.

2.1 Preliminaries

Most of reduction rules in FR_{FT} rely on specific functions to ensure safe reduction. In this section, we present the supporting functions required for this purpose.

As previously mentioned, the program store S maps locations ℓ with partial values $\langle v^{\perp} \rangle^{1}$. Hence, it is therefore useful to identify the location of a given lval as follows:

Definition 3.1 (Location) *For a given program store S, an lval ω and a lifetime $\mathtt{l}$, loc is a partial function that returns the location related to ω in S. Subsequently, we define $loc(S, \omega, \mathtt{l})$ as follows:*

$$loc(S, x, \mathtt{l}) = \ell_{\mathtt{m}::x} \text{ where } S(\ell_{\mathtt{m}::x}) = \langle . \rangle^{\mathtt{m}} \text{ and } \mathtt{m} \geq \mathtt{l}$$
$$loc(S, *\omega, \mathtt{l}) = \ell \text{ where } loc(S, \omega, \mathtt{l}) = \ell' \text{ and } S(\ell') = \langle \ell^{\eta} \rangle^{\mathtt{m}}$$

As previously indicated, we represent the location of a variable x in the program store S as $\ell_{\mathtt{m}::x}$. Additionally, the lifetime $\mathtt{l}$ represents the lifetime of the enclosing block in which $loc(S, \omega, \mathtt{l})$ is performed. This lifetime information is crucial to ensure that the values returned are bound to the current thread, considering that we have only one store for the entire FR_{FT} program. Furthermore, the notation ℓ^{η} designates any possible kind of locations: $\ell_a^{\blacksquare}, \ell_a^{\blacklozenge}, \ell_a^{\diamond}, \ell^{\circ}$. However, this means that $loc(S, \omega, \mathtt{l})$ not only handles simple variables where their values are in S but also handles reference values. For example, in the given program $\{\mathtt{let\ mut}\ x = \mathtt{trc(0)};\ \mathtt{let\ mut}\ a = \mathtt{\&mut}\ x\}^{\mathtt{m}}$, when attempting to retrieve the location of $^*a$, the above function is used as follows: $S = \{\ell_{\mathtt{m}::x} \mapsto \langle \ell_c^{\blacklozenge} \rangle^{\mathtt{m}},\ \ell_c \mapsto \langle 0 \rangle^{1},\ \ell_{\mathtt{m}::a} \mapsto \langle \ell^{\circ}_{\mathtt{m}::x} \rangle^{\mathtt{m}}\}$ is the existing store program and $loc(\{\ell_{\mathtt{m}::x} \mapsto \langle \ell_c^{\blacklozenge} \rangle^{\mathtt{m}},\ \ell_c \mapsto \langle 0 \rangle^{1},\ \ell_{\mathtt{m}::a} \mapsto \langle \ell^{\circ}_{\mathtt{m}::x} \rangle^{\mathtt{m}}\}, {}^*a, \mathtt{m}) = \ell_{\mathtt{m}::x}$. Finally, as outlined above, loc is a partial function, it manages undefined cases and returns an error when necessary. For instance, consider the same example again, if we have $loc(\{\ell_{\mathtt{m}::x} \mapsto \langle \ell_c^{\blacklozenge} \rangle^{\mathtt{m}},\ \ell_c \mapsto \langle 0 \rangle^{1},\ \ell_{\mathtt{m}::a} \mapsto \langle \ell^{\circ}_{\mathtt{m}::x} \rangle^{\mathtt{m}}\}, {}^{***}a, \mathtt{m})$, an error occurs at compile time since the location of $^{***}a$ has not been defined in S, thus preventing unauthorized reading and writing.

Having introduced the loc function to obtain the location of a given lval ω, the next step involves reading the value associated with that location in the program store S:

Definition 3.2 (Read) *For a given lval ω, the partial function $read(S, \omega, \mathtt{l})$ retrieves the value of ω stored in S as follows:*

$$read(S, \omega, \mathtt{l}) = S(\ell) \text{ where } loc(S, \omega, \mathtt{l}) = \ell$$

In the given program store $S = \{\ell_{\mathtt{m}::x} \mapsto \langle 0 \rangle^{\mathtt{m}}, \ell_{\mathtt{m}::y} \mapsto \langle \ell^{\circ}_{\mathtt{m}::x} \rangle^{\mathtt{m}}\}$, the value of $^*y$, according to definition 3.2 is read as follows:

- $loc(S, {}^*y, \mathtt{l}) = \ell_{\mathtt{m}::x}$ where $loc(S, y, \mathtt{l}) = \ell_{\mathtt{m}::y}$ and $S(\ell_{\mathtt{m}::y}) = \langle \ell^{\circ}_{\mathtt{m}::x} \rangle^{\mathtt{m}}$
- $read(S, {}^*y, \mathtt{l}) = \langle 0 \rangle^{\mathtt{m}}$

Now, we present the function utilised to update the value of a given lval ω in the program store S as follows:

Definition 3.3 (Write) *For a given lval ω, the partial function write$(S, \omega, v^{\perp})$ updates the value of ω stored in S as follows:*

$$\mathit{write}(S, \omega, v^{\perp}, \mathtt{1}) = S[\![\ell \mapsto \langle v^{\perp} \rangle^{\mathtt{m}}]\!] \text{ where } loc(S, \omega, \mathtt{1}) = \ell \text{ and } S(\ell) = \langle . \rangle^{\mathtt{m}}$$

The notation $S[\![\ell \mapsto \langle v^{\perp} \rangle^{\mathtt{m}}]\!]$ returns a program store S by modifying the existing value of ℓ with a partial value $(v^{\perp})$. It is important to note that the lifetime of ω remains unchanged, and only the value is updated. Both the read *read*$(S, \omega, \mathtt{1})$ and *write*$(S, \omega, v^{\perp}, \mathtt{1})$ functions are implicitly partial since the $loc(S, \omega, \mathtt{1})$ is a partial function. As an example, let us consider a given program store $S = \{\ell_{\mathtt{m}::x} \mapsto \langle 0 \rangle^{\mathtt{m}}\}$. Changing the value of x where $\ell_{\mathtt{m}::x} \in dom(S)$ according to the definition 3.3 can be demonstrated as follows: $loc(S, x, \mathtt{1}) = \ell_{\mathtt{m}::x}$ where $S(\ell_{\mathtt{m}::x}) = \langle 0 \rangle^{\mathtt{m}}$. Thence, *write*$(S, x, 1, \mathtt{1}) = S[\![\ell_{\mathtt{m}::x} \mapsto \langle 1 \rangle^{\mathtt{m}}]\!]$.

A key characteristic of *FR*$_{FT}$ is its ability to manage the memory without relying on a garbage collector. Consequently, in the subsequent function, we elucidate how *FR*$_{FT}$ safely drops values in the program store S.

Definition 3.4 (Drop) *Let S be a program store and let $\mathtt{m}$ be a lifetime. The drop$(S, \mathtt{m})$ function is responsible for deallocating values with a lifetime $\mathtt{m}$. Then, drop$(S, \mathtt{m})$ is defined as drop(S, ρ) where ρ is a drop set as follows:*

$$\begin{array}{lcl}
\mathit{drop}(S, \varnothing) & = & S \\
\mathit{drop}(S, \rho \cup \{v^{\perp}\}) & = & \mathit{drop}(S, \rho) \ \mathit{if}\ (v^{\perp} \neq \ell_a^{\blacksquare} \wedge v^{\perp} \neq \ell_a^{\blacklozenge} \wedge v^{\perp} \neq \ell_a^{\diamond}) \\
\mathit{drop}(S, \rho \cup \ell_a^{\blacksquare}) & = & \mathit{drop}(S - \{\ell_a \mapsto \langle v^{\perp} \rangle^{*}\}, \rho \cup \{v^{\perp}\}) \ \mathit{where}\ S(\ell_a) = \langle v^{\perp} \rangle^{*} \\
\mathit{drop}(S, \rho \cup \ell_a^{\blacklozenge}) & = & \begin{cases} \mathit{drop}(S - \{\ell_a \mapsto \langle v^{\perp} \rangle^{1}\}, \rho \cup \{v^{\perp}\}) & \mathit{where}\ S(\ell_a) = \langle v^{\perp} \rangle^{1} \\ \mathit{drop}(S[\![\ell_a \mapsto \langle v^{\perp} \rangle^{i}]\!], \rho\} & \mathit{where}\ S(\ell_a) = \langle v^{\perp} \rangle^{i+1} \end{cases} \\
\mathit{drop}(S, \rho \cup \ell_a^{\diamond}) & = & \mathit{drop}(S[\![\ell_a \mapsto \langle v^{\perp} \rangle^{i}]\!], \rho\} \ \mathit{where}\ S(\ell_a) = \langle v^{\perp} \rangle^{i+1}
\end{array}$$

As described in definition 3.4, *drop*(S, ρ) traverses owning references recursively, dropping the associated locations if necessary. Here, ρ identifies the locations allocated by a given block based on its lifetime $\mathtt{m}$. Partial values are dropped by traversing their defined parts. The deallocation of slots allocated by a `Trc` depends on the counter (represented by i), and deallocation occurs only when the counter decreases to 1. When the counter is 1, it indicates that there is only one reference to that slot, making the deallocation safe to perform. Moreover, it is assumed that only active `Trc`s are responsible for deallocating the locations they refer to. Specifically, when the lifetime of an inactive `Trc` expires, decrementing the counter by 1 is sufficient, as indicated above.

2.2 Reduction Rules

In this section, we present and explain the *FR*$_{FT}$ reduction rules for a given thread t beginning with the expressions and according to the syntax presented in Figure 3.5.

2.2.1 General Rules

As a start, we introduce the notation of evaluation contexts to demonstrate the reduction of the sub-expressions of our language.

Definition 3.5 (Evaluation Context) *An evaluation context is an expression containing a single occurrence of $[\![.]\!]$ (the hole) instead of a sub-expression. In other words, it is used to describe where the next reduction step takes place in the program. The evaluation contexts for the FR_{FT} expressions are defined as follows:*

$$E \;::= [\![.]\!] \mid E; \overline{e} \mid \texttt{let mut}\ x = E \mid \omega = E \mid \texttt{box}(E) \mid \texttt{trc}(E) \mid E \oplus v \mid v \oplus E \mid E \otimes v \mid v \otimes E \mid \texttt{spawn}(f(\overline{v}, E, \overline{e}))$$

Now, we employ one single rule for E:

$$\frac{\langle S \triangleright e \xrightarrow{T}_i S' \triangleright e' \rangle^{1}}{\langle S \triangleright E[\![e]\!] \xrightarrow{T}_i S' \triangleright E[\![e']\!] \rangle^{1}} \quad (\textit{R-Sub})$$

2.2.2 Read and Write

We proceed to present and explain the FR_{FT} reduction rules, beginning with the *R-Copy* rule, which is used to reduce the copy expression. For the sake of simplicity, we will use $(\rightarrow_i)$ in the reduction rule notation when there is no spawned thread involved.

$$\frac{\textit{read}(S, \omega, \texttt{1}) = \langle v \rangle^{\texttt{m}}}{\langle S \triangleright \hat{\omega} \rightarrow_0 S \triangleright v \rangle^{\texttt{1}}} (\textit{R-Copy}) \quad (\textit{R-Copy})$$

The *R-Copy* rule is introduced to create a copy of an lval ω using the *read* function. This rule does not modify the program store S, making it a non-destructive read. In situations where the *R-Copy* rule cannot be applied, the *R-Move* rule is defined to handle such situation:

$$\frac{\textit{read}(S_1, \omega, \texttt{1}) = \langle v \rangle^{\texttt{m}} \quad S_2 = \textit{write}(S_1, \omega, \bot, \texttt{1})}{\langle S_1 \triangleright \omega \rightarrow_0 S_2 \triangleright v \rangle^{\texttt{1}}} \quad (\textit{R-Move})$$

In contrast to *R-Copy*, *R-Move* enforces a destructive read by rendering lval ω inaccessible. As a consequence, this rule is responsible for reducing ω and effectively replacing its value in the resulting program store S_2 with $\bot$, signifying that ω is now read-inaccessible. The notation $(\bot)$ is used to indicate this inaccessibility. For instance, consider a variable x where $S(\ell_{1::x}) = \langle \bot \rangle^{1}$. In this scenario, $\textit{read}(\{\ell_{1::x} \mapsto \langle \bot \rangle^{1}\}, x, \texttt{1})$ is prohibited, and any attempt to read x would result in an error at compile time.

To provide a semantic understanding of the concept of copy and move, and to illustrate the performance of the *R-Copy* and *R-Move* rules, we present the following example written in FR_{FT} in a user level:

```
{let mut x = trc(0); let mut y = 0; {let mut a = y; let mut b = x} }
```

Listing 3.5: The effect of R-Copy and R-Move rules

Listing 3.5 demonstrates a valid program in FR_{FT}. It contains two blocks where, in the inner block, the *R-Copy* rule is applied to the following expression `let mut a = y`; precisely in case the value of `y` is a primitive type. On the other hand, the *R-Move* rule is automatically applied to the second expression in the inner block when the type of `x` is a `Trc` type, implementing move semantics. Furthermore, the `trc(0)` expression automatically allocates a new `Trc` on the heap, initialized to 0, with its counter set to 1. Consequently, after the reduction of the inner block, the program store S now contains only the variables `x` and `y`, with `x` having a partial value as follows: $S = \{\ell_{\texttt{m::x}} \mapsto \langle \bot \rangle^{\texttt{m}},\ \ell_{\texttt{m::y}} \mapsto \langle 0 \rangle^{\texttt{m}}\}$. Finally, this program presents the *source-level* expressions which could be written by a user. However, the syntax shown in Figure 3.5 only appears at execution time. To clarify, Listing 3.5 is represented in the following form $\{\texttt{let mut } x \texttt{ = trc}(0);\ \texttt{let mut } y \texttt{ = } 0;\ \{\texttt{let mut } a \texttt{ = } \hat{y};\ \texttt{let mut } b \texttt{ = } x;\}^{\texttt{n}}\}^{\texttt{m}}$. Last but not least, not all reference values are accessible at the source level (e.g. $x = \ell_a^{\blacklozenge}$, where $\ell_a^{\blacklozenge}$ is the reduction of the $\texttt{trc}(0)$ expression by the *R-Trc* rule). A dynamic syntax at execution time is necessary to accurately model runtime memory management in FR_{FT}, especially when adding memory locations to expressions.

Let us continue with the same example and introduce a new expression as follows:

```
{let mut x = trc(0); let mut y = 0;
    {let mut a = y; let mut b = x; let mut c = x} }
```

Listing 3.6: The effect of *R-Copy* and *R-Move* rules

In contrast to the previous example, this example is not valid in FR_{FT}. Since the content of `x` is moved by `b`, `x` becomes inaccessible until it is reinitialized. To maintain the ownership discipline and prevent any violations, we prohibit all the accesses to `x`, such as `let mut c = x`.

2.2.3 Box and Trc

Concerning dynamic allocation, FR_{FT} manages heap allocation in two distinct ways, depending on whether it involves a `Box` or a `Trc`. The *R-Box* rule is responsible for creating a new location in the program store S_1 to represent the newly allocated `Box`. In this case, the slots in the heap receive the global lifetime (*). In general, (*) corresponds to a static variable in C/C++, where the lifetime is equal to the duration of the program execution. The *R-Box* rule is defined as follows:

$$\frac{\ell_a \notin dom(S_1) \quad S_2 = S_1[\ell_a \mapsto \langle v \rangle^*]}{\langle S_1 \rhd \texttt{box}(v) \rightarrow_0 S_2 \rhd \ell_a^{\blacksquare} \rangle^1} \qquad (R\text{-}Box)$$

The *R-Box* rule creates a new location in the heap and initialises it with the value v. As demonstrated, the lifetime of the value in the heap is global (*). For example:

$$\{\texttt{let mut } x \texttt{ = box}(0);\ \{\texttt{let mut } y \texttt{ = } x;\}^{\texttt{m}}; x \texttt{ = box}(1)\}^{\texttt{l}} \qquad (3.1)$$

In Example 3.1, during the reduction of the first expression in the outer block, the *R-Box* rule comes into play to reduce the expression "$\texttt{box}(0)$". As a result, a new location ℓ_a is allocated

in the heap with a global lifetime "*", yielding $S = \{\ldots, \ell_a \mapsto \langle 0 \rangle^{*}\}$. Subsequently, we link this location to the variable x, implying that the deallocation of the contents of ℓ_a in the heap depends on the owner's lifetime, i.e., x. Later, in the inner block, we move the contents of x into y. Consequently, the new owner of the data in the heap becomes y. Hence, when the lifetime of y, `m`, expires, the contents of ℓ_a are automatically deallocated.

Reference counting is a widely adopted memory management technique in various programming languages like C++, Rust, and others. It proves particularly valuable in systems where garbage collection is not feasible or preferred, such as real-time systems, embedded systems, or those with limited resources. The following rule showcases the semantic effectiveness of our new reference counting, `Trc`:

$$\frac{\ell_a \notin dom(S_1) \quad S_2 = S_1[\ell_a \mapsto \langle v \rangle^{1}]}{\langle S_1 \triangleright \texttt{trc}(v) \rightarrow_0 S_2 \triangleright \ell_a^{\blacklozenge} \rangle^{1}} \quad \textit{(R-Trc)}$$

The *R-Trc* creates a *fresh* location in the program store, representing the active `Trc`. Unlike the `Box` case, where the lifetime is set to (*), `Trc` uses reference counting, so the counter i is set to 1. The i counter serves the purpose of keeping track of the number of references (active and inactive `Trc`'s) to a specific memory location. When the number of references drops to 1, the memory is automatically deallocated. This means that the location of the heap, ℓ_a, is determined by setting the counter to 1. Additionally, we can create multiple instances of inactive `Trc`'s that point to the same location on the heap as the active `Trc` source. This functionality is achieved using *R-Clone*, as presented below:

$$\frac{\langle \ell_a^{\blacklozenge} \rangle^{\mathtt{m}} = \mathit{read}(S_1, \omega, \mathtt{1}) \quad S_1(\ell_a) = \langle v \rangle^{\mathtt{i}} \quad S_2 = S_1[\![\ell_a \mapsto \langle v \rangle^{\mathtt{i+1}}]\!]}{\langle S_1 \triangleright \omega.clone \rightarrow_0 S_2 \triangleright \ell_a^{\diamond} \rangle^{1}} \quad \textit{(R-Clone)}$$

The *R-Clone* rule increments the number of references (i) that point to the same location on the heap; denoted as $\langle v \rangle^{i+1}$. Note that the following premise $S_2 = S_1[\![\ell_a \mapsto \langle v \rangle^{\mathtt{i+1}}]\!]$ refers to the updated existing location ℓ_a, where $\ell_a \in S_1$. To provide a clearer understanding of how active and inactive `Trc`'s are managed in FR_{FT}, let us consider the following example:

```
{let mut x = trc(0); let mut y = box(trc(0));
    {let mut a = x; let mut b = *y.clone }}
```

Listing 3.7: Memory Management in FR_{FT}

In Listing 3.7, in the outer block, we create a new location in the heap via `trc`(0) expression and bind it to `x`. In the second expression, we also create a new location in the heap, which contains another new location, and link it to `y`. In the inner block, we apply the *R-Move* rule to move the contents of `x` into `a`. Additionally, using the *R-Clone* rule with `*y.clone`, we create a copy of the active `Trc` (representing the contents of `y`) without affecting `y`, and then link it to `b`. Once all expressions in the inner block are successfully reduced, we apply the *drop* function (definition 3.4) to recursively deallocate the owned slots. For `a`, the contents (owned slots) on the heap are safely dropped (i.e., the counter is set to 1), while for `b`, the *drop* function recursively drops the slot owned by it. In more detail, the *drop* decrements the counter of the contents of the `Box` to 1. Finally, at the end of the outer block, the *drop* function safely and recursively deallocates the contents of `y`, leading to the program reducing to ϵ.

2.2.4 Borrow

Like Rust, FR_{FT} also supports the concept of borrowing. Hence, the *R-Borrow* rule is responsible for determining the location of an lval ω that already exists in the program store S, and it operates as follows:

$$\frac{loc(S, \omega, \mathtt{l}) = \ell}{\langle S \triangleright \&[\mathtt{mut}]\omega \rightarrow_0 S \triangleright \ell^{\circ} \rangle^{\mathtt{l}}} \quad (\textit{R-Borrow})$$

As illustrated in the above rule, to create a reference to a given lval ω, the *R-Borrow* rule retrieves the location of ω (which can be either $\ell_{1::x}$ or ℓ_a) in the program store S.

2.3 Assign and Declare

In the rest of this section, we introduce the reduction rules for the FR_{FT} expressions that can be more complex and may have sub-expressions (e.g., $x = \mathtt{box}(\mathtt{trc}(0))$). In such cases, and in accordance with definition 3.5, we utilise the *R-Sub* rule.

Now, let us move on to presenting the *R-Assign* rule for the reduction of assignments as follows:

$$\frac{\textit{read}(S_1, \omega, \mathtt{l}) = \langle v_1^{\perp} \rangle^{\mathtt{m}} \quad S_2 = \textit{drop}(S_1, \{v_1^{\perp}\}) \quad S_3 = \textit{write}(S_2, \omega, v_2, \mathtt{l})}{\langle S_1 \triangleright \omega = v_2 \rightarrow_0 S_3 \triangleright \epsilon \rangle^{\mathtt{l}}} \quad (\textit{R-Assign})$$

The *R-Assign* rule updates the value of a given lval ω by its new value using the *write* function 3.3. Before updating the value, it first drops the old value using the *drop* function. The *drop* function is responsible for deallocating any location associated with the value $v^{\perp}$.

To declare a new variable, a new corresponding location for the variable in the program store is added and the *R-Declare* rule is then introduced as follows:

$$\frac{S_2 = S_1[\ell_{1::x} \mapsto \langle v \rangle^{\mathtt{l}}]}{\langle S_1 \triangleright \mathtt{let\ mut}\ x = v \rightarrow_0 S_2 \triangleright \epsilon \rangle^{\mathtt{l}}} \quad (\textit{R-Declare})$$

Using the *R-Declare* rule, we create a new variable x and bind it to its corresponding location in S_1, denoted as $\ell_{1::x}$ with the lifetime $\mathtt{l}$. Additionally, as demonstrated by this rule, the new value assumes the lifetime $\mathtt{l}$ of the enclosing block. In this scenario, the function *write* cannot be utilised as it requires an lval as a parameter that already exists in S_1. However, since x is newly created with its new location, we utilise the following notation: $S_2 = S_1[\ell_{1::x} \mapsto \langle v \rangle^{\mathtt{l}}]$.

2.4 Sequence and Block

Reducing a sequence of expressions in FR_{FT} allows these expressions to be reduced sequentially from left-to-right. We introduce the following rule:

$$\frac{S_2 = \textit{drop}(S_1, \{v\})}{\langle S_1 \triangleright v; \overline{e} \rightarrow_0 S_2 \triangleright \overline{e} \rangle^{\mathtt{l}}} \quad (\textit{R-Seq})$$

The reduction with *R-Seq* rule involves removing the completed values from the left by utilising *drop*$(S_1, \{v\})$. The concept of the *drop* function remains the same as introduced in definition 3.4, but here it is specifically responsible for the recursive deallocation of v.

To handle the reduction of a block in FR_{FT}, two reduction rules have been introduced. The *R-BlockA* rule is responsible for reducing the block body until only one value remains, and it operates as follows:

$$\frac{\langle S_1 \triangleright e_1 \xrightarrow{T}_i S_2 \triangleright e_2 \rangle^{\mathtt{m}}}{\langle S_1 \triangleright \{e_1\}^{\mathtt{m}} \xrightarrow{T}_i S_2 \triangleright \{e_2\}^{\mathtt{m}} \rangle^{\mathtt{l}}} \quad \textit{(R-BlockA)}$$

The *R-BlockA* rule is responsible for reducing the expression e_1 within the block. If the reduction of e_1 results in a value, it indicates that the block has been completely reduced. In such cases, we introduce the second reduction rule, *R-BlockB*.

The *R-BlockB* rule is utilised to deallocate any remaining owned locations in the block using *drop*$(S_1, \mathtt{m})$. This ensures that all resources are properly managed and deallocated after the block's execution is completed:

$$\frac{S_2 = \textit{drop}(S_1, \mathtt{m})}{\langle S_1 \triangleright \{v\}^{\mathtt{m}} \rightarrow_0 S_2 \triangleright v \rangle^{\mathtt{l}}} \quad \textit{(R-BlockB)}$$

Subsequently, at the end of each block, the *drop* function is responsible for deallocating all the owned locations by retrieving all the variables associated with them and having a lifetime `m`.

Intuitively, for a $\{e\}^{\mathtt{l}}$ FR_{FT} program, the execution of e is the sequence of global states such that at the beginning, S is empty, denoted $S_\emptyset$ and, also T is an empty set. Then, let us thoroughly go over and highlight the rules for reducing block in FR_{FT} program using the syntax presented in Figure 3.5 as follows:

$$\begin{aligned} S_\emptyset \triangleright \{&\texttt{let mut } x \texttt{ = trc(0); let mut } y \texttt{ = box(box(5));} \\ &\{\texttt{let mut } a \texttt{ = } x.clone\texttt{; let mut } b \texttt{ = } *y\texttt{; } *b\}^{\mathtt{m}}\}^{\mathtt{l}} \end{aligned} \tag{3.2}$$

While reducing the first two instructions using the *R-Trc* and *R-Box* rules, a total of five locations are created: one for x, one for y, and the other three are dynamically allocated via `trc(0)`, `box(box(5))`, respectively. Once the evaluation is successfully completed, a new state is achieved, which can be described as follows:

$$\begin{aligned} \{\ell_{1::x} \mapsto \langle \ell_e^{\blacklozenge} \rangle^{\mathtt{l}},\ \ell_e \mapsto \langle 0 \rangle^{\mathtt{l}},\ \ell_{1::y} \mapsto \langle \ell_k^{\blacksquare} \rangle^{\mathtt{l}},\ \ell_k \mapsto \langle \ell_r^{\blacksquare} \rangle^{*},\ \ell_r \mapsto \langle 5 \rangle^{*}\} \triangleright \\ \{\{\texttt{let mut } a \texttt{ = } x.clone\texttt{; let mut } b \texttt{ = } *y\texttt{; } *b\}^{\mathtt{m}}\}^{\mathtt{l}} \end{aligned} \tag{3.3}$$

As x holds an owned reference at location ℓ_e, the location ℓ_e will not be released unless x is also released and the counter is equal to 1. Likewise, y holds an owned reference to location ℓ_k, which in turn holds another owned reference to location ℓ_r. Both locations ℓ_k and ℓ_r have a global lifetime (*) as they have been dynamically allocated by the `box` expression.

Similar to ℓ_e, ℓ_k will be dropped recursively when y is dropped. In the following rule, only one more location is created for the variable a, resulting in the following state:

$$\{\ell_{1::x} \mapsto \langle \ell_e^{\blacklozenge} \rangle^1,\ \ell_e \mapsto \langle 0 \rangle^2,\ \ell_{1::y} \mapsto \langle \ell_k^{\blacksquare} \rangle^1,\ \ell_k \mapsto \langle \ell_r^{\blacksquare} \rangle^*,\ \ell_r \mapsto \langle 5 \rangle^*,\ \ell_{\mathtt{m}::a} \mapsto \langle \ell_e^{\diamond} \rangle^{\mathtt{m}}\} \triangleright \{\{\mathtt{let\ mut}\ b = *y;\ *b\}^{\mathtt{m}}\}^1 \tag{3.4}$$

The value of a in the program store is a copy of the owned reference of x (i.e. it results from an inactive `Trc`). Again and as stated above, after creating the variable a, the counter of the reference is incremented by 1 and reaches 2, which is true since we have two references to the same value in the heap. The execution of the next expression moves the contents of y as follows:

$$\{\ell_{1::x} \mapsto \langle \ell_e^{\blacklozenge} \rangle^1,\ \ell_e \mapsto \langle 0 \rangle^2,\ \ell_{1::y} \mapsto \langle \ell_k^{\blacksquare} \rangle^1,\ \ell_k \mapsto \langle \bot \rangle^*,\ \ell_r \mapsto \langle 5 \rangle^*,\ \ell_{\mathtt{m}::a} \mapsto \langle \ell_e^{\diamond} \rangle^{\mathtt{m}},\ \ell_{\mathtt{m}::b} \mapsto \langle \ell_r^{\blacksquare} \rangle^{\mathtt{m}}\} \triangleright \{\{*b\}^{\mathtt{m}}\}^1 \tag{3.5}$$

As discussed in the *R-Move* rule, the contents of y became inaccessible, exemplified by $\bot$. Last but not least, when evaluating the final instruction using the *R-Seq* rule and the *R-Move* rule, the content of b will be returned as the value of the inner block. At this point, the inner block is completed, and all the locations associated with a and b must be dropped using the *R-BlockB* rule. As a result, the new state is as follows:

$$\{\ell_{1::x} \mapsto \langle \ell_e^{\blacklozenge} \rangle^1,\ \ell_e \mapsto \langle 0 \rangle^1,\ \ell_{1::y} \mapsto \langle \ell_k^{\blacksquare} \rangle^1,\ \ell_k \mapsto \langle \bot \rangle^*\} \triangleright \{\ \}^1 \tag{3.6}$$

Similar to the inner block, we must also complete the outer block by dropping all the existing locations in the program store. Finally, we reach the final state where S is empty at the end of the program:

$$S_{\varnothing} \triangleright 0 \tag{3.7}$$

2.5 Arithmetic and Conditional Operations in FR_{FT}

FR_{FT} provides conditional support required for control flow extension, as explained in chapter 5 (Section 1.4). The syntax in Figure 3.5 includes Boolean types, values, and standard comparison conditional operators (==, >, <), represented by $\otimes$. The corresponding semantic rule for these conditional operators is given as *R-Cond*, as shown below:

$$\frac{v_3 = v_1 \overline{\otimes} v_2}{\langle S \triangleright v_1 \overline{\otimes} v_2 \rightarrow_0 S \triangleright v_3 \rangle^1} \quad (\textit{R-Cond})$$

In this rule, we introduce the function ($\overline{\otimes}$), which takes two values and a comparison operator as input. The function evaluates the result based on the operator's parameters and returns a Boolean value. Following the same approach as the standard conditional operators, we now present the reduction rules that support arithmetic expressions (+, -), represented by $\oplus$, as follows:

$$\frac{v_3 = v_1 \overline{\oplus} v_2}{\langle S \triangleright v_1 \overline{\oplus} v_2 \rightarrow_0 S \triangleright v_3 \rangle^1} \quad (\textit{R-Arithm})$$

The *R-Arithm* rule incorporates the function ($\overline{\oplus}$), which operates on two values and an arithmetic operator to calculate the corresponding result. Let us consider a typical example to illustrate the usage of arithmetic expressions in FR_{FT}:

$$\{\texttt{let mut } x \texttt{ = box(1); let mut } y \texttt{ = 1; } \{\texttt{let mut } a \texttt{ = } {}^{*}x \texttt{ + } y\}^{\texttt{m}}\}^{\texttt{l}} \tag{3.8}$$

In this example, we will use the FR_{FT} syntax presented in Figure 3.5. During the execution of the declaration in the inner block, we create a variable "a" and initialize it to the sum of the contents of x and y.

2.6 Function Declaration in FR_{FT}

We explain how FR_{FT} accommodates the memory to add function declarations and invocations inside the `spawn` expression. Similar to FR, FR_{FT} does not allow variable shadowing, meaning that a variable can only be instantiated once in the stack of each thread. Since variables in the program store S are associated with a lifetime, therefore, in accordance with this approach, we associate each thread with a distinct lifetime that lies within the global lifetime of the FR_{FT} program. As a result, the variables associated with each thread have lifetimes that match or are nested within their respective thread lifetimes. For example, we assume that we have three threads: *th1, th2* and *th3* where each has a well-defined lifetime such as $\texttt{l}_1, \texttt{l}_2$ and $\texttt{l}_3$ respectively. Thus, according to the above approach, we have $* \succcurlyeq \texttt{l}_1$, $* \succcurlyeq \texttt{l}_2$ and $* \succcurlyeq \texttt{l}_3$, indicating that each thread's lifetime is nested within the global lifetime. However, there is no ordering or relationship between the lifetimes of the different threads. In other words, there is no relationship between $\texttt{l}_1, \texttt{l}_2$ and $\texttt{l}_3$. This ensures that variables associated with *th1* have a lifetime of $\texttt{l}_1$ or are nested within $\texttt{l}_1$, variables associated with *th2* have a lifetime of $\texttt{l}_2$ or are nested within $\texttt{l}_2$, and so on. This scoping mechanism ensures that variables are properly scoped and accessible within their corresponding threads, while maintaining separation and independence between threads.

According to this approach, when managing function invocation inside a `spawn` expression, we integrate the function body within a block. Specifically, $\{\texttt{spawn}(f(x))\}^{\texttt{l}}$ is equivalent to $\{\{\overline{e}\}^{\texttt{m}}\}^{*}$ where $\{\overline{e}\}^{\texttt{m}}$ is the body of f. The lifetime `m` is inside the global lifetime (i.e. $* \succcurlyeq \texttt{m}$), since the function is called in `spawn`. This means that the lifetime of the invoked function is tied to the lifetime of the thread in which it is called, and the function will terminate when the thread itself completely terminates.

3 FR_{FT} Cooperative Operational Semantics

FR_{FT} is as a synchronous reactive language, especially for concurrent programming languages. In FR_{FT}, threads are executed cooperatively within an environment where all threads have guaranteed access to the scheduler. Cooperative threading involves a paradigm where only one thread executes at a given time, and threads voluntarily relinquish control of the scheduler to enable other waiting threads to execute. This approach requires each thread to explicitly yield the processor once it has completed its task or is awaiting a signal. Notably, in cooperative threading, the operating system does not preempt the thread, implying that

the thread can run indefinitely if it does not yield the processor. Cooperative threading is often employed in scenarios where multiple threads need to collaborate and share resources, such as in user interface programming or event-driven systems. In FR_{FT}, threads follow a deterministic semantics, which is based on previous works involving the reactive approach [76, 27, 45, 47]. In this section, we present a model for synchronous cooperative scheduling in the cooperative part of FR_{FT}, wherein explicit cooperation is achieved using the `cooperate` expression.

Eventually, we need to present the reduction rules for each reactive construct in FR_{FT}. Let us begin with the semantic rule that allows us to create a thread. This is achieved through the *R-Spawn* rule, which reduces the expression $\texttt{spawn}(f(\overline{v}))$ as follows:

$$\frac{t \in \mathit{fresh} \quad \mathcal{D}(f) = \lambda(\overline{x})\{\overline{e}\}^{\mathtt{m}} \quad \Theta(* \Rightarrow \{\overline{e}\}^{\mathtt{m}}) = \{\overline{e}\}^{\mathtt{n}} \quad (S', \overline{v'}) = \mathit{activate}(S_1, \overline{v}) \quad S_2 = S'[\overline{\ell_{\mathtt{n}::x} \mapsto \langle v' \rangle^{\mathtt{n}}}]}{\langle S_1 \triangleright \texttt{spawn}(f(\overline{v})) \xrightarrow{\{(t,\{\overline{e}\}^{\mathtt{n}})\}}_0 S_2 \triangleright \epsilon \rangle^1} \quad \textit{(R-Spawn)}$$

The *R-Spawn* rule creates a new thread represented as a pair $\{(t, \{\overline{e}\}^{\mathtt{n}})\}$, where t is the name of the newly created thread, and $\{\overline{e}\}^{\mathtt{n}}$ is the expression that the thread t should execute with the lifetime $\mathtt{n}$. The declaration context $\mathcal{D}$ is also taken into account, as it is necessary to load the declared functions into the program. Notably, the `spawn` expression invokes a function f that will be executed at the next instant. To ensure that all lifetimes in the expression e are instantiated to lifetimes within the global lifetime $*$, while adhering to the partial order (e.g. $* \geqslant \mathtt{n}$), we use the $\Theta(* \Rightarrow e)$ function, which recursively performs the instantiation. Furthermore, inactive `Trc`'s become active in the runtime environment of the new thread, and this is achieved using the *activate* function from Definition 3.6. Therefore, with $(S', \overline{v'}) = \mathit{activate}(S_1, \overline{v})$, we perform the activation recursively along the sequence $\overline{v}$, starting with S_1.

Definition 3.6 (Activate) *Let S be a program store and let v be a value. Then, $\mathit{activate}(S, v)$ is used to recursively activate an inactive `Trc` value. It is defined as follows:*

$$\begin{array}{lcl} \mathit{activate}(S, \ell_a^{\diamond}) & = & (S, \ell_a^{\blacklozenge}) \\ \mathit{activate}(S, \ell_a^{\blacksquare}) & = & (S'', \ell_a^{\blacksquare}) \ \textit{where}\ S'' = S'[\![\ell_a \mapsto \langle v' \rangle^*]\!]\ s.t. \\ & & (S', v') = \mathit{activate}(S, v)\ \textit{and}\ \langle v \rangle^* = S(\ell_a) \\ \mathit{activate}(S, v) & = & (S, v)\ \textit{otherwise} \end{array}$$

Note that, the value of a borrowed reference (ℓ°) is excluded from this function. This function is specifically used when creating a new thread and does not allow references at this stage due to the reference's lifetime.

Cooperative Threading Model

As previously explained, in the FR_{FT} program scheduler, instants are defined as the time during which all threads are allowed to execute. However, a thread will not relinquish control

to the scheduler until it completes its execution or cooperatively reaches its next cooperation point. In FR_{FT}, a thread can suspend its execution using the `cooperate` construct. This allows the thread's execution to be divided into several successive instants, preserving its progress from the current location for the next instant. When the `cooperate` expression is executed, it signifies that the thread has completed its work for the current instant and will not resume control until the next instant. The *R-Cooperate* rule is utilised to reduce the `cooperate` expression as follows:

$$\frac{}{\langle S \triangleright \texttt{cooperate} \rightarrow_1 S \triangleright \epsilon \rangle^1} \quad \textit{(R-Cooperate)}$$

The *R-Cooperate* rule yields the value ϵ and is uniquely identified by the $(\rightarrow_1)$ notation, while all other rules have an index of 0 instead. The index 1 signifies that the current thread is giving control to another thread, which is crucial for facilitating the instant semantics in FR_{FT}.

3.1 Thread Execution

We have described the execution of a thread in a single step. In this section, we will introduce the reduction rules for executing a thread more than once (i.e. big-step). This involves the execution of a thread until it either completes its task or is suspended for its next cooperation point.

(1) The following rule enables the execution of a thread more than once, as long as it maintains control:

$$\frac{\langle S \triangleright e \xrightarrow{T_0}_0 S'' \triangleright e'' \rangle^1 \quad \langle S'' \triangleright e'' \xRightarrow{T_1} S' \triangleright e' \rangle^1}{\langle S \triangleright e \xRightarrow{T_0 \cup T_1} S' \triangleright e' \rangle^1} \quad \textit{(R-Thread)}$$

(2) The following rule indicates that the current thread has finished its execution:

$$\frac{}{\langle S \triangleright v \xRightarrow{\varnothing} S \triangleright v \rangle^1} \quad \textit{(R-ThreadTerm)}$$

(3) The following rule indicates that the current thread is suspended, meaning that it is cooperating:

$$\frac{\langle S \triangleright e \xrightarrow{T_0}_1 S' \triangleright e' \rangle^1}{\langle S \triangleright e \xRightarrow{T_0} S' \triangleright e' \rangle^1} \quad \textit{(R-ThreadCoop)}$$

In this case, the current thread relinquishes control to the scheduler, enabling another thread to execute.

3.2 Instant

Using the notion of *logical time* called *instant*, the scheduler defines instants during which all threads execute until their next point of cooperation. When a thread executes the

`cooperate` expression, it only resumes control at the beginning of the next instant. As previously discussed, *FR*$_{FT}$ employs a cooperative round-robin scheduler, where at the start of each instant, all threads are part of the set T, containing those ready for execution. During an instant, when a thread is executed, it is removed from the set T ($T_{\setminus t}$) and will be resumed in the subsequent round. The set T' represents the remaining threads after all the threads in T have been executed in that instant. This section outlines the rewriting rules that define the semantics of executing an instant:

1) The end of an instant is reached when all threads in the set T, which represents the threads that are ready for execution in the current instant, have been executed. This condition is expressed by the following rule:

$$\frac{}{\varnothing, S \Longrightarrow \varnothing, S} \qquad \textit{(R-InstantEnd)}$$

2) To execute an instant in *FR*$_{FT}$, we introduce the following *big-step* rule as follows:

$$\frac{(t, \{e\}^1) \in T \quad \langle S \triangleright e \overset{T_0}{\Longrightarrow} S' \triangleright e' \rangle^1 \qquad T_{\setminus t}, S' \Longrightarrow T', S''}{T, S \Longrightarrow T' \cup \{(t, \{e'\}^1)\} \cup T_0, S''} \qquad \textit{(R-Instant)}$$

The *R-Instant* rule states that for each thread belonging to the set T, we apply one of the semantic rules defined in section 3.1. The premise $T_{\setminus t}, S' \Longrightarrow T', S''$ indicates that when a thread finishes its execution, it will be added to the new set T' to be executed again in the next instant. It is important to note that threads (T_0) created during the current instant do not immediately execute after their creation to avoid any interference with running threads. Instead, they are added to T' as follows: $T' \cup (t, \{e'\}^1) \cup T_0$.

3.3 Chaining of Instants

FR$_{FT}$ relies on the fundamental concept of logical instants, which ensures deterministic and predictable behavior. By dividing program execution into discrete logical instants, *FR*$_{FT}$ guarantees consistent and reliable operation, independent of the hardware or software platform. In *FR*$_{FT}$, all threads execute sequentially at the same pace, while sharing the same instants and automatically synchronizing at the end of each instant. A program in *FR*$_{FT}$ consists of multiple instants, and the semantics aim to demonstrate that the program's execution is a sequence of states $(T, S), (T', S'), \ldots$ denoted as $(T, S) \Longrightarrow (T', S') \Longrightarrow \ldots$ In this context, we introduce a sequence of complete instants known as "chaining of instants." We define the operational semantics of a chaining of instants using the notation $\Longrightarrow^*$, which represents the reflexive transitive closure of $\Longrightarrow$.

4 Discussion

In this chapter, we introduce a subset of MSSL called *FR*$_{FT}$, which includes a new smart pointer type named `Trc`, along with two multi-threading constructs. The main goal of

this chapter is to demonstrate that the pointer `Trc` can be safely shared between multiple threads without using a mutex, and to formally model how a FR_{FT} program executes one or more instants.

Unlike Rust semantics, FR_{FT} does not support variable shadowing, which means that some examples accepted by the Rust type system may be rejected in FR_{FT}. As an illustration, consider the following example, which is not valid in FR_{FT}:

```
{
    let mut x = box(0);
    {
        let mut x = 1;
    }
}
```

Listing 3.8: Variable shadowing prohibited in FR_{FT}

This example is rejected by FR_{FT}, because when reducing the declaration in the inner block, the *R-Declare* rule creates a new location for x in the program store S. Otherwise, in S, there is already a location for a variable x. As a comparison, let us consider the same Example 3.8 where this time it is written in Rust syntax and compiled with **`rustc`**:

```
fn main() {

    let mut x = Box::new(0);
   { let mut x = 1;}
}
```

Listing 3.9: Variable shadowing allowed in Rust

Unlike Example 3.8, Example 3.9 is accepted by **`rustc`** (the Rust compiler). In this case, in the inner block, the value of variable `x` is 1, and at the end of the inner block, `x` reverts to its value in the outer block. Additionally, FR_{FT} supports lexical scope to manage memory, while the recent version of Rust supports NLL (non-lexical lifetimes, [67]). Nevertheless, we found the former approach to be satisfactory for achieving our objective.

Clearly, in the syntax presented in Figure 3.5, we have distinguished between copy and move semantics (using the notation $\hat{\omega}$ and ω , respectively). Furthermore, in chapter 6, we have enriched our type system with additional features. Leveraging inference typing, our type system can automatically determine whether variables in an MSSL program should use copy or move semantics. This allows for more concise and expressive code. For instance, consider the following example:

```
{
    let mut x = trc(0);
    let mut y = 1;
    {
        let mut a = x;
        let mut b = y;
```

```
7       }
8   }
```

Listing 3.10: Copy and Move Operations

In this example, from an implementation standpoint, the FR_{FT} type system intelligently infers that variable `x` should apply move semantics, while variable `y` should use copy semantics. However, when we translate this example into the syntax presented in Figure 3.5, we get the following result:

$$\{\texttt{let mut } x = \texttt{trc}(0);\ \texttt{let mut } y = 1; \{\texttt{let mut } a = x;\ \texttt{let mut } b = \hat{y}\}^{\texttt{n}}\}^{\texttt{m}}$$

Hence, it is clear that x is no longer the owner of `Trc` since `Trc` implements move semantics. Instead, a becomes the new owner. On the other hand, y, which uses copy semantics, remains accessible even after the last expression has been reduced.

Chapter 4

The Type System and Soundness of FR_{FT}

The difference between the right word and the almost right word is the difference between lightning and a lightning bug

– Mark Twain

In this chapter, we introduce the FR_{FT} type system, which addresses the issue of undesired data in the memory of threads when cooperation occurs, as another thread assumes control for execution. The FR_{FT} type system serves to safeguard shared data between threads, especially during cooperation, while ensuring the validity of references each time they are used. Furthermore, this chapter provides evidence of type, borrow, and concurrency safety theorem results for FR_{FT}. The latter theorem guarantees that a *well-typed* synchronous reactive program will execute one or more instants or reach a terminal state. A terminal state implies that the expression of all threads is a value.

Dans ce chapitre, nous présentons le système de types de FR_{FT}, qui traite le problème des données indésirables dans la mémoire des threads lors de la coopération, lorsqu'un autre thread prend le contrôle pour l'exécution. Le système de type de FR_{FT} sert à protéger les données partagées entre les threads, en particulier lors de la coopération, tout en garantissant la validité des références à chaque utilisation. De plus, ce chapitre fournit des preuves des résultats des théorèmes de sécurité de type, d'emprunt et de concurrence pour FR_{FT}. Ce dernier théorème garantit qu'un programme réactif synchrone bien typé exécutera un ou plusieurs instants ou atteindra un état terminal. Un état terminal implique que l'expression de tous les threads est une valeur.

1 Typing of FR_{FT} Expressions

A type system in a programming language enforces a set of rules and constraints on the types of values used in a program. Utilising a type system brings several advantages, including enhanced program reliability, code maintainability, and increased developer productivity. It serves the purpose of ensuring a program is *well-typed* and free from certain types of errors, such as type matching errors when using a type value in a context that requires a different type. One of the key benefits of a type system is its ability to detect many common programming errors during compile-time rather than run-time. This reduces the likelihood of bugs and facilitates the process of error identification and correction. In essence, *well-typed* expressions should either diverge or terminate on a value.

In this section, our type system aims not only to achieve typing safety but also to ensure type and borrowing safety between threads, thus preventing type and borrowing errors. The system's purpose is to guarantee that *well-typed* programs possess necessary properties, such as the ownership invariant for mutable references, to ensure valid references on every use. Additionally, it ensures that sharing data between threads does not result in inappropriate data being stored in thread memory. Hereunder is an example statically rejected by the FR_{FT} type system:

$$\begin{array}{c} \texttt{fn}\ f_1(\texttt{mut}\ x : \blacklozenge\texttt{int})\{\texttt{let mut}\ a = \&\texttt{mut}\ {}^*x; \texttt{cooperate}; \\ {}^*a = 1\}^{l_2} \\ \texttt{fn}\ f_2(\texttt{mut}\ y : \blacklozenge\texttt{int})\{\ \text{//additional code that uses } y\}^{l_3} \\ \{\texttt{let mut}\ x = \texttt{trc}(0);\ \texttt{spawn}(f_1(x.clone)); \\ \texttt{spawn}(f_2(x.clone))\}^{l_1} \end{array} \tag{4.1}$$

Example 4.1 is rejected by the FR_{FT} type system. This is because the thread that executes the f_1 function passes control to another thread at some point. Since these two threads share the same memory region, if the second thread modifies the contents of this memory region, it may lead to unintended data changes in the first thread's memory, such as a dangling pointer on variable a. Although our syntax imposes some restrictions to mitigate these types of errors, when dealing with structures, tuples, and similar constructs, the situation becomes more critical. Therefore, it becomes imperative to avoid using references in such cases. Let's examine the following example that is not valid according to our type system:

$$\{\texttt{let mut}\ x = \texttt{trc}(0);\ \texttt{let mut}\ y = x.clone;\ {}^*y = 1\}^{\texttt{m}} \tag{4.2}$$

Example 4.2 is considered *ill-typed* because it violates the uniqueness of `Trc`. In this case, both variables x and y can access the same memory block simultaneously, leading to potential conflicts and data integrity issues. Similar to Rust, the FR_{FT} type system also safeguards the use of references to prevent such problems. For instance:

$$\{\texttt{let mut}\ x = 0; \texttt{let mut}\ a = \&x; \texttt{let mut}\ b = \&\texttt{mut}\ a;\ \{\texttt{let mut}\ y = 0;\ *b = \&y\}^{\texttt{n}};\ b\}^{\texttt{m}} \tag{4.3}$$

Example 4.3 exhibits *unsafe* borrowing. The assignment "$*b = \&y$" in practice modifies the value of "a" by creating a borrowed reference to the variable y that exists outside of its lifetime. As a result, this example leads to a dangling pointer when using b after y has been

dropped. However, similar to Rust, our type system prevents such errors by enforcing the constraint that the type of y must be a subtype of the type of a. In *FR*$_{FT}$, a program can be considered safe in terms of type and borrowing when it satisfies the following conditions: (1) It guarantees the use of valid references, disallowing the use of dangling references. (2) It safeguards the property invariant for mutable borrowed references. (3) It preserves the uniqueness of `Trc` extension. In this chapter, we first introduce such a type system for *FR*$_{FT}$ expressions and then ensure its soundness by proving standard progress and preservation theorems [99].

1.1 The Type System in *FR*$_{FT}$

The type system of *FR*$_{FT}$ is defined inductively by the set of typing rules presented in the following. Each rule yields a judgment in the form: $\Gamma_1 \vdash \langle e : \tau \rangle^1_\sigma \dashv \Gamma_2$ where Γ_1 is the typing environment mapping variables to a slot type $\langle \tilde{\tau} \rangle^{\mathtt{m}}$ with an allocated lifetime $\mathtt{m}$. When evaluating the expression e under the typing environment Γ_1, a new environment Γ_2 is produced. The difference between the two environments represents the effect caused by the expression e. Specifically, as in Rust, type checking in *FR*$_{FT}$ is responsible for enforcing ownership discipline. This occurs in the *flow-sensitive* phase, where borrowing can have effects, like when borrowing a variable x immutably, which prohibits writing its contents as long as the reference exists to x. This constraint is captured by the borrow checker. Note that, as in chapter 3, we define the declaration context $\mathcal{D}$ in order to maintain the signatures of the functions present in the program. Then, for all f such that f `::= fn` $f(\overline{\texttt{mut}\ x : \mathtt{S}})\{\overline{e}\}^1$, we have $\mathcal{D}[f \mapsto (\overline{\mathtt{S}})]$. Going forward, `l` is the *lifetime context* and σ is the store typing. The presence of σ in typing judgment is necessary to keep track of the heap-allocated location as described in [56]. For example:

$$S_\varnothing \triangleright \{\texttt{let mut}\ x = \texttt{trc}(1)\}^1 \text{ then, we have: } \{\ell_a \mapsto \langle 1 \rangle^1\} \triangleright \{\texttt{let mut}\ x = \ell_a^\blacklozenge\}^1$$

As previously explained, Γ, maps each variable to its type. However, in this particular example, ℓ_a is not represented in any typing environment. Specifically, it refers to the location ℓ_a allocated in the heap and this appears in the program store S. This raises the question: "What is the type of location $\ell_a^\blacklozenge$ when typing the `let mut` $x = \ell_a^\blacklozenge$ expression?". In this context, our expression involves a concrete location, and its type depends on the content of the program store S that we start with. For instance, in this example, $\ell_a^\blacklozenge$ has the type $\blacklozenge$`int` since it is initialized in our program store by "`0`" (i.e. `int`). In other words, we can infer that the type of a location is dependent on the type of its current contents in S. Additionally, as mentioned in [56], the type of a location can be determined when it is created in memory. Therefore, even if we change the initial value stored in this location, the type remains the same, preserving the original value's type. To handle this scenario, we introduce the metavariable σ to describe the program store, allowing us to compute the type of locations without directly searching in S. For example: $\sigma \vdash \ell_a^\blacklozenge : \blacklozenge\texttt{int}$ or $\sigma(\ell_a^\blacklozenge) = \blacklozenge\texttt{int}$.

`Trc` types

Similar to the `Box` type in Rust, the `Trc` type implements the move semantics. In Rust, the ownership can either be completely consumed or temporarily transferred using the borrowing approach. To exemplify this point, consider the following:

$$\{\texttt{let mut } x \texttt{ = trc}(0);\ \texttt{let mut } y \texttt{ = } x\}^{1} \tag{4.4}$$

In this example, we create a reference of `Trc` type and bind it to x. Then we create the variable y by initializing it with the value of x. The `Trc` type applies the move semantics, whereby the value of x is moved to y. At this point, x is currently uninitialized and its value takes the lifetime of the new variable, which in this case is y. In the following example, instead of moving x, we create a copy of the reference:

$$\{\texttt{let mut } x \texttt{ = trc}(0);\ \texttt{let mut } y \texttt{ = } x.clone\}^{1} \tag{4.5}$$

The `clone` expression copies the reference into x and stores it into y. Due to the two categories of `Trc`, in this example, y is of type $\diamond x$. To manage this type correctly, the type system prohibits moving x as long as y exists. Our type system must be able of determining, based on the typing environment, whether a given variable is forbidden to move or not. For instance, it must identify if x is cloned into that environment (e.g. $\diamond x$). Now, we have the capability to temporarily move the ownership of x as follows:

$$\{\texttt{let mut } x \texttt{ = trc}(0);\ \{\texttt{let mut } y \texttt{ = } \&\texttt{mut } x\}^{\texttt{m}}\}^{1} \tag{4.6}$$

Unlike in 4.4, in this case, we have borrowed ownership of variable x for the lifetime of y (i.e., for lifetime `m`). After y is dropped, ownership reverts to x for its location. Similar to the approach in 4.5, our type system needs to capture this information from the typing environment. The last example is the following:

$$\{\texttt{let mut } x \texttt{ = box(trc}(0)\texttt{)};\ \{\texttt{let mut } y \texttt{ = } \&^{*}x\}^{\texttt{m}}\}^{1} \tag{4.7}$$

In 4.7, x refers to a heap location of `Box` type which, in turn, refers to another heap location of `Trc` type containing an integer. We can then see the equivalence between the program store and the typing environment after reducing the first instruction as follows: $\{\ell_{1::x} \mapsto \langle \ell_a^{\blacksquare} \rangle^{1}, \ell_a \mapsto \langle \ell_b^{\blacklozenge} \rangle^{*}, \ell_b \mapsto \langle 0 \rangle^{1}\} \sim \{x \mapsto \langle \blacksquare \blacklozenge \texttt{ int} \rangle^{1}\}$. Note that to ensure the safety of the `Trc` type, the type system prevents the possibility of having a `Trc` inside a `Trc` and so on (for example, $\blacklozenge \blacksquare \blacklozenge$`int` is not permitted but $\blacksquare \blacksquare$ `int` is allowed). Additionally, through the inner block, we have borrowed the contents of x for the lifetime of y.

1.2 Preliminaries

As in the previous chapter, we introduce some supportive functions that facilitates the explanation of the typing rules. In FR_{FT}, as well as in Rust, types carry the semantics copy and move. We can state that our type system encodes both type and borrow checking rules that are required to determine when it is safe to copy or move a variable.

Definition 4.1 (Copy and Move Types) *A type τ has a copy semantics denoted by $copy(\tau)$, when τ is either a basic type* `int`, `bool`, *or τ is a shared reference $\&\overline{\omega}$. Otherwise, all other types (mutable references,* `box` *and* `Trc`*) have move semantics.*

If we extend our syntax to `struct` or `tuple`, the copy semantics only apply if all its elements also apply the copy semantics. Moreover, ensuring safety in FR_{FT} demands the ability to distinguish between mutable, immutable borrowed, or cloned locations. For example, let Γ be a typing environment such that $\Gamma = \{x \mapsto \langle \blacklozenge \texttt{int} \rangle^{\texttt{l}}, y \mapsto \langle \&\texttt{mut}\ x \rangle^{\texttt{l}}\}$. Based on the information captured by Γ, we can guess that x is mutably borrowed by y and thus x cannot be assigned or moved. Henceforth, we say that x is read prohibited by y. In other words, as long as y exists as a reference to x in this environment, x is not allowed to be read or written. Accordingly, we require a mechanism to determine if a location is borrowed as mutable (readProhibited) or as immutable (writeProhibited) or if a location is cloned (TrcMoveProhibited):

Definition 4.2 (Path, Path Conflict and Type Containment) *A path π is defined as a sequence of zeros ($\pi = \epsilon$) or more dereferences ($\pi = \pi'.*$). In addition, the notation $u = \pi_x \mid x$ denotes a destructuring of an lval u into its base x and path π_x. Therefore, it is rewarding to verify if two given lvals share the same path. In that case, let $u = \pi_x \mid x$ and $\omega = \pi_y \mid y$ be lvals. Then, u is said to be in conflict with ω, denoted $u \bowtie \omega$, if $x = y$. Finally, for a given environment Γ and $x \in dom(\Gamma)$ s.t. $\Gamma(x) = \langle \tilde{\tau} \rangle^{\texttt{l}}$ for some* `l`*, we define that x contains the type τ', denoted $\Gamma \vdash x \rightsquigarrow \tau'$ and is set as contains$(\Gamma, \tilde{\tau}, \tau')$:*

$$\mathit{contains}(\Gamma, \tilde{\tau}, \tau') = \begin{cases} \mathit{contains}(\Gamma, \tilde{\tau}_1, \tau') & \text{if } \tilde{\tau} = \blacksquare\tilde{\tau}_1 \text{ or } \tilde{\tau} = \blacklozenge\tilde{\tau}_1 \\ \texttt{true} & \text{if } \tilde{\tau} = \tau' \\ \texttt{false} & \text{otherwise} \end{cases}$$

In the given example:

$$\{\texttt{let mut}\ x = \texttt{trc}(\texttt{box}(0));\ \texttt{let mut}\ y = \&x;\ \texttt{let mut}\ z = \&^*x\}^{\texttt{m}} \tag{4.8}$$

The typing environment can be represented as: $\Gamma = [x \rightarrow \langle \blacklozenge \blacksquare \texttt{int} \rangle^{\texttt{m}},\ y \rightarrow \langle \&x \rangle^{\texttt{m}},\ z \rightarrow \langle \&^*x \rangle^{\texttt{m}}]$. According to the provided Definition, the path of x is defined as $\pi = \epsilon$, while the path of $^*y$ implies the appending of a selector on another path (i.e. $\pi = \pi'.*$). Moreover, we observe that y and z are in conflict (i.e. $y \bowtie z$) since they both refer to the same base, x. Lastly, regarding the *contains* functions, if we denote $\Gamma(x) = \langle \blacklozenge \blacksquare \texttt{int} \rangle^{\texttt{m}}$, then it follows that *contains*$(\Gamma, \blacklozenge \blacksquare \texttt{int}, \blacksquare\texttt{int})$ holds `true`.

1.2.1 Read and Write Function

The FR_{FT} type system needs to ascertain, for a given lval ω, whether it is prohibited to read or write. Let's consider the following example:

$$\{\texttt{let mut}\ x = \texttt{trc}(0);\ \{\texttt{let mut}\ y = \&x;\ ^*x = 1\}^{\texttt{n}}\}^{\texttt{m}} \tag{4.9}$$

Example 4.9 is rejected by the FR_{FT} type system. After typing the "`let mut` y = $\&x$" expression in the inner block, we get the following typing environment: $\{x \mapsto \langle \blacklozenge \texttt{int} \rangle^{\texttt{m}}, y \mapsto \langle \&x \rangle^{\texttt{n}}\}$. Therefore, when typing the "$^*x$ = 1" expression, the borrow checker detects that x is already borrowed as immutable by y. In order to achieve these requirements, we introduce the following Definitions:

Definition 4.3 (Read Prohibited) *For a given lval ω, readProhibited(Γ, ω) is defined as follows: there exists $x \in dom(\Gamma)$ such that $\Gamma \vdash x \rightsquigarrow \&\texttt{mut}\ \overline{u} \wedge \exists\, i.(u_i \bowtie \omega)$.*

Definition 4.4 (Write Prohibited) *For a given lval ω, writeProhibited(Γ, ω) is defined as: there exists $x \in dom(\Gamma)$ such that $\Gamma \vdash x \rightsquigarrow \& \overline{u} \wedge \exists\, i.(u_i \bowtie \omega)$ or readProhibited(Γ, ω).*

1.2.2 A Safe Path

A crucial aspect of the FR_{FT} system is ensuring inter-thread memory safety. In essence, it aims to prevent one thread from compromising the stability of another thread's memory when they share data, which could potentially lead to issues like creating dangling pointers. Consequently, it becomes essential to verify, at specific points in a program, whether there are references to data shared between threads, especially during cooperation. For instance, when a thread cooperates, it means another thread takes control, highlighting the significance of avoiding the presence of borrowed shared data in the current thread's environment. This helps prevent other threads from triggering unexpected errors. Thus, we introduce the following mutually recursive Definitions:

Definition 4.5 (Safe Trc) *An environment Γ is said to be safeTrc if the content of an active* `Trc` *is not borrowed in Γ. Then, safeTrc(Γ) is a function defined as: for all $x \in dom(\Gamma)$ if $\Gamma \vdash x \rightsquigarrow \&[\texttt{mut}]\ \overline{u}$ then $\neg\exists\, i.(traversTrc(\Gamma, u_i))$.*

Definition 4.6 (TraversTrc) *Let Γ be an environment and let ω be an lval. Then, traversTrc function is responsible to determine whether the path describing ω traverses an active* `Trc` *type. Thus, for some* `l`, $\omega = \pi_x \mid x$ *s.t.* $\Gamma(x) = \langle \tilde{\tau} \rangle^{\texttt{l}}$, *traversTrc$(\Gamma, \omega)$ is defined as* `check`$(\Gamma, \pi_x \mid \tilde{\tau})$*:*

$$
\begin{aligned}
\textit{check}(\Gamma, \epsilon \mid \tau) &= \texttt{false} \\
\textit{check}(\Gamma, (\pi.*) \mid \diamond\overline{\omega}) &= \texttt{false} \\
\textit{check}(\Gamma, (\pi.*) \mid \blacklozenge\tau) &= \texttt{true} \\
\textit{check}(\Gamma, (\pi.*) \mid \blacksquare\tau) &= \textit{check}(\Gamma, \pi \mid \tau) \\
\textit{check}(\Gamma, (\pi.*) \mid \&[\texttt{mut}]\,\overline{\omega}) &= \bigvee_i \textit{traversTrc}(\Gamma, \pi.\omega_i)
\end{aligned}
$$

Figure 4.1 presents an example written in FR_{FT} at source level that illustrates this particular approach. Let's consider a scenario where `thread 1` executes the `createVec` function,

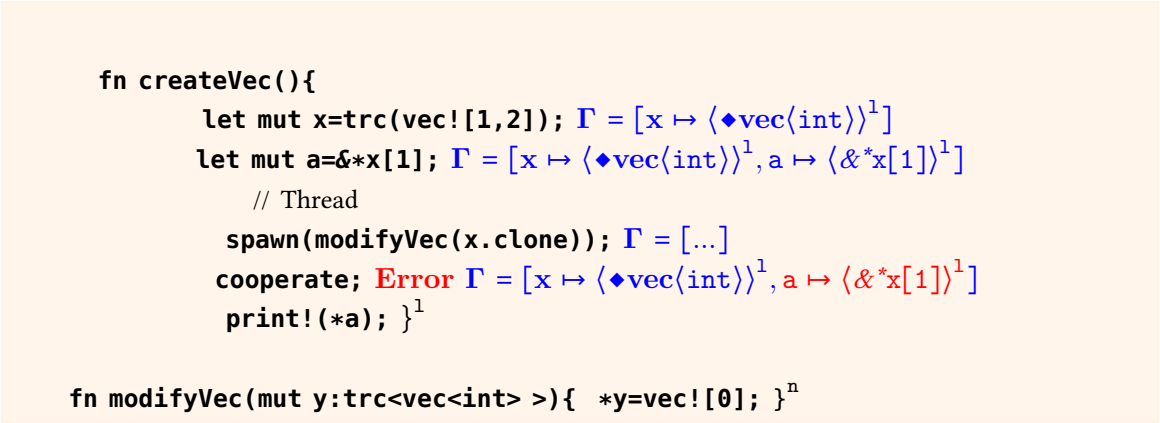

Figure 4.1: An Empowering Type System

and another `thread 2` executes the `modifyVec` function. In the `createVec` function, we intuitively create a heap location of `Trc` type, containing a `vector`. In Rust, the `vector` is a resizable contiguous array where the contents are allocated in the heap. Then, after creating the variable `x`, we create a reference to the second vector case and bind it to `a`. Following this, we create another thread (`thread 2`) using the `spawn` expression. When the first thread continues to execute, it cooperates and `thread 2` takes control. At this point, when `thread 2` executes the `modifyVec` function, it inadvertently creates a dangling pointer in the memory of `thread 1`, as depicted in Figure 4.1. To prevent such errors and maintain a well-typed FR_{FT} program, the type system rejects this program at cooperation expression typing time by employing the *safeTrc* function. Note that in the semantics of this thesis, we have not considered the semantics of a `vector`, `struct`, or `tuple`, however FR_{FT} supports all the functionality to add these types. Furthermore, to stress the efficiency of our semantics and type system, we have implemented in the experimental section the `tuple` type.

Next, we simulate an intriguing feature that the FR_{FT} type system must accommodate in Listing 4.1:

```
1 {
2     let mut x = trc(0);
3     let mut y = &x;
4     let mut a = &mut *y;// Error: "a" involves an immutable borrow
5     *a = 1;
6 }
```

Listing 4.1: An Lval Involving an Immutable Borrow

Listing 4.1 is rejected by both Rust and the FR_{FT} type system. In FR_{FT}, all variables are declared with the `mut` keyword, making them mutable by default. However, on line 4 of the example, creating a mutable reference to the contents of y contradicts the borrowing property, as it allows "a" to modify the contents of y afterward. Due to the path of "a" traversing an immutable reference, it is assumed to be immutable. Consequently, we must reject this example. To reinforce this feature, we introduce the following definition:

Definition 4.7 (Mutable) *An lval ω is said to be mutable if the path it describes never crosses an immutable borrow as mentioned above. Then, for some* `l`, $\omega = \pi_x \mid x$ *s.t.* $\Gamma(x) = \langle\widetilde{\tau}\rangle^{\mathtt{l}}$, *mut*$(\Gamma, \omega)$ *function is defined as* `mutable`$(\Gamma, \pi_x \mid \widetilde{\tau})$*:*

$$\begin{aligned}
\texttt{mutable}(\Gamma, \epsilon \mid \tau) &= \texttt{true} \\
\texttt{mutable}(\Gamma, (\pi.*) \mid \&\,\overline{\omega}) &= \texttt{false} \\
\texttt{mutable}(\Gamma, (\pi.*) \mid \diamond\overline{\omega}) &= \texttt{false} \\
\texttt{mutable}(\Gamma, (\pi.*) \mid \blacksquare\tau) &= \texttt{mutable}(\Gamma, \pi \mid \tau) \\
\texttt{mutable}(\Gamma, (\pi.*) \mid \blacklozenge\tau) &= \texttt{mutable}(\Gamma, \pi \mid \tau) \\
\texttt{mutable}(\Gamma, (\pi.*) \mid \&\texttt{mut}\ \overline{\omega}) &= \bigwedge_i \mathit{mut}(\Gamma, \pi.\omega_i)
\end{aligned}$$

The distinction between the definitions in 4.4 and 4.7 is highlighted as follows: in general, variables in FR_{FT} are mutable by default. The *mut*(.,.) function is employed to safeguard against having mutable access through immutable borrowing. For example $\{\texttt{let mut}\ a = \texttt{trc}(0); \texttt{let mut}\ b = \&a\}^{\mathtt{m}}$ where $\Gamma = \{a \mapsto \langle\blacklozenge\texttt{int}\rangle^{\mathtt{m}}, b \mapsto \langle\&a\rangle^{\mathtt{m}}\}$, we have $\neg\mathit{mut}(\Gamma, {*b})$ since ${*b}$ involves an immutable borrow. The *writeProhibited*(.,.) function is responsible for protecting against achieving mutable access by immutable borrowing. In the same example, a is write prohibited by b.

1.2.3 Move Function

As mentioned earlier, types in FR_{FT} encompass both move and copy semantics. Now, the question arises as to how our type system models the behavior of moving values. To address this, let us consider the following example:

$$\{\texttt{let mut}\ x = \texttt{trc}(0); \{\texttt{let mut}\ y = x\}^{\mathtt{n}}\}^{\mathtt{m}} \tag{4.10}$$

In example 4.10, the inner block "`let mut` $y = x$" signifies that the content of x is moved to y, and consequently, y becomes the new owner of the value. The type system effectively identifies this behavior by directly examining the typing environment, leading to the following representation: $\Gamma = [x \mapsto \langle\lfloor\blacklozenge\texttt{int}\rfloor\rangle^{\mathtt{m}}, y \mapsto \langle\blacklozenge\texttt{int}\rangle^{\mathtt{n}}]$. This implies that the type of x is currently undefined, denoted by $\lfloor\tau\rfloor$. Furthermore, as we discussed in chapter 2, Rust also supports the semantics of partial move, which indicates that a part of a `struct` or `tuple` is moved, for example. However, in FR_{FT}, partial moves are limited to the `Box` type exclusively. The example provided below illustrates such a case:

$$\{\texttt{let mut}\ x = \texttt{box}(\texttt{box}(0)); \{\texttt{let mut}\ y = {*x}\}^{\mathtt{n}}\}^{\mathtt{m}} \tag{4.11}$$

Following this example and after the execution of the declaration in the outer block, the typing environment is as follows: $\Gamma = [x \mapsto \langle\blacksquare\blacksquare\texttt{int}\rangle^{\mathtt{m}}]$. Subsequently, when we create the variable y, we move the content of x into y. As a consequence, the new type of x becomes $\Gamma(x) = \langle\blacksquare\lfloor\blacksquare\texttt{int}\rfloor\rangle^{\mathtt{m}}$, indicating that only a part of the `box` type is moved. Note that, akin to Rust's `Rc`, it is prohibited in FR_{FT} to move out of a `Trc`. In conclusion, to enforce move semantics in the type system, we define the following:

Definition 4.8 (Move) *The move*(Γ, ω) *partial function returns the resulting environment after moving the value of an lval* ω*. Then for a given lval* ω *such that* $\omega = \pi_x \mid x$ *where* $\Gamma(x) = \langle\tilde{\tau}_1\rangle^{\mathtt{l}}$ *for some lifetime* `l`*,* *move*(Γ, ω) *is defined as* $\Gamma[x \mapsto \langle\tilde{\tau}_2\rangle^{\mathtt{l}}]$ *where* $\tilde{\tau}_2 = \mathit{strike}(\pi_x \mid \tilde{\tau}_1)$ *defined as:*

$$\begin{aligned} \mathit{strike}(\epsilon \mid \tau_1) &= \lfloor \tau_1 \rfloor \\ \mathit{strike}((\pi.*) \mid \blacksquare\tilde{\tau}_1) &= \blacksquare\tilde{\tau}_2 \quad \mathit{where}\ \tilde{\tau}_2 = \mathit{strike}(\pi \mid \tilde{\tau}_1) \end{aligned}$$

Note that in the case of dereferencing, the *strike* function is exclusively applicable to `box` types. In the case of active and inactive `Trc` types, their contents cannot be moved.

1.2.4 `Trc` property safety

It is crucial not to permit a `Trc` type containing another `Trc` type, as it would compromise the property of the inactive `Trc`. For instance, the following scenario is not allowed in FR_{FT}:

$$\begin{aligned} &\mathtt{fn}\ f_1(\mathtt{mut}\ x : \blacklozenge\blacklozenge\ \mathtt{int})\{\mathtt{cooperate};\ {}^{**}x = 0\}^{\mathtt{n}} \\ &\mathtt{fn}\ f_2(\mathtt{mut}\ x : \blacklozenge\blacklozenge\ \mathtt{int})\{\mathtt{let\ mut}\ a = \mathtt{trc}(0);\ {}^{*}x = a.clone\}^{\mathtt{m}} \\ &\{\mathtt{let\ mut}\ x = \mathtt{trc}(\mathtt{trc}(0)); \mathtt{spawn}(f_1(x.clone)); \mathtt{spawn}(f_2(x.clone))\}^{\mathtt{l}} \end{aligned} \tag{4.12}$$

In this example, if the function f_1 is executed after f_2, it means that the content of x becomes an inactive `Trc` that is not allowed to access the data. Therefore, the execution of the ${}^{**}x$ expression is inconsistent and violates the *property* of the inactive `Trc`. Accordingly, in order to statically protect shared data between threads, namely data encapsulated in `Trc`s, we adhere to the following solution: avoid having complex `Trc` types (i.e. `Trc` within `Trc`) as follows:

Definition 4.9 (Contains Trc) *Let* Γ *be an environment and let* τ *be a type. Then, the function* *containsTrc*(Γ, τ) *is responsible for recursively verifying whether* τ *contains an active* `Trc` *type (respectively an inactive* `Trc` *type) and it is defined as follows:*

$$\mathit{containsTrc}(\Gamma, \tau) = \begin{cases} \mathit{containsTrc}(\Gamma, \tau') & \mathit{if}\ \tau = \blacksquare\tau'\ \mathit{or}\ (\tau = \&[\mathtt{mut}]\,\overline{\omega}\ \mathit{where}\ \forall_i.(\Gamma(\omega_i) = \tau')) \\ \mathit{true} & \mathit{if}\ \tau = \blacklozenge\tau'\ \mathit{or}\ \tau = \diamond\overline{\omega} \\ \mathit{false} & \mathit{otherwise} \end{cases}$$

Henceforth, the following example is akin to 4.12, with the exception that the `Trc` type is not composed of another `Trc`, as illustrated in 4.13:

$$\begin{aligned} &\mathtt{fn}\ f_1(\mathtt{mut}\ x : \blacklozenge\mathtt{int})\{\mathtt{cooperate};\ {}^{*}x = 0\}^{\mathtt{n}} \\ &\mathtt{fn}\ f_2(\mathtt{mut}\ x : \blacklozenge\mathtt{int})\{\mathtt{let\ mut}\ a = \mathtt{trc}(0); x = a.clone\}^{\mathtt{n}} \\ &\{\mathtt{let\ mut}\ x = \mathtt{trc}(0); \mathtt{spawn}(f_1(x.clone); \mathtt{spawn}(f_2(x.clone))\}^{\mathtt{l}} \end{aligned} \tag{4.13}$$

In comparison to 4.12, this example is completely safe and is accepted by the FR_{FT} type system.

An interesting property of moving `Trc` types is demonstrated in the following example:

```
1 {
2   let mut x = box(trc(0));
3   {
4     let mut y = *x.clone;
5     let mut a = x; // error: cannot move out
6     }              // of ‘x‘ as it is cloned
7 }
```

Listing 4.2: Preserving the Integrity of Move Semantics

On line 4, the typing environment is : $\Gamma = [x \mapsto \langle \blacksquare \blacklozenge \texttt{int} \rangle^1, y \mapsto \langle \diamond * x \rangle^1]$. Then, if we move the content of x on line 5, y becomes *ill-typed*. Hence, to ensure that y remains *well-typed*, we must prevent x from being moved or becoming inactive as long as y exists in Γ. This concept is akin to the idea of that x is borrowed in Γ. To enforce this property, the following definition is introduced:

Definition 4.10 (Trc Move Prohibited) *Let Γ be an environment and let ω be an lval, then ω is said to be move-prohibited, denoted TrcMoveProhibited(Γ, ω), if there exists $y \in dom(\Gamma)$ such that $\Gamma \vdash y \rightsquigarrow \diamond\overline{u} \wedge \exists_i(u_i \bowtie \omega)$.*

1.2.5 Type and Environment Join

Another crucial function is to determine the union of two types and, consequently, the union of two environments. This is necessary, for instance, for the control flow extension:

Definition 4.11 (Type Join) *For the given partial types $\tilde{\tau}_1$ and $\tilde{\tau}_2$, we define $\tilde{\tau}_3$ the resulting type from the union of $\tilde{\tau}_1$ and $\tilde{\tau}_2$, denoted $\tilde{\tau}_3 = \tilde{\tau}_1 \sqcup \tilde{\tau}_2$ and is defined as union$(\tilde{\tau}_1, \tilde{\tau}_2)$ according to Figure 4.2 as follows:*

$$
\begin{aligned}
\mathit{union}(\tilde{\tau}_1, \tilde{\tau}_2) &= \tilde{\tau}_2 \quad if\ \tilde{\tau}_1 \sqsubseteq \tilde{\tau}_2 \\
\mathit{union}(\tilde{\tau}_1, \tilde{\tau}_2) &= \tilde{\tau}_1 \quad if\ \tilde{\tau}_2 \sqsubseteq \tilde{\tau}_1 \\
\mathit{union}(\blacksquare\tilde{\tau}_1, \blacksquare\tilde{\tau}_2) &= \blacksquare \mathit{union}(\tilde{\tau}_1, \tilde{\tau}_2) \\
\mathit{union}(\blacklozenge\tau_1, \blacklozenge\tau_2) &= \blacklozenge \mathit{union}(\tau_1, \tau_2) \\
\mathit{union}(\diamond\overline{\omega}, \diamond\overline{u}) &= \diamond\overline{\omega}, \overline{u} \\
\mathit{union}(\&\overline{\omega}, \&\overline{u}) &= \&\overline{\omega}, \overline{u} \\
\mathit{union}(\&\texttt{mut}\ \overline{\omega}, \&\texttt{mut}\ \overline{u}) &= \&\texttt{mut}\ \overline{\omega}, \overline{u}
\end{aligned}
$$

Note that, we use $\sqsubseteq$ for sequences, as seen in **(W-Bor)** and **(W-Clone)** in Figure 4.2. There are constraints when it comes to joint borrowings or inactive `Trc`s. For example, we can have $\mathit{union}(\&x, \&y) = \&x, y$. However, the case $\mathit{union}(\&x, \&\texttt{mut}\ y)$ can never exist. Similarly, for `Trc` types, we can have $\mathit{union}(\diamond x, \diamond y) = \diamond x, y$. On the other hand, the case $\mathit{union}(\diamond x, \blacklozenge\texttt{int})$ is not possible, as an inactive `Trc` type cannot be simultaneously active

$$\frac{}{\widetilde{\tau}_1 \sqsubseteq \widetilde{\tau}_1}\ (\mathbf{W\text{-}Reflex}) \qquad \frac{\widetilde{\tau}_1 \sqsubseteq \widetilde{\tau}_2}{\blacksquare\widetilde{\tau}_1 \sqsubseteq \blacksquare\widetilde{\tau}_2}\ (\mathbf{W\text{-}Box}) \qquad \frac{\overline{u} \subseteq \overline{\omega}}{\&[\mathtt{mut}]\overline{u} \sqsubseteq \&[\mathtt{mut}]\overline{\omega}}\ (\mathbf{W\text{-}Bor})$$

$$\frac{\tau_1 \sqsubseteq \tau_2}{\lfloor\tau_1\rfloor \sqsubseteq \lfloor\tau_2\rfloor}\ (\mathbf{W\text{-}UndefA}) \qquad \frac{\tau_1 \sqsubseteq \tau_2}{\tau_1 \sqsubseteq \lfloor\tau_2\rfloor}\ (\mathbf{W\text{-}UndefB}) \qquad \frac{\widetilde{\tau}_1 \sqsubseteq \lfloor\tau_2\rfloor}{\blacksquare\widetilde{\tau}_1 \sqsubseteq \lfloor\blacksquare\tau_2\rfloor}\ (\mathbf{W\text{-}UndefC})$$

$$\frac{\tau_1 \sqsubseteq \tau_2}{\blacklozenge\tau_1 \sqsubseteq \blacklozenge\tau_2}\ (\mathbf{W\text{-}Trc}) \qquad \frac{\overline{u} \subseteq \overline{\omega}}{\lozenge\overline{u} \sqsubseteq \lozenge\overline{\omega}}\ (\mathbf{W\text{-}Clone})$$

Figure 4.2: Type Strengthening

and inactive. Another important feature to consider is the undefined type. This implies that certain information in a type is incomplete. When combining two undefined types, the resulting type contains the least amount of information to ensure an expected result and enforce the soundness of the type system. For example, suppose $\widetilde{\tau}_1$ is of the form $\blacksquare\lfloor\blacksquare\blacksquare\tau\rfloor$ and $\widetilde{\tau}_2$ is of the form $\blacksquare\blacksquare\lfloor\blacksquare\tau\rfloor$. The first type represents a `box` of undefined type, while the second type represents a `box` of a `box` of an undefined type. Obviously, the first type gives less information than the second one. Thence, the result of the joining between $\widetilde{\tau}_1$ and $\widetilde{\tau}_2$ returns $\widetilde{\tau}_1$ according to Figure 4.2.

Moreover, the same concept can be extended to the joining of the environments as follows:

Definition 4.12 (Environment Join) *Let Γ_1 and Γ_2 be typing environments. Then, Γ_1 reinforces Γ_2, denotes $\Gamma_1 \sqsubseteq \Gamma_2$, if $dom(\Gamma_1) = dom(\Gamma_2)$ and for each variable $x \in dom(\Gamma_1)$ where $\Gamma_1(x) = \langle\widetilde{\tau}_1\rangle^1$, we have its match in Γ_2 such as $\Gamma_2(x) = \langle\widetilde{\tau}_2\rangle^1$ where $\widetilde{\tau}_1 \sqsubseteq \widetilde{\tau}_2$. Additionally, for the given typing environments Γ_1 and Γ_2, we define Γ_3 the strongest resulting typing environment from the union of Γ_1 and Γ_2, denoted $\Gamma_3 = \Gamma_1 \sqcup \Gamma_2$ such that $\Gamma_1 \sqsubseteq \Gamma_3$ and $\Gamma_2 \sqsubseteq \Gamma_3$.*

1.2.6 Drop and Update

Figure 4.3 showcases several essential support functions, detailed as follows:

1. *drop*$(\Gamma, \mathtt{m})$ handles the deallocation of locations with a lifetime `m` by removing them from the environment Γ.

2. *write*$^k(\Gamma, \omega, \tau)$ is responsible for updating the type of a given lval ω, where $\omega = \pi_x \mid x$ and $\Gamma(x) = \langle\widetilde{\tau}_1\rangle^1$ for some lifetime `l`. In addition, for some $k \geq 0$, *write*$^k(\Gamma, \omega, \tau)$ is defined as $\Gamma_2[x \mapsto \langle\widetilde{\tau}_2\rangle^1]$ where $(\Gamma_2, \widetilde{\tau}_2) = \mathtt{update}^k(\Gamma_1, \pi_x \mid \widetilde{\tau}_1, \tau)$ as presented in Figure 4.3.

$$
\begin{array}{lcl}
\mathit{drop}(\Gamma, \mathtt{m}) & = & \Gamma - \{x \mapsto \langle \tilde{\tau} \rangle^{\mathtt{m}} \mid x \mapsto \langle \tilde{\tau} \rangle^{\mathtt{m}} \in \Gamma\} \\
\mathit{write}^{k}(\Gamma_1, \omega, \tau) & = & \Gamma_2[x \mapsto \langle \tilde{\tau}_2 \rangle^{\mathtt{l}}] \; s.t. \; (\Gamma_2, \tilde{\tau}_2) = \mathtt{update}^{k}(\Gamma_1, \pi_x \mid \tilde{\tau}_1, \tau) \\
 & & \mathit{where}\; \omega = \pi_x \mid x \; \mathit{and}\; k \geqslant 0 \\
\mathtt{update}^{0}(\Gamma_1, \epsilon \mid \tilde{\tau}_1, \tau_2) & = & (\Gamma_1, \tau_2) \\
\mathtt{update}^{k \geq 1}(\Gamma_1, \epsilon \mid \tau_1, \tau_2) & = & (\Gamma_1, \mathit{union}(\tau_1, \tau_2)) \\
\mathtt{update}^{k}(\Gamma_1, (\pi.*) \mid \blacksquare\tilde{\tau}_1, \tau) & = & (\Gamma_2, \blacksquare\tilde{\tau}_2) \quad \mathit{where}\; (\Gamma_2, \tilde{\tau}_2) = \mathtt{update}^{k}(\Gamma_1, \pi \mid \tilde{\tau}_1, \tau) \\
\mathtt{update}^{k}(\Gamma_1, (\pi.*) \mid \blacklozenge\tau_1, \tau) & = & (\Gamma_2, \blacklozenge\tau_2) \quad \mathit{where}\; (\Gamma_2, \tau_2) = \mathtt{update}^{k}(\Gamma_1, \pi \mid \tau_1, \tau) \\
\mathtt{update}^{k}(\Gamma_1, (\pi.*) \mid \&\mathtt{mut}\; \overline{u}, \tau) & = & (\sqcup_i \Gamma_i, \&\mathtt{mut}\; \overline{u}) \quad \mathit{where}\; \Gamma_i = \mathit{write}^{k+1}(\Gamma_1, \pi \mid u_i, \tau) \\
 & & \qquad\qquad\qquad\qquad\quad \mathit{and}\; 1 \leqslant i \leqslant \mathtt{len}(\overline{u})
\end{array}
$$

Figure 4.3: Drop and Update Functions

In line with Rust's borrow checker analysis, the FR_{FT} $write^{k}$ function supports two well-known features: *strong update* and *weak update*. The *strong update* approach is based on concepts from previous work on static analysis, such as pointer analysis [77]. In FR_{FT}, *strong update* is applicable to primitive types, references, `boxes`, and `Trc` types (i.e. when the `rank` k `= 0` in the *update* function in Figure 4.3). The *weak update* applies to mutable references when the rank $k \geq 0$. Let is consider this simple example of a *strong update*, which is accepted by the FR_{FT} type system:

```
1 {
2     let mut x = 0;
3     let mut y = 1;
4     let mut a = &mut x;
5     a = &mut y;
6     print!(*a); // 1
7 }
```

Listing 4.3: *Strong update* in FR_{FT}

As demonstrated in this example, the borrow checker quickly identifies that the value of "a" is overwritten in the assignment on line 5. Consequently, it can safely release the borrow on *x*. However, a slight modification to this example renders the *strong update* inapplicable, as shown below:

```
1 {
2     let mut x = 0;
3     let mut y = 1;
4     let mut a = &mut x;
5     let mut b = &mut a;
6
7     *b = &mut y;
8     print!(x);//Error: cannot borrow ‘x‘ as immutable
9             //because it is also borrowed as mutable
```

```
    print!(**a);
}
```

Listing 4.4: *Weak update* in FR_{FT}

In this example, we have changed the type of "a". Since Rust's borrow checker is well aware about invariants of this type (mutable reference), which allows for a *weak update*, this example is rejected by Rust and therefore by FR_{FT}. Similar to the example 4.4, the below example also applies the *weak update* using box types as follows:

```
{
    let mut x = 0;
    let mut y = 1;
    let mut a = Box::new(&mut x);
    *a = &mut y;
    print!(x);//Error: cannot borrow ‘x‘ as immutable
            //because it is also borrowed as mutable
    print!(**a);
}
```

Listing 4.5: *Weak update* in Rust using box type

Again, the same error as in Listing 4.4 except this time with the box type. Perhaps it makes more sense not to apply the *strong update* with the box type because $\blacksquare\tau$ (Box<T> in Rust) is only a *user-defined* type. On the other hand, it is not necessary for the borrow checker to have knowledge of its invariants. Research projects [61] have adopted the approach of *strong* and *weak updates*.

1.2.7 Compatibility of types

For an environment Γ, two partial types $\widetilde{\tau}_1$ and $\widetilde{\tau}_2$ are said to be *compatible type*, denoted as $\Gamma \vdash \widetilde{\tau}_1 \approx \widetilde{\tau}_2$, following the rules (**S-***) outlined in Figure 4.4. Notably, the **(S-Bor)** rule requires that the two types shall have the same *mutability* in order to be compared. For example, $\Gamma \vdash \& x \approx \& y$ is valid, but $\Gamma \vdash \& x \approx \&\,\texttt{mut}\ y$ is not. Moreover, compatibility for borrows disregards their lifetimes, assuming that borrows can be compatible even if the referred slots have different lifetimes. In the context of Trc types, an active Trc is compatible with an inactive Trc type (**S-ATrcL, S-ATrcR**). For instance, consider the code snippet $\{\texttt{let mut}\ x = \texttt{trc}(0); \texttt{let mut}\ y = x.clone; y = \texttt{trc}(1)\}$[1].

Before delving further, it is essential to address the *subtyping* feature in Rust and how FR_{FT} embraces and supports this aspect. This will be outlined in the following section.

$$\frac{}{\Gamma \vdash \texttt{int} \approx \texttt{int}}\ \textbf{(S–Int)} \qquad \frac{}{\Gamma \vdash \texttt{bool} \approx \texttt{bool}}\ \textbf{(S–Bool)} \qquad \frac{\Gamma \vdash \tau_1 \approx \widetilde{\tau}_2}{\Gamma \vdash \lfloor \tau_1 \rfloor \approx \widetilde{\tau}_2}\ \textbf{(S–UnL)}$$

$$\frac{\Gamma \vdash \widetilde{\tau}_1 \approx \widetilde{\tau}_2}{\Gamma \vdash \blacksquare\widetilde{\tau}_1 \approx \blacksquare\widetilde{\tau}_2}\ \textbf{(S–Box)} \qquad \frac{\Gamma \vdash \tau_1 \approx \tau_2}{\Gamma \vdash \blacklozenge\tau_1 \approx \blacklozenge\tau_2}\ \textbf{(S–ATrc)} \qquad \frac{\Gamma \vdash \widetilde{\tau}_1 \approx \tau_2}{\Gamma \vdash \widetilde{\tau}_1 \approx \lfloor \tau_2 \rfloor}\ \textbf{(S–UnR)}$$

$$\frac{\forall_{i,j}(\Gamma(u_i) = \tau_1 \wedge \Gamma(\omega_j) = \tau_2 \wedge \Gamma \vdash \tau_1 \approx \tau_2)}{\Gamma \vdash \&[\texttt{mut}]\overline{u} \approx \&[\texttt{mut}]\overline{\omega}}\ \textbf{(S–Bor)}$$

$$\frac{\forall_{i,j}(\Gamma(u_i) = \blacklozenge\tau_1 \wedge \Gamma(\omega_j) = \blacklozenge\tau_2 \wedge \Gamma \vdash \tau_1 \approx \tau_2)}{\Gamma \vdash \diamond\overline{u} \approx \diamond\overline{\omega}}\ \textbf{(S–ITrc)}$$

$$\frac{\forall_i(\Gamma \vdash \tau_1 \approx \tau_2 \wedge \Gamma(\omega_i) = \blacklozenge\tau_2)}{\Gamma \vdash \blacklozenge\tau_1 \approx \diamond\overline{\omega}}\ \textbf{(S–ATrcL)} \qquad \frac{\forall_i(\Gamma(u_i) = \blacklozenge\tau_1 \wedge \Gamma \vdash \tau_1 \approx \tau_2)}{\Gamma \vdash \diamond\overline{u} \approx \blacklozenge\tau_2}\ \textbf{(S–ATrcR)}$$

Figure 4.4: Compatible Type

1.3 Understanding the Subtyping Relationship

Subtyping can be generally defined as a relationship between types where, if `A` is a subtype of `B`, then the set of values belonging to `A` is a subset of the set of values belonging to `B`. This concept plays a crucial role in object-oriented programming (OOP) languages like Java, Python, and C++, as it enables the creation of modular, reusable, and extensible code. Drawing from a widely recognized example, in the context of subtyping, we can consider "Dog" as a subtype of "Animal". This relationship arises because the set of dogs is encompassed within the larger set of animals, effectively meaning that we can refer to all dogs as animals. On the aspect of typing, subtyping has significance in statically typed languages, making them more permissive and adaptive. By allowing subtyping between types, these languages gain increased flexibility in handling various data structures and interactions between different types.

In statically typed languages, subtyping allows for a certain degree of permissiveness and adaptability. However, it is important to note that in Rust, subtyping differs from traditional notions. Rust introduces a new subtyping relationship involving lifetimes. For instance, `&'a str` is a subtype of `&'b str` if `'a` outlives `'b`. This might seem counterintuitive since "sub" implies smaller, but in this case, the longest lifetime takes precedence, making it the subtype.

In the context of Rust, the type of references consists of two components: the lifetime and the type itself. This new subtyping relationship based on lifetimes adds to Rust's capabilities and uniqueness. Moreover, FR_{FT} also supports subtyping based on the lifetime relationship, following Rust's approach. Specifically, this applies to the reference and inactive `Trc` types, allowing for more nuanced and flexible handling of lifetimes and types, as follows:

$$\frac{}{\Gamma \vdash \texttt{int} \succeq \texttt{l}}\ \textbf{(L-Int)} \qquad \frac{}{\Gamma \vdash \texttt{bool} \succeq \texttt{l}}\ \textbf{(L-Bool)} \qquad \frac{\Gamma \vdash \tau \succeq \texttt{l}}{\Gamma \vdash \blacksquare\tau \succeq \texttt{l}}\ \textbf{(L-Box)}$$

$$\frac{\Gamma \vdash \tau \succeq \texttt{l}}{\Gamma \vdash \blacklozenge\tau \succeq \texttt{l}}\ \textbf{(L-AcTrc)} \qquad \frac{\overline{\Gamma \vdash \omega : \langle \blacklozenge\tau \rangle^{\texttt{m}}} \quad \overline{\texttt{m} \succeq \texttt{l}}}{\Gamma \vdash \diamond\overline{\omega} \succeq \texttt{l}}\ \textbf{(L-IncTrc)} \qquad \frac{\overline{\Gamma \vdash u : \langle \tau \rangle^{\texttt{m}}} \quad \overline{\texttt{m} \succeq \texttt{l}}}{\Gamma \vdash \&[\texttt{mut}]\,\overline{u} \succeq \texttt{l}}\ \textbf{(L-BoR)}$$

Figure 4.5: Well-formed Type

Definition 4.13 (Well-formed Type) *For a given environment Γ, a type τ is said to be well-formed with respect to a lifetime* `l`*, denoted $\Gamma \vdash \tau \succeq \texttt{l}$ according to the rules (**L-***) in Figure 4.5.*

The $\Gamma \vdash \tau \succeq \texttt{l}$ represents the subtyping requirements in FR_{FT}. To explain the subtyping relationship between lifetimes, let us consider the following example:

$$\{\texttt{let mut } x = 0;\ \texttt{let mut } a = \&x; \{\texttt{let mut } y = 1;\ a = \&y\}^{\texttt{m}};\ {}^{*}a\}^{\texttt{l}} \tag{4.14}$$

The FR_{FT} type system rejects Example 4.14 due to a specific issue that arises when the inner block is executed. In this scenario, the variable a becomes a reference to y instead of x. Consequently, when outside the inner block, a turns into a dangling pointer, as it points to y, which is dropped when the inner block is reduced. However, the lifetime of the new value must be greater than the lifetime of the current value. In the mentioned example, this condition is not met, as illustrated in Figure 4.5 (i.e. $\Gamma \vdash \&y \succcurlyeq \texttt{l}$ is not satisfied).

1.4 The Typing Rules

In this section, we describe the typing rules of FR_{FT}, explaining each of them in detail according to Figure 3.5. Let us define that ϵ in the typing rules stands for ϵ in the program store. We start with the typing of lvals in the FR_{FT} program as follows:

$$\frac{\Gamma(x) = \langle\tilde{\tau}\rangle^{\texttt{m}}}{\Gamma \vdash x : \langle\tilde{\tau}\rangle^{\texttt{m}}}\ (\textit{T-LvVar}) \qquad \frac{\Gamma \vdash \omega : \langle \blacklozenge\tau \rangle^{\texttt{m}}}{\Gamma \vdash {}^{*}\omega : \langle\tau\rangle^{\texttt{m}}}\ (\textit{T-LvTrc}) \qquad \frac{\Gamma \vdash \omega : \langle \blacksquare\tilde{\tau} \rangle^{\texttt{m}}}{\Gamma \vdash {}^{*}\omega : \langle\tilde{\tau}\rangle^{\texttt{m}}}\ (\textit{T-LvBox})$$

$$\frac{\Gamma \vdash \omega : \langle \&[\texttt{mut}]\overline{u} \rangle^{\texttt{n}} \quad \overline{\Gamma \vdash u : \langle\tau\rangle^{\texttt{m}}}}{\Gamma \vdash {}^{*}\omega : \langle \sqcup_i \tau_i \rangle^{\sqcap_i \texttt{m}_i}}\ (\textit{T-LvBorrow})$$

Based on the aforementioned rules (*T-Lv**), an lval denoted by ω is considered *well-typed* concerning the typing environment Γ, expressed as $\Gamma \vdash \omega : \langle\tilde{\tau}\rangle^{\texttt{m}}$. This premise ensures that ω belongs to Γ and currently possesses the type $\langle\tilde{\tau}\rangle^{\texttt{m}}$. In the *T-LvBorrow* rule, two notations are used: (1) $\sqcup_i \tau_i$ represents the join of the types of u, utilizing the *union* function 4.11. (2) $\sqcap_{\texttt{i}} \texttt{m}_{\texttt{i}}$ indicates the lowest lifetime of $\texttt{m}_0, \ldots, \texttt{m}_{\texttt{n}}$, ensuring that the sequence of active lifetimes remains defined within an expression.

Consider the following example: $\Gamma = \{y \mapsto \langle \blacklozenge\texttt{int} \rangle^{\texttt{m}}, b \mapsto \langle \diamond y \rangle^{\texttt{m}}, c \mapsto \langle \& y \rangle^{\texttt{m}}\}$ it follows that $\Gamma \vdash {}^{*}y : \texttt{int}$ and $\Gamma \vdash {}^{*}c : \blacklozenge\texttt{int}$. Since $\Gamma \vdash b : \diamond y$, then ${}^{*}b$ cannot be typed (i.e. only active `Trc` type can be accessed). In addition, lvals can have partial types as long as their internal "*path*" is defined (Definition 4.2). For instance, if $\Gamma = \{x \mapsto \blacksquare\lfloor\blacksquare\texttt{int}\rfloor\}$ it follows that x and ${}^{*}x$ can be typed while ${}^{**}x$ cannot since ${}^{**}x$ is a partial type(i.e. $\lfloor\blacksquare\texttt{int}\rfloor$) and we cannot access a value that has a partial type (e.g. we cannot read the content of ${}^{**}x$). Note that this situation is not possible with the active `Trc` type, as moving out of an active `Trc` type (i.e., $\blacklozenge\widetilde{\tau}$) is not permitted. For example:

$$\begin{array}{c}\{\texttt{let mut } x = 0;\ \{\texttt{let mut } y = \&\, x;\ \texttt{let mut } a = \&\texttt{mut } y;\\ \texttt{let mut } z = 1;\ {}^{*}a = \&\, z\}^{\texttt{m}}\}^{\texttt{l}}\end{array} \tag{4.15}$$

In example 4.15, in the inner block, we create a shared reference to x and link it to y. Subsequently, we create a mutable reference to y and bind it to a. Then we mutate the contents of a by creating a immutable reference to z. Consequently, in this case, the type of y becomes $(\& x, z)$ and the lifetime of its contents is determined by the lowest lifetime between x and z (i.e. `min(l,m)`).

1.4.1 Read and Write

After introducing the typing rules of an lval in an FR_{FT} program, next we will present and explain the typing rules for FR_{FT} expressions. The first rule is: *T-Const*, its role is to handle constant values that can have a FR_{FT} program such as: unit value, integer, Boolean and reference values using the store typing, σ, given in the form of the judgement as follows:

$$\frac{\sigma \vdash v : \tau}{\Gamma \vdash \langle v : \tau \rangle^{\texttt{l}}_{\sigma} \dashv \Gamma} \tag{*T-Const*}$$

As previously mentioned, the metavariable σ represents the program store S and helps determine the type of locations in our typing rules. Its significance becomes evident when typing an expression containing a reference value. However, to ensure the appropriate application of copy and move semantics, we need specific typing rules for each. For this purpose, we introduce the *T-Copy* and *T-Move* rules. The *T-Copy* rule handles copying value of an lval that has copy semantics 4.1:

$$\frac{\Gamma \vdash \omega : \langle \tau \rangle^{\texttt{m}} \quad \mathit{copy}(\tau) \quad \neg \mathit{readProhibited}(\Gamma, \omega)}{\Gamma \vdash \langle \hat{\omega} : \tau \rangle^{\texttt{l}}_{\sigma} \dashv \Gamma} \tag{*T-Copy*}$$

Therefore, it is considered safe to keep the output typing environment Γ unchanged in this rule. For this to apply, ω must not be borrowed as mutable, which is confirmed by $\neg \mathit{readProhibited}(\Gamma, \omega)$. Note that to copy an lval, its contents must be read. However, if the lval is borrowed as mutable, accessing its data is forbidden. With the *T-Move* rule, rather than copying the value, we move it out of ω that has move semantics:

$$\frac{\begin{array}{c}\Gamma_1 \vdash \omega : \langle \tau \rangle^{\texttt{m}} \quad \neg \mathit{writeProhibited}(\Gamma_1, \omega)\\ \neg \mathit{TrcMoveProhibited}(\Gamma_1, \omega) \quad \Gamma_2 = \mathit{move}(\Gamma_1, \omega)\end{array}}{\Gamma_1 \vdash \langle \omega : \tau \rangle^{\texttt{l}}_{\sigma} \dashv \Gamma_2} \tag{*T-Move*}$$

This rule must satisfy that ω must not be borrowed or cloned in Γ (checked by $\neg$ *writeProhibited*(Γ_1, ω) and $\neg$ *TrcMoveProhibited*(Γ_1, ω) respectively). Once validated, the output environment removes ω by using the *move* function which replaces exactly one occurrence of τ with $\lfloor \tau \rfloor$. Note that, only `box` type can move its content, while all other types cannot (i.e. similar to `Rc` in Rust, it is forbidden in FR_{FT} to move out of a `Trc`). Hence, to elaborate, we consider the following examples:

```
1 {
2     let mut x = trc(0);
3     let mut y = x.clone;
4     {
5         let mut z = x; //ERROR: x cannot move as long as y exists
6     }
7 }
```

Listing 4.6: An inactive `Trc` is *well-typed*

In Listing 4.6, on line 3, we create a copy of the reference in x (of the active `Trc`). The input environment is now as follows: $\Gamma = [x \mapsto \langle \blacklozenge\texttt{int} \rangle^1, y \mapsto \langle \diamond x \rangle^1]$. If we proceed with the expression reduction on line 5, x will be moved into z, leading to its type becoming a partial type (e.g. $\tilde{\tau}$). This results in the loss of consistency between the active and inactive `Trc` types since the type of y is $\diamond x$. Due to this inconsistency and in accordance with the *T-Move* rule, this program is rejected. Let's consider another example:

```
1 {
2     let mut x = trc(box(0));
3     let mut y = *x; //ERROR: cannot move out of x
4 }
```

Listing 4.7: Cannot move out of an active `Trc`

As in Rust, it is not allowed to move out of a reference counting. Hence, an error is encountered on line 3 when trying to move the contents of an active `Trc` x into y. This situation is addressed by defining the *move* function 4.8. However, the last example is accepted by the FR_{FT} type system:

```
1 {
2     let mut x = box(trc(0));
3     let mut y = *x; //OK
4 }
```

Listing 4.8: Move out of a `Box`

According to the *T-Move* rule, the variable x is not borrowed in its typing environment and the reduction of the expression on line 3 is therefore achieved. Thus, after the reduction on line 3, the type of x becomes $\blacksquare\lfloor\blacklozenge\texttt{int}\rfloor$.

1.4.2 Borrowing and inactive `Trc`

Considering the borrowing invariants, we encounter two possibilities: (1) having one mutable reference and no immutable reference, or (2) having no mutable reference and multiple immutable references. To ensure that our environment, Γ, adheres to these two possibilities, we introduce the following rules:

- The T-*MutBorrow* rule demands that the lval ω should not be *write-prohibited* to allow safe borrowing of the lval:

$$\frac{\Gamma \vdash \omega : \langle \tau \rangle^{\mathtt{m}} \quad \mathit{mut}(\Gamma, \omega) \quad \neg \mathit{writeProhibited}(\Gamma, \omega)}{\Gamma \vdash \langle \&\mathtt{mut}\ \omega : \&\mathtt{mut}\ \omega \rangle^{1}_{\sigma} \dashv \Gamma} \quad (\textit{T-MutBorrow})$$

Before creating a borrow to ω, the *T-MutBorrow* rule ensures two conditions are met. Firstly, ω should not already be borrowed as an immutable reference, which is checked using the *writeProhibited* function. Secondly, this rule verifies that ω is mutable by employing the *mut* function. The *mut* function confirms that the path of ω does not involve traversing an immutable reference, as defined in Definition 4.7.

- The *T-ImmBorrow* rule necessitates that the lval ω is not *read-prohibited*. In both situations, it is crucial for the type $\langle \tau \rangle^{\mathtt{m}}$ of ω to be well-formed, meaning it must not be a partial type:

$$\frac{\Gamma \vdash \omega : \langle \tau \rangle^{\mathtt{m}} \quad \neg \mathit{readProhibited}(\Gamma, \omega)}{\Gamma \vdash \langle \& \omega : \& \omega \rangle^{1}_{\sigma} \dashv \Gamma} \quad (\textit{T-ImmBorrow})$$

In the ensuing rules, we evaluate expressions e rather than lvals ω. Hence, the effect of evaluating e propagates outwards. As a result, we will have typing environments in the rules but they do not appear in the final evaluation result.

1.4.3 Box and `Trc`

In FR_{FT}, dynamic allocation is handled either by returning a `box` type with the *T-Box* rule, or an active `Trc` type with *T-Trc* rule. These types represent an owned pointer to a dynamically allocated location in the heap. Furthermore, to safely share an active `Trc` type between threads, the *T-Trc* rule necessitates that after evaluating e, the resulting type (τ) must not contain an active `Trc` type. This constraint is satisfied by the previously introduced function *containsTrc* 4.9 :

$$\frac{\Gamma_1 \vdash \langle e : \tau \rangle^{1}_{\sigma} \dashv \Gamma_2}{\Gamma_1 \vdash \langle \mathtt{box}(e) : \blacksquare \tau \rangle^{1}_{\sigma} \dashv \Gamma_2} \ (\textit{T-Box})$$

$$\frac{\Gamma_1 \vdash \langle e : \tau \rangle^{1}_{\sigma} \dashv \Gamma_2 \quad \neg \mathit{containsTrc}(\Gamma_2, \tau)}{\Gamma_1 \vdash \langle \mathtt{trc}(e) : \blacklozenge \tau \rangle^{1}_{\sigma} \dashv \Gamma_2} \ (\textit{T-Trc})$$

Using the *T-Box* rule, we begin by typing the expression e, which results in a new typing environment Γ_2. We then encapsulate the type of e within the `box` type. However, with

the *T-Trc* rule, an additional requirement is imposed to ensure that the type of e in Γ_2 does not include an active or inactive `Trc`. Lastly, the *T-Clone* rule returns an inactive `Trc` type, ensuring that ω has an active `Trc` type as follows:

$$\frac{\Gamma \vdash \omega : \langle \blacklozenge \tau \rangle^{\mathtt{m}}}{\Gamma \vdash \langle \omega.clone : \diamond \omega \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma} \quad (\textit{T-Clone})$$

The following example is rejected by the FR_{FT} type system:

```
{
    let mut x = box(trc(1));
    let mut y = trc(x);
}
```

Listing 4.9: Creation of an active `Trc` is unsafe

As shown in Listing 4.9, x is of type ■ ◆ `int` which means that x contains an active `Trc` type. Hence, using the *T-Trc* rule, this program is rejected.

1.4.4 Sequence and Block

The *T-Sequence* rule captures how variable environments are threaded into programs as follows:

$$\frac{\Gamma_1 \vdash \langle e_1 : \tau_1 \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2 \quad \ldots \quad \Gamma_n \vdash \langle e_n : \tau_n \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_{n+1}}{\Gamma_1 \vdash \langle \overline{e} : \tau_n \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_{n+1}} \quad (\textit{T-Sequence})$$

After evaluating an expression in a sequence, the resulting environment is straightforwardly carried over to the subsequent expression, as some expressions may alter the environment during type checking. Importantly, the type of a sequence is determined by the final expression within it, as outlined in the *T-sequence* rule.

Every block expression in FR_{FT} has a unique lifetime. The latter is essential to statically manage memory without the use of a garbage collector:

$$\frac{\Gamma_1 \vdash \langle e : \tau \rangle^{\mathtt{m}}_{\sigma} \dashv \Gamma_2 \quad \Gamma_2 \vdash \tau \succcurlyeq \mathtt{l} \quad \Gamma_3 = \mathit{drop}(\Gamma_2, \mathtt{m})}{\Gamma_1 \vdash \langle \{e\}^{\mathtt{m}} : \tau \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_3} \quad (\textit{T-Block})$$

This rule exploits the lifetime associated with a given block to determine which variables should be dropped while ensuring that there are no *dangling references*. Thus, the premise $\Gamma_2 \vdash \tau \succcurlyeq \mathtt{l}$ ensures that the resulting type τ does not contain a reference or inactive `Trc` pointing to a dropped location. Additionally, $\mathit{drop}(\Gamma, \mathtt{m})$ deallocates variables that have a lifetime `m`, as depicted in Figure 4.3.

1.4.5 Assign and Declare

With *T-Declare*, the creation of a new location (owner) requires that the variable should not exist in the environment. From then on, this rule produces an output environment,

Γ_3, by adding the new variable whose lifetime matches that of the enclosing block as the following:

$$\frac{x \notin dom(\Gamma_1) \quad \Gamma_1 \vdash \langle e : \tau \rangle^{1}_{\sigma} \dashv \Gamma_2 \quad \Gamma_3 = \Gamma_2\,[x \mapsto \langle\, \tau \,\rangle^{1}]}{\Gamma_1 \vdash \langle \texttt{let mut}\ x = e : \epsilon \rangle^{1}_{\sigma} \dashv \Gamma_3} \quad (\textit{T-Declare})$$

To assign a value to a given lval ω, the *T-Assign* rule is introduced as follows:

$$\frac{\begin{array}{c}\Gamma_1 \vdash \omega : \langle \tilde{\tau}_1 \rangle^{\mathtt{m}} \quad \Gamma_1 \vdash \langle e : \tau_2 \rangle^{1}_{\sigma} \dashv \Gamma_2 \quad \Gamma_2 \vdash \tilde{\tau}_1 \approx \tau_2 \\ \Gamma_2 \vdash \tau_2 \trianglerighteq \mathtt{m} \quad \Gamma_3 = \mathit{write}^{0}(\Gamma_2, \omega, \tau_2) \\ \neg \mathit{writeProhibited}(\Gamma_3, \omega) \quad \neg \mathit{TrcMoveProhibited}(\Gamma_3, \omega)\end{array}}{\Gamma_1 \vdash \langle \omega = e : \epsilon \rangle^{1}_{\sigma} \dashv \Gamma_3} \quad (\textit{T-Assign})$$

The *T-Assign* rule is subject to several conditions: (1) ω must not be borrowed or cloned. This is verified using the $\neg\, \mathit{writeProhibited}(\Gamma_3, \omega)$ and $\neg\, \mathit{TrcMoveProhibited}(\Gamma_3, \omega)$ functions to prevent unchecked mutation in the presence of aliasing. (2) $\tilde{\tau}_1$ and τ_2 must be *compatible type* ($\Gamma_2 \vdash \tilde{\tau}_1 \approx \tau_2$ according to Definition 4.4). (3) The new type τ_2 has to be *well-formed* with respect to lifetime $\mathtt{m}$ (i.e. $\tau_2 \trianglerighteq \mathtt{m}$ as a subtype requirement) as described in Definition 4.5. Furthermore, in this case, ω can have a partial type. The following example outlines point (3) mentioned above:

$$\{\texttt{let mut}\ x = \texttt{trc}(0);\ \texttt{let mut}\ y = \&\, x; \{\texttt{let mut}\ z = \texttt{box}(\texttt{trc}(0));\ y = \&^{*} z\}^{\mathtt{m}}\}^{\mathtt{l}}$$

This example is rejected by the FR_{FT} type system. Upon closer inspection, we see that this example creates a dangling pointer since the lifetime of $\&\, x$ is greater than that of $\&^{*} z$ (i.e. $\mathtt{l} \ngeq \mathtt{m}$). Therefore, this example must be rejected.

The arithmetic operations in FR_{FT} are straightforward since we only have integer values so far. The T-Arithm rule handles an arithmetic expression as follows:

$$\frac{\Gamma_1 \vdash \langle e_1 : \texttt{int} \rangle^{1}_{\sigma} \dashv \Gamma_2 \quad \Gamma_2 \vdash \langle e_2 : \texttt{int} \rangle^{1}_{\sigma} \dashv \Gamma_3}{\Gamma_1 \vdash \langle e_1 \oplus e_2 : \texttt{int} \rangle^{1}_{\sigma} \dashv \Gamma_3} \quad (\textit{T-Arithm})$$

The T-Arithm rule requires the operands to have an integer type. For completeness, a typing rule for conditional operations (e.g. equality comparator) is necessary as well, especially for control flow extension :

$$\frac{\begin{array}{c}\Gamma_1 \vdash \langle e_1 : \tau_1 \rangle^{1}_{\sigma} \dashv \Gamma_2 \quad \Gamma_2[\gamma \mapsto \langle \tau_1 \rangle^{1}] \vdash \langle e_2 : \tau_2 \rangle^{1}_{\sigma} \dashv \Gamma_3 \\ \Gamma_4 = \Gamma_3 \setminus \{\gamma \mapsto \langle \tau_1 \rangle^{1}\} \quad \Gamma_4 \vdash \tau_1 \approx \tau_2 \quad \mathit{copy}(\tau_1) \quad \mathit{copy}(\tau_2) \quad \gamma \in \mathit{fresh}\end{array}}{\Gamma_1 \vdash \langle e_1 \otimes e_2 : \texttt{bool} \rangle^{1}_{\sigma} \dashv \Gamma_4} \quad (\textit{T-Cond})$$

The *T-Cond* rule is more complex than the *T-Arithm* rule. Namely, when typing the left operand, we need to keep in mind its type in order to enforce the ownership invariants, especially for borrowing. In the following example $\{\texttt{let mut}\ x = 0;\ \texttt{let mut}\ y = 1;\ \texttt{if}\ (\&\, x! = \&\texttt{mut}\ x)\{x = y\}^{\mathtt{n}}\texttt{else}\{\ldots\}^{\mathtt{m}}\}^{\mathtt{l}}$, when typing the control flow, we create a new *fresh* variable for the left operand. However, this example is rejected by FR_{FT} because it violates the ownership invariant by creating two references to *x*, where one is immutable and the other mutable in the conditional part. Hence, to protect this ownership, the type of the

left operand is added in the typing environment Γ_2 via γ. In addition, Rust applies the *deref coercion* approach in different cases. For example, $\{\texttt{let mut } x \texttt{ = } 0\texttt{; let mut } y \texttt{ = } 1\texttt{; if (\&mut } x \texttt{ == \&mut } y\texttt{)}\{x \texttt{ = } y\texttt{;}\}^{\texttt{n}}\texttt{else}\{...\}^{\texttt{m}}\}^{\texttt{l}}$ here Rust applies *deref coercion* to both operands to compare the contents of these operands instead of comparing the two references themselves. Similarly, Rust applies the *deref coercion* for `box` types. Otherwise, as in FR, this example is rejected by the FR_{FT} type system as the concept of *deref coercion* does not apply. This is achieved by ensuring that the left and right operands are not copied using the *copy* function. Finally, the resulting typing environment, Γ_4, does not contain the type of the left or right operand, which means that once the comparison is performed, the values are dropped.

1.5 The Typing Function in FR_{FT}

FR_{FT} supports function extension, as explained in the previous chapter. There are limitations imposed by our type system since the invoked function can only be executed within the `spawn` expression. One of these limitations is that a reference cannot be used as a parameter because the lifetime of its content can be shorter than the lifetime of the thread that is created. These restrictions are manifested in the form of the signature that a FR_{FT} function can have, as depicted in Figure 3.5. For clarity, a signature with a form compatible with the inactive `Trc` type or a reference is not allowed. Note that in the next chapter, we will introduce the call function to FR_{FT}, extending its usage beyond the `spawn` expression and considering all the different conditions that arise from these extensions.

1.5.1 Establishing the Connection between Signatures and Types in FR_{FT}

The shape of the signatures in FR_{FT} is similar to that of the types, as illustrated in Figure 3.5. In the next chapter, we will introduce signatures that are not similar to types. In this case, we need a mechanism to establish a correspondence between the signature and the type when a function is declared or invoked [73]. We establish this correspondence as follows: $\Gamma \vdash (\overline{\texttt{S}}) \Longrightarrow (\overline{\tau})$. The concept behind the latter is to map the signature `S` of the declaration function to the typing environment. Alternatively, we ignore the return type in this mechanism since the function call inside the `spawn` expression does not return a value. Afterwards, it remains to present the typing rule dedicated to function declaration in FR_{FT} as follows:

$$\frac{\Gamma \vdash (\overline{\texttt{S}}) \Longrightarrow (\overline{\tau}) \quad \Gamma_1 = \Gamma[\overline{x \mapsto \langle \tau \rangle^{\texttt{m}}}] \qquad \Gamma_1 \vdash \langle \{\overline{e}\}^{\texttt{m}} : \epsilon \rangle^{\texttt{l}}_{\sigma} \dashv \Gamma_2}{\Gamma \vdash \langle \texttt{fn } f(\overline{\texttt{mut } x : \texttt{S}})\{\overline{e}\}^{\texttt{m}} : \epsilon \rangle^{\texttt{l}}_{\sigma} \dashv \Gamma} \quad (\textit{T-Function})$$

As demonstrated in the *T-Function* rule, the mechanism $\Gamma \vdash (\overline{\texttt{S}}) \Longrightarrow (\overline{\tau})$ is used to obtain an appropriate type $\overline{\tau}$. Thus, we declare the parameters in Γ with their suitable type, which produces Γ_1. Moreover, as the *T-Function* depicts, we type the body of f.

To illustrate the typing of an FR_{FT} program more comprehensively, we provide the fol-

lowing example:

$$\begin{gathered}\texttt{fn}\ f_1(\texttt{mut}\ x : \blacklozenge\texttt{int}, \texttt{mut}\ y : \blacksquare \blacklozenge\ \texttt{int})\{...\}^{1} \\ \{\texttt{let mut}\ x = \texttt{trc}(0);\ \texttt{let mut}\ y = \texttt{trc}(0);\ \texttt{spawn}(f_1(x.clone, \texttt{box}(y.clone)))\}^{\texttt{m}}\end{gathered} \tag{4.16}$$

We begin by typing the declared function f_1. At this stage, the *T-Function* rule comes into play, utilizing the mechanism: $\Gamma \vdash (\overline{\texttt{S}}) \Longrightarrow (\overline{\tau})$. Consequently, after mapping signatures to types, the typing environment Γ of f_1 is as follows: $\Gamma = [x \mapsto \langle \blacklozenge\texttt{int} \rangle^{1}, y \mapsto \langle \blacksquare \blacklozenge\ \texttt{int} \rangle^{1}]$, corresponding to the result of the mechanism: $\Gamma_1 \vdash (\blacklozenge\texttt{int}, \blacksquare \blacklozenge\ \texttt{int}) \Longrightarrow (\blacklozenge\texttt{int}, \blacksquare \blacklozenge\ \texttt{int})$. Subsequently, we can proceed to safely type the body of the function, and so on.

1.5.2 Function Invocation and Typing

In FR_{FT}, according to Figure 3.5, a potential case of the reduction function arises when a function is called within the `spawn` expression. In this section, we explore how the FR_{FT} type system manages the typing of function invocations while ensuring strong typing. Similar to function declaration, for function call typing, we also employ a mechanism to pass types from the typing environments to the function signatures. This mechanism is achieved by using the previous mechanism, but in a slightly different manner, where types are passed to signatures, such as: $\Gamma \vdash (\overline{\texttt{S}}) \Longleftarrow (\overline{\tau})$. It is worth noting that in this mechanism, we can observe that the execution of the function call has no effects on the typing environment Γ, thus leaving Γ unchanged. In essence, function invocation typing encompasses both the typing of arguments and the aforementioned mechanism, which enforces constraint resolution to allow the *two-way* flow of information: from types to signatures and from signatures to types.

Function Invocation inside the `spawn` expression

We consider a scenario where the function call is reduced within the `spawn` expression. As a consequence of the general idea of the `Trc` extension, the inactive `Trc` becomes active in the new thread environment. This raises the question of how to formally maintain this activation during the evaluation of the `spawn` expression. To address this, we must retrieve the argument types ($\overline{\tau}$) and activate all inactive `Trc` types accordingly. For this purpose, we introduce the following definition, which facilitates the transition from the inactive `Trc` type to the corresponding active `Trc`:

Definition 4.14 (Activate) *Let Γ be an environment and let τ be a type. Then, activate(Γ, τ) function is responsible for returning a suitable type for τ such that the type is safely activated when requested as follows:*

$$\begin{aligned} \mathit{activate}(\Gamma, \blacklozenge\omega) &= \tau_\omega \ \mathit{for\ some}\ \omega \in \mathit{dom}(\Gamma)\ \mathit{such\ as}\ \Gamma \vdash \omega : \langle \tau_\omega \rangle^{1} \\ \mathit{activate}(\Gamma, \blacksquare\tau) &= \blacksquare \mathit{activate}(\Gamma, \tau) \\ \mathit{activate}(\Gamma, \tau) &= \tau\ \mathit{otherwise} \end{aligned}$$

The Definition 4.14 assumes a typing environment Γ and a type τ. Then, if τ is an inactive `Trc`, it is necessarily of the form $\diamond\omega$. Therefore, we only need to retrieve the type of ω in Γ. Otherwise, if it is of type box $\blacksquare\tau$, *activate* function is recursively called to activate the type if needed (e.g. $\tau \approx \blacksquare \diamond \omega$). Note that, *activate* function does not support reference types or active `Trc`'s since we cannot have references in the arguments (i.e. due to the lifetime of references). Additionally, it is forbidden to have active `Trc`'s in the arguments. To shed light on the performance of the above functions, let's consider the following example written in FR_{FT} according to Figure 3.5 :

$$\begin{gathered}\texttt{fn}\ f_1(\texttt{mut}\ r : \blacklozenge\texttt{int},\ \texttt{mut}\ q : \blacksquare \blacklozenge\ \texttt{int})\{..\}^{1} \\ \{\texttt{let mut}\ a = \texttt{trc(0)};\ \texttt{let mut}\ b = \texttt{trc(0)};\ \texttt{let mut}\ x = a.clone; \\ \texttt{let mut}\ z = \texttt{box}(b.clone);\ \texttt{spawn}(f_1(x,\ z))\}^{\texttt{m}}\end{gathered} \tag{4.17}$$

Example 4.17 demonstrates a function f_1 that takes two parameters: (1) an active `Trc` type and (2) a `box` to an active `Trc` type. In the body of the program, we create an active `Trc` of type `integer` and bind it to *a*. It is similar to the *b* variable. Then we clone the reference into *a* and link it to *x*. Here, *x* is of type $\diamond a$. Same as *z* variable, except that we encapsulate the `Trc` in a `box` type ($\blacksquare \diamond b$). Thus far, our input environment Γ is: $\Gamma = [a \mapsto \langle \blacklozenge\texttt{int}\rangle^{\texttt{m}}, b \mapsto \langle \blacklozenge\texttt{int}\rangle^{\texttt{m}}, x \mapsto \langle \diamond a\rangle^{\texttt{m}}, z \mapsto \langle \blacksquare \diamond b\rangle^{\texttt{m}}]$. Then, when typing the `spawn` expression, we need to apply the *activate* function to activate the necessary type as follows:

$$\begin{gathered}\textit{activate}(\Gamma, \diamond a),\ \textit{activate}(\Gamma, \blacksquare \diamond b) \\ \tau_a \sqcup \blacksquare\textit{activate}(\Gamma, \diamond b) \\ \Gamma(a),\ \blacksquare\tau_b \\ \Gamma(a),\ \blacksquare\Gamma(b) \\ \overline{\tau'} = \blacklozenge\texttt{int}, \blacksquare \blacklozenge\ \texttt{int}\end{gathered}$$

As a result, the new sequence of argument types corresponding to the new thread is as follows: $\overline{\tau}' = \blacklozenge\texttt{int}, \blacksquare \blacklozenge\ \texttt{int}$ (i.e. $\overline{\tau} = \diamond a, \blacksquare \diamond b$).

For a given invocation function, it is required to verify if the types of these arguments correspond to the signatures of its parameters. Therefore we introduce the following definition:

Definition 4.15 (Signature and type compatibility) *For an environment Γ, a signature* `S` *and a type τ are said to be compatible, denoted as $\Gamma \vdash S \sim \tau$ according to the following rules:*

$$\frac{}{\Gamma \vdash \epsilon \sim \epsilon}\ \textit{C-Unit} \qquad \frac{}{\Gamma \vdash \texttt{int} \sim \texttt{int}}\ \textit{C-Int} \qquad \frac{}{\Gamma \vdash \texttt{bool} \sim \texttt{bool}}\ \textit{C-Bool}$$

$$\frac{\Gamma \vdash \texttt{S} \sim \tau}{\Gamma \vdash \blacksquare\texttt{S} \sim \blacksquare\tau}\ \textit{C-Box} \qquad \frac{\forall_i(\Gamma \vdash \texttt{S} \sim \tau \wedge \Gamma(\omega_i) = \blacklozenge\tau)}{\Gamma \vdash \blacklozenge\texttt{S} \sim \diamond\overline{\omega}}\ (\textbf{C-Trc})$$

As shown above, an active `Trc` signature is compatible with an inactive `Trc` type. However, the situation does not hold for an active (inactive, respectively) `Trc` signature being compatible with an active or (inactive, respectively) `Trc` type. The rationale behind this is that an active `Trc` serves as the entry point to the shared part of the heap, and the thread must not lose it. Furthermore, it should be emphasized that the case of references does not exist due to the limited lifetime of the reference.

Definition 4.16 (Argument Typing) *Let Γ and Γ' be environments, σ a store typing and* `l` *a lifetime. Let $\overline{e}$ be a sequence of zeros or more expressions and $\overline{\tau}$ a matching sequence of types. Typing the arguments over $\overline{e}$ results in a left-to-right typing of the expressions, denoted $\Gamma \vdash \langle \overline{e : \tau} \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma'$, and defined as $\Gamma' = \Gamma_n - \{\overline{\gamma - \Gamma_n(\gamma)}\}$ where $\Gamma_n = \mathtt{typeArg}^0(\Gamma, \overline{e})$:*

$$\mathtt{typeArg}^i(\Gamma_i, \varnothing) = \Gamma_i$$
$$\mathtt{typeArg}^i(\Gamma_i, e_i\ \overline{e}) = \mathtt{typeArg}^{i+1}(\Gamma_i'[\gamma_i \mapsto \langle \tau_i \rangle^{\mathtt{l}}], \overline{e}) \quad \textit{s.t. } \Gamma_i \vdash \langle e_i : \tau_i \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_i'$$

Here, γ represents a *freshly* introduced variable when typing the expression $\overline{e}$, as illustrated in definition 4.16. The purpose of γ is to maintain the constraints of the mutable reference and the inactive `Trc` while typing the function arguments.

Before presenting the specific typing rule for the `spawn` expression, let us first explain the necessary constraints that need to be checked during typing. The typing of the function call is limited to typing the arguments (definition 4.16 and checking the restrictions imposed by the mechanism: $\Gamma \vdash (\overline{\mathtt{S}}) \Longleftarrow (\overline{\tau})$. The imposed constraints are the following:

- The arguments must be *well-typed* in the typing environment.
- As discussed in section 1.5.2, inactive `Trc` types become active in the new thread environment. Hence, it is essential to ensure the uniqueness of `Trc` by guaranteeing that no two inactive `Trc` types point to the same location in memory. For example, if we consider the same example 4.17 and modify the `spawn` expression to $\mathtt{spawn}(f_1(x, box(a.clone)))$, it breaks the ownership invariant as when the inactive `Trc` is activated, the new thread has two active `Trc`s pointing to the same location in memory.
- With respect to active `Trc` types, we prevent an argument from having a type that contains an active `Trc`. The reason being that an active `Trc` serves as the primary access point to a location, and the thread must not lose it when passing it to another thread.
- Likewise, since the lifetime of a thread is static in FR_{FT}, we avoid arguments containing references in their type (i.e., the lifetime of a borrowing is shorter than that of a thread).
- It is necessary to ensure compatibility between the signatures of the function's parameters and the types of the function's arguments before activating the inactive `Trc` types. Additionally, the absence of references in the function call prevents the possibility of producing side effects in the output environment, hence Γ remains unchanged.

Now, let's consider the *T-Spawn* rule to handle the creation of a new thread that executes the function given in parameters, as follows:

$$\frac{\mathcal{D}(f) = (\overline{\mathtt{S}}) \quad \Gamma_1 \vdash \langle \overline{e : \tau} \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2 \qquad \Gamma_2 \vdash (\overline{\mathtt{S}}) \Longleftarrow (\overline{\tau})}{\Gamma_1 \vdash \langle \mathtt{spawn}(f(\overline{e})) : \epsilon \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2} \quad (\textit{T-Spawn})$$

As demonstrated by the *T-Spawn* rule, the invoked function does not have a return type (ϵ). The latter is limited to: (1) typing the arguments of the function f (definition 4.16), (2) retrieving the signature of the function f from the declaration context $\mathcal{D}$, and (3) enforcing two necessary constraints dictated by the mechanism ($\Gamma_2 \vdash (\overline{\mathtt{S}}) \Longleftarrow (\overline{\tau})$):

- It is important to ensure *uniqueness* of `Trc` by guaranteeing that there are no two inactive `Trc` types pointing to the same location in the memory. Knowing that, inactive `Trc` types become active in the new thread environment. In other words, for all $\tau_1, \tau_2 \in \overline{\tau}$ if *contains*$(\Gamma, \tau_1, \diamond\overline{\omega})$ and *contains*$(\Gamma, \tau_2, \diamond\overline{u})$, then $\neg\exists_{i,j}.(\omega_i \bowtie u_j)$.
- It is necessary to ensure the compatibility between signatures and types, denoted $\overline{\Gamma_2 \vdash S \sim \tau}$, by using the *C-** rules (definition 4.15). This compatibility prevents the presence of an active `Trc` or a reference in the argument types. After verifying the necessary constraints, the mechanism is able to activate the inactive `Trc` type in $\overline{\tau}$.

The main objective of this mechanism is to prevent the following scenarios: (1) having two inactive `Trc`s pointing to the same memory location, for instance, $\Gamma \vdash (\blacklozenge\mathtt{int}, \blacklozenge\mathtt{int}) \Longleftarrow\!\!\!\!\!\!/\;\; (\diamond x, \diamond x)$; and (2) having an active `Trc` as one of the function arguments, for example, $\Gamma \vdash (\blacklozenge\mathtt{int}) \Longleftarrow\!\!\!\!\!\!/\;\; (\blacklozenge\mathtt{int})$. To illustrate these constraints, let's consider a simple example that is rejected by the *FR*$_{FT}$ type system:

$$\begin{array}{c} \mathtt{fn}\ f_1(\mathtt{mut}\ x : \blacklozenge\mathtt{int},\ \mathtt{mut}\ y : \blacklozenge\mathtt{int})\{\ldots\}^{\mathtt{l}} \\ \{\mathtt{let\ mut}\ x = \mathtt{trc}(0);\ \mathtt{spawn}(f_1(x.clone, x.clone))\}^{\mathtt{m}} \end{array} \tag{4.18}$$

Example 4.18 is not permitted since the invoked function in the `spawn` expression has two inactive `Trc` arguments as parameters pointing to the same memory location. As the inactive `Trc`'s become active in the new thread's environment, the *uniqueness* property of the `Trc` extension (i.e. uniqueness of mutable location) must be preserved by preventing two active `Trc`'s pointing to the same memory location. Accordingly, we achieve an unsatisfactory constraint by the *T-Spawn* rule mechanism as follows: $\Gamma_1 \vdash (\blacklozenge\mathtt{int}, \blacklozenge\mathtt{int}) \Longleftarrow\!\!\!\!\!\!/\;\; (\diamond x, \diamond x)$. However, the following example is accepted by the *FR*$_{FT}$ type system:

$$\begin{array}{c} \mathtt{fn}\ f_1(\mathtt{mut}\ x : \blacklozenge\mathtt{int},\ \mathtt{mut}\ y : \blacklozenge\mathtt{int})\{\ldots\}^{\mathtt{l}} \\ \{\mathtt{let\ mut}\ x = \mathtt{trc}(0);\ \mathtt{let\ mut}\ y = \mathtt{box}(\mathtt{trc}(0));\ \\ \mathtt{spawn}(f_1(x.clone, {}^*y.clone))\}^{\mathtt{m}} \end{array} \tag{4.19}$$

Example 4.19 is permitted since the invoked function in the `spawn` expression has two inactive `Trc` arguments as parameters not pointing to the same memory location. As the inactive `Trc`'s become active in the new thread's environment, the *uniqueness* property of the `Trc` extension is preserved after the inactive `Trc` type is activated. Accordingly, we achieve a satisfactory constraint by the *T-Spawn* rule mechanism as follows: $\Gamma_1 \vdash (\blacklozenge\mathtt{int}, \blacklozenge\mathtt{int}) \Longleftarrow (\diamond x, \diamond {}^*y)$.

1.6 Cooperative Typing Rules for *FR*$_{FT}$

Up to this point, we have presented the *FR*$_{FT}$ type system for non-reactive expressions. In this section, we introduce the necessary typing rules for reactive expressions. The main

objective of the typing system presented in this section is to ensure that the synchronization and cooperation between threads do not compromise the safety of the threads' memory.

Our focus in this chapter is limited to the cooperative part of the FR_{FT} language, which includes only the `cooperate` expression as shown in Figure 3.5. Therefore, we introduce the typing rule to handle the `cooperate` expression. The `cooperate` expression signifies that the current thread has completed its execution at the current instant, allowing another thread to assume control. This implies that the other thread can share data with the current thread via the `Trc` extension. However, the `Trc` extension can potentially lead to memory safety issues, such as creating a dangling pointer (as illustrated in the example in Figure 4.1). Thus, it becomes crucial to safeguard shared data between threads, especially during cooperation. The *T-Cooperate* rule is introduced to serve this purpose:

$$\frac{\mathit{safeTrc}\,(\Gamma)}{\Gamma \vdash \langle \texttt{cooperate} : \epsilon \rangle^{1}_{\sigma} \dashv \Gamma} \qquad (\textit{T-Cooperate})$$

To maintain memory safety and prevent potential errors, it is crucial to ensure that shared data (i.e., the content of a `Trc`) is not borrowed in the typing environment of the current thread during cooperation. To achieve this, the *safeTrc*(Γ) function, introduced in definition 4.5, is utilised to verify this condition.

2 Soundness of FR_{FT}

In this section we present the soundness of the FR_{FT} type system that we introduced in Section 1 using the syntactic approach of Wright and Felleisen [48]. Accordingly, the soundness is a consequence of two lemmas: Progress and Preservation. Such results allow us to argue that a type-safe, borrowing-safe and concurrency-safe synchronous reactive program does not deadlock while it preserves borrow and `Trc` invariants. Furthermore, we can guarantee that at any given point in the program:

- References always point to valid locations in the program store, avoiding dangling pointers. This ensures that references are valid and do not point to non-existent or deallocated memory.
- The uniqueness of the active `Trc`'s in each thread's environment is maintained.
- References to shared data (active `Trc`) do not exist when a thread cooperates. This precautionary measure prevents any potential data corruption or inconsistencies from occurring.

The Progress lemma is typically responsible for ensuring that a *well-typed* expression in a program is either a value or a reducible expression. Meanwhile, the Preservation lemma guarantees that applying a semantic rule to a *well-typed* expression will result in a *well-typed* evaluation step of the same type. However, with our type system, which is similar to Rust, we inherit the sensitivity of the Borrow checker. Therefore, this sensitivity needs to be taken into account in these lemmas. In the case of an FR_{FT} program, the program's store is shared between different threads and at each point of the program. On top of that, it is crucial to guarantee that the execution of one thread does not violate the well-typed and memory-safe state of other threads at any point in the program. Also, an FR_{FT} program is

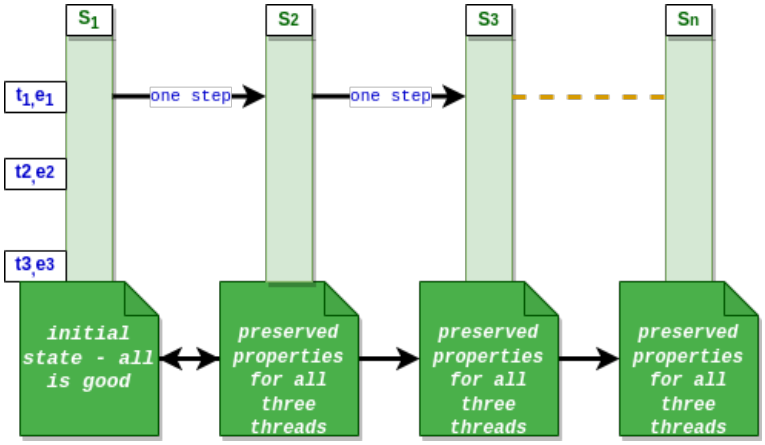

Figure 4.6: Step Preservation

divided into logical instants, and it must be ensured that a well-typed FR_{FT} program never gets stuck and progresses through one or more instants. Consequently, we decompose our proof to take into account the reactive nature of the language, as follows: First, we prove progress (4.9) and preservation (4.10) lemmas for individual execution steps. The main challenge, compared to FR [73], is to demonstrate that the reduction steps not only preserve the typing of the current thread but also do not affect the typing of other threads. Second, we prove progress (4.14) and preservation (4.15) for maximal thread executions (referred to as "slices"). Finally, we establish progress (4.16) and preservation (4.17) of an instant. Several additional definitions are necessary to introduce these lemmas, from which it follows that our type system is sound. Hence, for more clarity, according to Figure 4.6, after an execution (a step), the thread must preserve its typing. Furthermore, to ensure that the program remains *well-typed*, the thread must not modify the typing of the other threads.

The notion of the program store $S_{|\mathtt{l}}$ refers to S which is restricted to the lifetime `l`. More precisely, $S_{|\mathtt{l}}$ only maps the locations of the variables having the lifetime `l` (or less than `l`, $\forall \mathtt{n},\ \mathtt{l} \geqslant \mathtt{n}$) to their partial values and all other locations accessible from these values.

2.1 Local and Global Valid States

In essence, we are required to ensure that our state is always valid. The validity here is related to the notion of ownership. To better understand, we present an unsatisfied example in Figure **??**. The example in Figure **??** describes a program in FR_{FT} with two threads (Thread 1 and Thread 2). As we observe, the state of this program is considered invalid because: (1) between the expressions and the program store S of this program, there are two owning box references to ℓ_a. Furthermore, (2) from these two threads, we can access ℓ_a via two different paths, one through a `Trc` and the other outside the `Trc`. Hence, we must ensure that if there exists a path to a location ℓ_a in S, then there cannot be another one. For example, for a location (ℓ_a), either it is reachable by a single thread and there is thus only one path in S, or it is shared between several threads within a `Trc`. For this reason, we introduce the following definition:

Definition 4.17 (Unique Location) *Let S be a program store and let ℓ_a, ℓ_b be locations $\in S$. We note $S \vdash \ell_a \rightsquigarrow \ell_b$ if either $S(\ell_a) = \ell_b$ or there exists $\ell_c \in S$ such that $S(\ell_a) = \ell_c$ and $S \vdash \ell_c \rightsquigarrow \ell_b$. Then, ℓ_a is said to be a unique location to access ℓ_b, denoted $S \vdash_{\mathtt{u}} \ell_a \rightsquigarrow \ell_b$, if $S \vdash \ell_a \rightsquigarrow \ell_b$ and if there exists $\ell_c \in S$ such that $\ell_c \neq \ell_a$ and $S \vdash \ell_c \rightsquigarrow \ell_b$ then, either $S \vdash \ell_c \rightsquigarrow \ell_a$ or $S \vdash \ell_a \rightsquigarrow \ell_c$.*

In order to ensure the validity of the global state T, S, we must first guarantee the validity of the expressions e, then the validity of the program store S and lastly the validity of the local state $S \triangleright e$ as follows:

An expression e is considered a *well-typed* expression if and only if: (1) it does not contain distinct owning box references to the same location (e.g. $\{\mathtt{let\ mut}\ x = \ell_a^{\blacksquare};\ \mathtt{let\ mut}\ y = \ell_a^{\blacksquare}\}^{\mathtt{l}}$), and as well (2) it does not contain distinct owning `Trc` references to the same location (e.g. $\{\mathtt{let\ mut}\ x = \ell_a^{\blacklozenge};\ \mathtt{let\ mut}\ y = \ell_a^{\blacklozenge}\}^{\mathtt{l}}$):

Definition 4.18 (Valid expression) *Let e be an expression. Let $\overline{v}$ be the sequence of distinct values in e. Then e is said to be valid when $\neg\exists_{i,j}.(i \neq j \wedge \exists_{\ell_a^{\blacklozenge}}.(v_i = v_j = \ell_a^{\blacklozenge}))$ and $\neg\exists_{i,j}.(i \neq j \wedge \exists_{\ell_b^{\blacksquare}}.(v_i = v_j = \ell_b^{\blacksquare}))$.*

According to definition 4.18, a program store S is considered to be valid for some lifetime `l` if and only if: (1) it does not contain distinct owning box references to the same location (e.g. $S = \{\ell_a \mapsto \langle 0\rangle^*, \ell_b \mapsto \langle \ell_a^{\blacksquare}\rangle^*, \ell_c \mapsto \langle \ell_a^{\blacksquare}\rangle^*\}$), and (2) it does not contain distinct owning `Trc` references to the same location (e.g. $S = \{\ell_a \mapsto \langle 0\rangle^2, \ell_b \mapsto \langle \ell_a^{\blacklozenge}\rangle^1, \ell_c \mapsto \langle \ell_a^{\blacklozenge}\rangle^1, \ldots\}$). Moreover, if there are two owning box references to the same location, they must necessarily be under a `Trc` and among different threads:

Definition 4.19 (Valid Store) *Let S be a program store and let `l` be a lifetime. Let $\overline{v}$ be the sequence of distinct values in $S_{|\mathtt{l}}$. Then, S is said to be valid for `l` when $\neg\exists_{i,j}.(i \neq j \wedge \exists_{\ell_a^{\blacklozenge}}.(v_i = v_j = \ell_a^{\blacklozenge}))$ and $\neg\exists_{i,j}.(i \neq j \wedge \exists_{\ell_b^{\blacksquare}}.(v_i = v_j = \ell_b^{\blacksquare}))$.*

After establishing the validity of an expression e and a program store S for some lifetime `l`, we now define the validity of a local state $S \triangleright e$ as follows:

Definition 4.20 (Valid Local State) *Let $S \triangleright e$ be a local state and let `l` be a lifetime such that e is valid and S is valid for `l`. Let $\overline{v}$ be the sequence of distinct values in e and let $\overline{u}$ be the sequence of distinct values in $S_{|\mathtt{l}}$. Then, $S \triangleright e$ is said to be valid for `l` when $\neg\exists_{i,j}.(\exists_{\ell_a^{\blacklozenge}}.(v_i = u_j = \ell_a^{\blacklozenge}))$ and $\neg\exists_{i,j}.(\exists_{\ell_b^{\blacksquare}}.(v_i = u_j = \ell_b^{\blacksquare}))$.*

Finally, after introducing definition 4.20, we can derive the validity of the global state as follows:

Definition 4.21 (Valid Global State) *Let $T = \{t_i, \{e_i\}^{\mathtt{l_i}} \mid 1 \leqslant i \leqslant N\}$ be a set of threads, let S be a program store such that for all $i \in \{1, \ldots, N\}$, $S \triangleright e_i$ is a valid local state for $\mathtt{l_i}$.*

For some $i, j \in \{1, ..., N\}$ where $i \neq j$, let $\overline{v}$ be the sequence of distinct values in $S_{|\mathtt{l}_i}$ and e_i, and let $\overline{u}$ be the sequence of distinct values in $S_{|\mathtt{l}_j}$ and e_j. Then T, S is said to be valid if for all $\ell_a^{\blacksquare}$ such that $\ell_a^{\blacksquare} = v_i = u_j$, there exists a unique $\ell_b^{\blacklozenge}$ such that $S \vdash_u \ell_b \rightsquigarrow \ell_a$.

The progress lemma indicates that if a thread $(t, \{e\}^{\mathtt{l}}) \in T$ of an *FR*$_{FT}$ program starts from a valid global state and its expression e is *well-typed*, then it is guaranteed that the thread will eventually reduce to either a value or another expression. To ensure the safety of this progress and the connection between the program store S and the typing environment Γ, the following requirements must be satisfied:

- *Existence of locations in S*: the memory locations associated with the variables must exist in the program store S. The latter maintains the correspondence between the memory locations and their values.
- *Type-value match*: variable types defined in Γ must match their runtime values in S. This ensures that the program is consistent and that variables are assigned values of the correct type at runtime.

In the following section, we introduce the definitions needed to establish this safety connection.

2.2 Maintaining Safe Abstraction

As previously mentioned, a safe connection between the program store S and the typing environment Γ must be maintained. To illustrate this concern, let us go through an example: consider a program store S represented by $\{\ell_{\mathtt{l}::x} \mapsto \langle \ell_a^{\blacklozenge} \rangle^{\mathtt{l}}, \ell_a \mapsto \langle 0 \rangle^{\mathtt{l}}, \ell_{\mathtt{l}::y} \mapsto \langle \ell_{\mathtt{l}::x}^{\circ} \rangle^{\mathtt{l}}\}$, accompanied by a corresponding typing environment Γ denoted as: $\{x \mapsto \langle \blacklozenge \mathtt{int} \rangle^{\mathtt{l}}\}$. In this scenario, S does not serve as a safe abstraction of Γ due to the presence of the variable y in S with a runtime value, which currently does not exist in Γ. Now, let us explore another instance where an unsatisfied condition arises with the same program store S, but with a distinct typing environment Γ, defined as $\Gamma = \{x \mapsto \langle \blacksquare \mathtt{int} \rangle^{\mathtt{l}}, y \mapsto \langle \&\mathtt{mut}\ x \rangle^{\mathtt{l}}\}$. Once again, S fails to be a safe abstraction of Γ as the type of the variable x in S does not align with its runtime value. To establish a safe abstraction connection between S and Γ, we introduce the following definitions:

Definition 4.22 (Valid Value Type) *Let S be a program store, $\tilde{\tau}$ a partial type and $v^{\perp}$ a partial value, then $v^{\perp}$ is said to be abstracted by $\tilde{\tau}$ in S, denoted $S \vdash v^{\perp} \sim \tilde{\tau}$, according to Figure 4.7.*

In Section 1 (Chapter 4), we discussed the store typing σ, which is used when typing an expression e containing locations, for example: $\mathtt{let\ mut}\ x = \ell_a^{\blacklozenge}$. At this point, it is worthwhile guaranteeing the validity of σ when typing e as follows:

$$\frac{}{S \vdash \epsilon \sim \epsilon}(\textbf{V-Unit}) \qquad \frac{}{S \vdash c \sim \texttt{int}}(\textbf{V-Int}) \qquad \frac{}{S \vdash b \sim \texttt{bool}}(\textbf{V-Bool})$$

$$\frac{\exists\, i.(loc(S,\omega_i,\texttt{l}) = \ell)}{S \vdash \ell^{\circ} \sim \&[\texttt{mut}]\overline{\omega}}(\textbf{V-Borrow}) \qquad \frac{}{S \vdash \bot \sim \lfloor \tau \rfloor}(\textbf{V-Undef}) \qquad \frac{S(\ell_a) = \langle v^{\bot} \rangle^{*} \quad S \vdash v^{\bot} \sim \tilde{\tau}}{S \vdash \ell_a^{\blacksquare} \sim \blacksquare \tilde{\tau}}(\textbf{V-Box})$$

$$\frac{S(\ell_a) = \langle v \rangle^{\mathrm{i}} \quad S \vdash v \sim \tau}{S \vdash \ell_a^{\blacklozenge} \sim \blacklozenge \tau}(\textbf{V-Trc}) \qquad \frac{\exists\, i.(read(S,\omega_i,\texttt{l}) = \langle \ell_a^{\blacklozenge} \rangle^{\texttt{l}})}{S \vdash \ell_a^{\diamond} \sim \diamond \overline{\omega}}(\textbf{V-Clone})$$

Figure 4.7: Valid value type

Definition 4.23 (Valid Store Typing) *Let S be a program store, σ a store typing and e an expression. Let $\overline{v}$ be the sequence of distinct values in e. Then, σ is said to be valid for $S \triangleright e$, denoted $S \triangleright e \vdash \sigma$, if for all i we have $S \vdash v_i \sim \sigma(v_i)$.*

On top of that and to expand the notion of safe abstraction between S and Γ, we introduce Θ as a function defined as $\Theta(dom(\Gamma)) = \{\ell_{\texttt{m}::x} \mid x \mapsto \langle \tilde{\tau} \rangle^{\texttt{m}} \in \Gamma\}$. Additionally, we designate by $\mathcal{L}$ the set of all heap locations:

Definition 4.24 (Safe Abstraction) *Let Γ be a typing environment and S a valid program store for some lifetime $\texttt{l}$. Then, S is safely abstracted by Γ, denoted $S \sim \Gamma$, iff $(dom(S) \backslash \mathcal{L}) = \Theta(dom(\Gamma))$ and for all $x \in dom(\Gamma)$ such that $\Gamma(x) = \langle \tilde{\tau} \rangle^{\texttt{m}}$, there exists $v^{\bot}$ such that $S \vdash v^{\bot} \sim \tilde{\tau}$ where $S(\ell_{\texttt{m}::x}) = \langle v^{\bot} \rangle^{\texttt{m}}$.*

In chapter 3, the concept of S is elucidated. Within the program store S, there exists not only the information about variable locations and their respective values, but also about the heap locations. In contrast, the typing environment Γ associates variables with their types. To establish a safe abstraction between S and Γ, Definition 4.24 introduces $\mathcal{L}$ which encompasses all locations in the heap. In this context, $(\text{dom}(S) \backslash \mathcal{L})$ pertains to the subset of S that exclusively includes variable locations mapped to their values. Furthermore, for a precise equivalence alignment between the domains of S and Γ, the notation Θ is introduced.

2.3 Ensuring Borrow and `Trc` Invariance

Four important invariants about typing environments are to be observed: first, to avoid the presence of invalid borrowings as reflected below in (1) and (2). Second, to have precisely one mutable reference as shown in (3) (i.e. having one or more immutable references to a memory location or having exactly one mutable reference to this location). Finally to ensure that for every inactive `Trc`, there exists one or more active `Trc` as witnessed in (4). These properties are named *borrow and* `Trc` *invariance* and they are captured in the *well-formedness* property over environments as follows:

Definition 4.25 (Borrow and Trc Invariance) *Let Γ be a typing environment, then Γ is said to be well-formed with respect to some lifetime $\texttt{l}$ if :*

1. *for all* $x \in dom(\Gamma)$ *and* $\overline{\omega} \in Lval^+$ *where* $\Gamma \vdash x \rightsquigarrow \&[\mathtt{mut}]\overline{\omega} \wedge \Gamma(x) = \langle . \rangle^{\mathtt{n}}$, *we have* $\overline{\Gamma \vdash \omega : \langle \tau \rangle^{\mathtt{m}} \wedge \mathtt{m} \geq \mathtt{n}}$.
2. *for all* $x \in dom(\Gamma)$ *where* $\Gamma(x) = \langle . \rangle^{\mathtt{n}}$, *we have* $\mathtt{n} \geq \mathtt{l}$.
3. *for all* $\overline{u}, \overline{\omega} \in Lval^+$ *and for all* $x,\ y \in dom(\Gamma)$ *where* $\Gamma \vdash x \rightsquigarrow \&\mathtt{mut}\,\overline{\omega}$ *and* $\Gamma \vdash y \rightsquigarrow \&[\mathtt{mut}]\overline{u}$ *and* $\exists_{i,j}.(\omega_i \bowtie u_j)$, *we have* $x = y$.
4. *for all* $x \in dom(\Gamma)$ *and* $\overline{\omega} \in Lval^+$ *where* $\Gamma \vdash x \rightsquigarrow \diamond\overline{\omega}$, *we have* $\overline{\Gamma \vdash \omega : \langle \blacklozenge\tau \rangle^{\mathtt{m}} \wedge \mathtt{m} \geq \mathtt{l}}$.

2.4 Lemmas and Proofs

In this section, we introduce some additional lemmas needed to establish our soundness as follows:

2.4.1 Strengthening Type Lemmas

Lemma 4.1 (Safe Strengthening) *Let S be a program store. Let Γ be a well-formed typing environment with respect to a lifetime $\mathtt{l}$ such that $S \sim \Gamma$. Let $\tau_1,\ \tau_2$ be types such that $\tau_1 \sqsubseteq \tau_2$ and let v be a value. If $S \vdash v \sim \tau_1$ then $S \vdash v \sim \tau_2$.*

`Proof.` By induction on the structure of $\tau_1 \sqsubseteq \tau_2$ and according to Figure 4.2:

- *Base Case.* $[\tau_1 \sqsubseteq \tau_1]$. The result is immediate since by hypothesis we have $S \vdash v \sim \tau_1$.
- *Base Case.* $[\&[\mathtt{mut}]\,\overline{u} \sqsubseteq \&[\mathtt{mut}]\,\overline{\omega}]$. By hypothesis we have $\&[\mathtt{mut}]\,\overline{u} \sqsubseteq \&[\mathtt{mut}]\,\overline{\omega}$ and for some $v = \ell^{\circ}$ we have $S \vdash \ell^{\circ} \sim \&[\mathtt{mut}]\,\overline{u}$. By *W-Bor*, we have $\overline{u} \subseteq \overline{\omega}$ and by inspection of Figure 4.7, there exists i such that $loc(S, \omega_i, \mathtt{l}) = \ell$ and $S \vdash \ell^{\circ} \sim \&[\mathtt{mut}]\,\overline{\omega}$ (by *V-Borrow*).
- *Base Case.* $[\diamond\,\overline{u} \sqsubseteq \diamond\,\overline{\omega}]$. By hypothesis we have $\diamond\,\overline{u} \sqsubseteq \diamond\,\overline{\omega}$ and for some $v = \ell^{\diamond}$ we have $S \vdash \ell^{\diamond} \sim \diamond\overline{u}$. By *W-Clone*, we have $\overline{u} \subseteq \overline{\omega}$ and by inspection of Figure 4.7, there exists i such that $read(S, \omega_i, \mathtt{l}) = \langle \ell_a^{\blacklozenge} \rangle^{\mathtt{l}}$ and $S \vdash \ell^{\diamond} \sim \diamond\overline{\omega}$ (by *V-Clone*).
- *Inductive Case.* $[\blacksquare\tau_1 \sqsubseteq \blacksquare\tau_2]$. By hypothesis, $\blacksquare\tau_1 \sqsubseteq \blacksquare\tau_2$ and for some $v = \ell_a^{\blacksquare}$ we have $S \vdash \ell_a^{\blacksquare} \sim \blacksquare\tau_1$. Then, by inspection of definition 4.22, $\ell_a^{\blacksquare}$ is safely abstracted by $\blacksquare\tau_1$ and by *V-Box* (Figure 4.7), there exists v' in S such that $S(\ell_a) = \langle v' \rangle^*$ where $S \vdash v' \sim \tau_1$. By *W-Box*, $\tau_1 \sqsubseteq \tau_2$ and by induction hypothesis we have $S \vdash v' \sim \tau_1$ and $S \vdash v' \sim \tau_2$. Since, $S(\ell_a) = \langle v' \rangle^*$ and $S \vdash v' \sim \tau_2$ then, we can deduce by *V-Box* (Figure 4.7) that $S \vdash \ell_a^{\blacksquare} \sim \blacksquare\tau_2$.
- *Inductive Case.* $[\blacklozenge\tau_1 \sqsubseteq \blacklozenge\tau_2]$. By hypothesis, $\blacklozenge\tau_1 \sqsubseteq \blacklozenge\tau_2$ and for some $v = \ell_a^{\blacklozenge}$ we have $S \vdash \ell_a^{\blacklozenge} \sim \blacklozenge\tau_1$. Then, by inspection of definition 4.22, $\ell_a^{\blacklozenge}$ is safely abstracted by $\blacklozenge\tau_1$ and by *V-Trc* (Figure 4.7), there exists v' in S such that $S(\ell_a) = \langle v' \rangle^{\mathtt{i}}$ where $S \vdash v' \sim \tau_1$. By *W-Trc*, $\tau_1 \sqsubseteq \tau_2$ and by induction hypothesis we have $S \vdash v' \sim \tau_1$

and $S \vdash v' \sim \tau_2$. Since, $S(\ell_a) = \langle v' \rangle^{\mathtt{i}}$ and $S \vdash v' \sim \tau_2$ then, we can deduce by *V-Trc* (Figure 4.7) that $S \vdash \ell_a^{\blacklozenge} \sim \blacklozenge\tau_2$.

□

Lemma 4.2 (Transitive Strengthenning) *Let $\tilde{\tau}_1$, $\tilde{\tau}_2$ and $\tilde{\tau}_3$ be partial types. If $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_2$ and $\tilde{\tau}_2 \sqsubseteq \tilde{\tau}_3$, then $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_3$.*

`Proof.` By structural induction on the structure of $\tilde{\tau}_2$ according to Figure 3.5:

- *Base Case* $\tilde{\tau}_2 \triangleq [\epsilon]$. By hypothesis we have $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_2$ and $\tilde{\tau}_2 \sqsubseteq \tilde{\tau}_3$. By *W-Reflex*, according to Figure 4.2, we have $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_1$. Consequently, we can deduce that $\tilde{\tau}_1 = \epsilon$. If $\tilde{\tau}_3 = \epsilon$, then we can deduce that $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_3$ by *W-Reflex*. Now if $\tilde{\tau}_3 = \lfloor \tau \rfloor$, since $\tilde{\tau}_2 \sqsubseteq \tilde{\tau}_3$ then by *W-UndefB* we have $\tilde{\tau}_2 \sqsubseteq \tau$ and by *W-Reflex* we can deduce that $\tau = \epsilon$ since $\tilde{\tau}_2 = \epsilon$. Additionally, since $\tilde{\tau}_1 = \epsilon$ then $\tilde{\tau}_1 \sqsubseteq \tau$ by *W-Reflex* and $\tilde{\tau}_1 \sqsubseteq \lfloor \tau \rfloor$ by *W-UndefB*. Finally, $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_3$.

- *Base Case* $\tilde{\tau}_2 \triangleq [\mathtt{int}]$. By hypothesis we have $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_2$ and $\tilde{\tau}_2 \sqsubseteq \tilde{\tau}_3$. By *W-Reflex*, according to Figure 4.2, we have $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_1$. Consequently, we can deduce that $\tilde{\tau}_1 = \mathtt{int}$. If $\tilde{\tau}_3 = \mathtt{int}$, then we can deduce that $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_3$ by *W-Reflex*. Now if $\tilde{\tau}_3 = \lfloor \tau \rfloor$, since $\tilde{\tau}_2 \sqsubseteq \tilde{\tau}_3$ then by *W-UndefB* we have $\tilde{\tau}_2 \sqsubseteq \tau$ and by *W-Reflex* we can deduce that $\tau = \mathtt{int}$ since $\tilde{\tau}_2 = \mathtt{int}$. Additionally, since $\tilde{\tau}_1 = \mathtt{int}$ then $\tilde{\tau}_1 \sqsubseteq \tau$ by *W-Reflex* and $\tilde{\tau}_1 \sqsubseteq \lfloor \tau \rfloor$ by *W-UndefB*. Finally, $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_3$.

- *Base Case* $\tilde{\tau}_2 \triangleq [\mathtt{bool}]$. By hypothesis we have $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_2$ and $\tilde{\tau}_2 \sqsubseteq \tilde{\tau}_3$. By *W-Reflex*, according to Figure 4.2, we have $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_1$. Consequently, we can deduce that $\tilde{\tau}_1 = \mathtt{bool}$. If $\tilde{\tau}_3 = \mathtt{bool}$, then we can deduce that $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_3$ by *W-Reflex*. Now if $\tilde{\tau}_3 = \lfloor \tau \rfloor$, since $\tilde{\tau}_2 \sqsubseteq \tilde{\tau}_3$ then by *W-UndefB* we have $\tilde{\tau}_2 \sqsubseteq \tau$ and by *W-Reflex* we can deduce that $\tau = \mathtt{bool}$ since $\tilde{\tau}_2 = \mathtt{bool}$. Additionally, since $\tilde{\tau}_1 = \mathtt{bool}$ then $\tilde{\tau}_1 \sqsubseteq \tau$ by *W-Reflex* and $\tilde{\tau}_1 \sqsubseteq \lfloor \tau \rfloor$ by *W-UndefB*. Finally, $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_3$.

- *Base Case* $\tilde{\tau}_2 \triangleq [\&[\mathtt{mut}]\overline{z}]$. By hypothesis we have $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_2$ and $\tilde{\tau}_2 \sqsubseteq \tilde{\tau}_3$. By *W-Bor*, according to Figure 4.2, we have $\tilde{\tau}_1 = \&[\mathtt{mut}]\overline{u}$ such that $\overline{u} \subseteq \overline{z}$. Moreover, we have $\tilde{\tau}_3 = \&[\mathtt{mut}]\overline{\omega}$ such that $\overline{z} \subseteq \overline{\omega}$. Since, $\overline{u} \subseteq \overline{z}$ and $\overline{z} \subseteq \overline{\omega}$ then we have $\overline{u} \subseteq \overline{\omega}$ by the transitivity of subset relation. Therefore, by *W-Bor* we can deduce that $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_3$. Now, if $\tilde{\tau}_3 = \lfloor \tau \rfloor$ and since $\tilde{\tau}_2 \sqsubseteq \tilde{\tau}_3$ then, $\tilde{\tau}_2 \sqsubseteq \tau$ by *W-UndefB* according to Figure 4.2. If $\tau = \&[\mathtt{mut}]\overline{\omega}$ then we have $\overline{z} \subseteq \overline{\omega}$ and since we have $\overline{u} \subseteq \overline{z}$ hence, $\overline{u} \subseteq \overline{\omega}$. Therefore, by *W-UndefB* we can deduce that $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_3$.

- *Base Case* $\tilde{\tau}_2 \triangleq [\diamond\overline{z}]$. By hypothesis we have $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_2$ and $\tilde{\tau}_2 \sqsubseteq \tilde{\tau}_3$. By *W-Clone*, according to Figure 4.2, we have $\tilde{\tau}_1 = \diamond\overline{u}$ such that $\overline{u} \subseteq \overline{z}$. Moreover, we have $\tilde{\tau}_3 = \diamond\overline{\omega}$ such that $\overline{z} \subseteq \overline{\omega}$. Since, $\overline{u} \subseteq \overline{z}$ and $\overline{z} \subseteq \overline{\omega}$ then we have $\overline{u} \subseteq \overline{\omega}$ by the transitivity of subset relation. Therefore, by *W-Clone* we can deduce that $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_3$. Now, if $\tilde{\tau}_3 = \lfloor \tau \rfloor$ and since $\tilde{\tau}_2 \sqsubseteq \tilde{\tau}_3$ then, $\tilde{\tau}_2 \sqsubseteq \tau$ by *W-UndefB* according to Figure 4.2. If $\tau = \&[\mathtt{mut}]\overline{\omega}$ then we have $\overline{z} \subseteq \overline{\omega}$ and since we have $\overline{u} \subseteq \overline{z}$ hence, $\overline{u} \subseteq \overline{\omega}$. Therefore, by *W-UndefB* we can deduce that $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_3$.

- *Inductive Case* $\tilde{\tau}_2 \triangleq [\blacksquare\tilde{\tau}_a]$. By hypothesis we have $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_2$ and $\tilde{\tau}_2 \sqsubseteq \tilde{\tau}_3$. By *W-Box*, according to Figure 4.2, we have $\tilde{\tau}_1 = \blacksquare\tilde{\tau}_b$ such that $\tilde{\tau}_b \sqsubseteq \tilde{\tau}_a$. Moreover, we have $\tilde{\tau}_3 = \blacksquare\tilde{\tau}_c$ such that $\tilde{\tau}_a \sqsubseteq \tilde{\tau}_c$. By induction hypothesis, we have $\tilde{\tau}_b \sqsubseteq \tilde{\tau}_c$ and therefore we can deduce by *W-Box* that $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_3$. Now, if $\tilde{\tau}_3 = \lfloor\blacksquare\tau_c\rfloor$ and since $\tilde{\tau}_2 \sqsubseteq \tilde{\tau}_3$ hence, $\tilde{\tau}_a \sqsubseteq \lfloor\tilde{\tau}_c\rfloor$ by *W-UndefC*. Then, by induction hypothesis, we have $\tilde{\tau}_b \sqsubseteq \lfloor\tilde{\tau}_c\rfloor$. Therefore, by *W-UndefC* we can deduce that $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_3$.

- *Inductive Case* $\tilde{\tau}_2 \triangleq [\blacklozenge\tau_a]$. By hypothesis we have $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_2$ and $\tilde{\tau}_2 \sqsubseteq \tilde{\tau}_3$. By *W-Trc*, according to Figure 4.2, we have $\tilde{\tau}_1 = \blacklozenge\tau_b$ such that $\tau_b \sqsubseteq \tau_a$. Moreover, we have $\tilde{\tau}_3 = \blacklozenge\tau_c$ such that $\tau_a \sqsubseteq \tau_c$. By induction hypothesis, we have $\tau_b \sqsubseteq \tau_c$ and therefore we can deduce by *W-Trc* that $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_3$. Now, if $\tilde{\tau}_3 = \lfloor\tau\rfloor$ and since $\tilde{\tau}_2 \sqsubseteq \tilde{\tau}_3$ hence, $\blacklozenge\tau_a \sqsubseteq \tau$ by *W-UndefB*. Then, by induction hypothesis, we have $\blacklozenge\tau_b \sqsubseteq \tau$. Therefore, by *W-UndefB* we can deduce that $\tilde{\tau}_1 \sqsubseteq \lfloor\tau\rfloor$. Finally, $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_3$.

- *Inductive Case* $\tilde{\tau}_2 \triangleq [\lfloor\tau_a\rfloor]$. By hypothesis we have $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_2$ and $\tilde{\tau}_2 \sqsubseteq \tilde{\tau}_3$. By *W-UndefA*, according to Figure 4.2, we have $\tilde{\tau}_1 = \lfloor\tau_b\rfloor$ such that $\tau_b \sqsubseteq \tau_a$. Moreover, we have $\tilde{\tau}_3 = \lfloor\tau_c\rfloor$ such that $\tau_a \sqsubseteq \tau_c$. By induction hypothesis, we have $\tau_b \sqsubseteq \tau_c$ and therefore we can deduce by *W-UndefA* that $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_3$. Now, if $\tilde{\tau}_1 = \tau_b$ and since $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_2$ then we have $\tau_b \sqsubseteq \tau_a$ by *W-UndefB*. By induction hypothesis we have $\tau_b \sqsubseteq \tau_c$ then we can deduce that $\tilde{\tau}_1 \sqsubseteq \tilde{\tau}_3$ by *W-UndefB*.

□

2.4.2 Intermediate Preservation

In Definition 4.24, we stipulated that there exists a safe abstraction connecting the program store S and the typing environment Γ. Therefore, we can assume that for a *well-typed* lval ω in Γ, it always corresponds to a valid location in S according to Definition 4.22. Then, to establish this safety relation, we introduce the following lemma:

Lemma 4.3 (Location) *Let S be a program store. Let Γ be a well-formed typing environment with respect to a lifetime $\mathtt{l}$ such that $S \sim \Gamma$. Let $\tilde{\tau}$ be a partial type and let ω be an lval such that $\Gamma \vdash \omega : \langle\tilde{\tau}\rangle^{\mathtt{m}}$ for some lifetime $\mathtt{m}$. Then, $loc(S, \omega, \mathtt{l}) = \ell$ for some location ℓ such that $S(\ell) = \langle v^{\perp}\rangle^{\mathtt{n}}$ and $S \vdash v^{\perp} \sim \tilde{\tau}$ such that $\mathtt{n} \in \{\mathtt{m}, *\} \cup \mathbb{N}$.*

`Proof.` By structural induction on the structure of ω according to Figure 3.5:

- *Base Case.* $\omega \triangleq [x]$. By *T-LvVar*, we have $\Gamma(x) = \langle\tilde{\tau}\rangle^{\mathtt{m}}$. By Definition of Θ, we have $\ell_{\mathtt{m}::x} \in \Theta(dom(\Gamma))$. By hypothesis, $S \sim \Gamma$ then, by inspection of Definition 4.24, $(\mathrm{dom}(S)\backslash\mathcal{L}) = \Theta(dom(\Gamma))$ and for all $x \in dom(\Gamma)$ such that $\Gamma(x) = \langle\tilde{\tau}\rangle^{\mathtt{m}}$, there exists $v^{\perp}$ such that $S \vdash v^{\perp} \sim \tilde{\tau}$ where $\mathrm{S}(\ell_{\mathtt{m}::x}) = \langle v^{\perp}\rangle^{\mathtt{m}}$. From the *well-formedness* of Γ, we know that by inspection of Definition 4.25, for all $x \in dom(\Gamma)$ where $\Gamma(x) = \langle .\rangle^{\mathtt{m}}$, we have $\mathtt{m} \succcurlyeq \mathtt{l}$. Then, according to Definition 3.1, we have $loc(S, x, \mathtt{l}) = \ell_{\mathtt{m}::x}$ and $S(\ell_{\mathtt{m}::x}) = \langle v^{\perp}\rangle^{\mathtt{m}}$ such that $\mathtt{m} \succeq \mathtt{l}$ where $S \vdash v^{\perp} \sim \tilde{\tau}$.

- *Inductive case.* $\omega \triangleq [*u]$. We have $\Gamma \vdash *u : \langle \tilde{\tau} \rangle^{\mathtt{n}}$ and only *T-LvBox*, *T-LvTrc*, and *T-LvBorrow* apply. Then, We proceed by case:
  - If $\Gamma \vdash u : \langle \blacksquare \tilde{\tau} \rangle^{\mathtt{m}}$: by induction hypothesis, $loc(S, u, \mathtt{l}) = \ell'$ where $S(\ell') = \langle \ell_a^{\blacksquare} \rangle^{\mathtt{m}}$ and $S \vdash \ell_a^{\blacksquare} \sim \blacksquare \tilde{\tau}$. By inspection of rules of Figure 4.7, only *V-Box* applies, and there exists $\ell_a^{\blacksquare}$ and $v^{\perp}$ such that $S(\ell_a) = \langle v^{\perp} \rangle^{*}$ and $S \vdash v^{\perp} \sim \tilde{\tau}$. Furthermore, by inspection of Definition 3.1, we have $loc(S, *u, \mathtt{l}) = \ell_a$ and hence $S(\ell_a) = \langle v \rangle^{*}$ where $S \vdash v^{\perp} \sim \tilde{\tau}$.
  - If $\Gamma \vdash u : \langle \blacklozenge \tau \rangle^{\mathtt{m}}$: by induction hypothesis, $loc(S, u, \mathtt{l}) = \ell'$ where $S(\ell') = \langle \ell_a^{\blacklozenge} \rangle^{\mathtt{m}}$ and $S \vdash \ell_a^{\blacklozenge} \sim \blacklozenge \tau$. By inspection of rules of Figure 4.7, only *V-Trc* applies, and there exists $\ell_a^{\blacklozenge}$ and v such that $S(\ell_a) = \langle v \rangle^{\mathtt{i}}$ and $S \vdash v \sim \tau$. Furthermore, by inspection of Definition 3.1, we have $loc(S, *u, \mathtt{l}) = \ell_a$ and hence $S(\ell_a) = \langle v \rangle^{\mathtt{i}}$ where $S \vdash v \sim \tau$.
  - If $\Gamma \vdash u : \langle \&[\mathtt{mut}]\overline{q} \rangle^{\mathtt{m}}$: by induction hypothesis, $loc(S, u, \mathtt{l}) = \ell'$ where $S(\ell') = \langle \ell^{\circ} \rangle^{\mathtt{m}}$ and $S \vdash \ell^{\circ} \sim \&[\mathtt{mut}]\overline{q}$. By *T-LvBorrow* (section 1.4), we have $\overline{\Gamma \vdash q : \langle \tau \rangle^{\mathtt{n}}}$ for some $\mathtt{n} \succcurlyeq \mathtt{l}$ from the *well-formedness* of Γ. As $S \vdash \ell^{\circ} \sim \&[\mathtt{mut}]\overline{q}$ then, by inspection of Definition 4.22 and by Figure 4.7, only *V-Borrow* applies and there exists i such that $loc(S, q_i, \mathtt{l}) = \ell$ where $S(\ell) = \langle v \rangle^{\mathtt{n}}$ where $S \vdash v \sim \tau$ (since $S \sim \Gamma$ and by Definition 4.24). Since $S \sim \Gamma$ and $q_i \subseteq \overline{q}$ then by *W-Bor* (Figure 4.2) and by applying Lemma 4.1, we have $S \vdash v \sim \tau$ and $S \vdash v \sim \tau_{q_0} ... \tau_{q_k}$ (where k is the sequence length). Therefore, we can deduce that $S \vdash v \sim \sqcup \tau$. Finally, by inspection of Definition 3.1, we have $loc(S, *u, \mathtt{l}) = \ell$ where $S(\ell) = \langle v \rangle^{\mathtt{n}}$ and $S \vdash v \sim \sqcup \tau$.

Note that the following is prohibited: $\omega \triangleq [*u]$ where $\Gamma \vdash u : \langle \diamond q \rangle^{\mathtt{m}}$. In this case, we have u is an inactive `Trc`. Hence, u cannot access its own content. Moreover, this constraint is checked by the typing rules where there is no specific rule that defines this behavior in the typing part of *T-Lv** (Section 1.4). □

In the following, we introduce the intermediate Lemmas necessary to guarantee the notion of safe abstraction between the program store S and the typing environment Γ after reading, writing, and dropping. For example, ensuring that a value is not dropped when it is still needed, especially when the value is borrowed.

Lemma 4.4 (Read Preservation) *Let S be a program store. Let Γ be a well-formed typing environment with respect to a lifetime $\mathtt{l}$ such that $S \sim \Gamma$. Let τ be a type and let ω be an lval such that $\Gamma \vdash \omega : \langle \tau \rangle^{\mathtt{m}}$ for some lifetime $\mathtt{m}$. Then, $read(S, \omega, \mathtt{l}) = \langle v \rangle^{\mathtt{n}}$ for some value v and $\mathtt{n} \in \{\mathtt{m}, *\} \cup \mathbb{N}$ such that $S \vdash v \sim \tau$.*

`Proof.` By hypothesis we have $\Gamma \vdash \omega : \langle \tau \rangle^{\mathtt{m}}$ and $S \sim \Gamma$ then, by applying Lemma 4.3, we have $loc(S, \omega, \mathtt{l}) = \ell$ where $S(\ell) = \langle v \rangle^{\mathtt{n}}$ where $\mathtt{n} \in \{\mathtt{m}, *\} \cup \mathbb{N}$ and $S \vdash v \sim \tau$. Finally, according to Definition 3.2, we have $read(S, \omega, \mathtt{l}) = \langle v \rangle^{\mathtt{n}}$ such that $loc(S, \omega, \mathtt{l}) = \ell$ and $S(\ell) = \langle v \rangle^{\mathtt{n}}$. □

Lemma 4.5 (Drop Preservation) *Let S be a program store. Let Γ be a well-formed typing environment with respect to a lifetime $\mathtt{l}$ such that $S \sim \Gamma$. Then, $drop(S, \mathtt{l}) \sim drop(\Gamma, \mathtt{l})$.*

`Proof.` In order to establish the safe abstraction between $\mathit{drop}(S, \mathtt{l}) \sim \mathit{drop}(\Gamma, \mathtt{l})$, as per Definition 4.24, we need to validate the following three aspects:

1. Ensuring that $S' = \mathit{drop}(S, \mathtt{l})$ remains valid.
2. Let $\Gamma' = \mathit{drop}(\Gamma, \mathtt{l})$ then, showing that $(\mathrm{dom}(S')\backslash\mathcal{L}) = \Theta(dom(\Gamma'))$.
3. For all $x \in dom(\Gamma')$ such that $\Gamma'(x) = \langle\tilde{\tau}\rangle^{\mathtt{m}}$, there exists $v^{\perp}$ such that $S' \vdash v^{\perp} \sim \tilde{\tau}$ where $S'(\ell_{\mathtt{m}::x}) = \langle v^{\perp}\rangle^{\mathtt{m}}$.

By hypothesis, $S \sim \Gamma$ then, by inspection of Definition 4.24, we have:

1. S is valid for $\mathtt{l}$.
2. $(\mathrm{dom}(S)\backslash\mathcal{L}) = \Theta(dom(\Gamma))$.
3. For all $x \in dom(\Gamma)$ such that $\Gamma(x) = \langle\tilde{\tau}\rangle^{\mathtt{m}}$, there exists $v^{\perp}$ such that $S \vdash v^{\perp} \sim \tilde{\tau}$ where $\mathrm{S}(\ell_{\mathtt{m}::x}) = \langle v^{\perp}\rangle^{\mathtt{m}}$.

Firstly, according to Definition 4.19, S is valid for $\mathtt{l}$, indicating that all locations in the values of S have a unique occurrence. By applying the $\mathit{drop}(S, \mathtt{l})$ function, as defined in Definition 3.4, we can recursively remove all locations owned in the lifetime $\mathtt{l}$ from S. This implies that for all $v^{\perp}$ in $S_{|\mathtt{l}}$, if $v^{\perp} = \ell_a^{\blacksquare}$, the *drop* function recursively removes all the values accessible from ℓ_a. As S is valid for $\mathtt{l}$, there exists only one occurrence of ℓ_a in S, and removing ℓ_a does not invalidate S. Similarly, if $v^{\perp} = \ell_a^{\blacklozenge}$, the *drop* function verifies whether the counter i is 1. If the counter i is 1, it indicates that ℓ_a can only be accessed by the current thread, and thus the *drop* function recursively removes the values accessible from ℓ_a. Again, since S is valid for $\mathtt{l}$, there exists only one occurrence of ℓ_a in S, and removing ℓ_a does not invalidate S. Otherwise, the *drop* function decrements the counter by 1, and therefore ℓ_a is no longer accessible by the current thread. However, ℓ_a is always accessible via other threads. Lastly, if $v^{\perp} = \ell_a^{\diamond}$, the *drop* function decrements the counter by 1, as specified in Definition 3.4. Consequently, the resulting program store $S' = \mathit{drop}(S, \mathtt{l})$ contains all the locations in S except those owned in lifetime $\mathtt{l}$ while maintaining the validity of S'.

Secondly, we need to prove that $(\mathrm{dom}(S')\backslash\mathcal{L}) = \Theta(dom(\mathit{drop}(\Gamma, \mathtt{l})))$. By hypothesis, Γ is *well-formed* with respect to $\mathtt{l}$, referring to Definition 4.25, for all $x \in dom(\Gamma)$ where $\Gamma(x) = \langle . \rangle^{\mathtt{n}}$, we have $\mathtt{n} \succcurlyeq \mathtt{l}$. By applying the *drop* function depicted in Figure 4.3, we can remove all variables from the typing environment Γ that have the lifetime $\mathtt{l}$. Hence, the resulting typing environment $\Gamma' = \mathit{drop}(\Gamma, \mathtt{l})$ contains all the variables in $dom(\Gamma)$ except those with the lifetime $\mathtt{l}$. Since Γ is well-formed with respect to $\mathtt{l}$, Γ' remains well-formed. We need to demonstrate the equality of the domain in both directions: (1) for all $x \in dom(\Gamma')$ such that $\Gamma'(x) = \langle\tilde{\tau}\rangle^{\mathtt{m}}$, we have $\mathtt{l} \prec \mathtt{m}$. Consequently, we can deduce that $x \in dom(\Gamma)$. The latter indicates that there exists $\ell_{\mathtt{m}::x} \in dom(S)$ (since by hypothesis we have $S \sim \Gamma$). As $\mathtt{l} \prec \mathtt{m}$, then $\ell_{\mathtt{m}::x} \in dom(S')$. Hence, we have $(\mathrm{dom}(S')\backslash\mathcal{L}) \supseteq \Theta(dom(\Gamma'))$. (2) For all $\ell_{\mathtt{m}::x} \in dom(S')$, we have we have $\mathtt{l} \prec \mathtt{m}$ because we recursively removed all locations owned in the lifetime $\mathtt{l}$ from S. Consequently, we can deduce that $\ell_{\mathtt{m}::x} \in dom(S)$. The latter indicates that $x \in dom(\Gamma)$ (since by hypothesis we have $S \sim \Gamma$) such that $\Gamma(x) =$

$\langle\tilde{\tau}\rangle^{\mathtt{m}}$ and since $\mathtt{l} \prec \mathtt{m}$. Therefore, $x \in dom(\Gamma')$ as well. Hence, we have $(\mathrm{dom}(S')\backslash\mathcal{L}) \subseteq \Theta(dom(\Gamma'))$. Finally, we can deduce that $(\mathrm{dom}(S')\backslash\mathcal{L}) = \Theta(dom(\Gamma'))$.

Thirdly, we need to demonstrate that for all $x \in dom(\Gamma')$ such that $\Gamma'(x) = \langle\tilde{\tau}\rangle^{\mathtt{m}}$, there exists $v^{\perp}$ such that $S' \vdash v^{\perp} \sim \tilde{\tau}$ where $S'(\ell_{\mathtt{m}::x}) = \langle v^{\perp}\rangle^{\mathtt{m}}$. The proof proceeds by induction on $S' \vdash v^{\perp} \sim \tilde{\tau}$ according to Figure 4.7 as follows:

- *Base Case.* $[S' \vdash v^{\perp} \sim \tilde{\tau}]$. If $v^{\perp} = \epsilon$ or $v^{\perp} = \mathtt{int}$ or $v^{\perp} = \mathtt{bool}$, then only *V-Unit*, *V-Int*, and *V-Bool* apply respectively according to Figure 4.7. Now, if $v^{\perp} = \perp$ then only *V-Undef* rule applies.

- *Base Case.* $[S' \vdash \ell^{\circ} \sim \&[\mathtt{mut}]\overline{\omega}]$. Then, there exists $x \in dom(\Gamma')$ such that $\Gamma'(x) = \langle\&[\mathtt{mut}]\overline{\omega}\rangle^{\mathtt{m}}$ and there exists $v^{\perp}$ such that $S'(\ell_{\mathtt{m}::x}) = \langle v^{\perp}\rangle^{\mathtt{m}}$. Since $x \in dom(\Gamma')$ then, we have $x \in dom(\Gamma)$. By hypothesis, Γ is well-formed with respect to $\mathtt{l}$. By inspection of Definition 4.25, we have $\overline{\Gamma \vdash \omega : \langle\tau\rangle^{\mathtt{n}} \wedge \mathtt{m} \preccurlyeq \mathtt{n}}$. As discussed above, we have $\mathtt{l} \prec \mathtt{m}$, so we can deduce that $\mathtt{l} \prec \mathtt{n}$ and $\omega \in dom(\Gamma')$. Furthermore, since $\ell_{\mathtt{m}::x} \in dom(S')$ then, we have $\ell_{\mathtt{m}::x} \in dom(S)$. By hypothesis we have $S \sim \Gamma$ and by inspection of Definition 4.24 and Figure 4.7 (where only *V-Borrow* applies), there exists i such that $loc(S, \omega_i, \mathtt{l}) = \ell$ and $S \vdash \ell^{\circ} \sim \&[\mathtt{mut}]\overline{\omega}$ such that $v^{\perp} = \ell^{\circ}$. Since $\mathtt{l} \prec \mathtt{n}$, then ℓ is a location not owned in the lifetime $\mathtt{l}$ and we can deduce that $\ell \in dom(S')$. Hence, by *V-Borrow*, $S' \vdash \ell^{\circ} \sim \&[\mathtt{mut}]\overline{\omega}$.

- *Base Case.* $[S' \vdash \ell_a^{\diamond} \sim \diamond\overline{\omega}]$. Then, there exists $x \in dom(\Gamma')$ such that $\Gamma'(x) = \langle\diamond\overline{\omega}\rangle^{\mathtt{m}}$ and there exists $v^{\perp}$ such that $S'(\ell_{\mathtt{m}::x}) = \langle v^{\perp}\rangle^{\mathtt{m}}$. Since $x \in dom(\Gamma')$ then, we have $x \in dom(\Gamma)$. By hypothesis, Γ is well-formed with respect to $\mathtt{l}$. By inspection of Definition 4.25, we have $\overline{\Gamma \vdash \omega : \langle\tau\rangle^{\mathtt{n}} \wedge \mathtt{m} \preccurlyeq \mathtt{n}}$. As discussed above, we have $\mathtt{l} \prec \mathtt{m}$, so we can deduce that $\mathtt{l} \prec \mathtt{n}$ and $\omega \in dom(\Gamma')$. Furthermore, since $\ell_{\mathtt{m}::x} \in dom(S')$, we have $\ell_{\mathtt{m}::x} \in dom(S)$. By hypothesis, we have $S \sim \Gamma$ and by inspection of Definition 4.24 and Figure 4.7 (where only *V-Clone* applies), there exists i such that $read(S, \omega_i, \mathtt{l}) = \langle\ell_a^{\blacklozenge}\rangle^{\mathtt{l}}$ and $S \vdash \ell_a^{\diamond} \sim \diamond\overline{\omega}$ such that $v^{\perp} = \ell_a^{\diamond}$. Since $\mathtt{l} \prec \mathtt{n}$, then ℓ_a is a location not owned in the lifetime $\mathtt{l}$ and we can deduce that $\ell \in dom(S')$. Hence, by *V-Clone*, $S' \vdash \ell_a^{\diamond} \sim \diamond\overline{\omega}$.

- *Inductive Case.* $[S' \vdash \ell_a^{\blacksquare} \sim \blacksquare\tilde{\tau}]$. Then, there exists $x \in dom(\Gamma')$ such that $\Gamma'(x) = \langle\blacksquare\tilde{\tau}\rangle^{\mathtt{m}}$ and there exists $v^{\perp}$ such that $S'(\ell_{\mathtt{m}::x}) = \langle v^{\perp}\rangle^{\mathtt{m}}$. Since $x \in dom(\Gamma')$ then, we have $x \in dom(\Gamma)$ and since $\ell_{\mathtt{m}::x} \in dom(S')$ then, we have $\ell_{\mathtt{m}::x} \in dom(S)$. By hypothesis we have $S \sim \Gamma$ and by inspection of Definition 4.24, we have $S \vdash v^{\perp} \sim \blacksquare\tilde{\tau}$ such that $v^{\perp} = \ell_a^{\blacksquare}$ and $S(\ell_a) = \langle v'^{\perp}\rangle^{*}$ by *V-Box* (Figure 4.7). By induction hypothesis, we have $S'(\ell_a) = \langle v'^{\perp}\rangle^{*}$ and $S' \vdash \ell_a \sim \tilde{\tau}$. Then by (V-Box) (Figure 4.7) we can deduce that $S' \vdash \ell_a^{\blacksquare} \sim \blacksquare\tilde{\tau}$.

- *Inductive Case.* $[S' \vdash \ell_a^{\blacklozenge} \sim \blacklozenge\tilde{\tau}]$. Then, there exists $x \in dom(\Gamma')$ such that $\Gamma'(x) = \langle\blacklozenge\tilde{\tau}\rangle^{\mathtt{m}}$ and there exists $v^{\perp}$ such that $S'(\ell_{\mathtt{m}::x}) = \langle v^{\perp}\rangle^{\mathtt{m}}$. Since $x \in dom(\Gamma')$ then, we have $x \in dom(\Gamma)$ and since $\ell_{\mathtt{m}::x} \in dom(S')$ then, we have $\ell_{\mathtt{m}::x} \in dom(S)$. By hypothesis we have $S \sim \Gamma$ and by inspection of Definition 4.24, we have $S \vdash v^{\perp} \sim \blacklozenge\tilde{\tau}$ such that $v^{\perp} = \ell_a^{\blacklozenge}$ and $S(\ell_a) = \langle v'^{\perp}\rangle^{\mathtt{i}}$ by *V-Trc* (Figure 4.7). By induction

hypothesis, we have $S'(\ell_a) = \langle v'^{\perp} \rangle^{\mathtt{i}}$ and $S' \vdash \ell_a \sim \tilde{\tau}$. Then by (V-Trc) (Figure 4.7) we can deduce that $S' \vdash \ell_a^{\blacklozenge} \sim \blacklozenge\tilde{\tau}$.

Finally, having demonstrated the three aspects mentioned above, we can now infer that $\mathit{drop}(S, \mathtt{l}) \sim \mathit{drop}(\Gamma, \mathtt{l})$. □

The previous Lemma establishes that when a specific set of variables and their associated values are dropped from the typing environment Γ and the program store S respectively, the notion of safe abstraction is maintained. In addition, the subsequent Lemma guarantees that updating values in the program store S maintains the property of safe abstraction as follows:

Lemma 4.6 (Update Preservation) *Let S be a program store. Let Γ be a well-formed typing environment with respect to a lifetime $\mathtt{l}$ such that $S \sim \Gamma$. Let $\tilde{\tau}_1$, $\tilde{\tau}_2$ be partial types and let $v_1^{\perp}$, $v_2^{\perp}$ be partial values such that $S \vdash v_1^{\perp} \sim \tilde{\tau}_1$ and $S \vdash v_2^{\perp} \sim \tilde{\tau}_2$. Let ω be an lval such that $\Gamma \vdash \omega : \langle \tilde{\tau}_1 \rangle^{\mathtt{m}}$ for some lifetime $\mathtt{m}$. Then $\mathit{write}(\mathit{drop}(S, \{v_1^{\perp}\}), \omega, v_2^{\perp}, \mathtt{l}) \sim \mathit{write}^0(\Gamma, \omega, \tilde{\tau}_2)$.*

`Proof.` In order to establish the safe abstraction between $\mathit{write}(\mathit{drop}(S, \{v_1^{\perp}\}), \omega, v_2^{\perp}, \mathtt{l}) \sim \mathit{write}^0(\Gamma, \omega, \tilde{\tau}_2)$, as per Definition 4.24, we need to validate the following three aspects:

1. Ensuring that $S' = \mathit{write}(\mathit{drop}(S, \{v_1^{\perp}\}), \omega, v_2^{\perp}, \mathtt{l})$ remains valid.
2. Let $\Gamma' = \mathit{write}^0(\Gamma, \omega, \tilde{\tau}_2)$ then, showing that $(\mathrm{dom}(S')\backslash\mathcal{L}) = \Theta(dom(\Gamma'))$.
3. For all $x \in dom(\Gamma')$ such that $\Gamma'(x) = \langle \tilde{\tau} \rangle^{\mathtt{m}}$, there exists $v^{\perp}$ such that $S' \vdash v^{\perp} \sim \tilde{\tau}$ where $S'(\ell_{\mathtt{m}::x}) = \langle v^{\perp} \rangle^{\mathtt{m}}$.

By hypothesis, $S \sim \Gamma$ then, by inspection of Definition 4.24, we have:

1. S is valid for $\mathtt{l}$.
2. $(\mathrm{dom}(S)\backslash\mathcal{L}) = \Theta(dom(\Gamma))$.
3. For all $x \in dom(\Gamma)$ such that $\Gamma(x) = \langle \tilde{\tau} \rangle^{\mathtt{m}}$, there exists $v^{\perp}$ such that $S \vdash v^{\perp} \sim \tilde{\tau}$ where $\mathrm{S}(\ell_{\mathtt{m}::x}) = \langle v^{\perp} \rangle^{\mathtt{m}}$.

Firstly, according to Definition 4.19, S is valid for $\mathtt{l}$, indicating that all the locations in the values of S have a unique occurrence. Then, applying the $\mathit{drop}(S, \{v_1^{\perp}\})$ function, based on Definition 3.4, drops recursively the value $v_1^{\perp}$ from S. This implies that if $v_1^{\perp} = \ell_a^{\blacksquare}$, the *drop* function recursively remove all the values accessible from ℓ_a. As S is valid for $\mathtt{l}$, by inspection of Definition 4.19, there exists only one occurrence of ℓ_a in S, and removing ℓ_a does not invalidate S. Similarly, if $v_1^{\perp} = \ell_a^{\blacklozenge}$, the *drop* function verifies whether the counter i is 1. If the counter i is 1, it indicates that ℓ_a can only be accessed by the current thread, and thus the *drop* function recursively removes the values accessible from ℓ_a. Again, since S is valid for $\mathtt{l}$, there exists only one occurrence of ℓ_a in S, and removing

ℓ_a does not invalidate S. Otherwise, the *drop* function decrements the counter by 1, and therefore ℓ_a is no longer accessible by the current thread. However, ℓ_a is always accessible via other threads. If $v_1^\perp = \ell_a^\diamond$, the *drop* function decrements the counter by 1, as specified in Definition 3.4. Consequently, the resulting program store $S' = \mathit{drop}(S, \mathtt{l})$ contains all locations in S except of $v_1^\perp$, while maintaining the validity of S'. Furthermore, by applying the $\mathit{write}(\mathit{drop}(S, \{v_1^\perp\}), \omega, v_2^\perp, \mathtt{l})$, according to Definition 3.3, adds $v_2^\perp$ to S' as a new value of ω after dropping $v_1^\perp$. Consequently, S' contains all values in S except of $v_1^\perp$ which is replaced by $v_2^\perp$. By inspection of Definition 4.24, S is valid for $\mathtt{l}$ and since $S \vdash v_2^\perp \sim \tilde{\tau}_2$, we can deduce that S' is valid for $\mathtt{l}$.

Secondly, we need to prove that $(\mathrm{dom}(S')\backslash\mathcal{L}) = \Theta(dom(\Gamma'))$ such that $\Gamma' = \mathit{write}^0(\Gamma, \omega, \tilde{\tau}_2)$. Applying the $\mathit{write}^0(\Gamma, \omega, \tilde{\tau}_2)$ function, according to Figure 4.3, updates the type of ω in Γ by $\tilde{\tau}_2$. This function modifies Γ by replacing the type of ω with $\tilde{\tau}_2$. After the update, it is stated that all values in S' remain identical except for $v_1^\perp$, which is changed to $v_2^\perp$, and all variables in Γ' have the same type except for ω, which is replaced by $\tilde{\tau}_2$. Consequently we can deduce by inspection of Definition 4.24 that $(\mathrm{dom}(S')\backslash\mathcal{L}) = \Theta(dom(\Gamma'))$ inferred by $(\mathrm{dom}(S)\backslash\mathcal{L}) = \Theta(dom(\Gamma))$.

Thirdly, it should be demonstrated that for all $x \in dom(\Gamma')$ such that $\Gamma'(x) = \langle\tilde{\tau}\rangle^{\mathtt{m}}$, there exists $v^\perp$ such that $S' \vdash v^\perp \sim \tilde{\tau}$ where $S'(\ell_{\mathtt{m}::x}) = \langle v^\perp\rangle^{\mathtt{m}}$. The proof proceeds by induction on $S' \vdash v^\perp \sim \tilde{\tau}$ according to Figure 4.7. By hypothesis, we have $S \vdash v_2^\perp \sim \tilde{\tau}_2$ then we can deduce that $S' \vdash v_2^\perp \sim \tilde{\tau}_2$ and based on the proofs of Lemma 4.5 specifically for this case, we achieve the required result.

Finally, we have $\mathit{write}(\mathit{drop}(S, \{v_1^\perp\}), \omega, v_2^\perp, \mathtt{l}) \sim \mathit{write}^0(\Gamma, \omega, \tilde{\tau}_2)$. □

The following Lemma states that if an expression is considered a value, then the typing environment Γ remains unchanged from the input to the output:

Lemma 4.7 (Value Typing) *Let Γ_1 be a well-formed typing environment with respect to a lifetime $\mathtt{l}$. Let σ be a store typing and let v be a value such that $\Gamma_1 \vdash \langle v : \tau\rangle_\sigma^{\mathtt{l}} \dashv \Gamma_2$ for some τ and Γ_2. Then $\Gamma_1 = \Gamma_2$.*

`Proof.` The proof proceeds by induction on the structure of the typing derivation. Since we assume that the typed expression is a value v, we focus on the cases that can be used to type a value. These cases, according to the typing rules presented in section 1.4, include *T-Const, T-MutBorrow, T-ImmBorrow* and *T-Clone*. □

2.4.3 Borrow and `Trc` Invariance Lemma

The following lemma demonstrates that our typing rules (Section 1.4) guarantee the properties "Borrowing and `Trc` invariance" according to Definition 4.25. In other words, a *well-formed* typing environment is guaranteed to remain *well-formed* as follows:

Lemma 4.8 (Borrow and Trc Invariance) *Let $S \triangleright e$ be a valid local state for some lifetime $\mathtt{l}$. Let σ be a store typing such that $S \triangleright e \vdash \sigma$ and let Γ_1 be a well-formed typing environment*

with respect to $\mathtt{l}$ *such that* $S \sim \Gamma_1$. *Let* Γ_2 *be a typing environment and* $\Gamma_1 \vdash \langle e : \tau \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2$ *for some* τ. *Then* $\Gamma_2[\gamma \mapsto \langle \tau \rangle^{\mathtt{l}}]$ *is well-formed with respect to* $\mathtt{l}$ *for some* $\gamma \in$ *fresh.*

`Proof.` By structural induction on the form of e introduced by Figure 3.5:

- *Base Case* $e \triangleq [v]$. By *T-Const*, $\sigma \vdash v : \tau$ such that $\Gamma_1 = \Gamma_2$. By hypothesis, we have $S \rhd v \vdash \sigma$ then, by inspection of Definition 4.23 we have $S \vdash v \sim \sigma(v)$ where v is safely abstracted by τ. Hence, the properties outlined in Definition 4.25 are preserved and $\Gamma_1[\gamma \mapsto \langle \tau \rangle^{\mathtt{l}}]$ remains *well-formed* with respect to $\mathtt{l}$.

- *Base Case* $e \triangleq [\hat{\omega}]$. By hypothesis, Γ_1 is well-formed with respect to $\mathtt{l}$, so all conditions specified by Definition 4.25 are maintained. By *T-Copy*, $\Gamma_1 \vdash \omega : \langle \tau \rangle^{\mathtt{m}}$. Additionally, we have $copy(\tau)$ and $\neg readProhibited(\Gamma_1, \omega)$ functions where $\Gamma_1 = \Gamma_2$. By inspection of Definition 4.1, $copy(\tau)$ indicates that τ can be of the form $\mathtt{int}, \mathtt{bool}, \epsilon$ or shared reference. According to Definition 4.3, $\neg readProhibited(\Gamma_1, \omega)$ verifies that ω is not borrowed as mutable in Γ_1. This condition preserves point (3) in Definition 4.25 without invalidating points (1), (2), and (4). Then, if τ is $\mathtt{int}, \mathtt{bool}$ or ϵ, $\Gamma_2[\gamma \mapsto \langle \tau \rangle^{\mathtt{l}}]$ is *well-formed* with respect to $\mathtt{l}$. Otherwise, if τ is of the form $\& u$ for some u then, it is possible to have multiple shared references to u. In this case, all conditions specified by Definition 4.25 are satisfied and we can deduce that $\Gamma_2[\gamma \mapsto \langle \tau \rangle^{\mathtt{l}}]$ is *well-formed* with respect to $\mathtt{l}$.

- *Base Case* $e \triangleq [\omega]$. By hypothesis, Γ_1 is well-formed with respect to $\mathtt{l}$, so all conditions specified by Definition 4.25 are maintained. By *T-Move*, we have $\Gamma_1 \vdash \omega : \langle \tau \rangle^{\mathtt{m}}$ and from the well-formedness of Γ_1, we have $\mathtt{m} \succcurlyeq \mathtt{l}$. Additionally, we have $move(\Gamma_1, \omega)$, $\neg writeProhibited(\Gamma_1, \omega)$ and $\neg TrcMoveProhibited(\Gamma_1, \omega)$ functions. According to Definition 4.4, $\neg writeProhibited(\Gamma_1, \omega)$ ensures that ω is not borrowed in Γ_1, which protects points (1) and (2) in Definition 4.25. Similarly, by inspection of Definition 4.10, $\neg TrcMoveProhibited(\Gamma_1, \omega)$ ensures that ω is not cloned in Γ_1, protecting point (4) in Definition 4.25. By inspection of Definition 4.8, $move(\Gamma_1, \omega)$ replaces τ with $\lfloor \tau \rfloor$ in Γ_2 (if $\tau = \&\mathtt{mut}\ u$, then τ is not in Γ_2, which protects point (3)). Finally, all conditions specified by Definition 4.25 are satisfied and we can deduce that $\Gamma_2[\gamma \mapsto \langle \tau \rangle^{\mathtt{l}}]$ is *well-formed* with respect to lifetime $\mathtt{l}$.

- *Base Case* $e \triangleq [\&[\mathtt{mut}]\omega]$. By hypothesis, Γ_1 is well-formed with respect to $\mathtt{l}$, so all conditions specified by Definition 4.25 are maintained. By both *T-ImmBorrow* and *T-MutBorrow*, we have $\Gamma_1 \vdash \omega : \langle \tau \rangle^{\mathtt{m}}$ such that $\Gamma_1 = \Gamma_2$ and from the well-formedness of Γ_1, we have $\mathtt{m} \succcurlyeq \mathtt{l}$. By *T-ImmBorrow*, we have $\neg readProhibited(\Gamma_1, \omega)$. By inspection of Definition 4.3, this function ensures that ω is not borrowed as mutable in Γ_1, which protect point (3) in Definition 4.25 without invalidating points (1), (2), and (4). Similarly, by *T-MutBorrow*, we have $\neg writeProhibited(\Gamma_1, \omega)$. By inspection of Definition 4.4, this function guarantees that ω is not borrowed, mutable or immutable, protecting also point (3) in Definition 4.25 without invalidating points (1), (2), and (4). Hence $\Gamma_2[\gamma \mapsto \langle \&[\mathtt{mut}]\omega \rangle^{\mathtt{l}}]$ is *well-formed* with respect to lifetime $\mathtt{l}$.

- *Base Case* $e \triangleq [\omega.clone]$. By hypothesis, Γ_1 is well-formed with respect to $\mathtt{l}$, so all conditions specified by Definition 4.25 are maintained. By *T-Clone*, $\Gamma_1 = \Gamma_2$ and

$\Gamma \vdash \omega : \langle \blacklozenge \tau \rangle^{\mathtt{m}}$ which requires that ω is a well typed and is an active `Trc` type. From the well-formedness of Γ_1, we have $\mathtt{m} \geqslant \mathtt{l}$ which protect point (4) in Definition 4.25 without invalidating points (1), (2), and (3). Finally, all conditions specified by Definition 4.25 are satisfied and we can deduce that $\Gamma_2[\gamma \mapsto \langle \diamond \omega \rangle^{\mathtt{l}}]$ is *well-formed* with respect to lifetime `l`.

- *Base Case* $e \triangleq [\mathtt{cooperate}]$. By hypothesis, Γ_1 is well-formed with respect to `l`, so all conditions specified by Definition 4.25 are maintained. By *T-Cooperate*, we have $\Gamma_1 = \Gamma_2$. In this case, the current thread gives control to another thread, implying that when the thread resumes execution after cooperation, certain properties described in Definition 4.25 need to be preserved, specifically points (1) and (2). The *T-Cooperate* rule introduces the function *safeTrc*. By inspecting Definition 4.5, this function verifies that there are no borrowed shared data in Γ_1. In other words, if the latter exists Γ_1, then the path to that borrow is not under a `Trc` (i.e. via Definition 4.6). Hence, all conditions specified by Definition 4.25 are satisfied and we can deduce that $\Gamma_2[\gamma \mapsto \langle \epsilon \rangle^{\mathtt{l}}]$ is *well-formed* with respect to `l`.

- *Inductive Case* $e \triangleq [\mathtt{box}(e_2)]$. By *T-Box*, $\Gamma_1 \vdash \langle e_2 : \tau_2 \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2$ and by induction hypothesis, $\Gamma_2[\gamma \mapsto \langle \tau_2 \rangle^{\mathtt{l}}]$ is *well-formed*. Hence, $\Gamma_2[\gamma \mapsto \langle \blacksquare \tau_2 \rangle^{\mathtt{l}}]$ is *well-formed* with respect to `l`.

- *Inductive Case* $e \triangleq [\mathtt{trc}(e_2)]$. By *T-Trc*, $\Gamma_1 \vdash \langle e_2 : \tau_2 \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2$ and by induction hypothesis, $\Gamma_2[\gamma \mapsto \langle \tau_2 \rangle^{\mathtt{l}}]$ is *well-formed* with respect to `l`. Hence, $\Gamma_2[\gamma \mapsto \langle \blacklozenge \tau_2 \rangle^{\mathtt{l}}]$ is *well-formed* with respect to `l`.

- *Inductive Case* $e \triangleq [e_1 \oplus e_2]$. By *T-Arithm*, $\Gamma_1 \vdash \langle e_1 : \mathtt{int} \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2$ and $\Gamma_2 \vdash \langle e_2 : \mathtt{int} \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_3$. By induction hypothesis, we have $\Gamma_2[\gamma \mapsto \langle \mathtt{int} \rangle^{\mathtt{l}}]$ is *well-formed* with respect to `l` and $\Gamma_3[\gamma \mapsto \langle \mathtt{int} \rangle^{\mathtt{l}}]$ is *well-formed* with respect to `l`.

- *Inductive Case* $e \triangleq [e_1 \otimes e_2]$. By *T-Cond*, $\Gamma_1 \vdash \langle e_1 : \tau_1 \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2$ and $\Gamma_2 \vdash \langle e_2 : \tau_2 \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_3$. By induction hypothesis, we have $\Gamma_2[\gamma \mapsto \langle \tau_1 \rangle^{\mathtt{l}}]$ is *well-formed* with respect to `l` and $\Gamma_3[\gamma \mapsto \langle \tau_2 \rangle^{\mathtt{l}}]$ is *well-formed* with respect to `l`. By *T-Cond*, we have *copy*(τ_1) and *copy*(τ_2). Hence, the properties outlined in Definition 4.25 are preserved and $\Gamma_3[\gamma \mapsto \langle \mathtt{bool} \rangle^{\mathtt{l}}]$ is *well-formed* with respect to `l`.

- *Inductive Case* $e \triangleq [\mathtt{spawn}(f(\overline{e_2})]$. By *T-Spawn*, $\Gamma_1 \vdash \langle \overline{e_2 : \tau_2} \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2$. According to the induction hypothesis, we assume that $\Gamma_2[\overline{\gamma \mapsto \langle \tau_2 \rangle^{\mathtt{l}}}]$ is *well-formed* with respect to `l`. By *T-Spawn*, we have $\Gamma_2 \vdash (\overline{S}) \Longleftarrow (\overline{\tau})$, by the requirements implemented by the latter, references and active `Trc`s cannot be considered. In particular, by checking the compatibility between signatures and types via Definition 4.15. As a result, the properties outlined in Definition 4.25 are preserved in this case and therefore $\Gamma_2[\gamma \mapsto \langle \epsilon \rangle^{\mathtt{l}}]$ is *well-formed* with respect to `l`.

- *Inductive Case* $e \triangleq [\mathtt{let\ mut}\ x = e_2]$. By *T-Declare*, $\Gamma_1 \vdash \langle e_2 : \tau_2 \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma$. By induction hypothesis, we have $\Gamma[\gamma \mapsto \langle \tau_2 \rangle^{\mathtt{l}}]$ is *well-formed* with respect to `l`. This implies that $\Gamma[\gamma \mapsto \langle \tau_2 \rangle^{\mathtt{l}}]$ preserves the properties described in Definition 4.25. Therefore, by replacing γ with x, we obtain $\Gamma_2 = \Gamma[x \mapsto \langle \tau_2 \rangle^{\mathtt{l}}]$ by *T-Declare*, which is *well-formed* with respect to `l`. Finally, $\Gamma_2[\gamma \mapsto \langle \epsilon \rangle^{\mathtt{l}}]$ is *well-formed* with respect to `l`.

- *Inductive Case* $e \triangleq [\omega = e_2]$. By *T-Assign*, $\Gamma_1 \vdash \langle e_2 : \tau_2 \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma$. By induction hypothesis, we assume that $\Gamma[\gamma \mapsto \langle \tau_2 \rangle^{\mathtt{l}}]$ is *well-formed* with respect to $\mathtt{l}$. This implies that $\Gamma[\gamma \mapsto \langle \tau_2 \rangle^{\mathtt{l}}]$ preserves the properties described in Definition 4.25. By *T-Assign*, we have $\Gamma_1 \vdash \omega : \langle \tau_1 \rangle^{\mathtt{m}}$ such that $\Gamma \vdash \tau_2 \succcurlyeq \mathtt{m}$ and $\Gamma_2 = \mathit{write}^0(\Gamma, \omega, \tau_2)$. By inspection of Figure 4.5, if τ_2 contains a reference or an inactive `Trc`, that reference (resp. the inactive `Trc`) must live at least as long as the lifetime $\mathtt{m}$. This ensures points (1), (2) and (4) of Definition 4.25 without invalidate point (3). According to Figure 4.3, $\mathit{write}^0(\Gamma, \omega, \tau_2)$ updates the type of ω by τ_2. Consequently, all the variables in Γ_2 are the same as in Γ_1, with the exception of ω. Finally, by *T-Assign* we have $\neg \mathit{writeProhibited}(\Gamma_2, \omega)$. By inspection of Definition 4.4, this ensures that ω is not borrowed in Γ_2 which protects point (3) in Definition 4.25. As a result, $\Gamma_2[\gamma \mapsto \langle \epsilon \rangle^{\mathtt{l}}]$ remains *well-formed* with respect to $\mathtt{l}$.
- *Inductive Case* $e \triangleq [\{e\}^{\mathtt{m}}]$. By *T-Block*, we have $\Gamma_1 \vdash \langle e : \tau \rangle^{\mathtt{m}}_{\sigma} \dashv \Gamma$ for fresh $\mathtt{l} \succcurlyeq \mathtt{m}$ and by induction hypothesis $\Gamma[\gamma \mapsto \langle \tau \rangle^{\mathtt{m}}]$ is *well-formed* with respect to $\mathtt{l}$. By *T-Block*, we have $\Gamma_2 = \mathit{drop}(\Gamma, \mathtt{m})$. the *drop* function, as defined in Figure 4.3, deallocates variables that have a lifetime $\mathtt{m}$ by dropping them from Γ. Furthermore, the *T-Block* rule requires the premise $\Gamma_2 \vdash \tau \succcurlyeq \mathtt{l}$. According to Figure 4.5, this means that if τ contains a reference (resp. an inactive `Trc`), that reference (resp. the inactive `Trc`) must live at least as long as the lifetime $\mathtt{l}$. This condition is necessary to prevent the possibility of having a dropped location that is still needed (which protects points (1), (2) and (4) without invalidate point (3)). Hence, $\Gamma_2[\gamma \mapsto \langle \tau \rangle^{\mathtt{l}}]$ is *well-formed* with respect to $\mathtt{l}$.
- *Inductive Case* $e \triangleq [\overline{e}]$. By *T-Sequence* we have $\Gamma_1 \vdash \langle e_1 : \tau_1 \rangle^{\mathtt{m}}_{\sigma} \dashv \Gamma_2 \ldots \Gamma_n \vdash \langle e_n : \tau_n \rangle^{\mathtt{m}}_{\sigma} \dashv \Gamma_{n+1}$ and by induction hypothesis over the derivation of e_1, we have $\Gamma_2[\gamma \mapsto \langle \tau_1 \rangle^{\mathtt{l}}]$ is *well-formed* with respect to $\mathtt{l}$ and so on.

□

In the following proofs, we present progress and preservation lemmas at three different depths.

2.4.4 Progress and Preservation Step

The Step Progress lemma establishes that for a given thread $(t, \{e\}^{\mathtt{l}})$ where e is a *well-typed* expression then, one of the two cases must hold: either e is a value, or it can be reduced by at least one step to another expression e' as follows:

Lemma 4.9 (Step Progress) *Let $S_1 \rhd e_1$ be a valid local state for some lifetime $\mathtt{l}$. Let σ be a store typing such that $S_1 \rhd e_1 \vdash \sigma$ and let Γ_1 be a well-formed typing environment with respect to $\mathtt{l}$ such that $S_1 \sim \Gamma_1$ and $\Gamma_1 \vdash \langle e_1 : \tau \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2$ for some τ and Γ_2. Then either e_1 is a value or there exists some T and some state $S_2 \rhd e_2$ such that $\langle S_1 \rhd e_1 \xrightarrow{T}_i S_2 \rhd e_2 \rangle^{\mathtt{l}}$.*

`Proof.` By structural induction on the form of e_1 introduced by Figure 3.5:

- *Base Case* $e_1 \triangleq [v]$. The result is immediate as e_1 is already a value.
- *Base Case* $e_1 \triangleq [\hat{\omega}]$. By *T-Copy*, $\Gamma_1 \vdash \omega : \langle \tau \rangle^{\mathtt{m}}$. Since $S_1 \sim \Gamma_1$ then, by applying Lemma 4.4, *read*$(S_1, \omega, \mathtt{1})$ is defined. Hence, we can step with *R-Copy*.
- *Base Case* $e_1 \triangleq [\omega]$. By *T-Move* we have $\Gamma_1 \vdash \omega : \langle \tau \rangle^{\mathtt{m}}$. Since $S_1 \sim \Gamma_1$ then, by applying Lemma 4.3, we have $loc(S_1, \omega, \mathtt{1}) = \ell$ for some ℓ such that $S_1(\ell) = \langle v \rangle^{\mathtt{n}}$ for some $\mathtt{n} \in \{\mathtt{m}, {}^*\} \cup \mathbb{N}$ and $S_1 \vdash v \sim \tau$. Then, the functions *read*$(S_1, \omega, \mathtt{1})$ (deduced by Lemma 4.4) and *write*$(S_1, \omega, \bot, \mathtt{1})$ are defined. Thus, we can step with *R-Move*.
- *Base Case* $e_1 \triangleq [\&[\mathtt{mut}]\omega]$. By either *T-ImmBorrow* or *T-MutBorrow*, $\Gamma_1 \vdash \omega : \langle \tau \rangle^{\mathtt{m}}$. Since $S_1 \sim \Gamma_1$, by Lemma 4.3, $loc(S_1, \omega, \mathtt{1})$ is defined. Thus, we can step with *R-Borrow*.
- *Base Case* $e_1 \triangleq [\mathtt{box}(v)]$. For a fresh location $\ell_a \notin dom(S_1)$ we can step with *R-Box*.
- *Base Case* $e_1 \triangleq [\mathtt{trc}(v)]$. For a fresh location $\ell_a \notin dom(S_1)$ we can step with *R-Trc*.
- *Base Case* $e_1 \triangleq [\omega.clone]$. By *T-Clone*, we have $\Gamma_1 \vdash \omega : \langle \blacklozenge \tau \rangle^{\mathtt{m}}$. Since $S_1 \sim \Gamma_1$, by applying Lemma 4.4, *read*$(S_1, \omega, \mathtt{1})$ is defined such that *read*$(S_1, \omega, \mathtt{1}) = \langle \ell_a^{\blacklozenge} \rangle^{\mathtt{m}}$ and $S_1 \vdash \ell_a^{\blacklozenge} \sim \blacklozenge \tau$. Then by *V-Trc* (according to Figure 4.7), $S_1(\ell_a) = \langle v' \rangle^{\mathtt{i}}$ for some value v'. Hence, we can step with *R-Clone*.
- *Base Case* $e_1 \triangleq [\mathtt{let\ mut}\ x = v]$. By *T-Declare*, $x \notin dom(\Gamma_1)$. Since $S_1 \sim \Gamma_1$, then by inspection of Definition 4.24, we have $(dom(S_1) \setminus \mathcal{L}) = \Theta(dom(\Gamma_1))$ where $\mathcal{L}$ represents the set of all heap locations. Since $(dom(S_1) \setminus \mathcal{L})$ returns the locations of the variables existing in S_1 (by removing the heap locations) and $x \notin dom(\Gamma_1)$, we can conclude from the previous equality that $\ell_{1::x} \notin dom(S_1)$. Thus, we can step with *R-Declare*.
- *Base Case* $e_1 \triangleq [\omega = v]$. By hypothesis, $S_1 \sim \Gamma_1$ and by *T-Assign*, $\Gamma_1 \vdash \omega : \langle \tilde{\tau}_1 \rangle^{\mathtt{m}}$ then, applying Lemma 4.4, *read*$(S_1, \omega, \mathtt{1})$ is defined such that *read*$(S_1, \omega, \mathtt{1}) = \langle v_1^{\bot} \rangle^{\mathtt{n}}$ for some $v_1^{\bot}$ and $\mathtt{n}$. Since $S_1 \sim \Gamma_1$, by inspection of Definition 4.24, $v_1^{\bot}$ is a valid value where $S_1 \vdash v_1^{\bot} \sim \tilde{\tau}_1$. Hence, *drop*$(S_1, \{v_1^{\bot}\})$ is defined such that $S = $ *drop*$(S_1, \{v_1^{\bot}\})$. Now, to prove that *write*$(S_1, \omega, v, \mathtt{1})$ is defined, we need to demonstrate that there are no dropped locations between S_1 and S that could prevent the write operation (e.g. consider $S_1 = \{\ell_{1::x} \mapsto \langle \ell_a^{\blacklozenge} \rangle^{\mathtt{1}}, \ell_a \mapsto \langle v \rangle^{\mathtt{1}}\}$, where we have the expression: "$x = x.clone$"; same concept for borrowing). Since by *T-Assign* we have $\neg$*writeProhibited*(Γ_3, ω) and $\neg$*TrcMoveProhibited*(Γ_3, ω), then by inspection of Definitions 4.4 and 4.10, this prevents ω from being borrowed or cloned in the resulting environment Γ_3. Hence, *write*$(S_1, \omega, v, \mathtt{1})$ is defined and we can step with *R-Assign*.
- *Base Case* $e_1 \triangleq [v_1 \oplus v_2]$. The result is immediate and we can step with *R-Arithm*.
- *Base Case* $e_1 \triangleq [v_1 \otimes v_2]$. The result is immediate and we can step with *R-Cond*.
- *Base Case* $e_1 \triangleq [\{v\}^{\mathtt{m}}]$. By hypothesis we have $S_1 \sim \Gamma_1$ and according to the Definition of lifetimes we have $\mathtt{1} \succcurlyeq \mathtt{m}$. By inspection of Definition 4.24, $S_1 \sim \Gamma_1$ implies that for all variables in Γ_1, their location exists in S_1 and their type corresponds to their

run-time value. In addition, S_1 is valid for $\mathtt{l}$. As per Definition 4.19, this indicates that all the locations in the values of S_1 have a unique occurrence, thereby implying that all values in S_1 are valid. Hence, according to Definition 3.4, $\mathit{drop}(S_1, \mathtt{m})$ is defined. Thus we can step with *R-BlockB*.

- *Base Case* $e_1 \triangleq [v; \overline{e}]$. Since $S_1 \sim \Gamma_1$ then, v is a valid value (by inspection of Definition 4.24) and based on Definition 3.4, $\mathit{drop}(S_1, \{v\})$ is defined and we can step with *R-Seq*.

- *Base Case* $e_1 \triangleq [\mathtt{cooperate}]$. The result is immediate and we can step with *R-Cooperate*.

- *Base Case* $e_1 \triangleq [\mathtt{spawn}(f(\overline{v}))]$. The result is immediate and we can step with *R-Spawn*.

However, we now proceed under the assumption that e_2 is not a value v as follows:

- *Inductive Case* $e_1 \triangleq [\mathtt{box}(e_2)]$. In this case, we can decompose our expression e_2 into the evaluation context $\mathtt{box}(E)$ and redex e_2. Then, if e_2 is a value, we apply the base case. If it is not, by applying our induction hypothesis to the typing derivation for e_2, we know that e_2 steps to some e_3 and we can step with *R-Sub*.

- *Inductive Case* $e_1 \triangleq [\mathtt{trc}(e_2)]$. In this case, we can decompose our expression e_2 into the evaluation context $\mathtt{trc}(E)$ and redex e_2. Then, if e_2 is a value, we apply the base case. If it is not, by applying our induction hypothesis to the typing derivation for e_2, we know that e_2 steps to some e_3 and we can step with *R-Sub*.

- *Inductive Case* $e_1 \triangleq [\mathtt{let\ mut}\ x = e_2]$. In this case, we can decompose our expression e_2 into the evaluation context $\mathtt{let\ mut}\ x = E$ and redex e_2. Then, if e_2 is a value, we apply the base case. If it is not, by applying our induction hypothesis to the typing derivation for e_2, we know that e_2 steps to some e_3 and we can step with *R-Sub*.

- $e_1 \triangleq [\omega = e_2]$. In this case, we can decompose our expression e_2 into the evaluation context $\omega = E$ and redex e_2. Then, if e_2 is a value, we apply the base case. If it is not, by applying our induction hypothesis to the typing derivation for e_2, we know that e_2 steps to some e_3 and we can step with *R-Sub*.

- $e_1 \triangleq [e_2 \oplus e_3]$. In this case, we can decompose our expression e_2 into the evaluation context $E \oplus e_3$ and redex e_2. Then, if e_2 is a value, we apply the base case. If it is not, by applying our induction hypothesis to the typing derivation for e_2, we know that e_2 steps to some e' and we can step with *R-Sub*.

- $e_1 \triangleq [e_2 \otimes e_3]$. In this case, we can decompose our expression e_2 into the evaluation context $E \otimes e_3$ and redex e_2. Then, if e_2 is a value, we apply the base case. If it is not, by applying our induction hypothesis to the typing derivation for e_2, we know that e_2 steps to some e' and we can step with *R-Sub*.

- $e_1 \triangleq [\{e_2\}^{\mathtt{m}}]$. In this case, we can decompose our expression e_2 into the evaluation context $\{E\}^{\mathtt{m}}$ for some lifetime $\mathtt{l} \succeq \mathtt{m}$ and redex e_2. Then, if e_2 is a value, we apply the

base case. If it is not, by applying our induction hypothesis to the typing derivation for e_2, we know that e_2 steps to some e_3 and we can step with *R-BlockA*.

- $e_1 \triangleq [\overline{e}]$. In this case, we can decompose our expression e into the evaluation context $E; \overline{e}$ (i.e. $e; \overline{e}$) and redex e (i.e. E). Then, if e is a value, we apply the base case. If it is not, by applying our induction hypothesis to the typing derivation for e, we know that e steps to some e' and we can step with *R-Sub*.
- *Inductive Case* $e_1 \triangleq [\texttt{spawn}(f(e_2))]$, $e_2 \neq [v]$. We will proceed based on whether each expression $e_i \in e_2$ is a value or not. If it is, we apply the base case. Therefore, we decompose our expression into the evaluation context $e_2 \triangleq [\texttt{spawn}(f(v_1, \ldots, v_n, e_i, e_1, \ldots, e_n)]$ (i.e. $\texttt{spawn}(f(v_1, \ldots, v_n, E, e_1, \ldots, e_n))$ and redex e_i. By applying our induction hypothesis to e_i, we know that e_i steps to some e_i'. This satisfies our requirement since we can plug e_i' back into our evaluation context. Thus, we can step with *R-Sub*.

□

As already discussed, the Step Progress Lemma implies that if a thread's expression is *well-typed*, it can execute a step. Therefore, we introduce the Step Preservation Lemma for a *well-typed* expression e as follows:

Lemma 4.10 (Step Preservation) *Let* $T_1 = \{t_i, \{e_i\}^{\mathtt{l_i}} \mid 1 \leqslant i \leqslant N\}$, S_1, σ *such that* T_1, S_1 *is a valid global state and for all* $i \in \{1, ..., N\}$ *we have* $S_1 \triangleright e_i \vdash \sigma$, $S_{1|\mathtt{l_i}} \sim \Gamma_1^i$ *and* $\Gamma_1^i \vdash \langle e_i : \tau_i \rangle_\sigma^{\mathtt{l_i}} \dashv \Gamma_2^i$ *for some* Γ_1^i, Γ_2^i *and* τ_i. *Let* $i \in \{1, ..., N\}, T$ *and* $S_2 \triangleright e_i'$ *such that* $\langle S_1 \triangleright e_i \xrightarrow{T}_i S_2 \triangleright e_i' \rangle^{\mathtt{l_i}}$ *then, there exists* Γ' *and* σ' *such that* $T_{1\setminus t} \cup (t, \{e_i'\}^{\mathtt{l_i}}) \cup T, S_2$ *remains valid and* $\Gamma' \vdash \langle e_i' : \tau_i \rangle_{\sigma'}^{\mathtt{l_i}} \dashv \Gamma_2^i$ *and* $S_2 \triangleright e_i' \vdash \sigma'$ *and* $S_{2|\mathtt{l_i}} \sim \Gamma'$ *and for all* $j \in \{1, ..., i-1, i+1, .., N\}$ *we have* $\Gamma_1^j \vdash \langle e_j : \tau_j \rangle_{\sigma'}^{\mathtt{l_j}} \dashv \Gamma_2^j$ *and* $S_2 \triangleright e_j \vdash \sigma'$ *and* $S_{2|\mathtt{l_j}} \sim \Gamma_1^j$ *and for all* $(t, \{e\}^{\mathtt{l}}) \in T$ *we have* $\Gamma_1 \vdash \langle e : \tau \rangle_{\sigma'}^{\mathtt{l}} \dashv \Gamma_2$ *and* $S_2 \triangleright e \vdash \sigma'$ *and* $S_{2|\mathtt{l}} \sim \Gamma_1$ *for some* Γ_1, Γ_2 *and* τ.

The preservation Lemma states that for a given valid global state T_1, S_1, where all $(t, \{e\}^{\mathtt{l}}) \in T_1$ are *well-typed* under σ, if for a given thread $(t, \{e_i\}^{\mathtt{l_i}}) \in T_1$, we reduce e_i to e_i', resulting in a new state S_2. Then, there exists an intermediate typing environment Γ' and a store typing σ' such that the program store S_2 is safely abstracted with Γ'. The new global state $T_{1\setminus t} \cup (t, \{e_i'\}^{\mathtt{l_i}}) \cup T, S_2$ remains valid, and the new expression e_i' preserves the type τ_i under σ'. Furthermore, all existing threads in $T_{1\setminus t} \cup T$ remain *well-typed* under σ'. Finally, to prove Lemma 4.10, we split it up into three sub-lemmas as follows:

Firstly, Lemma 4.11 guarantees that when a thread executes a single step, it preserves the validity of the global state of the program store (Definition 4.21). This means that the execution of a thread does not violate the valid local state of other threads. In other words, it ensures that its execution does not create aliases in the memory of other threads:

Lemma 4.11 *(Step Alias Preservation) Let* T_1, S_1 *be a valid global state. For a given* $(t, \{e\}^{\mathtt{l}}) \in T_1$, *let* σ *be a store typing such that* $S_1 \triangleright e \vdash \sigma$ *and let* Γ_1 *be a well formed typing environ-*

ment with respect to $\mathtt{1}$ *such that* $S_{1|\mathtt{1}} \sim \Gamma_1$ *and* $\Gamma_1 \vdash \langle e : \tau \rangle^{\mathtt{1}}_{\sigma} \dashv \Gamma_2$ *for some* τ *and* Γ_2. *If* $\langle S_1 \triangleright e \xrightarrow{T}_i S_2 \triangleright e' \rangle^{\mathtt{1}}$ *for some* T *and* $S_2 \triangleright e'$, *then* $T_{1 \setminus t} \cup (t, \{e'\}^{\mathtt{1}}) \cup T, S_2$ *remains valid.*

`Proof.` By structural induction on the forms of e according to Figure 3.5. For each case and by inspection of Definition 4.21, the following must be demonstrated: (1) S_2 remains valid for $\mathtt{1}$. (2) e' remains valid. (3) $S_2 \triangleright e'$ remains valid for $\mathtt{1}$. (4) $T_{1 \setminus t} \cup (t, \{e'\}^{\mathtt{1}}) \cup T, S_2$ remains valid:

- *Base Case* e=$[v]$. The result is immediate as there is no reduction rule for this case.
- *Base Case* e=$[\hat{\omega}]$. By *R-Copy* we have $\mathit{read}(S_1, \omega, \mathtt{1}) = \langle v \rangle^{\mathtt{m}}$. By hypothesis, S_1 is valid for $\mathtt{1}$. This means that all locations in the values of $S_{1|\mathtt{1}}$ have a unique occurrence by inspection of Definition 4.19, and since v is in $S_{1|\mathtt{1}}$, v is valid. Moreover, S_2 is valid for $\mathtt{1}$ since $S_1 = S_2$ by *R-Copy*. By *T-Copy*, $\mathit{copy}(\tau)$ means that: neither $v \triangleq [\ell^{\blacksquare}_a]$ nor $v \triangleq [\ell^{\blacklozenge}_b]$ can hold, then $S_2 \triangleright v$ is valid for $\mathtt{1}$, and also $S_2 \triangleright v$ cannot violate the validity of the global state since v cannot be a `box` or a `trc` location value. Hence, $T_{1 \setminus t} \cup (t, \{v\}^{\mathtt{1}}) \cup T, S_2$ remains valid.
- *Base Case* e=$[\omega]$. By *R-Move*, we have $\mathit{read}(S_1, \omega, \mathtt{1}) = \langle v \rangle^{\mathtt{m}}$. By hypothesis, S_1 is valid for $\mathtt{1}$. This means that all locations in the values of $S_{1|\mathtt{1}}$ have a unique occurrence by inspection of Definition 4.19, and since v is in $S_{1|\mathtt{1}}$, v is valid. By *R-Move* we have $S_2 = \mathit{write}(S_1, \omega, \bot, \mathtt{1})$, which implies that the values of $S_{2|\mathtt{1}}$, except for v, are the same values of $S_{1|\mathtt{1}}$ and therefore any location within the values of $S_{2|\mathtt{1}}$ has only one occurrence, so S_2 is valid for $\mathtt{1}$. Next, we need to establish the validity of $S_2 \triangleright v$. We suppose that $v = \ell^{\blacksquare}_a$ and $\ell^{\blacksquare}_a$ in $S_{2|\mathtt{1}}$. Then, $\ell^{\blacksquare}_a$ in $S_{2|\mathtt{1}}$ implies that $\ell^{\blacksquare}_a$ in $S_1[\![\ell \rightarrow \langle \bot \rangle^{\mathtt{m}}]\!]$ (i.e. $\ell = \mathit{loc}(S_1, \omega, \mathtt{1})$) and thus $\ell^{\blacksquare}_a$ appears with two occurrences in S_1, which is in contradiction with our hypothesis. Consequently, $S_2 \triangleright v$ is valid. Finally, we need to prove that $T_{1 \setminus t} \cup (t, \{v\}^{\mathtt{1}}) \cup T, S_2$ remains valid. In such a case, ω can take two different forms: (1) If $(\omega = \epsilon \mid x)$, we suppose that $v = \ell^{\blacksquare}_a$, by hypothesis we have T_1, S_1 is valid and by inspection of Definition 4.21, we know that $\ell^{\blacksquare}_a$ appears only once in S_1 (i.e. since the path of ω is ϵ), so $T_{1 \setminus t} \cup (t, \{v\}^{\mathtt{1}}) \cup T, S_2$ remains valid. (2) If $(\omega = (\pi.*) \mid x)$ and $v = \ell^{\blacksquare}_a$, then according to the *T-Move* rule and the *move* function (Definition 4.8), the type of x cannot be an active `Trc` type since we cannot move out of an active `Trc`. Therefore, in this case, $\nexists\, \ell^{\blacklozenge}_b$ such that $S \vdash_u \ell_b \rightsquigarrow \ell_a$ where there is only one occurence of $\ell^{\blacksquare}_a$. For the other cases ($v = \ell^{\blacklozenge}_a$, $v = \ell^{\lozenge}_a$ or $v = \ell^{\circ}$), we have S_1 is valid for $\mathtt{1}$ and it follows that v is not in $S_{2|\mathtt{1}}$. Hence, $T_{1 \setminus t} \cup (t, \{v\}^{\mathtt{1}}) \cup T, S_2$ remains valid.
- *Base Case* e=$[\&[\mathtt{mut}]\omega]$. By *R-Borrow*, $v = \ell^{\circ}$ for some $\ell \in \mathit{dom}(S_1)$. Since ℓ is a valid location, v is valid. By hypothesis, S_1 is valid for $\mathtt{1}$. This means that all locations in the values of $S_{1|\mathtt{1}}$ have a unique occurrence by inspection of Definition 4.19. Since $S_1 = S_2$ by *R-Borrow*, S_2 is valid for $\mathtt{1}$. Moreover, as $v = \ell^{\circ}$ (i.e. $v \neq \ell^{\blacklozenge}_a$ and $v \neq \ell^{\blacksquare}_a$), $S_2 \triangleright \ell^{\circ}$ is valid for $\mathtt{1}$. Finally, we can deduce in this case that $T_{1 \setminus t} \cup (t, \{\ell^{\circ}\}^{\mathtt{1}}) \cup T, S_2$ remains valid.
- *Base Case* e=$[\mathtt{box}(v_2)]$. By hypothesis, $S_1 \triangleright \mathtt{box}(v_2)$ is valid for $\mathtt{1}$ and by inspection of Definition 4.20 this indicates that all locations in the values of $S_{1|\mathtt{1}}$ are not within

the values of $\texttt{box}(v_2)$. Then, we can deduce that $S_1 \triangleright v_2$ is valid for $\texttt{l}$. In addition, based on Definition 4.20, S_1 is valid for $\texttt{l}$ and v_2 is valid. By *R-Box*, $S_2 = S_1[\ell_a \mapsto \langle v_2 \rangle^*]$ for some $\ell_a \notin dom(S_1)$. Since both S_1 and v_2 are valid then $S_1[\ell_a \mapsto \langle v_2 \rangle^*]$ remains valid for $\texttt{l}$, and thus S_2 is valid for $\texttt{l}$. Furthermore, by *R-Box*, $v = \ell_a^{\blacksquare}$ and since $\texttt{box}(v_2)$ is valid, v is valid. Next, we need to establish the validity of $S_2 \triangleright v$. We suppose that $v = \ell_a^{\blacksquare}$ and $\ell_a^{\blacksquare} \in S_2$, the latter implies that $\ell_a^{\blacksquare} \in S_1[\ell_a \mapsto \langle v_2 \rangle^*]$, which is not possible since with *R-Box* we create a new location ℓ_a and accordingly $\ell_a^{\blacksquare} \notin S_2$. Then, $S_2 \triangleright v$ is valid for $\texttt{l}$. Finally, $\ell_a^{\blacksquare} \notin S_2$ and therefore there is only one occurrence of $\ell_a^{\blacksquare}$ in $T_{1\setminus t}, T_2 \cup (t, \{v\}^1) \cup T, S_2$, so the latter remains valid.

- *Base Case* e=$[\texttt{trc}(v_2)]$. By hypothesis, $S_1 \triangleright \texttt{trc}(v_2)$ is valid for $\texttt{l}$ and by inspection of Definition 4.20 this indicates that all locations in the values of $S_{1|\texttt{l}}$ are not within the values of $\texttt{trc}(v_2)$. Then, we can deduce that $S_1 \triangleright v_2$ is valid. Additionally, based on Definition 4.20, S_1 is valid for $\texttt{l}$ and v_2 is valid for $\texttt{l}$. By *R-Trc*, $S_2 = S_1[\ell_a \mapsto \langle v_2 \rangle^1]$ for some $\ell_a \notin dom(S_1)$. Since both S_1 and v_2 are valid then $S_1[\ell_a \mapsto \langle v_2 \rangle^1]$ remains valid for $\texttt{l}$ and thus S_2 is valid for $\texttt{l}$. Furthermore, by *R-Trc*, $v = \ell_a^{\blacklozenge}$ and since $\texttt{trc}(v_2)$ is valid, v is valid. Next, we need to establish the validity of $S_2 \triangleright v$. We suppose that $v = \ell_a^{\blacklozenge}$ and $\ell_a^{\blacklozenge} \in S_2$, the latter implies that $\ell_a^{\blacklozenge} \in S_1[\ell_a \mapsto \langle v_2 \rangle^1]$, which is not possible since, with *R-Trc*, we create a new location ℓ_a and accordingly $\ell_a^{\blacklozenge} \notin S_2$. Afterwards, $S_2 \triangleright v$ is valid for $\texttt{l}$. Finally, we need to prove that $T_{1\setminus t} \cup (t, \{v\}^1) \cup T, S_2$ remains valid. If $v_2 = [\ell_b^{\blacksquare}]$, by hypothesis, T_1, S_1 is valid, meaning that if there are more than one occurrence of $\ell_b^{\blacksquare}$ in T_1, S_1 then there is a unique $\ell_c^{\blacklozenge}$ such that $S_1 \vdash_u \ell_c \rightsquigarrow \ell_b$. This designates that $\ell_b^{\blacksquare}$ as a value is moved out of an $\texttt{Trc}$, which is not possible, since our type system prevents moving out of an $\texttt{Trc}$. As a result, there is only one occurrence of $\ell_b^{\blacksquare}$, which is v_2 such that $S_2 \vdash_u \ell_a \rightsquigarrow \ell_b$. Moreover, v_2 cannot be either $\ell_b^{\blacklozenge}$ or $\ell_b^{\diamond}$, which is prohibited by the *containsTrc* function 4.9 and if $v_2 = \ell^{\circ}$ this does not invalidate the validity of the global state according to Definition 4.21. Therefore, $T_{1\setminus t}$, $T_2 \cup (t, \{v\}^1) \cup T, S_2$ remains valid.

- *Base Case* e=$[\omega.clone]$. By *R-Clone*, we have $read(S_1, \omega, \texttt{l}) = \langle \ell_a^{\blacklozenge} \rangle^{\texttt{m}}$ for some $\ell_a \in dom(S_1)$ where $v = \ell_a^{\diamond}$. By hypothesis, S_1 is valid for $\texttt{l}$ this means that all locations in the values of $S_{1|\texttt{l}}$ have a unique occurrence by inspection of Definition 4.19 and since $\ell_a^{\blacklozenge} \in S_1$ then $\ell_a^{\blacklozenge}$ is valid. Therefore $\ell_a^{\diamond}$ is valid. By *R-Clone*, $S_1 = S_2$, so S_2 is valid for $\texttt{l}$. As $v = \ell_a^{\diamond}$ (i.e. $v \neq \ell_a^{\blacklozenge}$ and $v \neq \ell_a^{\blacksquare}$), $S_2 \triangleright \ell_a^{\diamond}$ is valid for $\texttt{l}$. Thus, we can deduce that $T_{1\setminus t} \cup (t, \{\ell_a^{\diamond}\}^1) \cup T, S_2$ remains valid.

- *Base Case* e=$[\texttt{let mut } x = v_2]$. By hypothesis, $S_1 \triangleright \texttt{let mut } x = v_2$ is valid for $\texttt{l}$. By inspection of Definition 4.20, $\texttt{let mut } x = v_2$ is valid (hence v_2 is valid) and S_1 is valid for $\texttt{l}$. According to Definition 4.20, v_2 is not in $S_{1|\texttt{l}}$. Afterwards, adding v_2 to S_1 cannot invalidate S_1. By *R-Declare*, $S_2 = S_1[\ell_{1::x} \mapsto \langle v_2 \rangle^1]$ where $\ell_{1::x} \notin dom(S_1)$, so S_2 is valid for $\texttt{l}$. By *R-Declare*, $v = \epsilon$, thus $S_2 \triangleright \epsilon$ is valid for $\texttt{l}$. Finally, we need to prove that $T_{1\setminus t} \cup (t, \{v\}^1) \cup T, S_2$ remains valid. If $v_2 = [\ell_b^{\blacksquare}]$, by hypothesis, T_1, S_1 is valid, meaning that if there are more than one occurrence of $\ell_b^{\blacksquare}$ in T_1, S_1 then there is a unique $\ell_c^{\blacklozenge}$ such that $S \vdash_u \ell_c \rightsquigarrow \ell_b$. This designates that $\ell_b^{\blacksquare}$ as a value is moved out of an $\texttt{Trc}$, which is not possible, since our type system prevents moving out of an $\texttt{Trc}$. As a result, there is only one occurrence of $\ell_b^{\blacksquare}$, which is v_2. For the other cases ($v_2 = \ell_b^{\blacklozenge}$, $v_2 = \ell_b^{\diamond}$ or $v_2 = \ell^{\circ}$), we know that v_2 is not in $S_{1|\texttt{l}}$ and adding v_2 to S_1

does not invalidate the validity of the global state. Therefore $T_{1\setminus t} \cup (t, \{v\}^{\mathtt{l}}) \cup T, S_2$ remains valid.

- *Base Case* e=$[\omega = v_2]$. By hypothesis, $S_1 \triangleright \omega = v_2$ is valid for $\mathtt{l}$ and by inspection of Definition 4.20, S_1 is valid for $\mathtt{l}$ and $\omega = v_2$ is also valid. Moreover, $S_1 \triangleright \omega = v_2$ indicates that all locations in the values of $S_{1|\mathtt{l}}$ are not within the values of $\omega = v_2$ and then we can deduce that $S_1 \triangleright v_2$ is valid for $\mathtt{l}$. By *R-Assign*, we have $S_2 = \mathit{write}(S_1, \omega, v_2, \mathtt{l})$ then, all the values of $S_{2|\mathtt{l}}$, except for v_2, are the same values of $S_{1|\mathtt{l}}$ and therefore any location in the values of $S_{2|\mathtt{l}}$ has only one occurrence. Since $S_1 \triangleright v_2$ is valid for $\mathtt{l}$, adding v_2 to S_1 does not invalidate S_1 and so $S_2 = S_1[\![\ell \rightarrow \langle v_2 \rangle^{\mathtt{l}}]\!]$ is valid for $\mathtt{l}$. Therefore, by *R-Assign*, $v = \epsilon$ thus $S_2 \triangleright \epsilon$ is valid for $\mathtt{l}$. Finally, we need to prove that $T_{1\setminus t} \cup (t, \{v\}^{\mathtt{l}}) \cup T, S_2$ remains valid. If $v_2 = [\ell_b^{\blacksquare}]$, by hypothesis, T_1, S_1 is valid, meaning that if there are more than one occurrence of $\ell_b^{\blacksquare}$ in T_1, S_1 then there is a unique $\ell_c^{\blacklozenge}$ such that $S \vdash_u \ell_c \rightsquigarrow \ell_b$. This designates that $\ell_b^{\blacksquare}$ as a value is moved out of an `Trc`, which is not possible, since our type system prevents moving out of an `Trc`. As a result, there is only one occurrence of $\ell_b^{\blacksquare}$, which is v_2. For the other cases ($v_2 = \ell_b^{\blacklozenge}$, $v_2 = \ell_b^{\diamond}$ or $v_2 = \ell^{\circ}$), we know that v_2 is not in $S_{1|\mathtt{l}}$ and adding v_2 to S_1 does not invalidate the validity of the global state. Therefore $T_{1\setminus t} \cup (t, \{v\}^{\mathtt{l}}) \cup T, S_2$ remains valid.

- *Base Case* $e_1 \triangleq [v_1 \oplus v_2]$. By hypothesis, $S_1 \triangleright v_1 \oplus v_2$ is valid for $\mathtt{l}$ and by inspection of Definition 4.20, S_1 is valid for $\mathtt{l}$ and $v_1 \oplus v_2$ is also valid. By *R-Arithm*, $v_3 = v_1 \oplus v_2$ then we can deduce that v_3 is valid. Additionally, by *R-Arithm*, $S_1 = S_2$ and since S_1 is valid for $\mathtt{l}$, S_2 is valid for $\mathtt{l}$. Moreover, $S_2 \triangleright v_3$ is valid for $\mathtt{l}$ since $S_1 \triangleright v_3$ is valid for $\mathtt{l}$ (by hypothesis). Finally, we need to prove that $T_{1\setminus t} \cup (t, \{v_3\}^{\mathtt{l}}) \cup T, S_2$ remains valid. by *T-Arithm*, we have $\Gamma_1 \vdash \langle e_1 : \mathtt{int} \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2$ and $\Gamma_2 \vdash \langle e_2 : \mathtt{int} \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_3$. This means that neither $v_3 = \ell_a^{\blacksquare}$ nor $v_3 = \ell_a^{\blacklozenge}$ can hold. Hence we can deduce that $T_{1\setminus t} \cup (t, \{v_3\}^{\mathtt{l}}) \cup T, S_2$ is valid.

- *Base Case* $e_1 \triangleq [v_1 \otimes v_2]$. By hypothesis, $S_1 \triangleright v_1 \otimes v_2$ is valid for $\mathtt{l}$ and by inspection of Definition 4.20, S_1 is valid for $\mathtt{l}$ and $v_1 \otimes v_2$ is also valid. By *R-Cond*, $v_3 = v_1 \oplus v_2$ then we can deduce that v_3 is valid. Additionally, by *R-Cond*, $S_1 = S_2$ and since S_1 is valid for $\mathtt{l}$, then, S_2 is valid for $\mathtt{l}$. Moreover, $S_2 \triangleright v_3$ is valid for $\mathtt{l}$ since $S_1 \triangleright v_3$ is valid for $\mathtt{l}$ (by hypothesis). Finally, we need to prove that $T_{1\setminus t} \cup (t, \{v_3\}^{\mathtt{l}}) \cup T, S_2$ remains valid. by *T-Cond*, we have $\Gamma_1 \vdash \langle v_1 \otimes v_2 : \mathtt{bool} \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2$. This means that neither $v_3 = \ell_a^{\blacksquare}$ nor $v_3 = \ell_a^{\blacklozenge}$ can hold. Hence we can deduce that $T_{1\setminus t} \cup (t, \{v_3\}^{\mathtt{l}}) \cup T, S_2$ is valid.

- *Base Case* e=$[\{v\}^{\mathtt{m}}]$. By hypothesis, $S_1 \triangleright \{v\}^{\mathtt{m}}$ is valid for $\mathtt{l}$ and by inspection of Definition 4.20, S_1 is valid for $\mathtt{l}$ and $\{v\}^{\mathtt{m}}$ is valid then, we can deduce that v is valid. By *R-BlockB* and for some $\mathtt{l} \succeq \mathtt{m}$, $S_2 = \mathit{drop}(S_1, \mathtt{m})$. According to Definition 3.4, S_2 contains all the values in S_1 except those owned in lifetime $\mathtt{m}$. Since S_1 is valid for $\mathtt{l}$ then S_2 remains valid for $\mathtt{l}$. Next, we need to establish the validity of $S_2 \triangleright v$. Since we have $S_1 \triangleright v$ is valid for $\mathtt{l}$ and $S_2 = \mathit{drop}(S_1, \mathtt{m})$ by *R-Block* thus, we can deduce that $S_2 \triangleright v$ is valid for $\mathtt{l}$. Finally, we need to prove that $T_{1\setminus t} \cup (t, \{v\}^{\mathtt{l}}) \cup T, S_2$ remains valid. If $v = [\ell_b^{\blacksquare}]$, by hypothesis, T_1, S_1 is valid, meaning that if there are more than one occurrence of $\ell_b^{\blacksquare}$ in T_1, S_1 then there is a unique $\ell_c^{\blacklozenge}$ such that $S \vdash_u \ell_c \rightsquigarrow \ell_b$. This

designates that $\ell_b^{\blacksquare}$ as a value is moved out of an `Trc`, which is not possible, since our type system prevents moving out of an `Trc`. As a result, there is only one occurrence of $\ell_b^{\blacksquare}$, which is v. For the other cases ($v = \ell_b^{\blacklozenge}$, $v = \ell_b^{\diamond}$ or $v = \ell^{\circ}$), we know that v is not in $S_{2|1}$. Therefore $T_{1\setminus t} \cup (t, \{v\}^1) \cup T, S_2$ remains valid.

- *Base Case* e=$[v; \overline{e}]$. By hypothesis, $S_1 \triangleright v; \overline{e}$ is valid for `1` and by inspection of Definition 4.20, S_1 is valid for `1` and $v; \overline{e}$ is valid. Intuitively, the validity of $v; \overline{e}$ infers that v and $\overline{e}$ are valid. By *R-Seq* we have $S_2 = \mathit{drop}(S_1, \{v\})$ then if $v = [\ell_a^{\blacksquare}]$ or $v = [\ell_b^{\blacklozenge}]$, according to Definition 3.4, the *drop* function traverses that location, dropping its values recursively. Moreover, $S_2 = \mathit{drop}(S_1, \{v\})$ means that all the values of S_2, except for v, are in S_1 and since S_1 is valid for `1` then S_2 remains valid for `1`. Finally, given that $S_1 \triangleright \overline{e}$ is valid for `1`, $S_2 \triangleright \overline{e}$ is valid for `1`. Therefore, $T_{1\setminus t} \cup (t, \{\overline{e}\}^1) \cup T, S_2$ remains valid.

- *Base Case* e=$[\texttt{cooperate}]$. By *R-Cooperate*, $S_1 = S_2$ then, the result is immediate. Hence, by *R-Cooperate* $v \triangleq [\epsilon]$ and $T_{1\setminus t} \cup (t, \{\epsilon\}^1) \cup T, S_2$ remains valid.

- *Base Case* e=$[\texttt{spawn}(f(\overline{v}))]$. By hypothesis, $S_1 \triangleright \texttt{spawn}(f(\overline{v}))$ is valid for `1` and by inspection of Definition 4.20, S_1 is valid for `1` and $\texttt{spawn}(f(\overline{v}))$ is valid and also implies that all locations in the values of $S_{1|1}$ are not within the values of $\texttt{spawn}(f(\overline{v}))$. By *R-Spawn*, we have for $t_1 \in \mathit{fresh}$, let $* \geq 1_1$ (indicating a fresh lifetime included in the global lifetime, *) such that $S_2 = S' \cup [\overline{\ell_{1_1::x} \mapsto \langle v' \rangle^{1_1}}]$ where $\overline{(S', v') = \mathit{activate}(S_1, v)}$. Here, $[\overline{\ell_{1_1::x} \mapsto \langle v' \rangle^{1_1}}]$ represents a new stack for the new thread t_1 restricted to the lifetime 1_1 (i.e. $(t_1, \{\overline{e}\}^{1_1})$). Consequently, by *R-Spawn*, $S_{1|1} = S_{2|1}$ such that S_2 is valid for `1` since S_1 is valid for `1`. Therefore, we can state that $S_2 \triangleright \epsilon$ is valid for `1`. Finally, we need to prove that $T_{1\setminus t} \cup (t, \{\epsilon\}^1) \cup T, S_2$ remains valid. To demonstrate global validity, it is necessary first to prove that the local state of the new thread is valid. By *R-Spawn*, we have $\overline{(S', v') = \mathit{activate}(S_1, v)}$, and according to Definition 3.6, this function recursively switches the values in $\overline{v}$ from the form $(\ell_a^{\diamond})$ to $(\ell_a^{\blacklozenge})$. Additionally, by *T-Spawn* we have the following mechanism: $\Gamma_2 \vdash (\overline{S}) \Longleftarrow (\overline{\tau})$ that ensures: (1) the locations in $\overline{v}$ carrying active `Trc` type cannot exist (based on Definition 4.15) as well as (2) two locations having the same inactive `Trc` type should not exist, thus avoiding the creation of aliases via `Trc` in the local state of the new thread. Adhering to these constraints, we conclude that all locations in $\overline{v'}$ have a unique occurrence and therefore S_2 is valid for 1_1. Hence, $S_2 \triangleright \overline{e}$ is valid for 1_1, $\overline{e}$ as an expression does not contain a location values. Finally, since all locations in $\overline{v'}$ are in $\overline{v}$ and, specifically, all $\ell_a^{\blacksquare}$ in $\overline{v'}$ are present in $\overline{v}$, adding $\overline{v'}$ to S' does not violate the validity of the global state. Therefore, we deduce that $T_{1\setminus t} \cup (t, \{\epsilon\}^1) \cup T, S_2$ is valid.

However, we now proceed under the assumption that e_1 is not a value v as follows:

- *Inductive Case* $e_1 \triangleq [\texttt{box}(e)]$. By *T-Box*, $\Gamma_1 \vdash \langle e : \tau \rangle_{\sigma}^{1} \dashv \Gamma_2$ and by applying our induction hypothesis over the derivation of e, we know that $\langle S_1 \triangleright e \xrightarrow{T}_i S_2 \triangleright e' \rangle^1$ such that e' is valid, S_2 is valid for `1`, $S_2 \triangleright e'$ is valid for `1`, and $T_{1\setminus t} \cup (t, \{e'\}^1) \cup T, S_2$ is valid. Therefore, $\texttt{box}(e')$ is valid, $S_2 \triangleright \texttt{box}(e')$ is valid for `1` and $T_{1\setminus t} \cup (t, \{\texttt{box}(e')\}^1) \cup T, S_2$ is valid.

- *Inductive Case* $e_1 \triangleq [\texttt{trc}(e)]$. By *T-Trc*, $\Gamma_1 \vdash \langle e : \tau \rangle_\sigma^{\texttt{l}} \dashv \Gamma_2$ and by applying our induction hypothesis over the derivation of e, we know that $\langle S_1 \triangleright e \xrightarrow{T}_i S_2 \triangleright e' \rangle^{\texttt{l}}$ such that e' is valid, S_2 is valid for $\texttt{l}$, $S_2 \triangleright e'$ is valid for $\texttt{l}$, and $T_{1 \setminus t} \cup (t, \{e'\}^{\texttt{l}}) \cup T, S_2$ is valid. Therefore, $\texttt{trc}(e')$ is valid, $S_2 \triangleright \texttt{trc}(e')$ is valid for $\texttt{l}$ and $T_{1 \setminus t} \cup (t, \{\texttt{trc}(e')\}^{\texttt{l}}) \cup T, S_2$ is valid.

- *Inductive Case* $e_1 \triangleq [e_1 \oplus e_2]$. By *T-Arithm*, $\Gamma_1 \vdash \langle e_1 : \tau_1 \rangle_\sigma^{\texttt{l}} \dashv \Gamma_2$ and $\Gamma_2 \vdash \langle e_2 : \tau_2 \rangle_\sigma^{\texttt{l}} \dashv \Gamma_3$. By applying our induction hypothesis over the derivation of e_1, we know that $\langle S_2 \triangleright e_1 \xrightarrow{T}_i S_2 \triangleright e' \rangle^{\texttt{l}}$ such that e' is valid, S_2 is valid for $\texttt{l}$, $S_2 \triangleright e'$ is valid for $\texttt{l}$, and $T_{1 \setminus t} \cup (t, \{e'\}^{\texttt{l}}) \cup T, S_2$ is valid. Therefore, $e' \oplus e_2$ is valid, $S_2 \triangleright e' \oplus e_2$ is valid for $\texttt{l}$ and $T_{1 \setminus t} \cup (t, \{e' \oplus e_2\}^{\texttt{l}}) \cup T, S_2$ is valid.

- *Inductive Case* $e_1 \triangleq [e_1 \otimes e_2]$. By *T-Cond*, $\Gamma_1 \vdash \langle e_1 : \tau_1 \rangle_\sigma^{\texttt{l}} \dashv \Gamma_2$ and $\Gamma_2 \vdash \langle e_2 : \tau_2 \rangle_\sigma^{\texttt{l}} \dashv \Gamma_3$. By applying our induction hypothesis over the derivation of e_1, we know that $\langle S_2 \triangleright e_1 \xrightarrow{T}_i S_2 \triangleright e' \rangle^{\texttt{l}}$ such that e' is valid, S_2 is valid for $\texttt{l}$, $S_2 \triangleright e'$ is valid for $\texttt{l}$, and $T_{1 \setminus t} \cup (t, \{e'\}^{\texttt{l}}) \cup T, S_2$ is valid. Therefore, $e' \otimes e_2$ is valid, $S_2 \triangleright e' \otimes e_2$ is valid for $\texttt{l}$ and $T_{1 \setminus t} \cup (t, \{e' \otimes e_2\}^{\texttt{l}}) \cup T, S_2$ is valid.

- *Inductive Case* $e_1 \triangleq [\texttt{let mut } x = e]$. By *T-Declare*, $\Gamma_1 \vdash \langle e : \tau \rangle_\sigma^{\texttt{l}} \dashv \Gamma_2$ and by applying our induction hypothesis over the derivation of e, we know that $\langle S_1 \triangleright e \xrightarrow{T}_i S_2 \triangleright e' \rangle^{\texttt{l}}$ such that e' is valid, S_2 is valid for $\texttt{l}$, $S_2 \triangleright e'$ is valid for $\texttt{l}$, and $T_{1 \setminus t} \cup (t, \{e'\}^{\texttt{l}}) \cup T, S_2$ is valid. Therefore, $\texttt{let mut } x = e'$ is valid, $S_2 \triangleright \texttt{let mut } x = e'$ is valid for $\texttt{l}$ and $T_{1 \setminus t} \cup (t, \{\texttt{let mut } x = e'\}^{\texttt{l}}) \cup T, S_2$ is valid.

- $e_1 \triangleq [\omega = e]$. By *T-Assign*, $\Gamma_1 \vdash \langle e : \tau \rangle_\sigma^{\texttt{l}} \dashv \Gamma_2$ and by applying our induction hypothesis over the derivation of e, we know that $\langle S_1 \triangleright e \xrightarrow{T}_i S_2 \triangleright e' \rangle^{\texttt{l}}$ such that e' is valid, S_2 is valid for $\texttt{l}$, $S_2 \triangleright e'$ is valid for $\texttt{l}$, and $T_{1 \setminus t} \cup (t, \{e'\}^{\texttt{l}}) \cup T, S_2$ is valid. Therefore, $\omega = e'$ is valid, $S_2 \triangleright \omega = e'$ is valid for $\texttt{l}$ and $T_{1 \setminus t} \cup (t, \{\omega = e'\}^{\texttt{l}}) \cup T, S_2$ is valid.

- $e_1 \triangleq [\{e\}^{\texttt{m}}]$. In this case, by *R-BlockA* we have $\langle S_1 \triangleright e \xrightarrow{T}_i S_2 \triangleright e' \rangle^{\texttt{m}}$ for some $\texttt{l} \geq \texttt{m}$, and by induction hypothesis over the derivation of e gives as e' is valid, S_2 is valid for $\texttt{l}$ and $S_2 \triangleright e'$ remains valid for $\texttt{l}$, and $T_{1 \setminus t} \cup (t, \{e'\}^{\texttt{l}}) \cup T, S_2$ is valid. Therefore, $\{e'\}^{\texttt{m}}$ is valid, $S_2 \triangleright \{e'\}^{\texttt{m}}$ is valid for $\texttt{l}$ and $T_{1 \setminus t} \cup (t, \{\{e'\}^{\texttt{m}}\}^{\texttt{l}}) \cup T, S_2$ is valid.

- $e_i \triangleq [\overline{e}]$. By *T-Sequence*, $\Gamma_1 \vdash \langle e_1 : \tau \rangle_\sigma^{\texttt{l}} \dashv \Gamma_{1_1} \ldots \Gamma_{1_n} \vdash \langle e_n : \tau_i \rangle_\sigma^{\texttt{l}} \dashv \Gamma_2$ and by applying our induction hypothesis over the derivation of e_1, we know that $\langle S_1 \triangleright e_1 \xrightarrow{T}_i S_2 \triangleright e_1' \rangle^{\texttt{l}}$ such that e_1' is valid, S_2 is valid for $\texttt{l}$ and $S_2 \triangleright e_1'$ remains valid for $\texttt{l}$, and $T_{1 \setminus t} \cup (t, \{e_1'\}^{\texttt{l}}) \cup T, S_2$ is valid. Therefore, $e_1'; \overline{e}$ is valid, $S_2 \triangleright e_1'; \overline{e}$ is valid for $\texttt{l}$ and $T_{1 \setminus t} \cup (t, \{e_1'; \overline{e}\}^{\texttt{l}}) \cup T, S_2$ is valid.

- *Inductive Case* $e_i \triangleq [\texttt{spawn}(f(\overline{e}))]$. As the previous case, by *T-Spawn* and by induction hypothesis the result is immediate.

□

Lemma 4.12 ensures that when reducing a *well-typed* expression e with type τ, the resulting expression e' remains *well-typed*. Moreover, if e' is a value v, then v is safely abstracted by τ.

Lemma 4.12 (Step Expression Preservation) *Let* $T_1 = \{t_i, \{e_i\}^{1_i} \mid 1 \leqslant i \leqslant N\}$, S_1, σ *such that* T_1, S_1 *is a valid global state and for all* $i \in \{1, ..., N\}$ *we have* $S_1 \triangleright e_i \vdash \sigma$, $S_{1|1_i} \sim \Gamma_1^i$ *and* $\Gamma_1^i \vdash \langle e_i : \tau_i \rangle_\sigma^{1_i} \dashv \Gamma_2^i$ *for some* Γ_1^i, Γ_2^i *and* τ_i. *Let* $i \in \{1, ..., N\}, T$ *and* $S_2 \triangleright e_i'$ *such that* $\langle S_1 \triangleright e_i \xrightarrow{T}_i S_2 \triangleright e_i' \rangle^{1_i}$ *then, either* e_i' *is a value* v *such that* $S_2 \vdash v \sim \tau_i$ *or there are* Γ' *and* σ' *such that* $\Gamma' \vdash \langle e_i' : \tau_i \rangle_{\sigma'}^{1_i} \dashv \Gamma_2^i$ *and* $S_2 \triangleright e_i' \vdash \sigma'$ *and for all* $j \in \{1, ..., i-1, i+1, .., N\}$ *we have* $\Gamma_1^j \vdash \langle e_j : \tau_j \rangle_{\sigma'}^{1_j} \dashv \Gamma_2^j$ *and* $S_2 \triangleright e_j \vdash \sigma'$ *and for all* $(t, \{e\}^1) \in T$ *we have* $\Gamma_1 \vdash \langle e : \tau \rangle_{\sigma'}^1 \dashv \Gamma_2$ *and* $S_2 \triangleright e \vdash \sigma'$ *for some* Γ_1, Γ_2 *and* τ.

`Proof.` By structural induction on the forms of e according to Figure 3.5. In the case where e' is a value v, we can establish $\Gamma' = \Gamma_2$:

- *Base Case* $e_i \triangleq [v]$. The result is immediate as there is no reduction rule for this case.
- *Base Case* $e_i \triangleq [\hat{\omega}]$. By *R-Copy* we have $read(S_1, \omega, 1) = \langle v \rangle^n$ where $S_1 = S_2$, and by *T-Copy* we have $\Gamma_1 \vdash \omega : \langle \tau \rangle^m$ such that $\tau_i = \tau$ and $n \in \{m, *\} \cup \mathbb{N}$. Since $S_{1|1_i} \sim \Gamma_1^i$, by applying Lemma 4.4 we have $S_2 \vdash v \sim \tau$.
- *Base Case* $e_i \triangleq [\omega]$. By *T-Move*, $\Gamma_1 \vdash \omega : \langle \tau \rangle^m$ such that $\tau_i = \tau$ and by *R-Move*, $read(S_1, \omega, 1) = \langle v \rangle^n$ some n. Since $S_{1|1_i} \sim \Gamma_1^i$, by applying Lemma 4.4 we have $S_2 \vdash v \sim \tau$ such that $n \in \{m, *\} \cup \mathbb{N}$.
- *Base Case* $e_i \triangleq [\&[mut]\omega]$. By *R-Borrow*, $loc(S_1, \omega, 1) = \ell$ such that $S_1 = S_2$ and by *T-Borrow*, $\Gamma_1 \vdash \omega : \langle \tau \rangle^m$. Since $S_{1|1_i} \sim \Gamma_1^i$, by applying Lemma 4.3 we have $S_2(\ell) = \langle v \rangle^n$ and $S_2 \vdash v \sim \tau_i$ such that $n \in \{m, *\} \cup \mathbb{N}$.
- *Base Case* $e_i \triangleq [box(v_2)]$. By *R-Box*, $S_2 = S_1[\ell_a \mapsto \langle v_2 \rangle^*]$ for some $\ell_a \notin dom(S_1)$ where $v = [\ell_a^{\blacksquare}]$. By *T-Box*, $\Gamma_1 \vdash \langle v_2 : \tau_2 \rangle_\sigma^1 \dashv \Gamma_2$. Since $S_{1|1_i} \sim \Gamma_1^i$, according to Figure 4.7, by *V-Box*, we have $S_1(\ell_a) = \langle v_2 \rangle^*$ and $S_1 \vdash v_2 \sim \tau_2$. Hence we can deduce by *V-Box* that $S_2 \vdash \ell_a^{\blacksquare} \sim \blacksquare\tau_2$.
- *Base Case* $e_i \triangleq [trc(v_2)]$. By *R-Trc*, $S_2 = S_1[\ell_a \mapsto \langle v_2 \rangle^1]$ for some $\ell_a \notin dom(S_1)$ where $v = [\ell_a^{\blacklozenge}]$. By *T-Trc*, $\Gamma_1 \vdash \langle v_2 : \tau_2 \rangle_\sigma^1 \dashv \Gamma_2$. Since $S_{1|1_i} \sim \Gamma_1^i$, according to Figure 4.7, by *V-Trc*, we have $S_1(\ell_a) = \langle v_2 \rangle^i$ and $S_1 \vdash v_2 \sim \tau_2$. Hence we can deduce by *V-Trc* that $S_2 \vdash \ell_a^{\blacklozenge} \sim \blacklozenge\tau_2$.
- *Base Case* $e_i \triangleq [\omega.clone]$. By *R-Clone*, we have $read(S_1, \omega, 1) = \langle \ell_a^{\blacklozenge} \rangle^n$ for some $\ell_a \in dom(S_1)$ and by *T-Clone* we have $\Gamma_1 \vdash \omega : \langle \blacklozenge\tau \rangle^m$ such that $n \in \{m, *\} \cup \mathbb{N}$. Since $S_{1|1_i} \sim \Gamma_1^i$, according to Figure 4.7, *V-Clone*, we have $S_{2|1_i} \vdash \ell_a^{\diamond} \sim \diamond\omega$.
- *Base Case* $e_1 \triangleq [v_1 \oplus v_2]$. By *R-Arithm*, $v = v_1 \oplus v_2$ such that $S_1 = S_2$ and by *T-Arithm*, $\Gamma_1 \vdash \langle v_1 \oplus v_2 : int \rangle_\sigma^{1_i} \dashv \Gamma_2$. Since $S_{1|1_i} \sim \Gamma_1^i$, according to Figure 4.7, by *V-Int*, we have $S_2 \vdash int \sim int$.

- *Base Case* $e_i \triangleq [v_1 \otimes v_2]$. By *R-Cond*, $v = v_1 \otimes v_2$ such that $S_1 = S_2$ and by *T-Cond*, $\Gamma_1 \vdash \langle v_1 \otimes v_2 : \texttt{bool} \rangle_\sigma^{\texttt{l}_\texttt{i}} \dashv \Gamma_2$. Since $S_{1|\texttt{l}_\texttt{i}} \sim \Gamma_1^i$, according to Figure 4.7, by *V-Bool*, we have $S_2 \vdash \texttt{bool} \sim \texttt{bool}$.

- *Base Case* $e_i \triangleq [\texttt{let mut } x = v_2]$. By *R- Declare*, $S_2 = S_1[\ell_{\texttt{l}_\texttt{i}::x} \mapsto \langle v_2 \rangle^{\texttt{l}_\texttt{i}}]$ for some $\ell_{\texttt{l}_\texttt{i}::x} \notin dom(S_1)$ such that $v = \epsilon$. By *T-Declare*, $\Gamma_1 \vdash \langle v_2 : \tau_2 \rangle_\sigma^{\texttt{l}_\texttt{i}} \dashv \Gamma_2$ and $\Gamma_2 = \Gamma_1[x \mapsto \langle \tau_2 \rangle^{\texttt{l}_\texttt{i}}]$ such that $\tau_i = \epsilon$. Since $S_{1|\texttt{l}_\texttt{i}} \sim \Gamma_1^i$, according to Figure 4.7, we have $S_2 \vdash \epsilon \sim \epsilon$ (*V-Unit*).

- *Base Case* $e_i \triangleq [\omega = v_2]$. Similar to the previous case, by *R-Assign*, we have $v = \epsilon$ and by *T-Assign*, $\Gamma_1 \vdash \langle v_2 : \tau_2 \rangle_\sigma^{\texttt{l}_\texttt{i}} \dashv \Gamma_2$ such that *T-Const*, $\sigma \vdash v_2 : \tau_2$ and $\tau_i = \epsilon$. Since $S_{1|\texttt{l}_\texttt{i}} \sim \Gamma_1^i$, according to Figure 4.7, we have $S_2 \vdash \epsilon \sim \epsilon$ (*V-Unit*).

- *Base Case* $e_i \triangleq [\{v'\}^{\texttt{m}}]$. By *R-BlockB*, $v = v'$. By *T-Block*, $\Gamma_1 \vdash \langle v' : \tau \rangle_\sigma^{\texttt{l}_\texttt{i}} \dashv \Gamma_2$ such that *T-Const* $\sigma \vdash v' : \tau$ and $\tau_i = \tau$. Then, we can deduce that $S_2 \vdash v' \sim \tau'$.

- *Base Case* $e_i \triangleq [v; \overline{e}]$. By *T-Sequence* and by Lemma 4.7, $\Gamma_1 \vdash \langle v : \tau \rangle_\sigma^{\texttt{l}_\texttt{i}} \dashv \Gamma_1$ $\Gamma_1 \vdash \langle \overline{e} : \tau_i \rangle_\sigma^{\texttt{l}_\texttt{i}} \dashv \Gamma_2$ such that by *T-Sequence*, τ_i is the type of the last expression in $\overline{e}$. Then, the result immediate since by *R-Seq*, we have $S_2 \triangleright \overline{e}$ where $\Gamma_1 \vdash \langle \overline{e} : \tau_i \rangle_\sigma^{\texttt{l}_\texttt{i}} \dashv \Gamma_2$.

- *Base Case* $e_i \triangleq [\texttt{cooperate}]$. By *R-Cooperate*, we have $[v = \epsilon]$. Since $S_{1|\texttt{l}_\texttt{i}} \sim \Gamma_1^i$, then according to Figure 4.7, we have $S_2 \vdash \epsilon \sim \epsilon$ (*V-Unit*).

- *Base Case* $e_i \triangleq [\texttt{spawn}(f(\overline{v_2}))]$. By *T-Spawn* and By *T-Const* we have, $\overline{\sigma \vdash v_2 : \tau_2}$. By *R-Spawn*, we create a new thread $(t, \{\overline{e}\}^{\texttt{n}})$ for some $* \succcurlyeq \texttt{n}$ such as $S_2 = S' \cup [\overline{\ell_{\texttt{n}::x} \mapsto \langle v_2' \rangle^{\texttt{n}}}]$ where $\overline{(S', v_2') = activate(S_{1|\texttt{l}_\texttt{i}}, v_2)}$. By inspection of Definition 3.6, the *activate* function recursively activates an inactive `Trc` value. Additionally, based on Figure 4.4, both the *S-ATrcR* and *S-ATrcL* rules indicate that an inactive `Trc` type is compatible with an active `Trc` type. In other words, $\overline{\sigma \vdash v_2' : \tau_2' \approx \tau_2}$ for some $\overline{\tau_2'}$ such that $\overline{\tau_2'}$ is compatible with $\overline{\tau}_2$. Therefore, by *R-Spawn*, $v = \epsilon$ and since $S_{1|\texttt{l}_\texttt{i}} \sim \Gamma_1^i$, then according to Figure 4.7, we have $S_2 \vdash \epsilon \sim \epsilon$ (*V-Unit*).

  We will now prove the cases where $e_i \neq [v]$. Then the reduction of e_i produces an alternative expression e_i'. In this case, an intermediate typing environment Γ' exists such that the typing of e_i' in Γ' remains *well-typed*. Therefore, we can say that there exists a valid store typing σ' where for all $v \in e_i'$ we have $S_2 \vdash v \sim \sigma'(v)$. Our proof is based on the structural induction hypothesis as follows:

- *Inductive Case* $e_i \triangleq [\texttt{box}(e)]$. By *T-Box*, $\Gamma_1 \vdash \langle e : \tau \rangle_\sigma^{\texttt{l}_\texttt{i}} \dashv \Gamma_2$ and by applying our induction hypothesis over the derivation of e, we know that $\langle S_1 \triangleright e \xrightarrow{T}_i S_2 \triangleright e' \rangle^{\texttt{l}_i}$ then either e' is a value v such that $S_2 \vdash v \sim \tau$ or there are Γ' and σ'' such that $\Gamma' \vdash \langle e' : \tau \rangle_{\sigma''}^{\texttt{l}_\texttt{i}} \dashv \Gamma_2^i$ and $S_2 \triangleright e' \vdash \sigma''$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$ we have $\Gamma_1^j \vdash \langle e_j : \tau_j \rangle_{\sigma''}^{\texttt{l}_\texttt{j}} \dashv \Gamma_2^j$ and $S_2 \triangleright e_j \vdash \sigma''$ and for all $(t, \{e_1\}^{\texttt{l}}) \in T$, we have $\Gamma_1 \vdash \langle e_1 : \tau_1 \rangle_{\sigma''}^{\texttt{l}} \dashv \Gamma_2$ and $S_2 \triangleright e_1 \vdash \sigma''$ for some Γ_1, Γ_2 and τ_1. In the first case when e' is a value v, we have $\sigma' = \sigma[v \mapsto \tau]$, which implies that all the values of σ' are in σ except v and, since σ is valid and v is safely abstracted by τ (by inspection of Definition 4.22), σ' remains valid. Hence, $S_2 \triangleright \texttt{box}(e') \vdash \sigma'$ such that $\Gamma' \vdash \langle \texttt{box}(e') : \blacksquare\tau \rangle_{\sigma'}^{\texttt{l}_\texttt{i}} \dashv \Gamma_2^i$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$, we

have $\Gamma_1^j \vdash \langle e_j : \tau_j \rangle_{\sigma'}^{1_j} \dashv \Gamma_2^j$ and $S_2 \triangleright e_j \vdash \sigma'$ and for all $(t, \{e_1\}^1) \in T$, we have $\Gamma_1 \vdash \langle e_1 : \tau_1 \rangle_{\sigma'}^1 \dashv \Gamma_2$ and $S_2 \triangleright e_1 \vdash \sigma'$ for some Γ_1, Γ_2 and τ_1. In the second case, we can deduce that $\Gamma' \vdash \langle \texttt{box}(e') : \blacksquare\tau \rangle_{\sigma''}^{1_i} \dashv \Gamma_2^i$ and the rest is immediate.

- *Inductive Case* $e_i \triangleq [\texttt{trc}(e)]$. By *T-Trc*, $\Gamma_1 \vdash \langle e : \tau \rangle_{\sigma}^{1_i} \dashv \Gamma_2$ and by applying our induction hypothesis over the derivation of e, we know that $\langle S_1 \triangleright e \xrightarrow{T}_i S_2 \triangleright e' \rangle^{1_i}$ then either e' is a value v such that $S_2 \vdash v \sim \tau$ or there are Γ' and σ'' such that $\Gamma' \vdash \langle e' : \tau \rangle_{\sigma''}^{1_i} \dashv \Gamma_2^i$ and $S_2 \triangleright e' \vdash \sigma''$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$ we have $\Gamma_1^j \vdash \langle e_j : \tau_j \rangle_{\sigma''}^{1_j} \dashv \Gamma_2^j$ and $S_2 \triangleright e_j \vdash \sigma''$ and for all $(t, \{e_1\}^1) \in T$, we have $\Gamma_1 \vdash \langle e_1 : \tau_1 \rangle_{\sigma''}^1 \dashv \Gamma_2$ and $S_2 \triangleright e_1 \vdash \sigma''$ for some Γ_1, Γ_2 and τ_1. In the first case when e' is a value v, we have $\sigma' = \sigma[v \mapsto \tau]$, which implies that all the values of σ' are in σ except v and, since σ is valid and v is safely abstracted by τ (by inspection of Definition 4.22), σ' remains valid. Hence, $S_2 \triangleright \texttt{trc}(e') \vdash \sigma'$ such that $\Gamma' \vdash \langle \texttt{trc}(e') : \blacklozenge\tau \rangle_{\sigma'}^{1_i} \dashv \Gamma_2^i$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $\Gamma_1^j \vdash \langle e_j : \tau_j \rangle_{\sigma'}^{1_j} \dashv \Gamma_2^j$ and $S_2 \triangleright e_j \vdash \sigma'$ and for all $(t, \{e_1\}^1) \in T$, we have $\Gamma_1 \vdash \langle e_1 : \tau_1 \rangle_{\sigma'}^1 \dashv \Gamma_2$ and $S_2 \triangleright e_1 \vdash \sigma'$ for some Γ_1, Γ_2 and τ_1. In the second case, we can deduce that $\Gamma' \vdash \langle \texttt{trc}(e') : \blacklozenge\tau \rangle_{\sigma''}^{1_i} \dashv \Gamma_2^i$ and the rest is immediate.

- *Inductive Case* $e_i \triangleq [\texttt{let mut } x = e]$. By *T-Declare*, $\Gamma_1 \vdash \langle e : \tau \rangle_{\sigma}^{1_i} \dashv \Gamma_2$ and by applying our induction hypothesis over the derivation of e, we know that $\langle S_1 \triangleright e \xrightarrow{T}_i S_2 \triangleright e' \rangle^{1_i}$, then either e' is a value v such that $S_2 \vdash v \sim \tau$ or there are Γ' and σ'' such that $\Gamma' \vdash \langle e' : \tau \rangle_{\sigma''}^{1_i} \dashv \Gamma_2^i$ and $S_2 \triangleright e' \vdash \sigma''$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $\Gamma_1^j \vdash \langle e_j : \tau_j \rangle_{\sigma''}^{1_j} \dashv \Gamma_2^j$ and $S_2 \triangleright e_j \vdash \sigma''$ and for all $(t, \{e_1\}^1) \in T$, we have $\Gamma_1 \vdash \langle e_1 : \tau_1 \rangle_{\sigma''}^1 \dashv \Gamma_2$ and $S_2 \triangleright e_1 \vdash \sigma''$ for some Γ_1, Γ_2 and τ_1. In the first case when e' is a value v, we have $\sigma' = \sigma[v \mapsto \tau]$, which implies that all the values of σ' are in σ except v and, since σ is valid and v is safely abstracted by τ (by inspection of Definition 4.22), σ' remains valid. Hence, $S_2 \triangleright \texttt{let mut } x = e' \vdash \sigma'$ such that $\Gamma' \vdash \langle \texttt{let mut } x = e' : \epsilon \rangle_{\sigma'}^{1_i} \dashv \Gamma_2^i$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $\Gamma_1^j \vdash \langle e_j : \tau_j \rangle_{\sigma'}^{1_j} \dashv \Gamma_2^j$ and $S_2 \triangleright e_j \vdash \sigma'$ and for all $(t, \{e_1\}^1) \in T$, we have $\Gamma_1 \vdash \langle e_1 : \tau_1 \rangle_{\sigma'}^1 \dashv \Gamma_2$ and $S_2 \triangleright e_1 \vdash \sigma'$ for some Γ_1, Γ_2 and τ_1. In the second case, we can deduce that $\Gamma' \vdash \langle \texttt{let mut } x = e' : \epsilon \rangle_{\sigma''}^{1_i} \dashv \Gamma_2^i$ and the rest is immediate.

- $e_i \triangleq [\omega = e]$. By *T-Assign*, $\Gamma_1 \vdash \langle e : \tau \rangle_{\sigma}^{1_i} \dashv \Gamma_2$ and by applying our induction hypothesis over the derivation of e, we know that $\langle S_1 \triangleright e \xrightarrow{T}_i S_2 \triangleright e' \rangle^{1_i}$ then either e' is a value v such that $S_2 \vdash v \sim \tau$ or there are Γ' and σ'' such that $\Gamma' \vdash \langle e' : \tau \rangle_{\sigma''}^{1_i} \dashv \Gamma_2^i$ and $S_2 \triangleright e' \vdash \sigma''$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $\Gamma_1^j \vdash \langle e_j : \tau_j \rangle_{\sigma''}^{1_j} \dashv \Gamma_2^j$ and $S_2 \triangleright e_j \vdash \sigma''$ and for all $(t, \{e_1\}^1) \in T$, we have $\Gamma_1 \vdash \langle e_1 : \tau_1 \rangle_{\sigma''}^1 \dashv \Gamma_2$ and $S_2 \triangleright e_1 \vdash \sigma''$ for some Γ_1, Γ_2 and τ_1. In the first case when e' is a value v, we have $\sigma' = \sigma[v \mapsto \tau]$, which implies that all the values of σ' are in σ except v and, since σ is valid and v is safely abstracted by τ (by inspection of Definition 4.22), σ' remains valid. Hence, $S_2 \triangleright \omega = e' \vdash \sigma'$ such that $\Gamma' \vdash \langle \omega = e' : \epsilon \rangle_{\sigma'}^{1_i} \dashv \Gamma_2^i$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $\Gamma_1^j \vdash \langle e_j : \tau_j \rangle_{\sigma'}^{1_j} \dashv \Gamma_2^j$ and $S_2 \triangleright e_j \vdash \sigma'$ and for all $(t, \{e_1\}^1) \in T$, we have $\Gamma_1 \vdash \langle e_1 : \tau_1 \rangle_{\sigma'}^1 \dashv \Gamma_2$ and $S_2 \triangleright e_1 \vdash \sigma'$ for some Γ_1, Γ_2

and τ_1. In the second case, we can deduce that $\Gamma' \vdash \langle \omega = e' : \epsilon \rangle^{1_i}_{\sigma''} \dashv \Gamma_2^i$ and the rest is immediate.

- $e_i \triangleq [e_1 \oplus e_2]$. By *T-Arithm*, $\Gamma_1 \vdash \langle e_1 : \texttt{int} \rangle^{1_i}_{\sigma} \dashv \Gamma_2$ and $\Gamma_2 \vdash \langle e_2 : \texttt{int} \rangle^{1_i}_{\sigma} \dashv \Gamma_3$. By applying our induction hypothesis over the derivation of e_1, we know that $\langle S_1 \triangleright e_1 \xrightarrow{T}_i S_2 \triangleright e' \rangle^{1_i}$ then either e' is a value v such that $S_2 \vdash v \sim \tau$ or there are Γ' and σ'' such that $\Gamma' \vdash \langle e' : \tau \rangle^{1_i}_{\sigma''} \dashv \Gamma_2^i$ and $S_2 \triangleright e' \vdash \sigma''$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $\Gamma_1^j \vdash \langle e_j : \tau_j \rangle^{1_j}_{\sigma''} \dashv \Gamma_2^j$ and $S_2 \triangleright e_j \vdash \sigma''$ and for all $(t, \{e_1\}^1) \in T$, we have $\Gamma_1 \vdash \langle e_1 : \tau_1 \rangle^{1}_{\sigma''} \dashv \Gamma_2$ and $S_2 \triangleright e_1 \vdash \sigma''$ for some Γ_1, Γ_2 and τ_1. In the first case when e' is a value v, we have $\sigma' = \sigma[v \mapsto \tau]$, which implies that all the values of σ' are in σ except v and, since σ is valid and v is safely abstracted by τ (by inspection of Definition 4.22), σ' remains valid. Hence, $S_2 \triangleright e' \oplus e_2 \vdash \sigma'$ such that $\Gamma' \vdash \langle e' \oplus e_2 : \texttt{int} \rangle^{1_i}_{\sigma'} \dashv \Gamma_2^i$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $\Gamma_1^j \vdash \langle e_j : \tau_j \rangle^{1_j}_{\sigma'} \dashv \Gamma_2^j$ and $S_2 \triangleright e_j \vdash \sigma'$ and for all $(t, \{e_1\}^1) \in T$, we have $\Gamma_1 \vdash \langle e_1 : \tau_1 \rangle^{1}_{\sigma'} \dashv \Gamma_2$ and $S_2 \triangleright e_1 \vdash \sigma'$ for some Γ_1, Γ_2 and τ_1. In the second case, we can deduce that $\Gamma' \vdash \langle e' \oplus e_2 : \texttt{int} \rangle^{1_i}_{\sigma''} \dashv \Gamma_2^i$ and the rest is immediate.

- $e_i \triangleq [e_1 \otimes e_2]$. By *T-Cond*, $\Gamma_1 \vdash \langle e_1 : \tau \rangle^{1_i}_{\sigma} \dashv \Gamma_2$ and $\Gamma_2 \vdash \langle e_2 : \tau' \rangle^{1_i}_{\sigma} \dashv \Gamma_3$. By applying our induction hypothesis over the derivation of e_1, we know that $\langle S_1 \triangleright e_1 \xrightarrow{T}_i S_2 \triangleright e' \rangle^{1_i}$ then either e' is a value v such that $S_2 \vdash v \sim \tau$ or there are Γ' and σ'' such that $\Gamma' \vdash \langle e' : \tau \rangle^{1_i}_{\sigma''} \dashv \Gamma_2^i$ and $S_2 \triangleright e' \vdash \sigma''$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $\Gamma_1^j \vdash \langle e_j : \tau_j \rangle^{1_j}_{\sigma''} \dashv \Gamma_2^j$ and $S_2 \triangleright e_j \vdash \sigma''$ and for all $(t, \{e_1\}^1) \in T$, we have $\Gamma_1 \vdash \langle e_1 : \tau_1 \rangle^{1}_{\sigma''} \dashv \Gamma_2$ and $S_2 \triangleright e_1 \vdash \sigma''$ for some Γ_1, Γ_2 and τ_1. In the first case when e' is a value v, we have $\sigma' = \sigma[v \mapsto \tau]$, which implies that all the values of σ' are in σ except v and, since σ is valid and v is safely abstracted by τ (by inspection of Definition 4.22), σ' remains valid. Hence, $S_2 \triangleright e' \otimes e_2 \vdash \sigma'$ such that $\Gamma' \vdash \langle e' \otimes e_2 : \texttt{bool} \rangle^{1_i}_{\sigma'} \dashv \Gamma_2^i$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $\Gamma_1^j \vdash \langle e_j : \tau_j \rangle^{1_j}_{\sigma'} \dashv \Gamma_2^j$ and $S_2 \triangleright e_j \vdash \sigma'$ and for all $(t, \{e_1\}^1) \in T$, we have $\Gamma_1 \vdash \langle e_1 : \tau_1 \rangle^{1}_{\sigma'} \dashv \Gamma_2$ and $S_2 \triangleright e_1 \vdash \sigma'$ for some Γ_1, Γ_2 and τ_1. In the second case, we can deduce that $\Gamma' \vdash \langle e' \otimes e_2 : \texttt{bool} \rangle^{1_i}_{\sigma''} \dashv \Gamma_2^i$ and the rest is immediate.

- $e_i \triangleq [\{e\}^{\mathtt{m}}]$. By *T-Block*, $\Gamma_1 \vdash \langle e : \tau \rangle^{1_i}_{\sigma} \dashv \Gamma_2$ and by applying our induction hypothesis over the derivation of e, we know that by *R-BlockA*, $\langle S_1 \triangleright e \xrightarrow{T}_i S_2 \triangleright e' \rangle^{1_i}$ then either e' is a value v such that $S_2 \vdash v \sim \tau$ or there are Γ' and σ' such that $\Gamma' \vdash \langle e' : \tau \rangle^{1_i}_{\sigma'} \dashv \Gamma_2^i$ and $S_2 \triangleright e' \vdash \sigma'$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $\Gamma_1^j \vdash \langle e_j : \tau_j \rangle^{1_j}_{\sigma'} \dashv \Gamma_2^j$ and $S_2 \triangleright e_j \vdash \sigma'$ and for all $(t, \{e_1\}^1) \in T$, we have $\Gamma_1 \vdash \langle e_1 : \tau_1 \rangle^{1}_{\sigma'} \dashv \Gamma_2$ and $S_2 \triangleright e \vdash \sigma'$ for some Γ_1, Γ_2 and τ_1.

- $e_i \triangleq [\overline{e}]$. By *T-Sequence*, $\Gamma_1 \vdash \langle e_1 : \tau \rangle^{1_i}_{\sigma} \dashv \Gamma_{1_1} \ldots \Gamma_{1_n} \vdash \langle e_n : \tau_i \rangle^{1_i}_{\sigma} \dashv \Gamma_2$ and by applying our induction hypothesis over the derivation of e_1, we know that $\langle S_1 \triangleright e_1 \xrightarrow{T}_i S_2 \triangleright e' \rangle^{1_i}$ then either e' is a value v such that $S_2 \vdash v \sim \tau$ or there are Γ' and σ'' such that $\Gamma' \vdash \langle e' : \tau \rangle^{1_i}_{\sigma''} \dashv \Gamma_2^i$ and $S_2 \triangleright e' \vdash \sigma''$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $\Gamma_1^j \vdash \langle e_j : \tau_j \rangle^{1_j}_{\sigma''} \dashv \Gamma_2^j$ and $S_2 \triangleright e_j \vdash \sigma''$ and for all $(t, \{e_1\}^1) \in T$, we have $\Gamma_1 \vdash \langle e_1 : \tau_1 \rangle^{1}_{\sigma''} \dashv \Gamma_2$ and $S_2 \triangleright e_1 \vdash \sigma''$ for some Γ_1, Γ_2 and

τ_1. In the first case when e' is a value v, we have $\sigma' = \sigma[v \mapsto \tau]$, which implies that all the values of σ' are in σ except v and, since σ is valid and v is safely abstracted by τ (by inspection of Definition 4.22), σ' remains valid. Hence, $S_2 \triangleright e'; \overline{e} \vdash \sigma'$ such that $\Gamma' \vdash \langle e'; \overline{e} : \tau_i \rangle^{1_i}_{\sigma'} \dashv \Gamma^i_2$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $\Gamma^j_1 \vdash \langle e_j : \tau_j \rangle^{1_j}_{\sigma'} \dashv \Gamma^j_2$ and $S_2 \triangleright e_j \vdash \sigma'$ and for all $(t, \{e_1\}^1) \in T$, we have $\Gamma_1 \vdash \langle e_1 : \tau_1 \rangle^1_{\sigma'} \dashv \Gamma_2$ and $S_2 \triangleright e_1 \vdash \sigma'$ for some Γ_1, Γ_2 and τ_1. In the second case, we can deduce that $\Gamma' \vdash \langle e'; \overline{e} : \tau_i \rangle^{1_i}_{\sigma''} \dashv \Gamma^i_2$ and the rest is immediate.

- *Inductive Case* $e_i \triangleq [\texttt{spawn}(f(\overline{e}))]$. By *T-Spawn*, we have $\Gamma_1 \vdash \langle \overline{e : \tau} \rangle^{1_i}_{\sigma} \dashv \Gamma_2$ and by applying our induction hypothesis over the derivation of e_0, we know that $\langle S_1 \triangleright e_0 \xrightarrow{T}_i S_2 \triangleright e' \rangle^{1_i}$ then either e' is a value v such that $S_2 \vdash v \sim \tau$ or there are Γ' and σ' such that $\Gamma' \vdash \langle e' : \tau \rangle^{1_i}_{\sigma'} \dashv \Gamma^i_2$ and $S_2 \triangleright e' \vdash \sigma'$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $\Gamma^j_1 \vdash \langle e_j : \tau_j \rangle^{1_j}_{\sigma'} \dashv \Gamma^j_2$ and $S_2 \triangleright e_j \vdash \sigma'$ and for all $(t, \{e_1\}^1) \in T$, we have $\Gamma_1 \vdash \langle e_1 : \tau_1 \rangle^1_{\sigma'} \dashv \Gamma_2$ and $S_2 \triangleright e \vdash \sigma'$ for some Γ_1, Γ_2 and τ_1. In the first case when e' is a value v, we have $\sigma' = \sigma[v \mapsto \tau]$, which implies that all the values of σ' are in σ except v and, since σ is valid and v is safely abstracted by τ (by inspection of Definition 4.22), σ' remains valid. Hence, $S_2 \triangleright \texttt{spawn}(f(e'; \overline{e})) \vdash \sigma'$ such that $\Gamma' \vdash \langle \texttt{spawn}(f(e'; \overline{e})) : \epsilon \rangle^{1_i}_{\sigma'} \dashv \Gamma^i_2$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $\Gamma^j_1 \vdash \langle e_j : \tau_j \rangle^{1_j}_{\sigma'} \dashv \Gamma^j_2$ and $S_2 \triangleright e_j \vdash \sigma'$ and for all $(t, \{e_1\}^1) \in T$, we have $\Gamma_1 \vdash \langle e_1 : \tau_1 \rangle^1_{\sigma'} \dashv \Gamma_2$ and $S_2 \triangleright e_1 \vdash \sigma'$ for some Γ_1, Γ_2 and τ_1. In the second case, we can deduce that $\Gamma' \vdash \langle \texttt{spawn}(f(e'; \overline{e})) : \epsilon \rangle^{1_i}_{\sigma''} \dashv \Gamma^i_2$ and the rest is immediate.

□

Lemma 4.13 ensures that after a local reduction for a given thread t, the resulting program store remains safely abstracted by the resulting typing environment. Furthermore, this preservation of safe abstraction extends to all existing threads within the program as follows:

Lemma 4.13 (Step Store Preservation) *Let* $T_1 = \{t_i, \{e_i\}^{1_i} \mid 1 \leqslant i \leqslant N\}$, S_1, σ *such that* T_1, S_1 *be a valid global state and for all* $i \in \{1, ..., N\}$ *we have* $S_1 \triangleright e_i \vdash \sigma$, $S_{1|1_i} \sim \Gamma^i_1$ *and* $\Gamma^i_1 \vdash \langle e_i : \tau_i \rangle^{1_i}_{\sigma} \dashv \Gamma^i_2$ *for some* Γ^i_1, Γ^i_2 *and* τ_i. *Let* $i \in \{1, ..., N\}, T$ *and* $S_2 \triangleright e'_i$ *such that* $\langle S_1 \triangleright e_i \xrightarrow{T}_i S_2 \triangleright e'_i \rangle^{1_i}$ *then, there exists* Γ' *such that* $S_{2|1_i} \sim \Gamma'$ *and for all* $j \in \{1, ..., i-1, i+1, .., N\}$, *we have* $S_{2|1_j} \sim \Gamma^j_1$ *and for all* $(t, \{e\}^1) \in T$ *such that* $\Gamma_1 \vdash \langle e : \tau \rangle^1_{\sigma'} \dashv \Gamma_2$, *we have* $S_{2|1} \sim \Gamma_1$ *for some* $\Gamma_1, \Gamma_2, \sigma'$ *and* τ.

`Proof.` By structural induction on the forms of e according to Figure 3.5. For each case and by inspection of Definition 4.24, the following must be demonstrated: (1) S_2 remains valid, (2) $(\text{dom}(S_2) \backslash \mathcal{L}) = \Theta(dom(\Gamma'))$, and (3) for all $x \in dom(\Gamma')$ such that $\Gamma'(x) = \langle \tilde{\tau} \rangle^{\mathtt{m}}$, there exists $v^{\perp}$ such that $S_2 \vdash v^{\perp} \sim \tilde{\tau}$ where $S_2(\ell_{\mathtt{m}::x}) = \langle v^{\perp} \rangle^{\mathtt{m}}$. In the case where e' is a value v, we can establish $\Gamma' = \Gamma_2$:

- *Base Case* $e \triangleq [v]$. The result is immediate as there is no reduction rule for this case.

- *Base Case* $e \triangleq [\hat{\omega}]$. By *R-Copy* we have $S_1 = S_2$ then, S_2 is valid for $\mathtt{l_i}$ since S_1 is valid for $\mathtt{l_i}$. By *T-Copy*, $\Gamma_1^i = \Gamma_2^i$. Consequently, we can deduce that $(\text{dom}(S_{2|\mathtt{l_i}}) \backslash \mathcal{L}) = \Theta(dom(\Gamma_2^i))$ and for all $x \in dom(\Gamma_2^i)$ such that $\Gamma_2^i(x) = \langle \tilde{\tau} \rangle^{\mathtt{m}}$, there exists $v^{\perp}$ such that $S_2 \vdash v^{\perp} \sim \tilde{\tau}$ where $S_2(\ell_{\mathtt{m}::x}) = \langle v^{\perp} \rangle^{\mathtt{m}}$. As a result, $S_{2|\mathtt{l_i}} \sim \Gamma_2^i$ since $S_{1|\mathtt{l_i}} \sim \Gamma_1^i$. Moreover, since $S_1 = S_2$ then, for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $S_{2|\mathtt{l_j}} \sim \Gamma_1^j$ and for all $(t, \{e\}^{\mathtt{l}}) \in T$ such that $\Gamma_1 \vdash \langle e : \tau \rangle_{\sigma}^{\mathtt{l}} \dashv \Gamma_2$, we have $S_{2|\mathtt{l}} \sim \Gamma_1$ for some Γ_1, Γ_2 and τ.

- *Base Case* $e \triangleq [\omega]$. By *R-Move*, $\mathit{read}(S_1, \omega, \mathtt{l}) = \langle v \rangle^{\mathtt{m}}$ and $S_2 = \mathit{write}(S_1, \omega, \perp, \mathtt{l})$, which implies that the values of $S_{2|\mathtt{l_i}}$, except for v, are the same values of $S_{1|\mathtt{l_i}}$, with v is replaced by $\perp$. Since S_1 is valid for $\mathtt{l_i}$ then S_2 remains valid for $\mathtt{l_i}$. By *T-Move*, we have the following two premises: $\Gamma_1 \vdash \omega : \langle \tau \rangle^{\mathtt{m}}$ and $\Gamma_2 = \mathit{move}(\Gamma_1, \omega)$. By applying the second premise, the new type of ω in Γ_2 becomes $\lfloor \tau \rfloor$, which implies that Γ_2 is exactly the same as Γ_1 with the exception of ω. Consequently, we can deduce that $(\text{dom}(S_{2|\mathtt{l_i}}) \backslash \mathcal{L}) = \Theta(dom(\Gamma_2^i))$. By inspection of Figure 4.7 we have $S_2 \vdash \perp \sim \lfloor \tau \rfloor$. Furthermore, the *T-Move* rule guarantees that $\neg \mathit{writeProhibited}(\Gamma_1, \omega)$ and $\neg \mathit{TrcMoveProhibited}(\Gamma_1, \omega)$ hold, preventing violations of borrow and `Trc` invariance (meaning that ω is neither borrowed nor cloned, as defined in Definitions 4.4 and 4.10). As a result, for all $x \in dom(\Gamma_2^i)$ such that $\Gamma_2^i(x) = \langle \tilde{\tau} \rangle^{\mathtt{m}}$, there exists $v^{\perp}$ such that $S_2 \vdash v^{\perp} \sim \tilde{\tau}$ where $S_2(\ell_{\mathtt{m}::x}) = \langle v^{\perp} \rangle^{\mathtt{m}}$. Hence, we can deduce that $S_{2|\mathtt{l_i}} \sim \Gamma_2^i$. Finally, as v in $S_{1|\mathtt{l_i}}$, this does not violate the notion of safe abstraction between S and Γ to other threads hence, for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $S_{2|\mathtt{l_j}} \sim \Gamma_1^j$ and for all $(t, \{e\}^{\mathtt{l}}) \in T$ such that $\Gamma_1 \vdash \langle e : \tau \rangle_{\sigma}^{\mathtt{l}} \dashv \Gamma_2$, we have $S_{2|\mathtt{l}} \sim \Gamma_1$ for some Γ_1, Γ_2 and τ.

- *Base Case* $e \triangleq [\&[\mathtt{mut}]\omega]$. By *R-Borrow* we have $S_1 = S_2$ then, S_2 is valid for $\mathtt{l_i}$ since S_1 is valid for $\mathtt{l_i}$. By both *T-ImmBorrow* and *T-MutBorrow*, $\Gamma_1^i = \Gamma_2^i$. Consequently, we can deduce that $(\text{dom}(S_{2|\mathtt{l_i}}) \backslash \mathcal{L}) = \Theta(dom(\Gamma_2^i))$ and for all $x \in dom(\Gamma_2^i)$ such that $\Gamma_2^i(x) = \langle \tilde{\tau} \rangle^{\mathtt{m}}$, there exists $v^{\perp}$ such that $S_2 \vdash v^{\perp} \sim \tilde{\tau}$ where $S_2(\ell_{\mathtt{m}::x}) = \langle v^{\perp} \rangle^{\mathtt{m}}$. As a result, $S_{2|\mathtt{l_i}} \sim \Gamma_2^i$ since $S_{1|\mathtt{l_i}} \sim \Gamma_1^i$. Moreover, since $S_1 = S_2$ then, for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $S_{2|\mathtt{l_j}} \sim \Gamma_1^j$ and for all $(t, \{e\}^{\mathtt{l}}) \in T$ such that $\Gamma_1 \vdash \langle e : \tau \rangle_{\sigma}^{\mathtt{l}} \dashv \Gamma_2$, we have $S_{2|\mathtt{l}} \sim \Gamma_1$ for some Γ_1, Γ_2 and τ.

- *Base Case* $e \triangleq [\mathtt{box}(v_2)]$. By *R-Box*, $v = \ell_a^{\blacksquare}$ such that $S_2 = S_1[\ell_a \mapsto \langle v_2 \rangle^*]$ for some $\ell_a \notin dom(S_1)$. Since ℓ_a is a heap location ($\ell_a \in \mathcal{L}$ such that $\mathcal{L}$ is the set of heap locations) then, by inspection of Definition 4.24, we have $(\text{dom}(S_2) \backslash \mathcal{L}) = (\text{dom}(S_1) \backslash \mathcal{L})$. Since S_1 is valid for $\mathtt{l_i}$ then, S_2 is valid for $\mathtt{l_i}$. Therefore, by Lemma 4.7 and by *T-Box*, we have $\Gamma_1^i \vdash \langle v_2 : \tau_2 \rangle_{\sigma}^{\mathtt{l_i}} \dashv \Gamma_2^i$ such that $\Gamma_1^i = \Gamma_2^i$. Consequently, we can deduce that $(\text{dom}(S_{2|\mathtt{l_i}}) \backslash \mathcal{L}) = \Theta(dom(\Gamma_2^i))$ and for all $x \in dom(\Gamma_2^i)$ such that $\Gamma_2^i(x) = \langle \tilde{\tau} \rangle^{\mathtt{m}}$, there exists $v^{\perp}$ such that $S_2 \vdash v^{\perp} \sim \tilde{\tau}$ where $S_2(\ell_{\mathtt{m}::x}) = \langle v^{\perp} \rangle^{\mathtt{m}}$. As a result, $S_{2|\mathtt{l_i}} \sim \Gamma_2^i$ since $S_{1|\mathtt{l_i}} \sim \Gamma_1^i$. Finally, as $\ell_a \in \mathcal{L}$ then, for all $j \in \{1, ..., i-1, i+1, .., N\}$, we can deduce that $(\text{dom}(S_{2|\mathtt{l_j}}) \backslash \mathcal{L}) = (\text{dom}(S_{1|\mathtt{l_j}} \backslash \mathcal{L})$ and hence, $S_{2|\mathtt{l_j}} \sim \Gamma_1^j$ and for all $(t, \{e\}^{\mathtt{l}}) \in T$ such that $\Gamma_1 \vdash \langle e : \tau \rangle_{\sigma}^{\mathtt{l}} \dashv \Gamma_2$, we have $S_{2|\mathtt{l}} \sim \Gamma_1$ for some Γ_1, Γ_2 and τ since $(\text{dom}(S_{2|\mathtt{l}}) \backslash \mathcal{L}) = (\text{dom}(S_{1|\mathtt{l}} \backslash \mathcal{L})$.

- *Base Case* $e \triangleq [\mathtt{trc}(v_2)]$. By *R-Trc*, $v = \ell_a^{\blacklozenge}$ such that $S_2 = S_1[\ell_a \mapsto \langle v_2 \rangle^1]$ for

some $\ell_a \notin dom(S_1)$. Since ℓ_a is a heap location ($\ell_a \in \mathcal{L}$ such that $\mathcal{L}$ is the set of heap locations) then, by inspection of Definition 4.24, we have (dom($S_{2|1_i}$) $\setminus\mathcal{L}$) = (dom($S_{1|1_i}$) $\setminus\mathcal{L}$). Since S_1 is valid for 1_i then, S_2 is valid for 1_i. Therefore, by Lemma 4.7 and by *T-Trc*, we have $\Gamma_1^i \vdash \langle v_2 : \tau_2\rangle_\sigma^{1_i} \dashv \Gamma_2^i$ such that $\Gamma_1^i = \Gamma_2^i$. Consequently, we can deduce that (dom($S_{2|1_i}$)$\setminus\mathcal{L}$) = $\Theta(dom(\Gamma_2^i))$ and for all $x \in dom(\Gamma_2^i)$ such that $\Gamma_2^i(x) = \langle\tilde{\tau}\rangle^{\mathtt{m}}$, there exists $v^\perp$ such that $S_2 \vdash v^\perp \sim \tilde{\tau}$ where $S_2(\ell_{\mathtt{m}::x}) = \langle v^\perp\rangle^{\mathtt{m}}$. As a result, $S_{2|1_i} \sim \Gamma_2^i$ since $S_{1|1_i} \sim \Gamma_1^i$. Finally, as $\ell_a \in \mathcal{L}$ then, for all $j \in \{1, ..., i-1, i+1, .., N\}$, we can deduce that (dom($S_{2|1_j}$) $\setminus\mathcal{L}$) = (dom($S_{1|1_j}\setminus\mathcal{L}$) and hence, $S_{2|1_j} \sim \Gamma_1^j$ and for all $(t, \{e\}^1) \in T$ such that $\Gamma_1 \vdash \langle e : \tau\rangle_\sigma^1 \dashv \Gamma_2$, we have $S_{2|1} \sim \Gamma_1$ for some Γ_1, Γ_2 and τ since (dom($S_{2|1}$) $\setminus\mathcal{L}$) = (dom($S_{1|1}\setminus\mathcal{L}$).

- *Base Case* $e \triangleq [\omega.clone]$. By *R-Clone*, $v = \ell_a^\diamond$ such that $S_2 = S_1[\![\ell_a \mapsto \langle v\rangle^{i+1}]\!]$ for some $\ell_a \in dom(S_1)$ where ℓ_a is a heap location. By inspection of Definition 4.24, $\ell_a \in \mathcal{L}$ and we have ($dom(S_{2|1_i})$ $\setminus\mathcal{L}$) = (dom($S_{1|1_i}$) $\setminus\mathcal{L}$). Since S_1 is valid for 1_i then, S_2 is valid for 1_i. By *T-Clone*, $\Gamma_1^i = \Gamma_2^i$. Consequently, we can deduce that (dom($S_{2|1_i}$)$\setminus\mathcal{L}$) = $\Theta(dom(\Gamma_2^i))$ and for all $x \in dom(\Gamma_2^i)$ such that $\Gamma_2^i(x) = \langle\tilde{\tau}\rangle^{\mathtt{m}}$, there exists $v^\perp$ such that $S_2 \vdash v^\perp \sim \tilde{\tau}$ where $S_2(\ell_{\mathtt{m}::x}) = \langle v^\perp\rangle^{\mathtt{m}}$. As a result, $S_{2|1_i} \sim \Gamma_2^i$ since $S_{1|1_i} \sim \Gamma_1^i$. Finally, as $\ell_a \in \mathcal{L}$ (ℓ_a is already in S_1) then, for all $j \in \{1, ..., i-1, i+1, .., N\}$, we can deduce that (dom($S_{2|1_j}$) $\setminus\mathcal{L}$) = (dom($S_{1|1_j}\setminus\mathcal{L}$) and hence, $S_{2|1_j} \sim \Gamma_1^j$ and for all $(t, \{e\}^1) \in T$ such that $\Gamma_1 \vdash \langle e : \tau\rangle_\sigma^1 \dashv \Gamma_2$, we have $S_{2|1} \sim \Gamma_1$ for some Γ_1, Γ_2 and τ since (dom($S_{2|1}$) $\setminus\mathcal{L}$) = (dom($S_{1|1}\setminus\mathcal{L}$).

- *Base Case* $e \triangleq [\mathtt{let\ mut}\ x = v_2]$. By *R-Declare*, we have $S_2 = S_1[\ell_{1_i::x} \mapsto \langle v_2\rangle^{1_i}]$ such that $\ell_{1_i::x} \notin dom(S_1)$. This implies that all locations in $S_{2|1_i}$ are in $S_{1|1_i}$, expanded by $\ell_{1_i::x}$. Since S_1 is valid for 1_i then, S_2 is valid for 1_i. By *T-Declare* and by lemma 4.7, $\Gamma_1^i \vdash \langle v_2 : \tau_2\rangle_\sigma^{1_i} \dashv \Gamma_2^i$ where $\Gamma_1^i = \Gamma_2^i$. Thus, by *T-Declare* we have and $\Gamma_2^i = \Gamma_1^i[x \mapsto \langle\tau_2\rangle^{1_i}]$ such that $x \notin dom(\Gamma_1^i)$. This implies that all variables in Γ_2^i are in Γ_1^i , expanded by x. As (dom($S_{1|1_i}$) $\setminus\mathcal{L}$) = $\Theta(dom(\Gamma_1^i))$ and by inspection of Definition 4.24 we can deduce that (dom($S_{2|1_i}$) $\setminus\mathcal{L}$) = $\Theta(dom(\Gamma_2^i))$ and for all $x \in dom(\Gamma_2^i)$ such that $\Gamma_2^i(x) = \langle\tilde{\tau}\rangle^{\mathtt{m}}$, there exists $v^\perp$ such that $S_2 \vdash v^\perp \sim \tilde{\tau}$ where $S_2(\ell_{\mathtt{m}::x}) = \langle v^\perp\rangle^{\mathtt{m}}$. As a result, $S_{2|1_i} \sim \Gamma_2^i$. Finally, since $\ell_{1_i::x} \in dom(S_{2|1_i})$, this does not violate the notion of safe abstraction between S and Γ to other threads. Hence, for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $S_{2|1_j} \sim \Gamma_1^j$ and for all $(t, \{e\}^1) \in T$ such that $\Gamma_1 \vdash \langle e : \tau\rangle_\sigma^1 \dashv \Gamma_2$, we have $S_{2|1} \sim \Gamma_1$ for some Γ_1, Γ_2 and τ.

- *Base Case* $e \triangleq [\omega = v_2]$. By *R-Assign*, we have $read(S_1, \omega, 1) = \langle v_1^\perp\rangle^{\mathtt{m}}$ and $S_2 = write(drop(S_1, \{v_1^\perp\}), \omega, v_2, 1)$. By *T-Assign*, $\Gamma_1^i \vdash \omega : \langle\tilde{\tau}_1\rangle^{\mathtt{m}}$ and by hypothesis, $S_{1|1_i} \sim \Gamma_1^i$. By inspection of Definition 4.24, we have (dom($S_{1|1_i}$)$\setminus\mathcal{L}$) = $\Theta(dom(\Gamma_1^i))$ and for all $x \in dom(\Gamma_1^i)$ such that $\Gamma_1^i(x) = \langle\tilde{\tau}\rangle^{\mathtt{m}}$, there exists $v^\perp$ such that $S \vdash v^\perp \sim \tilde{\tau}$ where $S_{1|1_i}(\ell_{\mathtt{m}::x}) = \langle v^\perp\rangle^{\mathtt{m}}$. Therefore, we can deduce that $S_1 \vdash v_1^\perp \sim \tilde{\tau}_1$ by applying Lemma 4.4. Furthermore, by *T-Assign*, $\Gamma_1^i \vdash \langle v_2 : \tau_2\rangle_\sigma^{1_i} \dashv \Gamma_2^i$, $\Gamma_2^i \vdash \tilde{\tau}_1 \approx \tau_2$, and $write^0(\Gamma_2^i, \omega, \tau_2)$. By the first premise, $S_1 \vdash v_2 \sim \tau_2$ and by applying Lemma 4.7, we have $\Gamma_1^i = \Gamma_2^i$. By the second premise, according to Figure 4.4, τ_1 must be compatible with τ_2. Since we have $S_{1|1_i} \sim \Gamma_1^i$, $S_1 \vdash v_1^\perp \sim \tilde{\tau}_1$, and $S_1 \vdash v_2 \sim \tau_2$ then, by applying Lemma 4.6, we have $write(drop(S_1, \{v_1^\perp\}), \omega, v_2, 1) \sim write^0(\Gamma_1^i, \omega, \tau_2)$ and

for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $S_{2|\mathtt{l}_j} \sim \Gamma_1^j$ and for all $(t, \{e\}^{\mathtt{l}}) \in T$ such that $\Gamma_1 \vdash \langle e : \tau \rangle_\sigma^{\mathtt{l}} \dashv \Gamma_2$, we have $S_{2|\mathtt{l}} \sim \Gamma_1$ for some Γ_1, Γ_2 and τ.

- *Base Case* $e \triangleq [v_1 \oplus v_2]$. By *R-Arithm*, $S_1 = S_2$ then, S_2 is valid for $\mathtt{l}_i$ since S_1 is valid for $\mathtt{l}_i$. By *T-Arithm* and by Lemma 4.7, $\Gamma_1^i \vdash \langle v_1 : \tau_1 \rangle_\sigma^{\mathtt{l}_i} \dashv \Gamma_1^i$ and $\Gamma_1^i \vdash \langle v_2 : \tau_2 \rangle_\sigma^{\mathtt{l}_i} \dashv \Gamma_1^i$ such that $\Gamma_1^i = \Gamma_2^i$. Consequently, we can deduce that $(\text{dom}(S_2) \backslash \mathcal{L}) = \Theta(dom(\Gamma_2^i))$ and for all $x \in dom(\Gamma_2^i)$ such that $\Gamma_2^i(x) = \langle \tilde{\tau} \rangle^{\mathtt{m}}$, there exists $v^\perp$ such that $S_2 \vdash v^\perp \sim \tilde{\tau}$ where $S_2(\ell_{\mathtt{m}::x}) = \langle v^\perp \rangle^{\mathtt{m}}$. As a result, $S_{2|\mathtt{l}_i} \sim \Gamma_2^i$ since $S_{1|\mathtt{l}_i} \sim \Gamma_1^i$. Finally, for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $S_{2|\mathtt{l}_j} \sim \Gamma_1^j$ and for all $(t, \{e\}^{\mathtt{l}}) \in T$ such that $\Gamma_1 \vdash \langle e : \tau \rangle_\sigma^{\mathtt{l}} \dashv \Gamma_2$, we have $S_{2|\mathtt{l}} \sim \Gamma_1$ for some Γ_1, Γ_2 and τ.

- *Base Case* $e \triangleq [v_1 \otimes v_2]$. By *R-Cond*, $S_1 = S_2$ then, S_2 is valid for $\mathtt{l}_i$ since S_1 is valid for $\mathtt{l}_i$. By *T-Cond* and by Lemma 4.7, $\Gamma_1^i \vdash \langle v_1 : \tau_1 \rangle_\sigma^{\mathtt{l}_i} \dashv \Gamma_1^i$ and $\Gamma_1^i \vdash \langle v_2 : \tau_2 \rangle_\sigma^{\mathtt{l}_i} \dashv \Gamma_1^i$ such that $\Gamma_1^i = \Gamma_2^i$. Consequently, we can deduce that $(\text{dom}(S_2) \backslash \mathcal{L}) = \Theta(dom(\Gamma_2^i))$ and for all $x \in dom(\Gamma_2^i)$ such that $\Gamma_2^i(x) = \langle \tilde{\tau} \rangle^{\mathtt{m}}$, there exists $v^\perp$ such that $S_2 \vdash v^\perp \sim \tilde{\tau}$ where $S_2(\ell_{\mathtt{m}::x}) = \langle v^\perp \rangle^{\mathtt{m}}$. As a result, $S_{2|\mathtt{l}_i} \sim \Gamma_2^i$ since $S_{1|\mathtt{l}_i} \sim \Gamma_1^i$. Finally, for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $S_{2|\mathtt{l}_j} \sim \Gamma_1^j$ and for all $(t, \{e\}^{\mathtt{l}}) \in T$ such that $\Gamma_1 \vdash \langle e : \tau \rangle_\sigma^{\mathtt{l}} \dashv \Gamma_2$, we have $S_{2|\mathtt{l}} \sim \Gamma_1$ for some Γ_1, Γ_2 and τ.

- *Base Case* $e \triangleq [\{v\}^{\mathtt{m}}]$. By *R-BlockB*, $S_2 = drop(S_1, \mathtt{m})$. According to Definition 3.4, S_2 contains all the values in S_1 except those owned with a lifetime $\mathtt{m}$. By *T-Block*, we have $\Gamma_1^i \vdash \langle v : \tau \rangle_\sigma^{\mathtt{m}} \dashv \Gamma_2^i$ and $\Gamma_2^i = drop(\Gamma_1^i, \mathtt{m})$. By applying Lemma 4.7, $\Gamma_1^i = \Gamma_2^i$. According to Figure 4.3, Γ_2 contains all the variables in Γ_1 except those with a lifetime $\mathtt{m}$. Since $S_{1|\mathtt{l}_i} \sim \Gamma_1^i$ then by applying Lemma 4.5, $drop(S_1, \mathtt{m}) \sim drop(\Gamma_1^i, \mathtt{m})$. Consequently, $S_{2|\mathtt{l}_i} \sim \Gamma_2^i$. Furthermore, if v represents a value located within a `Trc` (otherwise, the reasoning is straightforward), the contents of v will not be destroyed by the *drop* function 3.4 unless the counter of the `Trc` is equal to 1. When the counter is 1, it indicates that v can only be accessed by the current thread. In this case, deallocating v does not break the notion of safe abstraction for other threads, thus for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $S_{2|\mathtt{l}_j} \sim \Gamma_1^j$ and for all $(t, \{e\}^{\mathtt{l}}) \in T$ such that $\Gamma_1 \vdash \langle e : \tau \rangle_\sigma^{\mathtt{l}} \dashv \Gamma_2$, we have $S_{2|\mathtt{l}} \sim \Gamma_1$ for some Γ_1, Γ_2 and τ. On the other hand, if the counter is greater than 1, it will be safely decremented by 1 without affecting other threads' program store, and then the result is immediate.

- *Base Case* $e \triangleq [v; \overline{e}]$. By *R-Seq*, $S_2 = drop(S_1, \{v\})$. As per Definition 3.4, if v is an owning reference, the *drop* function recursively traverses v, dropping its contents. Therefore, we can conclude that S_2 includes all the values from S_1, except for v. By *T-Sequence* and by Lemma 4.7, $\Gamma_1^i \vdash \langle v : \tau_1 \rangle_\sigma^{\mathtt{l}_i} \dashv \Gamma_1$and $\Gamma_1^i \vdash \langle \overline{e} : \tau_i \rangle_\sigma^{\mathtt{l}_i} \dashv \Gamma_2^i$. By hypothesis, $S_{1|\mathtt{l}_i} \sim \Gamma_1^i$ then, based on Lemma 4.5 we can deduce that $S_{2|\mathtt{l}_i} \sim \Gamma_2^i$. Additionally, if v represents a value located within a `Trc` (otherwise, the reasoning is straightforward), the contents of v will not be destroyed by the *drop* function 3.4 unless the counter of the `Trc` is equal to 1. When the counter is 1, it indicates that v can only be accessed by the current thread. In this case, deallocating v does not break the notion of safe abstraction for other threads, thus for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $S_{2|\mathtt{l}_j} \sim \Gamma_1^j$ and for all $(t, \{e\}^{\mathtt{l}}) \in T$ such that $\Gamma_1 \vdash \langle e : \tau \rangle_\sigma^{\mathtt{l}} \dashv \Gamma_2$, we have $S_{2|\mathtt{l}} \sim \Gamma_1$ for some Γ_1, Γ_2 and τ. On the other hand, if the counter is greater than 1, it

will be safely decremented by 1 without affecting other threads' program store, and then the result is immediate.

- *Base Case* $e \triangleq [\texttt{cooperate}]$. By *R-Cooperate*, $S_1 = S_2$ then, S_2 is valid for $\texttt{1}_\texttt{i}$ since S_1 is valid for $\texttt{1}_\texttt{i}$. By *T-Cooperate* we have $\Gamma_1^i = \Gamma_2^i$ and $safeTrc(\Gamma_1^i)$. By inspection of Definition 4.5, we know that there is no borrowed shared data present in Γ_1. Consequently, we have $(\text{dom}(S_2) \backslash \mathcal{L}) = \Theta(dom(\Gamma_2^i))$ and for all $x \in dom(\Gamma_2^i)$ such that $\Gamma_2^i(x) = \langle \tilde{\tau} \rangle^\texttt{m}$, there exists $v^\perp$ such that $S_2 \vdash v^\perp \sim \tilde{\tau}$ where $S_2(\ell_{\texttt{m}::x}) = \langle v^\perp \rangle^\texttt{m}$. As a result, $S_{2|\texttt{1}_\texttt{i}} \sim \Gamma_2^i$ since $S_{1|\texttt{1}_\texttt{i}} \sim \Gamma_1^i$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $S_{2|\texttt{1}_\texttt{j}} \sim \Gamma_1^j$. For all $(t, \{e\}^\texttt{1}) \in T$ such that $\Gamma_1 \vdash \langle e : \tau \rangle_\sigma^\texttt{1} \dashv \Gamma_2$, we have $S_{2|\texttt{1}} \sim \Gamma_1$ for some Γ_1, Γ_2 and τ.

- *Base Case* $e \triangleq [\texttt{spawn}(f(\overline{v}))]$. By *R-Spawn*, we create a new thread $(t', \{e\}^\texttt{n})$ for some $* \succcurlyeq \texttt{n}$ such as $T \cup (t', \{e\}^\texttt{n})$ and $S_2 = S' \cup [\overline{\ell_{\texttt{n}::x} \mapsto \langle v_2' \rangle^\texttt{n}}]$ where $\overline{(S', v') = activate(S_1, v)}$. By inspection of Definition 3.6, the *activate* function recursively activates an inactive `Trc` value. This implies that all locations in the values of $\overline{v'}$ are in $\overline{v}$, but the only difference is that the values of an inactive `Trc` are activated in $\overline{v'}$, and by applying Lemma 4.12, we know that $\overline{v'}$ remain safely abstracted with their types. Moreover, by *R-Spawn* we have $(\text{dom}(S_{2|\texttt{1}_\texttt{i}}) \backslash \mathcal{L}) = (\text{dom}(S_{1|\texttt{1}_\texttt{i}} \backslash \mathcal{L})$. Thus, we can deduce that $S_{2|\texttt{1}_\texttt{i}} = S_{1|\texttt{1}_\texttt{i}}$, and S_2 is valid for $\texttt{1}_\texttt{i}$ since S_1 is valid for $\texttt{1}_\texttt{i}$. By *T-Spawn*, $\Gamma_1^i \vdash \langle \overline{v : \tau} \rangle_\sigma^{\texttt{1}_\texttt{i}} \dashv \Gamma_2^i$ and by applying Lemma 4.7, $\Gamma_1^i = \Gamma_2^i$. Consequently, we can deduce that $(\text{dom}(S_{2|\texttt{1}_\texttt{i}}) \backslash \mathcal{L}) = \Theta(dom(\Gamma_2^i))$ and for all $x \in dom(\Gamma_2^i)$ such that $\Gamma_2^i(x) = \langle \tilde{\tau} \rangle^\texttt{m}$, there exists $v^\perp$ such that $S_2 \vdash v^\perp \sim \tilde{\tau}$ where $S_2(\ell_{\texttt{m}::x}) = \langle v^\perp \rangle^\texttt{m}$. As a result, $S_{2|\texttt{1}_\texttt{i}} \sim \Gamma_2^i$. Since $\overline{v'}$ are values associated to the new thread, adding $\overline{v'}$ to S_2 cannot break the notion of safe abstraction for other existing threads, thus for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $S_{2|\texttt{1}_\texttt{j}} \sim \Gamma_1^j$ and for all $(t, \{e\}^\texttt{1}) \in T$ such that $\Gamma_1 \vdash \langle e : \tau \rangle_\sigma^\texttt{1} \dashv \Gamma_2$, we have $S_{2|\texttt{1}} \sim \Gamma_1$ for some Γ_1, Γ_2 and τ. Finally, it remains to demonstrate that there exists a safe abstraction between S and Γ of the new thread. By *T-Spawn*, $\Gamma_1^i \vdash \langle \overline{v : \tau} \rangle_\sigma^{\texttt{1}_\texttt{i}} \dashv \Gamma_1^i$ and $\Gamma_2 \vdash \overline{\texttt{S}} \Longleftarrow \overline{\tau}$. Based on Definition 4.16, we know that $\overline{v}$ are *well-typed* in Γ_1^i. Additionally, as per the Definition of 4.15, $\Gamma_2 \vdash \overline{\texttt{S}} \Longleftarrow \overline{\tau}$ ensures that $\overline{\texttt{S}}$ are compatible with $\overline{\tau}$. Since by *R-Spawn*, $S'_{|\texttt{n}}[\overline{\ell_{\texttt{n}::x} \mapsto \langle v_2' \rangle^\texttt{n}}]$ and by *T-Function* , $\Gamma_1[\overline{x \mapsto \langle \tau \rangle^\texttt{n}}]$ ($\overline{x}$ represent the arguments of the function f) such that $\overline{S_2 \vdash v_2' \sim \tau}$ (Definition 4.15), we can derive that $S_{2|\texttt{n}} \sim \Gamma_1$.

We will now prove the cases where $e_i \neq [v]$. Then the reduction of e_i produces an alternative expression e_i'. In this case, an intermediate typing environment Γ' exists such that $S_{2|\texttt{1}_\texttt{i}} \sim \Gamma'$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $S_{2|\texttt{1}_\texttt{j}} \sim \Gamma_1^j$ and for all $(t, \{e\}^\texttt{1}) \in T$ such that $\Gamma_1 \vdash \langle e : \tau \rangle_{\sigma'}^\texttt{1} \dashv \Gamma_2$, we have $S_{2|\texttt{1}} \sim \Gamma_1$ for some $\Gamma_1, \Gamma_2, \sigma'$ and τ. Then, our proof is based on the structural induction hypothesis as follows:

- *Inductive Case* $e_i \triangleq [\texttt{box}(e)]$. By *T-Box*, $\Gamma_1 \vdash \langle e : \tau \rangle_\sigma^{\texttt{1}_\texttt{i}} \dashv \Gamma_2$ and by applying our induction hypothesis over the derivation of e, we know that $\langle S_1 \triangleright e \xrightarrow{T}_i S_2 \triangleright e' \rangle^{\texttt{1}_i}$ and there exists Γ' such that $S_{2|\texttt{1}_\texttt{i}} \sim \Gamma'$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$, we

have $S_{2|\mathtt{l}_j} \sim \Gamma_1^j$ and for all $(t, \{e\}^{\mathtt{l}}) \in T$ such that $\Gamma_1 \vdash \langle e : \tau \rangle_{\sigma'}^{\mathtt{l}} \dashv \Gamma_2$, we have $S_{2|\mathtt{l}} \sim \Gamma_1$ for some $\Gamma_1, \Gamma_2, \sigma'$ and τ hence the result is immediate.

- *Inductive Case* $e_i \triangleq [\mathtt{trc}(e)]$. By *T-Trc*, $\Gamma_1 \vdash \langle e : \tau \rangle_{\sigma}^{\mathtt{l}_i} \dashv \Gamma_2$ and by applying our induction hypothesis over the derivation of e, we know that $\langle S_1 \triangleright e \xrightarrow{T}_i S_2 \triangleright e' \rangle^{\mathtt{l}_i}$ and there exists Γ' such that $S_{2|\mathtt{l}_i} \sim \Gamma'$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $S_{2|\mathtt{l}_j} \sim \Gamma_1^j$ and for all $(t, \{e\}^{\mathtt{l}}) \in T$ such that $\Gamma_1 \vdash \langle e : \tau \rangle_{\sigma'}^{\mathtt{l}} \dashv \Gamma_2$, we have $S_{2|\mathtt{l}} \sim \Gamma_1$ for some $\Gamma_1, \Gamma_2, \sigma'$ and τ hence the result is immediate.

- *Inductive Case* $e_i \triangleq [\mathtt{let\ mut}\ x = e]$. By *T-Declare*, $\Gamma_1 \vdash \langle e : \tau \rangle_{\sigma}^{\mathtt{l}_i} \dashv \Gamma_2$ and by applying our induction hypothesis over the derivation of e, we know that $\langle S_1 \triangleright e \xrightarrow{T}_i S_2 \triangleright e' \rangle^{\mathtt{l}_i}$ and there exists Γ' such that $S_{2|\mathtt{l}_i} \sim \Gamma'$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $S_{2|\mathtt{l}_j} \sim \Gamma_1^j$ and for all $(t, \{e\}^{\mathtt{l}}) \in T$ such that $\Gamma_1 \vdash \langle e : \tau \rangle_{\sigma'}^{\mathtt{l}} \dashv \Gamma_2$, we have $S_{2|\mathtt{l}} \sim \Gamma_1$ for some $\Gamma_1, \Gamma_2, \sigma'$ and τ hence the result is immediate.

- $e_1 \triangleq [\omega = e]$. By *T-Assign*, $\Gamma_1 \vdash \langle e : \tau \rangle_{\sigma}^{\mathtt{l}_i} \dashv \Gamma_2$ and by applying our induction hypothesis over the derivation of e, we know that $\langle S_1 \triangleright e \xrightarrow{T}_i S_2 \triangleright e' \rangle^{\mathtt{l}_i}$ and there exists Γ' such that $S_{2|\mathtt{l}_i} \sim \Gamma'$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $S_{2|\mathtt{l}_j} \sim \Gamma_1^j$ and for all $(t, \{e\}^{\mathtt{l}}) \in T$ such that $\Gamma_1 \vdash \langle e : \tau \rangle_{\sigma'}^{\mathtt{l}} \dashv \Gamma_2$, we have $S_{2|\mathtt{l}} \sim \Gamma_1$ for some $\Gamma_1, \Gamma_2, \sigma'$ and τ hence the result is immediate.

- $e_1 \triangleq [e_1 \oplus e_2]$. By *T-Arithm*, $\Gamma_1 \vdash \langle e_1 : \mathtt{int} \rangle_{\sigma}^{\mathtt{l}_i} \dashv \Gamma_2$ and $\Gamma_2 \vdash \langle e_2 : \mathtt{int} \rangle_{\sigma}^{\mathtt{l}_i} \dashv \Gamma_3$. By applying our induction hypothesis over the derivation of e_1, we know that $\langle S_1 \triangleright e_1 \xrightarrow{T}_i S_2 \triangleright e' \rangle^{\mathtt{l}_i}$ and there exists Γ' such that $S_{2|\mathtt{l}_i} \sim \Gamma'$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $S_{2|\mathtt{l}_j} \sim \Gamma_1^j$ and for all $(t, \{e\}^{\mathtt{l}}) \in T$ such that $\Gamma_1 \vdash \langle e : \tau \rangle_{\sigma'}^{\mathtt{l}} \dashv \Gamma_2$, we have $S_{2|\mathtt{l}} \sim \Gamma_1$ for some $\Gamma_1, \Gamma_2, \sigma'$ and τ hence the result is immediate.

- $e_1 \triangleq [e_1 \otimes e_2]$. By *T-Cond*, $\Gamma_1 \vdash \langle e_1 : \mathtt{int} \rangle_{\sigma}^{\mathtt{l}_i} \dashv \Gamma_2$ and $\Gamma_2 \vdash \langle e_2 : \mathtt{int} \rangle_{\sigma}^{\mathtt{l}_i} \dashv \Gamma_3$. By applying our induction hypothesis over the derivation of e_1, we know that $\langle S_1 \triangleright e_1 \xrightarrow{T}_i S_2 \triangleright e' \rangle^{\mathtt{l}_i}$ and there exists Γ' such that $S_{2|\mathtt{l}_i} \sim \Gamma'$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $S_{2|\mathtt{l}_j} \sim \Gamma_1^j$ and for all $(t, \{e\}^{\mathtt{l}}) \in T$ such that $\Gamma_1 \vdash \langle e : \tau \rangle_{\sigma'}^{\mathtt{l}} \dashv \Gamma_2$, we have $S_{2|\mathtt{l}} \sim \Gamma_1$ for some $\Gamma_1, \Gamma_2, \sigma'$ and τ hence the result is immediate.

- $e_1 \triangleq [\{e\}^{\mathtt{m}}]$. In this case, by *R-BlockA* we have $\langle S_1 \triangleright e \xrightarrow{T}_i S_2 \triangleright e' \rangle^{\mathtt{m}}$ for some $\mathtt{l} \geq \mathtt{m}$, and by induction hypothesis over the derivation of e that reduces to e', we assume that there exists Γ' such that $S_{2|\mathtt{l}_i} \sim \Gamma'$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $S_{2|\mathtt{l}_j} \sim \Gamma_1^j$ and for all $(t, \{e\}^{\mathtt{l}}) \in T$ such that $\Gamma_1 \vdash \langle e : \tau \rangle_{\sigma'}^{\mathtt{l}} \dashv \Gamma_2$, we have $S_{2|\mathtt{l}} \sim \Gamma_1$ for some $\Gamma_1, \Gamma_2, \sigma'$ and τ.

- $e_i \triangleq [\overline{e}]$. By *T-Sequence*, $\Gamma_1 \vdash \langle e_1 : \tau \rangle_{\sigma}^{\mathtt{l}_i} \dashv \Gamma_{1_1} \ldots \Gamma_{1_n} \vdash \langle e_n : \tau_i \rangle_{\sigma}^{\mathtt{l}_i} \dashv \Gamma_2$ and by applying our induction hypothesis over the derivation of e_1, we know that $\langle S_1 \triangleright e_1 \xrightarrow{T}_i S_2 \triangleright e_1' \rangle^{\mathtt{l}_i}$ and there exists Γ' such that $S_{2|\mathtt{l}_i} \sim \Gamma'$ and for all $j \in$

$\{1, ..., i-1, i+1, .., N\}$, we have $S_{2|1_j} \sim \Gamma_1^j$ and for all $(t, \{e\}^1) \in T$ such that $\Gamma_1 \vdash \langle e : \tau \rangle^1_{\sigma'} \dashv \Gamma_2$, we have $S_{2|1} \sim \Gamma_1$ for some $\Gamma_1, \Gamma_2, \sigma'$ and τ hence the result is immediate.

- *Inductive Case* $e_i \triangleq [\texttt{spawn}(f(\overline{e}))]$. By *T-Spawn*, we have $\Gamma_1 \vdash \langle \overline{e : \tau} \rangle^{1_i}_{\sigma} \dashv \Gamma_2$ and by applying our induction hypothesis over the derivation of e_0, we know that $\langle S_1 \triangleright e_0 \xrightarrow{T}_i S_2 \triangleright e' \rangle^{1_i}$ and there exists Γ' such that $S_{2|1_i} \sim \Gamma'$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $S_{2|1_j} \sim \Gamma_1^j$ and for all $(t, \{e\}^1) \in T$ such that $\Gamma_1 \vdash \langle e : \tau \rangle^1_{\sigma'} \dashv \Gamma_2$, we have $S_{2|1} \sim \Gamma_1$ for some $\Gamma_1, \Gamma_2, \sigma'$ and τ hence the result is immediate.

□

Now, we can proceed to the proof of Lemma 4.10:

The preservation Lemma serves to uphold the following aspects during the reduction of a specific thread $(t, \{e_i\}^{1_i})$: the validity of the global state, the *well-typed* expression of all existing threads in $T_1 \cup (t, \{e_i'\}^{1_i}) \cup T$, and the notion of safe abstraction. To achieve this objective, the preservation proof relies on various families of Lemmas that follow a consistent pattern, which can be outlined as follows:

`Proof.` Firstly, by applying Lemma 4.11, we know that executing a thread from a valid global state preserves the validity of the resulting global state. This Lemma ensures that after the execution of a step by a given thread, its local state remains valid and that it also preserves the validity of the local states of the other threads. Essentially, it guarantees that no aliases are created in the global state. Secondly, by applying Lemma 4.12, tells us that when a thread takes a step, the type of its reduced expression remains unchanged (i.e., it remains well-typed). This ensures that the *well-typed* expressions of other threads in the program, including new threads, are not violated. It also guarantees the presence of a valid store typing σ'. Lastly, by applying Lemma 4.13, we know that after a given transition for a thread, its typing environment remains a safe abstraction with the runtime program's store. This property also extends to all other threads. The preservation proof relies on these three Lemmas to fulfill its ultimate objective. Additionally, we can augment the preservation by including a property that, during each step execution, the resulting environment upholds the invariants of the borrows and `Trcs` (as defined in 4.25). This aspect is ensured by applying Lemma 4.8. □

2.4.5 Progress and Preservation Slice

Lemma 4.9 proves that if a thread is *well-typed*, it can take a step. In this section, we introduce the Slice Progress Lemma that says that if a thread is *well-typed*, it can take one or more steps until it terminates or until it cooperates as follows:

Lemma 4.14 (Slice Progress) *Let $S_1 \triangleright e_1$ be a valid local state for some lifetime $\mathtt{l}$. Let σ be a store typing such that $S_1 \triangleright e_1 \vdash \sigma$ and let Γ_1 be a well-formed typing environment with*

respect to $\mathtt{l}$ *such that* $S \sim \Gamma_1$ *and* $\Gamma_1 \vdash \langle e_1 : \tau \rangle_\sigma^{\mathtt{l}} \dashv \Gamma_2$ *for some* τ *and* Γ_2. *Then, there are some* T *and some state* $S_2 \triangleright e_2$ *such that* $\langle S_1 \triangleright e_1 \overset{T}{\Longrightarrow} S_2 \triangleright e_2 \rangle^{\mathtt{l}}$.

Proof. We suppose that each reduction by the relation ($\overset{T}{\Longrightarrow}$) has N reductions, where N is a finite number of steps ($\overset{T}{\rightarrow}_i$). Starting from a valid state $S_1 \triangleright e_1$ and a *well-typed* expression e_1, we proceed by induction on the maximum number of reductions (i.e. N). By hypothesis and by Lemma 4.9, we establish that we can take one step with e_1. According to Lemma 4.9, if e_1 is a value, this implies that N = 0, and thus, the *R-ThreadTerm* applies. On the other hand, N⩾1, we have two cases, $S_1 \triangleright e_1 \overset{T}{\rightarrow}_{1|0} S_1' \triangleright e_1'$ denotes a reduction by 1 or 0 for some $S_1' \triangleright e_1'$ and T. In the first case ($\rightarrow_1$) the *R-ThreadCoop* applies. In the second case ($\rightarrow_0$), by applying Lemma 4.10, we can conclude that $S_1' \triangleright e_1'$ remains valid, e_1' remains *well-typed*, and hence we are able to take a step with e_1'. Consequently, by applying the induction hypothesis on the maximum number of steps (reduction) from $S_1' \triangleright e_1'$ (i.e. N-1), we can proceed with the *R-Thread* rule. Finally, note that regardless of the considered configuration, only one semantic rule will be applied at most. In other words, the operational semantics of our language is *deterministic*. □

Lemma 4.10 guarantees that when a thread takes a step, it preserves the validity of the global state, its expression remains *well-typed* and it satisfies the notion of safe abstraction. In addition, these properties are preserved for other threads. With Lemma 4.15, we demonstrate the same results for maximal thread executions as follows:

Lemma 4.15 (Slice Preservation) *Let* $T_1 = \{t_i, \{e_i\}^{\mathtt{l}_i} \mid 1 \leqslant i \leqslant N\}$, S_1, σ *such that* T_1, S_1 *is a valid global state and for all* $i \in \{1, ..., N\}$, *we have* $S_1 \triangleright e_i \vdash \sigma$, $S_{1|\mathtt{l}_i} \sim \Gamma_1^i$ *and* $\Gamma_1^i \vdash \langle e_i : \tau_i \rangle_\sigma^{\mathtt{l}_i} \dashv \Gamma_2^i$ *for some* Γ_1^i, Γ_2^i *and* τ_i. *Let* $i \in \{1, ..., N\}, T$ *and* $S_2 \triangleright e_i'$ *such that* $\langle S_1 \triangleright e_i \overset{T}{\Longrightarrow} S_2 \triangleright e_i' \rangle^{\mathtt{l}_i}$ *then, there are* Γ' *and* σ' *such that* $T_{1\setminus t} \cup (t, \{e_i'\}^{\mathtt{l}_i}) \cup T, S_2$ *remains valid,* $\Gamma' \vdash \langle e_i' : \tau_i \rangle_{\sigma'}^{\mathtt{l}_i} \dashv \Gamma_2^i$, $S_2 \triangleright e_i' \vdash \sigma'$, $S_{2|\mathtt{l}_i} \sim \Gamma'$ *and for all* $j \in \{1, ..., i-1, i+1, .., N\}$, *we have* $\Gamma_1^j \vdash \langle e_j : \tau_j \rangle_{\sigma'}^{\mathtt{l}_j} \dashv \Gamma_2^j$ *and* $S_2 \triangleright e_j \vdash \sigma'$ *and* $S_{2|\mathtt{l}_j} \sim \Gamma_1^j$ *and for all* $(t, \{e\}^{\mathtt{l}}) \in T$, *we have* $\Gamma_1 \vdash \langle e : \tau \rangle_{\sigma'}^{\mathtt{l}} \dashv \Gamma_2$ *and* $S_2 \triangleright e \vdash \sigma'$ *and* $S_{2|\mathtt{l}} \sim \Gamma_1$ *for some* Γ_1, Γ_2 *and* τ.

Proof. We suppose that each reduction by the relation ($\overset{T}{\Longrightarrow}$) has N reductions, where N is a finite number of steps ($\overset{T}{\rightarrow}_i$). Starting from a valid state $S_1 \triangleright e_1$ and a *well-typed* expression e_1, we proceed by induction on the maximum number of reductions (i.e. N). By hypothesis and by Lemma 4.10, which relies on the sub-lemmas 4.11, 4.12 and 4.13, we can deduce that after one step of reduction from e_i to e_i', there are Γ' and σ' such that $T_{1\setminus t} \cup (t, \{e_i'\}^{\mathtt{l}_i}) \cup T, S_2$ remains valid and $\Gamma' \vdash \langle e_i' : \tau_i \rangle_{\sigma'}^{\mathtt{l}_i} \dashv \Gamma_2^i$ and $S_2 \triangleright e_i' \vdash \sigma'$ and $S_{2|\mathtt{l}_i} \sim \Gamma'$ and for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $\Gamma_1^j \vdash \langle e_j : \tau_j \rangle_{\sigma'}^{\mathtt{l}_j} \dashv \Gamma_2^j$ and $S_2 \triangleright e_j \vdash \sigma'$ and $S_{2|\mathtt{l}_j} \sim \Gamma_1^j$ and for all $(t, \{e\}^{\mathtt{l}}) \in T$, we have $\Gamma_1 \vdash \langle e : \tau \rangle_{\sigma'}^{\mathtt{l}} \dashv \Gamma_2$ and $S_2 \triangleright e \vdash \sigma'$ and $S_{2|\mathtt{l}} \sim \Gamma_1$ for some Γ_1, Γ_2 and τ. Thus, by induction hypothesis on the maximum number of steps (reduction) from $S_2 \triangleright e_i'$ (i.e. N-1), we can achieve the desired result. □

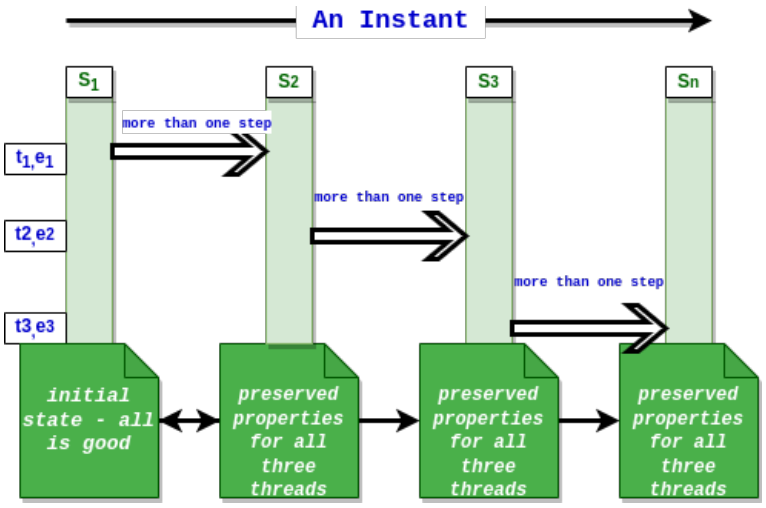

Figure 4.8: Instant Preservation

2.4.6 Progress and Preservation Instant

Based on Figure 4.6, we can derive Figure 4.8, which illustrates that when all threads in a program are *well-typed*, they can be executed more than once ($\overset{T}{\Longrightarrow}$). In this context, the execution of a thread preserves the properties of the other threads, ensuring that the execution of one thread does not freeze the execution of the others. As a result, the execution of a FR_{FT} program can progress over multiple instants. The following lemma states that when each thread in the set T_1 undergoes multiple executions, the program takes an instant:

Lemma 4.16 (Instant Progress) *Let* $T_1 = \{t_i, \{e_i\}^{1_i} | 1 \leqslant i \leqslant N\}$, S_1, σ *such that* T_1, S_1 *be a valid global state and for all* $i \in \{1, ..., N\}$, *we have* $S_1 \triangleright e_i \vdash \sigma$, $S_{1|1_i} \sim \Gamma_1^i$ *and* $\Gamma_1^i \vdash \langle e_i : \tau_i \rangle_\sigma^{1_i} \dashv \Gamma_2^i$ *for some* Γ_1^i, Γ_2^i *and* τ_i. *Then, either* T_1, S_1 *is a terminal state or there exists some* T_1', S_1' *such that* $T_1, S_1 \Longrightarrow T_1', S_1'$.

`Proof.` Let N be the cardinality of T_1 (i.e. T_1 is a finite set of threads). We proceed by induction on the cardinality of T_1 (i.e. N). Then, if T_1 is an empty set, where N=0, then the *R-InstantEnd* applies. In other hands, where N≥1, for a given $(t, \{e\}^1) \in T_1$ such that e is *well-typed*, we apply Lemma 4.14 where we have $\langle S_1 \triangleright e \overset{T}{\Longrightarrow} S_1' \triangleright e' \rangle^1$ for some e'. Furthermore, we apply Lemma 4.15 where we have T_1', S_1' remains valid, e' remains *well-typed*, $S_1' \triangleright e' \vdash \sigma'$ and $S_{1|1}' \sim \Gamma'$ for some Γ' and σ'. Moreover, for all $j \in \{1, ..., i-1, i+1, .., N\}$, we have $\Gamma_1^j \vdash \langle e_j : \tau_j \rangle_{\sigma'}^{1_j} \dashv \Gamma_2^j$ and $S_1' \triangleright e_j \vdash \sigma'$ and $S_{1|1_j}' \sim \Gamma_1^j$. Therefore, by induction hypothesis on the cardinality of $(T_{1 \setminus t})$ (i.e. N-1), we proceed with *R-Instant*. □

Lemma 4.16 ensures that when all threads in T_1 execute more than once, the program takes an instant. The following lemma guarantees that when a program takes an instant, all the threads that exist in the resulting set T_1' remain *well-typed* such that the global state also remains valid.

Lemma 4.17 (Instant Preservation) *Let* $T_1 = \{t_i, \{e_i\}^{1_i} \mid 1 \leqslant i \leqslant N\}$, S_1, σ *such that* T_1, S_1 *be a valid global state and for all* $i \in \{1, ..., N\}$, *we have* $S_1 \triangleright e_i \vdash \sigma$, $S_{1|1_i} \sim \Gamma_1^i$ *and* $\Gamma_1^i \vdash \langle e_i : \tau_i \rangle_\sigma^{1_i} \dashv \Gamma_2^i$ *for some* Γ_1^i, Γ_2^i *and* τ_i. *If* $T_1, S_1 \Longrightarrow T_1', S_1'$ *for some* T_1', S_1', *then* T_1', S_1' *remains valid and there exists* σ' *such that for all* $j \in \{1, ..., M\}$, *where M is the cardinality of* T_1', *we have* $S_1' \triangleright e_j' \vdash \sigma'$, $S_{1|1_j}' \sim \Gamma^j$ *and* $\Gamma^j \vdash \langle e_j' : \tau_j \rangle_{\sigma'}^{1_j} \dashv \Gamma_2^j$ *for some* Γ^j, Γ_2^j, e_j' *and* τ_j.

`Proof.` Let N be the cardinality of T_1 (i.e. T_1 is a finite set of threads). We proceed by induction on the cardinality of T_1 (i.e. N). Then, by hypothesis and by applying lemma 4.15, we know that, for some $i \in \{1, ..., N\}$, the maximal execution for a given thread $(t, \{e_i\}^{1_i}) \in T_1$ produces e_i' and preserves the following: T_1', S_1' remains valid, $S_1' \triangleright e_i' \vdash \sigma'$, $S_{1|1_i}' \sim \Gamma^i$ and $\Gamma^i \vdash \langle e_i' : \tau_i \rangle_{\sigma'}^{1_i} \dashv \Gamma_2^j$ for some Γ^i, Γ_2^i, e_i' and τ_i and also for all $(t, \{e\}^1) \in T_{1\setminus t}'$, we have $\Gamma_1 \vdash \langle e : \tau \rangle_{\sigma'}^1 \dashv \Gamma_2$ and $S_1' \triangleright e \vdash \sigma'$ and $S_{1|1}' \sim \Gamma$ for some Γ_1, Γ_2 and τ. Thus, by induction hypothesis on the cardinality of $(T_{1\setminus t})$ (i.e. N-1), we reach the required result.
□

2.5 The Type, Borrow, and Concurrency Safety Theorem

Finally, we present the type, borrow and concurrency safety theorem, which establishes that a *well-typed* synchronous reactive program is guaranteed to execute zero or more instants or achieve a terminal state, which implies that for all $(t, \{e\}^1) \in T_1$, e is a value. Note that, $S_\varnothing$ and $\sigma_\varnothing$ denote an empty program store and an empty store typing respectively.

Theorem 4.18 (Type, Borrow and Concurrency Safety) *Let e be an expression and let* Γ *be a well-formed typing environment with respect to a lifetime* 1 *such that* $\varnothing \vdash \langle e : \tau \rangle_{\sigma_\varnothing}^1 \dashv \Gamma$ *for some* τ. *If* $(t, \{e\}^1), S_\varnothing \Longrightarrow^* T, S$ *for some* T, S *then, either* T, S *is a terminal state or there exists some* T', S' *such that* $T, S \Longrightarrow T', S'$.

`Proof.` The proof of this theorem is given by Lemmas 4.17 and 4.16 as follows: by applying Lemma 4.17 we have T, S remains valid and there exists σ' such that for all $j \in \{1, ..., M\}$, where $T = \{t_j, \{e_j\}^{1_j} \mid 1 \leqslant j \leqslant M\}$, we have $S \triangleright e_j \vdash \sigma'$, $S_{|1_j} \sim \Gamma^j$ and $\Gamma^j \vdash \langle e_j : \tau_j \rangle_{\sigma'}^{1_j} \dashv \Gamma_2^j$ for some Γ^j, Γ_2^j and τ_j. Afterwards, by applying Lemma 4.16 we have either T, S is a terminal state or there exists some T', S' such that $T, S \Longrightarrow T', S'$. □

Chapter 5

MSSL, Extensions

We're here to put a dent in the universe.
Otherwise why else even be here?

– Steve Jobs

In this chapter, we enhance FR_{FT} by incorporating new reactive and non-reactive extensions. Specifically, these extensions encompass a control flow and function calls outside the `spawn` expression, necessitating the inclusion of effects in our type system. Furthermore, we introduce the pivotal concept of signals in MSSL. These signals are efficiently managed in memory through a reference counting mechanism, enabling effective memory handling and manipulation. Lastly, we provide comprehensive semantic and typing rules for each of these extensions.

Dans ce chapitre, nous améliorons FR_{FT} en incorporant de nouvelles extensions réactives et non réactives. Plus précisément, ces extensions incluent les flots de contrôle et des appels de fonctions en dehors de l'expression `spawn`*, ce qui nécessite l'inclusion des effets dans notre système de types. De plus, nous introduisons le concept essentiel de signaux dans MSSL. Ces signaux sont gérés efficacement en mémoire grâce à un mécanisme de comptage de références, permettant une manipulation et une gestion efficaces de la mémoire. Enfin, nous fournissons des règles sémantiques et de typage détaillées pour chacune de ces extensions*

Values	v	::=	$\epsilon_0 \mid \epsilon_1 \mid n \mid \texttt{true} \mid \texttt{false} \mid \ell_a^{\blacksquare} \mid \ell_a^{\blacklozenge} \mid \ell_a^{\lozenge} \mid \ell_a^{\circ} \mid \ell_{\texttt{m::}x}^{\circ} \mid \ell_a^{s}$
Partial Values	$v^{\perp}$	::=	$v \mid \perp$
Types	τ	::=	$\epsilon \mid \texttt{int} \mid \texttt{bool} \mid \texttt{\&mut}\ \overline{\omega} \mid \texttt{\&}\ \overline{\omega} \mid \lozenge\overline{\omega} \mid \blacklozenge\tau \mid \blacksquare\tau$
Partial Types	$\widetilde{\tau}$	::=	$\tau \mid \blacksquare\widetilde{\tau} \mid \lfloor\widetilde{\tau}\rfloor$
LVals	ω	::=	$x \mid *\omega$
Expressions	e	::=	$v \mid \omega \mid \hat{\omega} \mid \overline{e} \mid \{e\}^{1} \mid \texttt{let mut}\ x = e \mid \texttt{box}(e) \mid \texttt{\&[mut]}\ \omega \mid \omega = e$
			$\mid \texttt{trc}(e) \mid \omega.clone \mid e_1 \oplus e_2 \mid e_1 \otimes e_2 \mid \texttt{if}(e)\ \{e_1\}^{n}\ \texttt{else}\ \{e_2\}^{m}$
			$\mid f(\overline{e};\overline{s}) \mid \texttt{spawn}(f(\overline{e};\overline{s})) \mid \texttt{cooperate} \mid \texttt{Sig}\ s$
			$\mid \texttt{emit}(s) \mid \texttt{when}(s)\ \{e_1\}^{m} \mid \texttt{watch}(s)\ \{e_1\}^{m}$
Functions	f	::=	$\texttt{fn}\ f(\overline{\texttt{mut}\ x : \texttt{S}};\overline{s}) \rightarrow_{\kappa} \texttt{S}\{\overline{e}\}^{1}$
Signatures	S	::=	$\epsilon \mid \texttt{int} \mid \texttt{bool} \mid \blacksquare\texttt{S} \mid \blacklozenge\texttt{S} \mid \lozenge\texttt{S} \mid \texttt{\&'l mut S} \mid \texttt{\&'l S}$
Programs	p	::=	$f\ p \mid \{e\}^{1}$

Figure 5.1: Full MSSL syntax

1 Extensions of MSSL

MSSL, as a language, focuses on the development of a robust type system and provides borrowing safety, ensuring reliable data sharing between threads. Furthermore, MSSL introduces reactivity to Rust by providing reactive constructs and enabling thread synchronization. As mentioned earlier, FR_{FT} serves as the cooperative kernel of MSSL, utilising a type system based on Rust's ownership and borrowing model, following FR's semantic approach [73]. However, FR_{FT} lacks signal synchronization operations. This chapter introduces MSSL as a full language, incorporating the following extensions to FR_{FT}: (1) the `if/else` extension, (2) the function calls outside the `spawn` expression, including the lifetime annotation for references (section 1.5.5) and (3) a complete cooperative extension that includes signals and distinguishes between cycle and instant (section 1.6). Henceforth, we will refer to the language as MSSL instead of FR_{FT} in the rest of this chapter.

1.1 Full Syntax of MSSL

The addition of the new extensions entails making certain adjustments to the semantics of the FR_{FT} model and its type system. Initially, we outline the complete syntax of MSSL, incorporating the new constructs. As depicted in Figure 5.1 and compared to Figure 3.5, we introduce the necessary syntactic extensions to support the following: (1) the `if/else` extension, (2) function calls, (3) signal creation and emission, (4) implicit cooperation (i.e. waiting for a signal), and (5) the *weak watching preemption* primitive (i.e. the `watch` expression). The modifications are implemented as follows:

Values. We introduce two forms for the ϵ values: ϵ_1 and ϵ_0. To differentiate, a ϵ_1 value is produced by the `cooperate` expression. Otherwise, ϵ_0 is produced by an expression that finishes (e.g. $\{\texttt{let mut}\ x = 1\}^{1}$). Regarding memory locations, Figure 5.1 now distinguishes

six forms, with the previous five remaining unchanged, and a new one is added: the value ℓ_a^s, representing an *owning* reference that denotes the location of a specific signal resulting from the reduction of a `Sig` s expression. As far as types are concerned, no new ones are added.

Expressions. We add six new expressions to MSSL, including reactive expressions such as: (1) `Sig` s creates a new signal and binds it to the identifier s. Note that s represents the signal name, distinct from variables in MSSL's semantics. (2) `emit`(s) emits the signal specified by the parameter s. (3) `when`(s) $\{e_1\}^{\mathtt{m}}$, for a given signal s, if it is emitted, the block $\{e_1\}^{\mathtt{m}}$ is executed; otherwise, the current thread will cooperate until the next cycle. And (4) `watch`(s) $\{e_1\}^{\mathtt{m}}$ executes the block $\{e_1\}^{\mathtt{m}}$ without checking if the signal s (specified as a parameter) is present. If the `watch`'s body did not terminate, it will wait until the end of the current instant to verify if s has been emitted during the instant. If s is emitted, it invokes function 5.7 (explained later), which terminates the non-terminating body immediately, and the corresponding thread regains control in the next instant. In addition, we introduce two non-reactive expressions: (5) `if`(e) $\{e_1\}^{\mathtt{n}}$ `else` $\{e_2\}^{\mathtt{m}}$ represents a control flow expression and (6) the invoked function $f(\overline{e};\overline{s})$ that takes two sequences as parameters: $\overline{v}$ (the sequence of values denoted by $\overline{e}$) and $\overline{s}$ (the sequence of signals, if any). When the function is invoked, we increment the counter for each signal in $\overline{s}$, and we decrement it when the function finishes its execution. Similarly, the expression `spawn` now assumes a sequence of signals. When `spawn` is executed, we increment the counter for each signal, and when the function's execution is complete, we decrement the counters.

Functions and Signatures. Allowing the function to be invoked outside the `spawn` expression imposes more effort. In such cases, we can have a return type, and the function can have parameters of type reference or an inactive `Trc`. To ensure the validity of references after the function is invoked, we introduce the lifetime annotation syntax for references (similar to Rust). This annotation becomes a part of the signature of a reference (e.g., $\&'a$ [`mut`]`S`). On top of that, in MSSL, we introduce forms of signatures that are different from types. For example, $\diamond$`S` and $\&'a$ [`mut`]`S` are signatures but not types according to the syntax. In order to ensure memory safety when data sharing between threads is allowed, we associate an effect κ with each function declaration. The effect κ is defined as a Boolean: `0` indicates that this function does not contain `cooperate` or `when` expressions and `1` indicates that the function body can have at least one reactive construct (`cooperate` or `when`) (e.g. as `Throw Exception` for Java). Having a type system with effect κ is essential to avoid scenarios where one thread calls a function and cooperates before the function terminates, while another thread takes control and creates unsuitable values in the memory of the former one. Therefore, the function signature provides information at typing time about whether the function contains a reactive construct or not, allowing us to handle references to shared data appropriately when the function is called.

1.2 Operational Semantics

To ensure the proper creation and transmission of signals, we need to include essential information in our semantics. As a result, we introduce a new form of states, denoted as $\psi, S \triangleright e$ where ψ is a signal environment in MSSL. Note that in MSSL, there is a single

heap represented in S; however, for the simplicity of semantics, we have specifically specified ψ to serve as an environment dedicated to signals. This signal environment maps locations to signal values, and S is a program store that maps locations to partial values $\langle v^{\perp} \rangle^{1}$. Hence, we supplement the small-step semantics with a signal environment, ψ, such that all reduction rules take the form: $\langle \psi, S \triangleright e \xrightarrow{T}_{i} \psi', S' \triangleright e' \rangle^{1}$. For instance, the rule $\langle \psi_{\varnothing}, S_{\varnothing} \triangleright \{\texttt{Sig}\ s\}^{1} \rightarrow_{0} \{\ell_a \mapsto \langle 0 \rangle^{1}\}, \{\ell_{\,1::s} \mapsto \langle \ell_a^{s} \rangle^{1}\} \triangleright \epsilon_0 \rangle^{1}$ illustrates simple reduction rule to create a signal s. In this case, the signal s is associated with a location $\ell_{1::s}$ in S, allocated with the lifetime $\texttt{1}$, and its value is $\texttt{0}$, as indicated in the signal environment ψ (which is absent by default). The expression ϵ_0 immediately terminates, indicating that the signal creation is complete. According to the semantics, ℓ_a is a heap-allocated location stored in ψ with the counter initialized to 1. Finally, we implicitly assume that all existing MSSL reduction rules are extended to the said form in an obvious way or instance, the *R-Copy* rule is updated as follows:

$$\frac{\mathit{read}(S, \omega, \texttt{1}) = \langle v \rangle^{\texttt{m}}}{\langle \psi, S \triangleright \hat{\omega} \rightarrow_0 \psi, S \triangleright v \rangle^{\texttt{1}}} (\textit{R-Copy}) \qquad (\textit{R-Copy})$$

Thread. Next, for the execution of a thread by the scheduler, we employ the updated reduction rule: $\langle \psi, S \triangleright e \overset{T_0}{\Longrightarrow} \psi', S' \triangleright e' \rangle^{1}$. Moreover, we assume that at the beginning of each instant, all threads are gathered in a set T. As a thread executes, it is removed from T. When all threads from T have been executed, it marks the completion of the current cycle. This concept can be expressed as follows: $T, \psi, S \Rrightarrow T', \psi', S'$.

Instant. Lastly, a complete instant is defined as a sequence of cycles in which all threads execute at their maximum during the current instant. It can be represented as follows: $T, \psi, S \Mapsto T', \psi', S'$.

1.3 Preliminaries

Before introducing the new extensions, we conduct a review of the functions defined in chapter 3 to make the required modifications as follows:

In order to enable the invocation of a function outside of the `spawn` expression in MSSL, we need to make adjustments to the loc function (defined in chapter 3). This modification becomes essential to identify the locations associated with such functions, as elaborated in detail in Section 1.5.5.

Definition 5.1 (Expanded Location) *Let S be a program store, ω an lval and $\texttt{l}$ a lifetime. The partial function $loc(S, \omega, \texttt{l})$ returns the location related to ω in S. Then, we define $loc(S, \omega, \texttt{l})$ as follows:*

$$loc(S, x, \texttt{l}) = \ell_{\texttt{m}::x} \textit{ where } S(\ell_{\texttt{m}::x}) = \langle . \rangle^{\texttt{m}} \textit{ and } \neg \exists \texttt{n}.(\texttt{m} \succeq \texttt{n} \wedge S(\ell_{\texttt{n}::x}) = \langle . \rangle^{\texttt{n}}) \wedge \texttt{m} \succeq \texttt{l}$$

...

According to this Definition, the condition $\neg\exists \mathtt{n}.(\mathtt{m} \geq \mathtt{n} \wedge S(\ell_{\mathtt{n}::x}) = \langle . \rangle^{\mathtt{n}})$ ensures the correct retrieval of variable locations within S when a function is invoked. Specifically, the last location associated with the variable requested as an argument is returned.

Furthermore, as depicted in Section 1.1, MSSL is specialized in handling signals as well. In other words, signals are considered as a reference counting similar to `Trc`. Therefore, to accommodate signal management, we extend the *drop* function in Definition 3.4 to include the capability of dropping signals when required, as demonstrated below:

Definition 5.2 (Expanded Drop) *Let S be a program store and let $\mathtt{m}$ be a lifetime. The $drop(S, \mathtt{m})$ function is used to deallocate values with the lifetime $\mathtt{m}$. Then, $drop(S, \mathtt{m})$ relies on $drop(\pounds, \rho)$, where $\pounds$ can refer to either S or ψ, and ρ represents a drop set as follows:*

$$
\begin{array}{lcl}
drop(S, \varnothing) & = & S \\
drop(S, \rho \cup \{v^{\perp}\}) & = & drop(S, \rho) \ \ where\ (v^{\perp} \neq \ell_a^{\blacksquare} \wedge v^{\perp} \neq \ell_a^{\blacklozenge} \wedge v^{\perp} \neq \ell_a^{\diamond} \wedge v^{\perp} \neq \ell_a^{s}) \\
\ldots & & \\
drop(\psi, \rho \cup \ell_a^{s}) & = & \begin{cases} drop(\psi - \{\ell_a \mapsto \langle v \rangle^{1}\}, \rho \cup \{v\}) \ \ where\ \psi(\ell_a) = \langle v \rangle^{1} \\ drop(\psi[\![\ell_a \mapsto \langle v \rangle^{i}]\!], \rho\} \ \ where\ \psi(\ell_a) = \langle v \rangle^{i+1} \end{cases}
\end{array}
$$

Similar to `Trc`, the dealloaction of slots allocated by a `Sig` expression depends on the counter (i). As soon as the counter falls to 1, it is reliable to perform the deallocation. Note that signals in MSSL adhere to the copy semantics, which will be explained in detail later.

Finally, we will update our evaluation context to include the new extensions and ensure that *R-Sub* is always applicable. The updated evaluation context is as follows:

Definition 5.3 (Evaluation Context) *An evaluation context is an expression containing a single occurrence of $[\![.]\!]$ (the hole) instead of a sub-expression. In other words, it is used to describe where the next reduction step takes place in the program as follows:*

$$E \ ::= [\![.]\!] \mid \ldots \mid \mathtt{if}(E)\ \{e_1\}^{\mathtt{n}}\ \mathtt{else}\ \{e_2\}^{\mathtt{m}} \mid f(\overline{v}, E, \overline{e}; \overline{s}) \mid \mathtt{spawn}(f(\overline{v}, E, \overline{e}; \overline{s}))$$

We employ one single rule for E:

$$\frac{\langle \psi, S \triangleright e \rightarrow_i \psi', S' \triangleright e' \rangle^{1}}{\langle \psi, S \triangleright E[\![e]\!] \rightarrow_i \psi', S' \triangleright E[\![e']\!] \rangle^{1}} \quad (R\text{-}Sub)$$

The purpose of this chapter is to provide three extensions to FR_{FT}: (1) control flow, (2) call functions outside the `spawn` expression, and (3) reactive constructs using the notion of signals. These extensions are crucial for incorporating essential programming features such as code reusability, modularity, and structured design through the use of functions. Moreover, the inclusion of reactive constructs empowers MSSL with enhanced reactivity, enabling threads to communicate through signals. In the subsequent Section, we detail the semantics and typing of the `if/else` expression in MSSL.

1.4 Control Flow Extension in MSSL

Control flow constitutes a fundamental aspect of programming languages, enabling developers to create sophisticated algorithms and intricate programs by regulating the sequence of instructions. As a subset of Rust, MSSL facilitates the extension of control flow to enhance program functionality. Generally, the operational semantics of this control flow extension adhere to standard practices employed in various programming languages. However, the primary challenge lies in the typing aspect, necessitating the merging of typing environments at the junction points of the control flow graph. To achieve this, we leverage the Definitions of Type Join 4.11 and Environment Join 4.12 Definitions introduced in chapter 4.

1.4.1 Semantics of Control Flow in MSSL

The `if` expression consists of a block expression that is executed when the condition is true. Additionally, it includes a second branch known as `else`, which is executed when the condition evaluates to false, as depicted below:

$$\frac{}{\langle \psi, S \triangleright \texttt{if (true)}\ \{\overline{e}_1\}^{\texttt{n}}\ \texttt{else}\ \{\overline{e}_2\}^{\texttt{m}} \rightarrow_0 \psi, S \triangleright \{\overline{e}_3\}^{\texttt{n}} \rangle^{\texttt{l}}} \quad \textit{(R-IfTrue)}$$

$$\frac{}{\langle \psi, S \triangleright \texttt{if (false)}\ \{\overline{e}_1\}^{\texttt{n}}\ \texttt{else}\ \{\overline{e}_2\}^{\texttt{m}} \rightarrow_0 \psi, S \triangleright \{\overline{e}_3\}^{\texttt{m}} \rangle^{\texttt{l}}} \quad \textit{(R-IfFalse)}$$

Let's consider the following example to illustrate the control flow in MSSL:

$$\{\texttt{let mut}\ x = 0;\ \texttt{let mut}\ y = 1; \{\texttt{let mut}\ a = \&y;\ \texttt{if}\ (x == y)\ \{a = \&x\}^{\texttt{n}} \texttt{else} \{x = 1\}^{\texttt{r}}\}^{\texttt{m}}\}^{\texttt{l}} \tag{5.1}$$

This example written in MSSL using syntax 5.1 is satisfactory. However, after the execution of the `if/else` expression, one can be curious about the type of a, which can be $\&x$ if the `if` block is executed and $\&y$ otherwise. Nevertheless, during compile time, there is no information available about the branch that will be executed at runtime. Hence, to ensure all possibilities are covered, we need to consider the common case and combine these two types, resulting in a new type $\&x, y$. In the next Section, we will provide a detailed explanation of the approach employed by MSSL to achieve this outcome. Once again, let us take the same example and replace the immutable reference with a mutable reference as follows:

$$\{\texttt{let mut}\ x = 0;\ \texttt{let mut}\ y = 1; \{\texttt{let mut}\ a = \texttt{\&mut}\ y;\ \texttt{if}\ (x == y)\ \{a = \texttt{\&mut}\ x\}^{\texttt{n}} \texttt{else} \{\}^{\texttt{r}}\}^{\texttt{m}}\}^{\texttt{l}} \tag{5.2}$$

The following program is not valid in MSSL nor in Rust. As in Rust, MSSL successfully protects the invariant ownership. In other words, in this example, y is borrowed as mutable in the inner block. Therefore, when attempting to verify the equality in the `if` condition, an error occurs because the contents of y cannot be read as long as a exists.

1.4.2 Control Flow Typing in MSSL

As mentioned earlier, MSSL incorporates control flow extensions, including the `if/else` expression. Due to the approximate nature of the type system in determining which branch of the `if/else` expression will be executed, it becomes necessary to analyze both environments. To merge the environments of both the `if` branch and the `else` branch, we utilise the Environment Join 4.12 Definition, specifically the *union* function. Finally, we introduce the following *T-IF* rule to type the `if/else` expression:

$$\frac{\Gamma_1 \vdash \langle e : \texttt{bool} \rangle^{\texttt{l}}_{\sigma} \dashv \Gamma_2 \quad \Gamma_2 \vdash \langle \{e_1\}^{\texttt{n}} : \tau_1 \rangle^{\texttt{l}}_{\sigma} \dashv \Gamma_3 \quad \Gamma_2 \vdash \langle \{e_2\}^{\texttt{m}} : \tau_2 \rangle^{\texttt{l}}_{\sigma} \dashv \Gamma_4}{\Gamma_1 \vdash \langle \texttt{if}(e)\ \{e_1\}^{\texttt{n}}\ \texttt{else}\ \{e_2\}^{\texttt{m}} : \mathit{union}(\tau_1, \tau_2) \rangle^{\texttt{l}}_{\sigma} \dashv \Gamma_3 \sqcup \Gamma_4} \quad (\textit{T-IF})$$

This rule mandates that the condition expression must have a type of `bool` beforehand. Additionally, the type of the `if/else` expression is determined by the union of the types of each branch. Similarly, the output typing environment is created by joining the typing environments of each branch, utilizing the notation $\Gamma_3 \sqcup \Gamma_4$ (refer back to Definition 4.12 for clarity). To illustrate this process, let us refer back to Example 5.1, when we apply *T-IF* rule to type the `if/else` expression, we first type the `if` branch as follows: $\Gamma_3 = [x \mapsto \langle \texttt{int} \rangle^{\texttt{l}},\ y \mapsto \langle \texttt{int} \rangle^{\texttt{l}},\ a \mapsto \langle \& x \rangle^{\texttt{m}}]$. Next, we type the `else` branch as follows: $\Gamma_4 = [x \mapsto \langle \texttt{int} \rangle^{\texttt{l}},\ y \mapsto \langle \texttt{int} \rangle^{\texttt{l}},\ a \mapsto \langle \& y \rangle^{\texttt{m}}]$. Then we join the type of these two branches (i.e. blocks) using the *union* function. Finally, we join the environments of these branches, which also results in $\Gamma_3 \sqcup \Gamma_4 = [x \mapsto \langle \texttt{int} \rangle^{\texttt{l}},\ y \mapsto \langle \texttt{int} \rangle^{\texttt{l}},\ a \mapsto \langle \& x, y \rangle^{\texttt{m}}]$. Consequently, the resulting environment indicates that if we add other expressions after the `if/else` expression, x and y are considered as immutable borrowed by a.

Let us consider Example 5.3, which is rejected by the MSSL type system:

$$\begin{aligned}\{&\texttt{let mut } x = \texttt{trc}(0);\ \texttt{let mut } y = x.\mathit{clone};\ \texttt{let mut } z = \texttt{trc}(0);\\ &\texttt{if(cond)}\{y = \texttt{trc}(1)\}^{\texttt{m}}\ \texttt{else}\ \{y = z.\mathit{clone}\}^{\texttt{n}}\}^{\texttt{l}}\end{aligned} \tag{5.3}$$

Example 5.3 is rejected by the MSSL type system. According to the *T-IF* rule, when typing the `if/else` expression, we merge the types of the two branches. However, the *union* function (defined in 4.11 in Chapter 4), does not allow y to be both inactive and active at the same time. Now, let us examine Example 5.4:

$$\begin{aligned}\{&\texttt{let mut } x = \texttt{trc}(0);\ \texttt{let mut } y = x.\mathit{clone};\ \texttt{let mut } z = \texttt{trc}(1);\ \texttt{let mut } a = \texttt{trc}(2);\\ &\texttt{if(cond)}\{y = z.\mathit{clone}\}^{\texttt{m}}\ \texttt{else}\ \{y = a.\mathit{clone}\}^{\texttt{n}}\}^{\texttt{l}}\end{aligned} \tag{5.4}$$

Example 5.4 is accepted by the MSSL type system. According to the *T-IF* rule, when typing the `if/else` expression, we join the types of the two branches. In the first branch, the type of y is $\diamond z$ and in the second branch, it is $\diamond a$. Applying the *union* function, we have $\mathit{union}(\diamond z, \diamond a) = \diamond z, a$.

1.5 Function Extension in MSSL

Within this Section, we will elaborate on the semantics of the function call both inside and outside the `spawn` expression, providing a comprehensive explanation of its typing rules.

1.5.1 Declaration of Functions in MSSL

In Section 2.6 (Chapter 3), we have gone through MSSL, namely on how it handles the invocation function inside the `spawn` expression. The main idea is that the variables allocated to each thread in the program store S are associated with their respective lifetime. Furthermore, by considering the partial order of lifetimes, it is possible to determine the location of the variables associated to each thread. In this Section, we expand on the invocation function to include cases outside the `spawn` expression. Consequently, we need to determine the locations of the variables allocated for each function call. The distinction arises when invoking a function inside or outside the `spawn` expression. When a function is invoked outside the `spawn` expression, there exists a partial order relationship between the lifetime of the function's block and the enclosing block where the function is called. On the other hand, when a function is invoked inside the `spawn` expression, as previously explained, there is no relation between the lifetime of the function's block and the enclosing block where the `spawn` expression is called.

Moving forward, let us examine how MSSL handles the program store S in order to add the invocation function outside the `spawn` expression. Like FR, FR_{FT} does not support variable shadowing, where a variable can only be instantiated once in a program. MSSL achieves this extension by utilizing the concept of block lifetimes, allowing for the unique identification of variables declared within each block, where each block represents a stack frame. To provide further clarity, let us consider the following example:

$$\psi_{\emptyset}, S_{\emptyset} \triangleright \{\texttt{let mut } x = 0; \{\texttt{let mut } x = 1; \{\texttt{let mut } x = 2;\ \texttt{let mut } y = x\}^{\texttt{r}}\}^{\texttt{n}}\}^{\texttt{m}} \quad (5.5)$$

After executing Example 5.5 and progressing through several additional steps, the resulting state is as follows:

$$\psi_{\emptyset}, \{\ell_{\texttt{m}::x} \mapsto \langle 0\rangle^{\texttt{m}}, \ell_{\texttt{n}::x} \mapsto \langle 1\rangle^{\texttt{n}}, \ell_{\texttt{r}::x} \mapsto \langle 2\rangle^{\texttt{r}}\} \triangleright \{\{\{\texttt{let mut } y = x\}^{\texttt{r}}\}^{\texttt{n}}\}^{\texttt{m}} \quad (5.6)$$

Within the program store S, three locations are assigned to the variable x, each with a different lifetime. Consequently, when encountering the last expression that requires reading the content of x, the question arises: which x should be considered? To address this concern, MSSL makes use of the nesting of lifetimes and adopts a strategy of examining the shortest lifetime. The approach begins by searching for a variable x with the lifetime `r`. If the location of x is not found within the block of lifetime `r`, it looks for the lifetime that strictly encloses it ($\ell_{\texttt{n}::x}$) and so on. This approach is developed in Definition 5.1.

Thence, applying the aforementioned approach, the function body will be integrated in a block in MSSL as long as $\{f(x; s_1);\}^{\texttt{l}}$ is equivalent to $\{\{\overline{e}\}^{\texttt{m}}\}^{\texttt{l}}$ where $\{\overline{e}\}^{\texttt{m}}$ is the body of f and `l` is the lifetime of the enclosing block. However, this approach raises concerns when we deal with recursive cases. For instance $\{\{\{...\}^{\texttt{m}}\}^{\texttt{m}}\}^{\texttt{l}}$ is inconsistent and contradicts with the idea presented above. To mitigate the problem, the solution is simply to substitute the lifetimes with lifetimes during integration. In other words, we substitute `m` by a new lifetime `n` while respecting the partial order relation such as $\texttt{l} \succcurlyeq \texttt{n}$.

We revise the declaration context, $\mathcal{D}$, introduced in Chapter 3 as follows: $\mathcal{D}[f \mapsto_{\kappa} \lambda(\overline{x}; \overline{s})\{\overline{e}\}^{\texttt{m}}]$.

The function invocation reduction rule is depicted below using the *R-Invoke* rule:

$$\frac{\mathcal{D}(f) = \lambda(\overline{x};\overline{s})\{\overline{e}\}^{\mathtt{m}} \quad \Theta(\mathtt{l} \Rightarrow \{\overline{e}\}^{\mathtt{m}}) = \{\overline{e}\}^{\mathtt{n}} \quad \overline{\langle \ell_a^s \rangle^{\mathtt{r}} = read(S_1, s, \mathtt{l})} \quad S_2 = S_1[\overline{\ell_{\mathtt{n}::x} \mapsto \langle v \rangle^{\mathtt{n}}}, \overline{\ell_{\mathtt{n}::s} \mapsto \langle \ell_a^s \rangle^{\mathtt{n}}}] \quad \psi_2 = \psi_1[\overline{\ell_a \mapsto \langle v \rangle^{\mathtt{i+1}}}]}{\langle \psi_1, S_1 \rhd f(\overline{v};\overline{s}) \rightarrow_0 \psi_2, S_2 \rhd \{\overline{e}\}^{\mathtt{n}} \rangle^{\mathtt{l}}} \quad \textit{(R-Invoke)}$$

The *R-Invoke* rule retrieves the declaration of the function invoked from $\mathcal{D}$ using its unique name. At this point, we introduce the $(\Theta(\mathtt{l} \Rightarrow e) = \{e\}^{\mathtt{n}})$ function which is responsible for *instantiating* the lifetime of an expression e. It simultaneously *instantiates* all lifetimes of e to fresh lifetimes included in $\mathtt{l}$ (e.g. $\mathtt{l} \succcurlyeq \mathtt{n}$). The following is a simple example demonstrating the declaration of the function in MSSL using the syntax 5.1:

$$\begin{aligned} &\mathtt{fn}\ foo(\mathtt{mut}\ x : \&'a\ \mathtt{mut}\ \mathtt{int}, \mathtt{mut}\ y : \blacklozenge\mathtt{int}) \rightarrow_0 \blacklozenge\mathtt{int}\{{}^*x = 1; \{\mathtt{let}\ \mathtt{mut}\ a = y.clone\}^{\mathtt{m}};\ y\}^{\mathtt{n}} \\ &\qquad \{\mathtt{let}\ \mathtt{mut}\ x = 0;\ \mathtt{let}\ \mathtt{mut}\ y = \mathtt{trc}(0);\ y = foo(\&\mathtt{mut}\ x, y)\}^{\mathtt{l}} \end{aligned} \tag{5.7}$$

Example 5.7 is accepted by the MSSL type system. The $(\rightarrow_0)$ informs the type system that the function foo does not contain reactive constructs such as `cooperate` or `when` (see Section 1.5.2). Furthermore, when the function foo is invoked, it is represented by $\{\ \{{}^*y = {}^*x; \{\mathtt{let}\ \mathtt{mut}\ a = y.clone\}^{\mathtt{k}}\ y\}^{\mathtt{r}}\ \}^{\mathtt{l}}$ where `r` is the lifetime instantiated using the Θ function where $\mathtt{l} \succcurlyeq \mathtt{r}$. Additionally, for each block in the body of foo, a lifetime is instantiated within `r` (i.e. $\mathtt{r} \succcurlyeq \mathtt{k}$).

1.5.2 Typing Functions in MSSL

In this Section, we explore how the type system types the function calls outside the `spawn` expression. To achieve this, we rely on a *type system with effects* [87]. The significance of introducing the notion of *effect* is to safeguard shared data between threads at cooperation time. To elaborate, let us consider the same example depicted in Figure 4.1 with some modifications, as illustrated in Figure 5.2. We suppose that one thread executes the `createVec` function, and another thread executes the `modifyVec` function. Obviously, the `createVec` function calls a second function `createTh`. Within the body of `createTh`, we create a new thread (which will execute the `modifyVec` function), and then the current thread explicitly cooperates using the cooperate expression. A similar error, as seen in Figure 4.1, occurs. However, in this example, the error arises implicitly through the `createTh` function due to the κ effect.

We introduce a slight modification to our typing judgments before jumping into function typing in more detail. In MSSL, signals are distinct from standard variables; they are special variables that cannot be treated as expressions. Additionally, their access is restricted to their creation and emission. Furthermore, we extend the typing rules with a declaration context of signals, denoted as $\mathscr{L}$, such that all rules have the following form: $\mathscr{L}_1, \Gamma_1 \vdash \langle e : \tau \rangle_\sigma^{\mathtt{l}} \dashv \mathscr{L}_2, \Gamma_2$. The set of signals $\mathscr{L}$ contains the signals declared during execution. Moreover, we assume that all existing MSSL typing rules are extended to adhere to this format in a straightforward manner. For example:

$$\frac{\Gamma \vdash \omega : \langle \tau \rangle^{\mathtt{m}} \quad copy(\tau) \quad \neg readProhibited(\Gamma, \omega)}{\mathscr{L}, \Gamma \vdash \langle \hat{\omega} : \tau \rangle_\sigma^{\mathtt{l}} \dashv \mathscr{L}, \Gamma} \quad \textit{(T-Copy)}$$

```
fn createVec()→κ ϵ{
        let mut x=trc(vec![1,2]); Γ = [x ↦ ⟨◆vec⟨int⟩⟩^1]
        let mut a=&*x[1]; Γ = [x ↦ ⟨◆vec⟨int⟩⟩^1, a ↦ ⟨&*x[1]⟩^1]
        createTh(x.clone);Γ = [x ↦ ⟨◆vec⟨int⟩⟩^1, a ↦ ⟨&*x[1]⟩^1]
        print!(*a); }^1

fn createTh(mut x:trc<vec<int> >)→κ ϵ{
            // Thread
          spawn(modifyVec(x.clone)); Γ = [...]
          cooperate;  Γ = [...]
                                  }^m

fn modifyVec(mut y:trc<vec<int> >)→κ ϵ{  *y=vec![0]; }^n
```

Figure 5.2: MSSL's Type and Effect System.

1.5.3 Effects of Functions in MSSL

We associate each well-typed function signature with an effect κ, which is defined as a predicate taking values 0 or 1. Specifically, when κ is 0, it indicates that the body of the function does not contain cooperative expressions like `cooperate` or `when` expressions. Conversely, when κ is 1, it denotes that the function is *cooperative*, allowing threads to cooperate when necessary to safeguard memory safety. Thus, we can deduce that if a function includes `cooperate` or `when` expressions at least once in its body, its effect κ is inevitably 1. To accommodate this aspect, we extend the effect as a flow of information in the typing rules, explicitly incorporating it into the judgment as follows: $\mathscr{L}_1, \Gamma_1 \vdash_\kappa \langle e : \tau \rangle^1_\sigma \dashv \mathscr{L}_2, \Gamma_2$. For instance, the example 5.8 is rejected by the MSSL type system, as shown below:

$$\begin{gathered}\texttt{fn } f_1(\texttt{mut } x : \blacklozenge\texttt{int}) \to_0 \epsilon_0\{{}^*x = 1;\ \texttt{cooperate}\}^m \\ \texttt{fn } f_2(\texttt{mut } x : \blacklozenge\texttt{int}) \to_0 \epsilon_0\{\ldots\}^n \\ \{\texttt{let mut } x = \texttt{trc}(0);\ \texttt{spawn}(f_1(x.clone));\ \texttt{spawn}(f_2(x.clone))\}^1\end{gathered} \tag{5.8}$$

The MSSL type system rejects the aforementioned example immediately at the typing of the body of the function f_1. Specifically, the rejection occurs when typing the `cooperate` expression, where κ is expected to be 1. However, the effect κ defined in the signature of this function is 0, indicating that the block of f_1 does not contain cooperative constructs. Moreover, in Example 5.2, we need an effect κ for both the `createVec` and `createTh` functions to be 1. When the function is invoked in `createVec`, our type system is aware that the invoked function contains at least one cooperative construct due to its effect κ. Consequently, this example will be rejected since *x* (shared data) is borrowed by *a*.

1.5.4 Suitable Types for Functions in MSSL

There exists a crucial link between signatures and types in MSSL. For instance, as depicted in Figure 5.1, the signature $\diamond\texttt{int}$ is not considered a type in MSSL. Therefore, we require a mechanism to establish a correspondence between the signature and the type when a function is declared or invoked [73]. To address this, we use the following mechanism: $\Gamma_1 \vdash (\overline{\texttt{S}} \rightarrow \texttt{S}) \Longrightarrow (\overline{\tau} \rightarrow \tau)$. The underlying concept behind this mechanism is to map the signature S of the declared function to the typing environment. For example, if the signature is $\diamond$S, then according to the MSSL syntax, the corresponding type must have the form $\diamond\omega$. Hence, this matching is achieved through the above mechanism in the following manner: $\Gamma \vdash (\diamond\texttt{int}) \Longrightarrow (\diamond\gamma)$, where $\Gamma = \{\gamma \mapsto \langle\blacklozenge\texttt{int}\rangle^{\texttt{l}}\}$. Here, γ serves as an anonymous or *fresh* variable introduced by the mechanism into the typing environment Γ as a *suitable parameter*. Similarly, the same principle applies to reference signatures, for example: $\Gamma \vdash (\&'\texttt{l int}) \Longrightarrow (\&\gamma)$, where $\Gamma = \{\gamma \mapsto \langle\texttt{int}\rangle^{\texttt{l}}\}$. To formalize this interaction between signatures and types in MSSL, we introduce the following Definition:

Definition 5.4 (*Suitable Types*) *Given an environment* Γ *and a signature* S*, we denote* $\Gamma \vdash \texttt{S} \Rightarrow \tau$*, which signifies the process of reducing the signature* S *of declarations into a suitable type* τ*, potentially leading to an updated typing environment, based on the following rules:*

$$\frac{}{\Gamma \vdash \texttt{int} \Rightarrow \texttt{int}}\ \textit{Su-Int} \qquad \frac{}{\Gamma \vdash \texttt{bool} \Rightarrow \texttt{bool}}\ \textit{Su-Bool}$$

$$\frac{\Gamma \vdash \texttt{S} \Rightarrow \tau}{\Gamma \vdash \blacksquare\texttt{S} \Rightarrow \blacksquare\tau}\ \textit{Su-Box} \qquad \frac{\Gamma \vdash \texttt{S} \Rightarrow \tau}{\Gamma \vdash \blacklozenge\texttt{S} \Rightarrow \blacklozenge\tau}\ \textit{Su-AcTrc}$$

$$\frac{\gamma \in \textit{fresh} \quad \Gamma \vdash \texttt{S} \Rightarrow \tau}{\Gamma \sqcup \{\gamma \mapsto \blacklozenge\tau\} \vdash \diamond\texttt{S} \Rightarrow \diamond\gamma}\ \textit{Su-IncTrc}$$

$$\frac{\gamma \in \textit{fresh} \quad \Gamma \vdash \texttt{S} \Rightarrow \tau}{\Gamma \sqcup \{\gamma \mapsto \langle\tau\rangle^{\texttt{a}}\} \vdash \&'\texttt{a}[\texttt{mut}]\texttt{S} \Rightarrow \&[\texttt{mut}]\gamma}\ \textit{Su-Borrow}$$

The signatures of an inactive Trc and a reference require the establishment of anonymous variables, denoted as γ, into the typing environment to obtain a *suitable type*. Upon considering this Definition, it becomes apparent that the typing environment, Γ, supports both concrete and abstract lifetimes. Abstract lifetimes correspond to lifetime variables (`'a`, `'b`, etc.), whereas concrete lifetimes carry concrete information into the typing environment, representing *well-defined* regions in the context of the new version of Rust (denoted as a set of *loans*). For example, in the *Su-Borrow* rule, the resulting typing environment contains a new fresh variable γ with an abstract lifetime ($\Gamma(\gamma) = \langle\tau\rangle^{\texttt{a}}$). Hence, a *well-defined* relationship exists between lifetimes: (1) lifetimes, whether abstract or concrete, outlives itself (i.e. reflexivity). Therefore, (2) one abstract lifetime outlives another if there is a corresponding lifetime relation (i.e. transitivity). Moreover, (3) the abstract lifetime always outlives a concrete lifetime. Finally, (4) a concrete lifetime `l` survives `m` if the former occurs before the latter (e.g. the lifetime of the outer block outlives all the lifetimes of the inner blocks).

When typing a MSSL program, the type system considers and enforces several conditions, starting with the function declaration. The requirements for the function declaration ($\texttt{fn}\ f(\overline{\texttt{mut}\ x : \texttt{S}}; \overline{s}) \rightarrow_{\kappa} \texttt{S}\{\overline{e}\}^{\texttt{m}}$) are as follows:

- We start with an empty typing environment and according to Definition 5.4, we apply the premise $\overline{\Gamma \vdash \texttt{S} \Rightarrow \tau}$. This premise is responsible for reducing signatures in typing environments, resulting in Γ. Here, Γ may potentially contain type variables, e.g. γ. Since MSSL prohibits aliasing, this condition is ensured via $\overline{\Gamma \vdash \texttt{S} \Rightarrow \tau}$, where each anonymous location included in a signature corresponds to a unique anonymous location in the typing environment. This uniqueness is necessary to guarantee the soundness of the MSSL type system.
- Before typing the function's body, it is necessary to declare the list of arguments in Γ (i.e. $\Gamma_1 = \Gamma[\overline{x \mapsto \tau}]$) and the list of signals in $\mathscr{L}$ (i.e. $\mathscr{L}_1 = \mathscr{L} \sqcup \{\overline{s}\}$, note that $\mathscr{L}$ is initially empty).
- To guarantee a well-typed function, the type system compares whether the type of the body is compatible with the type obtained by $\Gamma_1 \vdash (\overline{\texttt{S}} \rightarrow \texttt{S}) \Longrightarrow (\overline{\tau} \rightarrow \tau)$. In the case of references and based on the subtyping relation (Figure 4.5), we need to compare not only the types of the references but also their assigned lifetime.

To illustrate the typing of an MSSL program, let us consider Example 5.9:

$$\begin{gathered}\texttt{fn}\ f_1(\texttt{mut}\ x : \&'\texttt{a}\ \texttt{mut}\&'\texttt{b}\ \texttt{int},\ \texttt{mut}\ y : \diamond\texttt{int}) \rightarrow_0 \epsilon_0\{...\}^{\texttt{l}} \\ \{\texttt{let mut}\ x = \texttt{trc}(0);\ \texttt{let mut}\ a = 1;\ \{\texttt{let mut}\ b = \&a;\ f_1(\&\texttt{mut}\ b, x.clone)\}^{\texttt{n}}\}^{\texttt{m}}\end{gathered} \tag{5.9}$$

We start by typing the declaration function f_1. To achieve this, we apply the *T-Function* rule, which utilizes the mechanism: $\Gamma \vdash (\overline{\texttt{S}} \rightarrow \texttt{S}) \Longrightarrow (\overline{\tau} \rightarrow \tau)$, to create an appropriate type. Thence, after mapping signatures to types using Definition 5.4, the resulting typing environment Γ of f_1 is as follows: $\Gamma = [\gamma_1 \mapsto \langle\texttt{int}\rangle^{\texttt{b}}, \gamma_2 \mapsto \langle\&\gamma_1\rangle^{\texttt{a}}$, $\gamma_3 \mapsto \langle\blacklozenge\texttt{int}\rangle^{\texttt{l}}]$ corresponds to the result of the mechanism: $\Gamma_1 \vdash (\&'\texttt{a}\ \texttt{mut}\&'\texttt{b}\ \texttt{int}, \diamond\texttt{int} \rightarrow \epsilon_0) \Longrightarrow (\&\texttt{mut}\gamma_2, \diamond\gamma_3 \rightarrow \epsilon)$. With this information, we can safely proceed to type the body of the function and continue accordingly.

1.5.5 Invocation of Functions in MSSL

In MSSL, as depicted in Figure 5.1, function calls can occur in two distinct cases: outside the `spawn` expression, which is the standard execution of a function as seen in other languages, and inside the `spawn` expression. Each case has different typing requirements. In this Section, we present how the MSSL type system handles the typing of function invocations outside `spawn` while ensuring strong typing.

Based on the approach outlined in Chapter 4, we adopt a mechanism for passing types from typing environments to function signatures. Since functions can now return values, we have slightly modified this mechanism to accommodate this feature. Furthermore, given the two different ways of reducing a function call in MSSL (inside and outside of `spawn`)

with different typing requirements, the type system needs to differentiate between the two cases. To achieve this, we introduce the following notation: $\Gamma_1 \vdash (\overline{\mathtt{S}} \to \mathtt{S}) \Longleftarrow_{\mathtt{b}} (\overline{\tau} \to \tau) \dashv \Gamma_2$. Here, the index `b` is a `boolean`, which can be either `tt` to indicate that the function call is inside the `spawn` expression or `ff` otherwise. Moreover, in this mechanism, we can spot the difference between Γ_1 and Γ_2. Actually, this difference comes from the possible effects that the invocation of the function can produce during the execution. In essence, the typing of function invocations involves both the typing of the arguments and the mechanism that applies constraint resolution to enable the flow of information, such as the subtyping relationship. We will now elaborate on the necessary semantic and typing rules for function calls outside the `spawn` expression and discuss the relevant modifications that impact function calls inside `spawn`.

Function Invocation Outside the `Spawn` Expression in MSSL

We introduce the typing rule for function invocations outside the `spawn` expression, taking into consideration the presence of lifetime parameters in signatures, particularly for references, and the suitable signature for the inactive `Trc` type. This process is governed by five constraints, as indicated by $\Gamma_1 \vdash (\overline{\mathtt{S}} \to \mathtt{S}) \Longleftarrow_{\mathtt{ff}} (\overline{\tau} \to \tau) \dashv \Gamma_2$, which ensure that the function call is safely typed while preserving the properties of the involved types. Let us examine these constraints in detail:

1. Moving arguments that contain an active `Trc` within their type is allowed, unlike in the case of a function call inside the `spawn` expression. However, it is essential to consider the behavior of the function body. For instance, Example 5.10 illustrates a scenario where the properties of the `Trc` type are not preserved after executing function f_2. Hence, it is crucial to avoid having both an inactive and an active `Trc` in arguments that point to the same memory location.

2. Similar reasoning applies to inactive `Trc`'s. In the same example 5.10, replacing $f_2(x, y)$ with $f_2(x.clone, y)$ results in two inactive `Trc`'s becoming active in the new typing environment. Thus, it is necessary to prevent the existence of two inactive `Trc`'s pointing to the same memory location.

3. Function arguments can be references, so it is essential to ensure that there is at least one suitable binding signature for the arguments. For instance, Example 5.11 is rejected because the variable x in f_2 has two different lifetimes, while f_1 requires them to have the same lifetime (i.e. the lifetime may not be long enough).

4. Functions inside the `spawn` expression do not have a return type, but in the case of a function call outside the `spawn` expression, there might be a return type. This makes typing the function call more challenging, as it requires additional constraints to allow bidirectional flow of information between signatures and types. For example, consider Example 5.12 where the function f_1 returns an inactive `Trc` type. Our mechanism rejects this example because there must be at least one parameter in the function with the same signature as the return type. However, Example 5.13 is accepted by the MSSL type system because the return type can be at least the type of y,

meaning the return type is a subtype of y. Thus, we can safely assume that the type of z is $\diamond y$, indicating that y is cloned into the typing environment after executing f_1. For reference types, Example 5.17 demonstrates a scenario where function f_1 returns a reference type with lifetime `'b`. Looking at the parameter types, the return type is a subtype of x. Similarly, in Example 5.16, the return type is the union of types x and y, with the minimum lifetime between them, as the return type is a subtype of both x and y. This concept of minimum lifetime applies to both references and inactive `Trc` types. Example 5.15 illustrates this concept.

5. The presence of a mutable reference in function parameters may lead to side effects. We ensure that all possible side effects that may occur in the environment due to the invocation are taken into account in the difference between Γ_1 and Γ_2. Example 5.18 illustrates how a function invocation can lead to side effects in the typing environment.

1.5.6 Examples of Functions in MSSL

For better understanding of the constraints imposed by the following mechanism : $\Gamma_1 \vdash (\overline{\mathrm{S}} \to \mathrm{S}) \Longleftarrow_{\mathtt{ff}} (\overline{\tau} \to \tau) \dashv \Gamma_2$, we consider some examples written in MSSL according to Figure 5.1. These examples outline the five aforementioned constraints:

The uniqueness of `Trc`. A notable observation about the mechanism is its ability to reinforce the uniqueness of `Trc` by preventing aliasing when calling the function.

$$\begin{gathered}\texttt{fn}\ f_1(\texttt{mut}\ x : \blacklozenge\texttt{int}) \to_0 \epsilon_0\{\texttt{let mut}\ y = x.clone; f_2(x, y)\} \\ \texttt{fn}\ f_2(\texttt{mut}\ x : \blacklozenge\texttt{int}, \texttt{mut}\ y : \diamond\texttt{int}) \to_0 \epsilon_0\{\texttt{spawn}(f_3(x.clone, y))\} \\ \texttt{fn}\ f_3(\texttt{mut}\ x : \blacklozenge\texttt{int}, \texttt{mut}\ y : \blacklozenge\texttt{int}) \to_0 \epsilon_0\{\ \texttt{//additional code that uses x and y}\ \}\end{gathered} \tag{5.10}$$

In Example 5.10, the function f_2 is invoked within the body of the function f_1. When the function is called, the MSSL type system cannot predict the effect of f_2 after its execution, and it is not allowed to have two `Trc`'s (inactive or active) pointing to the same location. In this case, the result of our mechanism is as follows: $\Gamma_1 \vdash (\blacklozenge\texttt{int}, \diamond\texttt{int} \to \epsilon_0) \Longleftarrow\!\!\!\!\!\!/_{\mathtt{ff}}$ $(\blacklozenge\texttt{int}, \diamond\ x \to \epsilon) \dashv \Gamma_1$. The reason for rejecting this example is: after typing the arguments, x will be moved (i.e. it has a partial type) and consequently our mechanism detects that $\diamond\ x$ is *ill-typed*.

Incompatible binding. A major challenge for the observer, concerning the mechanism, is to find a binding that is compatible with the lifetimes, as follows:

$$\begin{gathered}\texttt{fn}\ f_1(\texttt{mut}\ x : \&'\texttt{c mut}\ \&'\texttt{c int}) \to_0 \epsilon_0\{\ldots\}^{\texttt{l}} \\ \texttt{fn}\ f_2(\texttt{mut}\ x : \&'\texttt{a mut}\ \&'\texttt{b int}) \to_0 \epsilon_0\{f_1(x)\}^{\texttt{m}}\end{gathered} \tag{5.11}$$

The MSSL type system rejects this example due to the call of f_1 inside f_2. The function f_1 requires that the lifetime of x to be the same, but in f_2 the lifetime of x is different. However, the challenge is that the type system does not know whether `'a` is longer than `'b` (i.e.

it must be `'a: 'b`). As a result, there is no appropriate lifetime to use. Hence, we obtain a constraint of type as follows: $\Gamma_1 \vdash (\&'\texttt{c mut } \&'\texttt{c int} \rightarrow \epsilon_0) \Longleftarrow\!\!\!\!=_{\texttt{ff}} (\&\texttt{mut}\gamma_2 \rightarrow \epsilon) \dashv \Gamma_1$ where γ_2 is an anonymous variable created when the declaration function is typed.

Protective returns. Another important observation about the mechanism is its role in determining a *well-typed* return. Consider the following example to illustrate its significance:

$$\begin{array}{c}\texttt{fn } f_1(\texttt{mut } x : \blacklozenge\texttt{int}) \rightarrow_0 \diamond\texttt{int}\{...\}^{\texttt{l}} \\ \{\texttt{let mut } x = \texttt{trc}(0);\ \texttt{let mut } z = f_1(x)\}^{\texttt{m}}\end{array} \tag{5.12}$$

Example 5.12 is also rejected since, if the return type is an inactive `Trc`, we must have at least one inactive `Trc` in the signature parameters, which is not the case. Thus, we obtain the following constraints: $\Gamma_1 \vdash (\blacklozenge\texttt{int} \rightarrow \diamond\texttt{int}) \Longleftarrow\!\!\!\!=_{\texttt{ff}} (\blacklozenge\texttt{int} \rightarrow \texttt{?}) \dashv \Gamma_1$.

Protective returns. Let us consider another observation regarding the mechanism for finding a *well-typed* return. The following example illustrates this point:

$$\begin{array}{c}\texttt{fn } f_1(\texttt{mut } x : \blacklozenge\texttt{int}, \texttt{mut } y : \diamond\texttt{int}) \rightarrow_0 \diamond\texttt{int}\{...\}^{\texttt{l}} \\ \{\texttt{let mut } x = \texttt{trc}(0); \texttt{let mut } y = \texttt{trc}(0);\ \texttt{let mut } z = f_1(x, y.clone)\}^{\texttt{m}}\end{array} \tag{5.13}$$

In contrast to Example 5.12, Example 5.13 is accepted by the MSSL type system. The result of the mechanism is the following: $\Gamma_1 \vdash (\blacklozenge\texttt{int}, \diamond\texttt{int} \rightarrow \diamond\texttt{int}) \Longleftarrow_{\texttt{ff}} (\blacklozenge\texttt{int}, \diamond y \rightarrow \diamond y) \dashv \Gamma_1$.

Protective returns. A significant challenge in typing function invocations is effectively managing side effects, particularly to ensure a *well-typed* return type. The example highlights the following considerations:

$$\begin{array}{c}\texttt{fn } f_1(\texttt{mut } x : \&'\texttt{c mut } \blacklozenge\texttt{ int}) \rightarrow_0 \diamond\texttt{int}\{...\}^{\texttt{l}} \\ \{\texttt{let mut } x = \texttt{trc}(0);\ \texttt{let mut } y = f_1(\&\texttt{mut } x)\}^{\texttt{m}}\end{array} \tag{5.14}$$

Similar to Example 5.12, the MSSL type system rejects Example 5.14. The current challenge, regardless of the actual body given for the function f_1, is that we can assume that f_1 returns "$^*x$.`clone`", which would provide a copy of the reference of x with an abstract lifetime $'\texttt{c}$. Despite the Definition 4.5, our type system identifies that the lifetime of x is not within the return type's lifetime. Consequently, this leads to the following unreasonable constraint: $\Gamma_1 \vdash (\&'\texttt{c mut } \blacklozenge\texttt{ int} \rightarrow \diamond\texttt{int}) \Longleftarrow\!\!\!\!=_{\texttt{ff}} (\&\texttt{mut } x \rightarrow \diamond x) \dashv \Gamma_1$. Note that we have the same case even if the parameter signature is an immutable reference.

Let us examine another example to illustrate the process of computing a *well-typed* return type:

$$\begin{array}{c}\texttt{fn } f_1(\texttt{mut } x : \diamond\texttt{int}, \texttt{mut } y : \diamond\texttt{int}) \rightarrow_0 \diamond\texttt{int}\{...\}^{\texttt{l}} \\ \{\texttt{let mut } x = \texttt{trc}(0);\ \{\texttt{let mut } y = \texttt{trc}(0); \texttt{let mut } z = f_1(x.clone, y.clone)\}^{\texttt{n}}\}^{\texttt{m}}\end{array} \tag{5.15}$$

In Example 5.15, the type of z is determined by taking the union of the types $\diamond x$ and $\diamond y$ (i.e. $\diamond x, y$).

Protective lifetimes. Another noteworthy observation about our mechanism pertains to the selection of not only a *well-typed* return type but also an appropriate lifetime, as follows:

$$\begin{array}{c}\texttt{fn}\ f_1(\texttt{mut}\ x : \&'\texttt{a}\ \texttt{int}, \texttt{mut}\ y : \&'\texttt{a}\ \texttt{int}) \rightarrow_0 \&'\texttt{a}\ \texttt{int}\{...\}^{\texttt{l}} \\ \{\texttt{let mut}\ x = 0; \{\texttt{let mut}\ y = 0; \texttt{let mut}\ a = f_1(\&x, \&y)\}^{\texttt{n}}\}^{\texttt{m}}\end{array} \tag{5.16}$$

In Example 5.16, the function f_1 requires that the parameters x and y have the same lifetime (`'a`). However, during the typing of f_1 in the inner block, x and y have a different lifetime (i.e. concrete lifetime). As previously elaborated, the abstract lifetime outlives the concrete lifetime and therefore we obtain a satisfying constraint as follows: $\Gamma_1 \vdash (\&'\texttt{a}\ \texttt{int}, \&'\texttt{a}\ \texttt{int} \rightarrow \&'\texttt{a}\ \texttt{int}) \Longleftarrow_{\texttt{ff}} (\&x, \&y \rightarrow \&x, y) \dashv \Gamma_1$ where, lifetime `'a` is related to the minimum lifetime, which is 'n' here.

Protective lifetimes. Another notable observation regarding our mechanism is the selection of an appropriate lifetime. Here's a detailed explanation:

$$\begin{array}{c}\texttt{fn}\ f_1(\texttt{mut}\ x : \&'\texttt{b}\ \texttt{int}, \texttt{mut}\ y : \&'\texttt{a}\ \texttt{int}) \rightarrow_0 \&'\texttt{b}\ \texttt{int}\{...\}^{\texttt{l}} \\ \{\texttt{let mut}\ x = 0; \{\texttt{let mut}\ y = 0; \texttt{let mut}\ a = f_1(\&x, \&y)\}^{\texttt{n}}\}^{\texttt{m}}\end{array} \tag{5.17}$$

In Example 5.17, we have changed the lifetimes and therefore the function f_1 requires the parameters to have different lifetimes. The chosen lifetime is explicitly `'b` which is related to the lifetime of x.

Side effects. The final observation regarding the mechanism revolves around identifying the potential side effects that a function call can generate. Here is an example to illustrate this:

$$\begin{array}{c}\texttt{fn}\ f_1(\texttt{mut}\ x : \&'\texttt{c}\ \texttt{mut}\ \blacklozenge\ \texttt{int}, \texttt{mut}\ y : \&'\texttt{c}\ \texttt{mut}\ \diamond\ \texttt{int}) \rightarrow_0 \epsilon_0\{{}^*y = {}^*x.clone;\}^{\texttt{l}} \\ \{\texttt{let mut}\ x = \texttt{trc}(0);\ \texttt{let mut}\ y = \texttt{trc}(0);\ \texttt{let mut}\ z = y.clone;\ f_1(\&\texttt{mut}\ x, \&\texttt{mut}\ z);\}^{\texttt{m}}\end{array} \tag{5.18}$$

The challenge here is that the type of z must be carefully updated from $\diamond y$ to $\diamond x, y$ (before and after the invocation respectively). This occurs independently of the actual body given for f_1, since we must creatively assume that ${}^*y = {}^*x.clone$ may occur (even if it does not). This *side effect* is captured by the typing environment where Γ before the invocation has the form: $\Gamma = [x \mapsto \langle\blacklozenge\texttt{int}\rangle^{\texttt{m}}, y \mapsto \langle\blacklozenge\texttt{int}\rangle^{\texttt{m}}, z \mapsto \langle\diamond y\rangle^{\texttt{m}}]$ and after the invocation, Γ becomes as follows: $\Gamma = [x \mapsto \langle\blacklozenge\texttt{int}\rangle^{\texttt{m}}, y \mapsto \langle\blacklozenge\texttt{int}\rangle^{\texttt{m}}, z \mapsto \langle\diamond x, y\rangle^{\texttt{m}}]$. Finally, we obtain a satisfactory constraint: $\Gamma_1 \vdash (\&'\texttt{c}\ \texttt{mut}\ \blacklozenge\ \texttt{int}, \&'\texttt{c}\ \texttt{mut}\ \diamond\ \texttt{int} \rightarrow \epsilon_0) \Longleftarrow_{\texttt{ff}} (\&\texttt{mut}\ x, \&\texttt{mut}\ z \rightarrow \epsilon) \dashv \Gamma_1[z \mapsto \langle\diamond x, y\rangle^{\texttt{m}}]$.

After elucidating the constraints that must be satisfied when typing a function, we now present the following typing rules for handling the function call. To address the invoked function, we introduce two specific typing rules, outlined below:

$$\frac{\begin{array}{c}\mathcal{D}(f) = (\overline{\texttt{S}}; \overline{s}) \rightarrow_0 (\texttt{S}) \quad \mathscr{L}_1, \Gamma_1 \vdash \langle\overline{e : \tau}\rangle^{\texttt{l}}_{\sigma} \dashv \mathscr{L}_2, \Gamma_2 \\ \overline{s} \in \mathscr{L}_1 \quad \Gamma_2 \vdash (\overline{\texttt{S}} \rightarrow \texttt{S}) \Longleftarrow_{\texttt{ff}} (\overline{\tau} \rightarrow \tau) \dashv \Gamma_3\end{array}}{\mathscr{L}_1, \Gamma_1 \vdash_0 \langle\ f(\overline{e}; \overline{s}) : \tau\rangle^{\texttt{l}}_{\sigma} \dashv \mathscr{L}_2, \Gamma_3} \tag{\textit{T-InvokeA}}$$

$$\frac{\begin{array}{c}\mathcal{D}(f) = (\overline{\texttt{S}}; \overline{s}) \rightarrow_1 (\texttt{S}) \quad \textit{safeTrc}(\Gamma_1) \quad \overline{s} \in \mathscr{L}_1 \\ \mathscr{L}_1, \Gamma_1 \vdash \langle\overline{e : \tau}\rangle^{\texttt{l}}_{\sigma} \dashv \mathscr{L}_2, \Gamma_2 \quad \Gamma_2 \vdash (\overline{\texttt{S}} \rightarrow \texttt{S}) \Longleftarrow_{\texttt{ff}} (\overline{\tau} \rightarrow \tau) \dashv \Gamma_3\end{array}}{\mathscr{L}_1, \Gamma_1 \vdash_1 \langle\ f(\overline{e}; \overline{s}) : \tau\rangle^{\texttt{l}}_{\sigma} \dashv \mathscr{L}_2, \Gamma_3} \tag{\textit{T-InvokeB}}$$

One notable distinction between these rules pertains to the effect κ. As previously mentioned, when κ is 0, the body of the function does not contain cooperative constructs. This is evident in the *T-InvokeA* rule, where no constraints are imposed on the shared data between threads, allowing the current thread to safely execute the function. Conversely, when κ is 1, the invoked function is considered *cooperative*, indicating that its body contains at least one cooperative construct. In this case, it is not evident that the current thread immediately executes the function body, which could potentially lead to memory corruption by another thread. To address this, we handle this case as `cooperative` expressions by verifying if any shared data is being borrowed through the *safeTrc* function (Definition 4.5). Additionally, in this case, the effect κ of the current function (i.e. the enclosing block) must be 1.

Subsequently, in both rules, we verify the existence of all the required signals ($\overline{s} \in \mathscr{L}_1$) and type the expressions e provided as arguments via the Argument Typing function 4.16. Afterward, we need to perform the following step in our mechanism: $\Gamma_2 \vdash (\overline{\mathtt{S}} \to \mathtt{S}) \Longleftarrow_{\mathtt{ff}} (\overline{\tau} \to \tau) \dashv \Gamma_3$. This step must satisfy the aforementioned constraints. We represent the most crucial aspects in the following manner: (1) for all i, we have $\overline{\Gamma_2 \vdash S_i \sim \tau_i}$ (Expanded Signature and Type Compatibility Definition 5.6), (2) for all $\tau_1, \tau_2 \in \overline{\tau}$ if $\mathit{containsType}(\Gamma, \tau_1, \diamond\overline{\omega})$ and $\mathit{containsType}(\Gamma, \tau_2, \diamond\overline{u})$ then $\neg\exists_{i,j}.(\omega_i \bowtie u_j)$. Finally, (3) for all $\tau \in \overline{\tau}$ if $\mathit{containsType}(\Gamma, \tau_1, \diamond\overline{\omega})$ then $\overline{\Gamma \vdash \omega : \langle\tau\rangle^1}$ (This requirement is enforced by the Argument Typing function 4.16).

Based on the Definition of the *contains* function 4.2 defined in Chapter 4, we introduce the following:

Definition 5.5 (ContainsType) *Let Γ be an environment and let τ, τ' be types. Then, the $\mathit{containsType}(\Gamma, \tau, \tau')$ function is responsible for recursively verifying whether τ contains τ' and is defined as follows:*

$$\mathit{containsType}(\Gamma, \tau, \tau') = \begin{cases} \mathtt{true} & \mathit{if}\ \tau = \tau' \\ \mathit{containsType}(\Gamma, \tau_1, \tau') & \mathit{if}\ \tau = \blacksquare\tau_1\ \mathit{or}\ \tau = \blacklozenge\tau_1\ \mathit{or} \\ & (\tau = \diamond\overline{\omega}\ \mathit{or}\ \tau = \&[\mathtt{mut}]\overline{\omega}\ \ \mathit{s.t.} \\ & \forall_i(\Gamma(\omega_i) = \tau_1)) \\ \mathtt{false} & \mathit{otherwise} \end{cases}$$

To ensure that a given invocation function, whether it is under a `spawn` expression or not, is *well-typed*, it is necessary to verify if the types of its arguments are in accordance with the signatures of its parameters. Therefore, we have introduced new rules to the Definition 4.15 as follows:

Definition 5.6 (Expanded Signature and Type Compatibility) *For an environment Γ, a signature $\mathtt{S}$ and a type τ are said to be compatible, denoted as $\Gamma \vdash S \sim \tau$ according to the*

following rules:

$$\frac{}{\Gamma \vdash \texttt{int} \sim \texttt{int}}\ \textit{C-Int} \qquad \frac{}{\Gamma \vdash \texttt{bool} \sim \texttt{bool}}\ \textit{C-Bool} \qquad \frac{\Gamma \vdash \mathtt{S} \sim \tau}{\Gamma \vdash \blacksquare \mathtt{S} \sim \blacksquare \tau}\ \textit{C-Box}$$

$$\frac{\Gamma \vdash \mathtt{S} \sim \tau}{\Gamma \vdash \blacklozenge \mathtt{S} \sim \blacklozenge \tau}\ \textit{C-Trc} \qquad \frac{\forall_i (\Gamma \vdash \mathtt{S} \sim \tau \wedge \Gamma(\omega_i) = \blacklozenge \tau)}{\Gamma \vdash \blacklozenge \mathtt{S} \sim \lozenge \overline{\omega}}\ \textit{C-Clone}$$

$$\frac{\forall_i (\Gamma \vdash \mathtt{S} \sim \tau \wedge \Gamma(\omega_i) = \tau)}{\Gamma \vdash \&'\mathtt{a}[\mathtt{mut}]\ \mathtt{S} \sim \&[\mathtt{mut}]\ \overline{\omega}}\ \textit{C-Borrow}$$

Expanded Function Invocation within the `Spawn` Expression

In the preceding Section, we covered the function call outside the `spawn` expression. Now, let us go through into the modifications required for the semantics and typing of the spawn expression, as outlined in chapter 4. The *R-Spawn* rule is updated with the following adjustments when a sequence of signals is included in the function call arguments:

$$\frac{t \in \mathit{fresh} \quad \mathcal{D}(f) = \lambda(\overline{x}; \overline{s})\{\overline{e}\}^{\mathtt{m}} \quad \Theta(* \Rightarrow \{\overline{e}\}^{\mathtt{m}}) = \{\overline{e}\}^{\mathtt{n}} \quad (S', \overline{v'}) = \mathit{activate}(S_1, \overline{v}) \\ \overline{\langle \ell_a^s \rangle^{\mathtt{r}} = \mathit{read}(S', s, \mathtt{1})} \quad S_2 = S'[\overline{\ell_{\mathtt{n}::x} \mapsto \langle v \rangle^{\mathtt{n}}}, \overline{\ell_{\mathtt{n}::s} \mapsto \langle \ell_a^s \rangle^{\mathtt{n}}}] \quad \psi_2 = \psi_1[\overline{\ell_a \mapsto \langle v \rangle^{\mathtt{i+1}}}]}{\langle \psi_1, S_1 \triangleright \mathtt{spawn}(f(\overline{v}; \overline{s})) \xrightarrow{\{(t, \{\overline{e}\}^{\mathtt{n}})\}}_0 \psi_2, S_2 \triangleright \epsilon \rangle^{\mathtt{1}}}\ (\textit{R-Spawn})$$

The *R-Spawn* rule introduces several additional premises. Notably, since the invoked function now anticipates a sequence of signals $\overline{s}$, the rule obtains the location of each signal given as a parameter. Subsequently, it increments the respective counter in the signal environment ψ_1.

In Section 1.5.5, we have made some modifications to the mechanism as follows: $\Gamma_1 \vdash (\overline{\mathtt{S}} \to \mathtt{S}) \Longleftarrow_{\mathtt{b}} (\overline{\tau} \to \tau) \dashv \Gamma_2$. These changes also impact the typing of the function inside the `spawn` expression, as discussed in Chapter 4 (Section 1.5.2). Similar to the previous case, the typing of the function call is limited to typing the arguments and verifying the restrictions imposed by our mechanism: $\Gamma \vdash (\overline{\mathtt{S}} \to \mathtt{S}) \Longleftarrow_{\mathtt{tt}} (\overline{\tau} \to \tau)$. In the case of a function call within the `spawn` expression, there are no side effects, so we have removed Γ_2 from the mechanism. Additionally, the value of `b` is set to `tt` to inform our type system that we are inside the `spawn`. When compared to the scenario where the function is called outside the `spawn` expression, we encounter some distinct limitations (e.g. the prohibition of having references in the arguments). However, these restrictions align with those mentioned in Chapter 4 (Section 1.5.2). As a result, we present the *T-Spawn* typing rule below to handle this new form of function call:

$$\frac{\mathcal{D}(f) = (\overline{\mathtt{S}}; \overline{s}) \to_{\kappa'} (\epsilon) \quad \mathscr{L}_1, \Gamma_1 \vdash \langle \overline{e : \tau} \rangle_\sigma^1 \dashv \mathscr{L}_2, \Gamma_2 \\ \overline{s} \in \mathscr{L}_1 \quad \Gamma_2 \vdash (\overline{\mathtt{S}} \to \epsilon) \Longleftarrow_{\mathtt{tt}} (\overline{\tau} \to \epsilon)}{\mathscr{L}_1, \Gamma_1 \vdash_\kappa \langle \mathtt{spawn}(f(\overline{e}; \overline{s})) : \epsilon \rangle_\sigma^1 \dashv \mathscr{L}_2, \Gamma_2}\ (\textit{T-Spawn})$$

Since the current thread retains control during the execution of the spawn expression, there is no need to check the effect κ. Furthermore, as mentioned earlier, the restrictions imposed

by the following mechanism $\Gamma_2 \vdash (\overline{S} \rightarrow \epsilon) \Longleftarrow_{tt} (\overline{\tau} \rightarrow \epsilon)$ to ensure the typing safety of the `spawn` expression align with those in Chapter 4. However, considering the form of signatures presented in Figure 5.1, a function may have arguments of reference or active `Trc` types. Therefore, there is an additional requirement to enhance the mechanism to handle both of these cases. Hereafter, we provide a new formal Definition of the mechanism as follows:

1. for all $\tau_1, \tau_2 \in \overline{\tau}$ if *containsType*$(\Gamma, \tau_1, \diamond\overline{\omega})$ and *containsType*$(\Gamma, \tau_2, \diamond\overline{u})$ then, $\neg\exists_{i,j}.(\omega_i \bowtie u_j)$.

2. for all $\tau \in \overline{\tau}$, we have $\neg$*containsType*$(\Gamma, \tau, \blacklozenge\tau')$ and $\neg$*containsType*$(\Gamma, \tau, \&[\texttt{mut}]\overline{u})$ for some τ'.

3. After verifying the necessary constraints, the mechanism is capable of activating the inactive `Trc` type in $\overline{\tau}$ and ensuring compatibility between signatures and types as follows: (1) $\overline{\tau'}$ = *activate*$(\Gamma, \overline{\tau})$ (Definition 4.14 in Chapter 4) and (2) $\overline{\Gamma_2 \vdash S \sim \tau'}$ (Expanded Signature and Type Compatibility Definition 5.6).

In the following we present an example written in MSSL according to Figure 5.1, that showcases the effectiveness of the mechanism:

$$\begin{aligned}&\texttt{fn } f_1(\texttt{mut } x : \blacklozenge\texttt{int}, \texttt{mut } y : \blacksquare \blacklozenge \texttt{ int}, \texttt{mut } z : \blacklozenge\&'\texttt{a int}) \rightarrow_1 \epsilon_0\{...\}^1\\ &\quad \{\texttt{let mut } x = \texttt{trc}(0);\ \texttt{let mut } y = \texttt{trc}(0);\ \texttt{let mut } z = 1;\\ &\texttt{let mut } a = \texttt{box}(y.clone);\ \texttt{let mut } b = \texttt{trc}(\&z);\ \texttt{spawn}(f_1(x.clone, a, b.clone))\}^{\texttt{m}}\end{aligned} \tag{5.19}$$

Example 5.19 is rejected by the MSSL type system. When typing the function f_1 inside the `spawn` expression, the mechanism verifies the following constraints: constraint (1) is valid otherwise constraint (2) is invalid since $b.clone$ has the type ($\diamond$ b), and b has the type ($\blacklozenge\&z$), which contains a reference to z. The presence of a reference in the parameters of f_1 is unsafe because the type system is not aware of the lifetime of the newly created thread. As a result, the mechanism detects an unsatisfied constraint, indicating that the example is rejected.

1.6 Expanded Synchronous Cooperative Threading Model in MSSL

As previously mentioned, the third extension introduced to MSSL aims to enhance its reactive capabilities, allowing threads to communicate through signals, namely, emit and await signals. In this Section, we explore these new reactive constructs as outlined in Figure 5.1, and provide a comprehensive explanation of their semantics and typing rules.

1.6.1 MSSL Cooperative Operational Semantics

Signals are mechanisms for synchronisation and communication between threads. In essence, a thread utilises a signal when it needs to wait for a specific condition, which can avoid the

risk of thread deadlock. In comparison to Pthreads in C, signals in MSSL are akin to condition variables. As mentioned earlier, threads in MSSL are cooperatively executed by the round-robin scheduler. When a thread emits a signal, it is broadcasted to all threads waiting for that signal simultaneously with its emission. The following Sections elucidate how a thread creates a signal, emits it, and awaits its occurrence. To begin, the creation of a signal is illustrated by the *R-Sig* rule, as shown below:

$$\frac{\ell_a \notin dom(\psi_1) \quad \psi_2 = \psi_1[\ell_a \mapsto \langle 0 \rangle^1] \quad S_2 = S_1[\ell_{1::s} \mapsto \langle \ell_a^s \rangle^1]}{\langle \psi_1, S_1 \triangleright \texttt{Sig}\ s \rightarrow_0 \psi_2, S_2 \triangleright \epsilon_0 \rangle^1} \quad \textit{(R-Sig)}$$

The *R-Sig* rule is utilised to create a signal, denoted as s, which promptly terminates. Subsequently, a *fresh* location ℓ_a is generated and stored in the ψ_1 signal environment with an initial value of 0. The handling of this signal depends on a counter (denoted as "i"), initialised to 1 upon its creation. Each thread can obtain a copy of this signal by incrementing its counter by 1. In MSSL, memory management is performed without the need for a garbage collector. Consequently, the deallocation of slots ℓ_a is carried out automatically based on the counter (i). This deallocation is executed only when the counter is 1, as prescribed by the *drop* function 5.2. When the counter is 1, it indicates that there is only one reference to this location, ensuring safe deallocation. Finally, *R-Sig* updates the program store S_1 by creating a new signal named "s", associated with a value ℓ_a^s and having a lifetime of the enclosing block, `1`.

Every thread has the ability to obtain a copy of each signal. Signals, as a reference counting type, implement copy semantics. However, unlike the explicit $clone$ expression used for the active `Trc` type, the signal copy is implicit. When a function is invoked, whether within a `spawn` expression or not, the counter of each signal used as an argument in the invoked function is automatically incremented by 1. In this context, we make use of the *R-Copy* rule, which was introduced earlier in Chapter 3, to handle such situations:

$$\frac{\textit{read}(S_1, s, \texttt{1}) = \langle \ell_a^s \rangle^{\texttt{m}}}{\langle \psi_1, S_1 \triangleright s \rightarrow_0 \psi_1, S_1 \triangleright \ell_a^s \rangle^1} \quad \textit{(R-Copy)}$$

The *R-Copy* rule facilitates copying the value of the signal s without causing any destructive reading. To achieve this, we employ the function *read* 3.2 (Chapter 3), which allows us to retrieve the value of the specified signal.

In MSSL, a cooperative thread can synchronize with other threads through the use of signals. The following rule outlines how a thread in MSSL emits a signal:

$$\frac{\textit{read}(S_1, s, \texttt{1}) = \langle \ell_a^s \rangle^{\texttt{m}} \quad \psi_2 = \psi_1[\ell_a \mapsto \langle 1 \rangle^{\texttt{i}}]}{\langle \psi_1, S_1 \triangleright \texttt{emit}(s) \rightarrow_0 \psi_2, S_1 \triangleright \epsilon_0 \rangle^1} \quad \textit{(R-Emit)}$$

The *R-Emit* rule facilitates the immediate emission of the specified signal, indicated by the expression `emit`(s). This rule effectively changes the value of the signal from 0 to 1. Subsequently, the scheduler takes charge of broadcasting the signal s to all threads that are awaiting it. This broadcasting feature enhances modularity as there is no requirement to keep track of which thread is waiting for a particular signal.

Cooperation model

In accordance with chapter 3, MSSL supports two forms of cooperation points: (1) explicit by using the `cooperate` expression and (2) implicit, threads wait for signals that are not emitted at the same instant (where it can be emitted later). On top of that, a thread can be suspended using the `when` and `cooperate` constructs. The purpose of introducing ϵ_0 and ϵ_1 values is to separate expressions that instantly terminate (e.g. assignment) from expressions that do not instantaneously complete (i.e. `cooperate`). Specifically, to manage the end of the instant in our semantics, it is imperative to identify the threads that have cooperated to be unblocked in the subsequent instant. This information is captured using ϵ_1. A minor modification is made to the *R-Cooperate* reduction rule, as follows:

$$\frac{}{\langle \psi, S \triangleright \texttt{cooperate} \rightarrow_1 \psi, S \triangleright \epsilon_1 \rangle^{\texttt{l}}} \quad \textit{(R-Cooperate)}$$

To facilitate waiting for a signal, a thread surrenders control to the scheduler. The execution of the current thread is then put on hold until the specified signal is emitted in subsequent cycles or instants. Once the signal is emitted by any other thread, the suspended thread resumes its execution. To implement this functionality, we introduce three reduction rules for awaiting signals, as follows:

- The *R-WhenFalse* rule denotes that for a given signal, it is absent if its value in the signal environement ψ is `"0"`:

$$\frac{\textit{read}(S_1, s, \texttt{l}) = \langle \ell_a^s \rangle^{\texttt{m}} \quad \psi_1(\ell_a) = \langle 0 \rangle^{\texttt{i}}}{\langle \psi_1, S_1 \triangleright \texttt{when}(s)\ \{e_1\}^{\texttt{m}} \rightarrow_1 \psi_1, S_1 \triangleright \texttt{when}(s)\ \{e_1\}^{\texttt{m}} \rangle^{\texttt{l}}} \quad \textit{(R-WhenFalse)}$$

The *R-WhenFalse* rule involves retrieving the value of the signal s from ψ_1 by reading its location from S_1 using the *read* function. In this case, the value is `"0"`. Subsequently, the current thread cooperates at the next cooperation point, relinquishing control back to the scheduler.

- On the other hand, the *R-WhenTrue* rule specifies that a given signal is present if it is emitted by another thread during the current instant. In such a scenario, the current thread continues its execution without any suspension:

$$\frac{\textit{read}(S_1, s, \texttt{l}) = \langle \ell_a^s \rangle^{\texttt{m}} \quad \psi_1(\ell_a) = \langle 1 \rangle^{\texttt{i}} \quad \langle \psi_1, S_1 \triangleright \{e_1\}^{\texttt{m}} \rightarrow_i \psi_2, S_2 \triangleright \{e_2\}^{\texttt{m}} \rangle^{\texttt{l}}}{\langle \psi_1, S_1 \triangleright \texttt{when}(s)\ \{e_1\}^{\texttt{m}} \rightarrow_i \psi_2, S_2 \triangleright \texttt{when}(s)\ \{e_2\}^{\texttt{m}} \rangle^{\texttt{l}}} \quad \textit{(R-WhenTrue)}$$

Similarly to the *R-WhenFalse* rule, the *R-WhenTrue* rule retrieves the value of the signal s given as a parameter, which is `"1"` in this case. And contrary to the previous rule, the current thread keeps executing the body of the when expression e_1.

- Finally, the *R-WhenTerm* rule specifies that once the expression within the `when` construct is completed, the execution of the `when` expression terminates immediately.

$$\frac{}{\langle \psi, S \triangleright \texttt{when}(s)\ \{\epsilon_i\}^{\texttt{m}} \rightarrow_i \psi, S \triangleright \{\epsilon_i\}^{\texttt{m}} \rangle^{\texttt{l}}} \quad \textit{(R-WhenTerm)}$$

Since ϵ_0 and ϵ_1 are introduced in our semantics to distinguish the reduction of a `cooperative` expression from other expressions. Thus, hereafter, we add a new sequence reduction rule to the one defined in Chapter 3, as follows:

$$\frac{}{\langle \psi, S \rhd \epsilon_1; \overline{e} \rightarrow_1 \psi, S \rhd \epsilon_1; \overline{e} \rangle^1} \quad \textit{(R-SeqCoop)}$$

As the *R-SeqCoop* rule demonstrates, when a thread executes a cooperative expression, it must remain blocked until the end of the instant.

Before diving into the semantics of the last reactive expression in MSSL, we first introduce the function that is essential for modeling the principal of the *weak watchdog* statement. The function is presented below:

Definition 5.7 (Inter-Instant) *Let* $T = \{t_i, \{e_i\}^{l_i} | 1 \leqslant i \leqslant N\}$ *be a set of threads and let* ψ *be a signal environment. The* `Kill` *function aims to reduce the* `watch` *expression and unblocks cooperative threads between the end of the current instant and the beginning of the next instant. Then,* $\texttt{Kill}(T, \psi)$ *is defined as* $T' = \{t_i, \{e_i'\}^{l_i} | 1 \leqslant i \leqslant N\}$ *such that* $e_i' = \texttt{KillExp}(e_i, \psi)$*:*

$$\texttt{KillExp}(\texttt{watch}(\ell_a^s)\ e, \psi) = \begin{cases} \epsilon_0 & \textit{if } \psi(\ell_a^s) = \langle 1 \rangle^i \\ \texttt{watch}(\ell_a^s)\ \texttt{KillExp}(e, \psi) & \textit{if } \psi(\ell_a^s) = \langle 0 \rangle^i \end{cases} \tag{5.20}$$

$$\texttt{KillExp}(\epsilon_1; e, \psi) = \epsilon_0; e$$

Function 5.7 serves to terminate the body of the `watch` expression immediately if the specified signal is emitted, as it returns ϵ_0. On the other hand, if the signal is not emitted, nothing happens, and the thread continues to wait for the next instant. It is important to mention that the second `KillExp` function is required to the threads that are waiting for the next instant (those cooperating at the current instant). This allows these threads to resume their execution for the subsequent instant.

In the following rule, *R-Watch*, it is indicated that during the current instant, the current thread executes the `watch` body immediately. In other words, the execution of the `watch` expression takes place in the inter-instant stage.

$$\frac{\langle \psi_1, S_1 \rhd \{e_1\}^m \rightarrow_i \psi_2, S_2 \rhd \{e_2\}^m \rangle^l}{\langle \psi_1, S_1 \rhd \texttt{watch}(s)\ \{e_1\}^m \rightarrow_i \psi_2, S_2 \rhd \texttt{watch}(s)\ \{e_2\}^m \rangle^l} \quad \textit{(R-Watch)}$$

Consequently, as soon as the body of `watch` is completed, the execution of `watch` terminates immediately as illustrated by the *R-WatchTerm* rule:

$$\frac{}{\langle \psi_1, S_1 \rhd \texttt{watch}(s)\ \{\epsilon_i\}^m \rightarrow_i \psi_1, S_1 \rhd \{\epsilon_i\}^m \rangle^l} \quad \textit{(R-WatchTerm)}$$

1.6.2 Expanded Thread Execution in MSSL

In Chapter 3 (Section 3.1), we have presented the reduction rules for the execution of a thread in more than one step. In this Section, we focus on the modified general form of the rule, while the context remains the same.

- The following rule executes a thread more than once:

$$\frac{\langle \psi, S \triangleright e \xrightarrow{T_0}_0 \psi'', S'' \triangleright e'' \rangle^1 \quad \langle \psi'', S'' \triangleright e'' \xRightarrow{T_1} \psi', S' \triangleright e' \rangle^1}{\langle \psi, S \triangleright e \xRightarrow{T_0 \cup T_1} \psi', S' \triangleright e' \rangle^1} \quad \textit{(R-Thread)}$$

- A thread can terminate its execution in three situations: (1) when it has completed its entire execution, (2) when it has finished executing for the current instant (i.e., it is explicitly cooperating), or (3) when it is waiting for a signal that is not emitted during the current instant (i.e., it is implicitly cooperating). The following rule exemplifies the first point:

$$\frac{}{\langle \psi, S \triangleright v \xRightarrow{\varnothing} \psi, S \triangleright v \rangle^1} \quad \textit{(R-ThreadTerm)}$$

- The following rule indicates that the current thread is suspended, meaning it is either waiting for a signal that is not currently emitted or is cooperating:

$$\frac{\langle \psi, S \triangleright e \xrightarrow{T_0}_1 \psi', S' \triangleright e' \rangle^1}{\langle \psi, S \triangleright e \xRightarrow{T_0} \psi', S' \triangleright e' \rangle^1} \quad \textit{(R-ThreadSuspend)}$$

In Chapter 3, there was some confusion between the concepts of a round (cycle) and an instant. However, with the introduction of signals, a clear distinction emerges between the concepts of a cycle and an instant. In the following Section, we provide the semantic rules for the execution of a cycle.

1.6.3 Cycles in MSSL

Using the concept of logical instants, the scheduler defines instants during which all threads execute until their next cooperation point. When a thread encounters a cooperate expression, it assumes control and waits until the next instant. Similarly, if a thread is waiting for a signal that has not been emitted yet, it can be resumed when the signal is eventually emitted by another thread. It is important to note that if a signal is emitted during an instant, the current instant is not terminated, and the threads are re-executed in a new cycle within the same instant.

As mentioned earlier, in MSSL, threads are executed cooperatively by a round-robin scheduler. At the beginning of each round, all threads that are ready for execution belong to the set T. During the round, each thread in T is executed for a maximum number of steps and then removed from T ($T' = T_{\setminus t}$). The set T' represents the remaining threads after the execution of the current round. In this Section, we will detail the rewrite rules that define the semantics of a round in MSSL.

1) Given that the set T comprises threads ready to be executed in the current cycle (round), the cycle is considered complete when all these threads have been executed. This is defined by the following rule:

$$\frac{}{\varnothing, \psi, S \Rrightarrow \varnothing, \psi, S} \quad \textit{(R-CycleEnd)}$$

2) To perform a cycle in MSSL, we present the following *big-step* rule:

$$\frac{(t, \{e\}^1) \in T \quad \langle \psi, S \triangleright e \xRightarrow{T_0} \psi', S' \triangleright e' \rangle^1 \quad T_{\setminus t}, \psi', S' \Rrightarrow T', \psi'', S''}{T, \psi, S \Rrightarrow T' \cup (t, \{e'\}^1) \cup T_0, \psi'', S''} \quad \textit{(R-Cycle)}$$

The *R-Cycle* rule stipulates that each thread in T executes more than once (recall Section 1.6.2). In addition, the premise $T_{\setminus t}, \psi', S' \Rrightarrow T', \psi'', S''$ means that after a thread is executed, it will be added to the new set T' to be executed again at the next instant. As mentioned earlier, the threads in the set T_0, created during the current instant, will not execute immediately after their creation to avoid interference with the running threads. Instead, they are included in the new state as follows: $T' \cup (t, \{e'\}^1) \cup T_0$.

1.6.4 Instants in MSSL

MSSL is a synchronous reactive programming language designed to provide deterministic and predictable behavior based on the core concept of logical instants. Additionally, when the scheduler schedules shared instants between threads, those threads can execute at the same rate with automatic synchronization at the end of each instant.

After defining the cycle, we can deduce that an instant can consist of one or more cycles. Now, the question arises: how is an instant managed? When a cycle is completed, there are two possibilities for the instant: (1) if a signal is emitted during the current cycle, the instant is not ended, and a new cycle starts. (2) If a signal is not emitted during a cycle, the instant is considered ended. Next, we will introduce the rewrite rules that define the semantics for executing an instant in MSSL.

1) The following rule indicates that the current instant is not yet terminated, and a new cycle will start within the same instant.

$$\frac{T_1, \psi, S \Rrightarrow T_2, \psi', S' \quad \psi \neq \psi' \quad T_2, \psi', S' \Longmapsto T_3, \psi'', S''}{T_1, \psi, S \Longmapsto T_3, \psi'', S''} \quad \textit{(R-Instant)}$$

In the *R-Instant* rule, the premise $\psi \neq \psi'$ indicates that one or more signals are emitted during the cycle. This implies that there could be threads waiting for a signal emitted during this cycle, and therefore, the current instant is not yet terminated. For this reason, the scheduler restarts a new cycle within the same instant. It is important to note that the signals are reset at the beginning of each instant. Hence, we present below a function responsible for reinitialising all the signals emitted during an instant as follows:

Definition 5.8 (Reset) *For a given signal environment ψ, we define the* `reset`*(ψ) function as follows:*

$$\texttt{reset}(\psi) = \forall \ell_a \in \psi, \psi[\ell_a \mapsto \langle 0 \rangle^{\texttt{i}}]$$

This function is responsible for resetting all the signals in ψ by changing their value from 1 to 0 (absent state).

2) The following rule defines the end of the instant. In this case, the *inter-instant* stage must be applied before the next instant begins, as shown below:

$$\frac{T_1, \psi, S \Rrightarrow T_2, \psi', S' \quad \psi = \psi'}{T_1, \psi, S \mapsto \texttt{Kill}(T_2, \psi'), \texttt{reset}(\psi'), S'} \quad (\textit{R-InstantEnd})$$

Unlike the previous rule, the premise $\psi = \psi'$ in the *R-InstantEnd* rule indicates that no new signal was emitted during the current cycle. As a result, the scheduler determines the end of the instant. Besides, as previously stated, when the end of the instant is determined, an additional step called "*inter-instant*" 5.7 is performed. The purpose of this step is to unblock a thread at the next instant. Additionally, the rule states that all signals emitted during the current instant are reset to 0.

1.6.5 Chaining of Instants in MSSL

A program in MSSL is composed of several instants. Then, the main purpose of semantics is to reveal that the execution of a program in MSSL can result in an infinite sequence of instants, following the pattern:

$$T_1, \psi, S \mapsto \texttt{Kill}(T_2, \psi'), \psi'', S' \qquad T_2, \psi'', S' \mapsto \texttt{Kill}(T_3, \psi'''), \psi'''', S'' \quad \ldots.$$

Where $\psi'' = \texttt{reset}(\psi')$ and $\psi'''' = \texttt{reset}(\psi''')$ ...

In this Section, we define a sequence of complete instants as a chain of instants. Furthermore, we introduce the operational semantics of chaining instants denoted by $\mapsto^*$. This notation corresponds to the *reflexive transitive closure* of $\mapsto$.

1.6.6 MSSL Cooperative Typing Rules

In this Section, we focus on the typing rules for the fundamental reactive expressions in MSSL. In addition to cooperate, threads can synchronize themselves using signals. As mentioned elsewhere, a thread waiting for a signal that is not emitted during the current instant is considered implicit cooperation. However, this behavior should be treated as explicit cooperation since the thread relinquishes control back to the scheduler. Consequently, another thread executes and might create undesirable data in the memory of other threads. Therefore, our type system is responsible for checking scenarios where a thread could potentially compromise the safety of other threads' memory.

Before delving into the specific typing rules for the new reactive expressions, let us briefly review the *T-Cooperate* rule as follows:

$$\frac{\textit{safeTrc}\,(\Gamma)}{\mathscr{L}, \Gamma \vdash_1 \langle \texttt{cooperate} : \epsilon \rangle^1_\sigma \dashv \mathscr{L}, \Gamma} \quad (\textit{T-Cooperate})$$

The *T-Cooperate* rule utilises the *safeTrc* function 4.5 to address the aforementioned issue. Its purpose is to ensure that shared data between threads is not borrowed from the current

thread's typing environment. Additionally, the *T-Cooperate* rule mandates that the current function has an effect κ of 1, which is crucial for conveying information during function calls.

The following rule is necessary to type the expression for creating a signal:

$$\frac{s \notin \mathscr{L}_1 \quad \mathscr{L}_2 = \mathscr{L}_1 \sqcup \{s_1\}}{\mathscr{L}_1, \Gamma_1 \vdash_\kappa \langle \texttt{Sig}\ s : \epsilon \rangle^1_\sigma \dashv \mathscr{L}_2, \Gamma_1} \quad (\textit{T-Sig})$$

In MSSL, variable shadowing is not allowed for consistency, and the same applies to signal shadowing. Hence, the premise $s \notin \mathscr{L}_1$ ensures that signal s is not already declared. Similarly to variables, we must ascertain the signals generated within each block to deallocate them at the block's end. Consequently, when adding the new signal s to $\mathscr{L}_1$, we link it with the lifetime of the enclosing block, as evident from the premise $\mathscr{L}_1 \sqcup \{s_1\}$. However, for copying a signal s, the *T-CopySig* rule requires that s already exists in the signal environment $\mathscr{L}$ as follows:

$$\frac{s \in \mathscr{L}}{\mathscr{L}, \Gamma \vdash_\kappa \langle s : \epsilon \rangle^1_\sigma \dashv \mathscr{L}, \Gamma} \quad (\textit{T-CopySig})$$

The *T-Emit* rule governs the emission of a signal in an MSSL program and is defined as follows:

$$\frac{s \in \mathscr{L}}{\mathscr{L}, \Gamma \vdash_\kappa \langle \texttt{emit}(s) : \epsilon \rangle^1_\sigma \dashv \mathscr{L}, \Gamma} \ (\textit{T-Emit})$$

To emit a signal, the *T-Emit* rule mandates that the specified signal, provided as a parameter must already be declared, as indicated by the premise: $s \in \mathscr{L}$. For typing the `when` expression, we present the following rule:

$$\frac{s \in \mathscr{L}_1 \quad \textit{safeTrc}(\Gamma_1) \quad \mathscr{L}_1, \Gamma_1 \vdash_1 \langle \{e_1\}^{\mathtt{m}} : \tau \rangle^1_\sigma \dashv \mathscr{L}_2, \Gamma_2}{\mathscr{L}_1, \Gamma_1 \vdash_1 \langle \texttt{when}(s)\ \{e_1\}^{\mathtt{m}} : \tau \rangle^1_\sigma \dashv \mathscr{L}_2, \Gamma_2} \quad (\textit{T-When})$$

Similar to the previous cases, the *T-When* rule also necessitates that the specified signal is already declared. Moreover, like the *T-Cooperate* rule, when a signal is not yet emitted, the current thread cooperates until the next cooperation point, where another thread takes control. To safeguard the shared data between threads, the *safeTrc* function is employed, as demonstrated in the *T-When* rule. Additionally, this rule stipulates that the effect κ of the current function is 1. Lastly, the typing rule for the `watch` expression is straightforward and coherent with the earlier rules. The *T-Watch* rule is presented as follows:

$$\frac{s \in \mathscr{L}_1 \quad \mathscr{L}_1, \Gamma_1 \vdash_1 \langle \{e_1\}^{\mathtt{m}} : \tau \rangle^1_\sigma \dashv \mathscr{L}_2, \Gamma_2}{\mathscr{L}_1, \Gamma_1 \vdash_\kappa \langle \texttt{watch}(s)\ \{e_1\}^{\mathtt{m}} : \tau \rangle^1_\sigma \dashv \mathscr{L}_2, \Gamma_2} \quad (\textit{T-Watch})$$

Indeed, the *T-Watch* rule simply requires the presence of a signal in the parameter of the `watch` expression.

2 Example of MSSL

Let us explore a short example to gain a better understanding of MSSL's synchronous cooperative threading model. The main objective of this example is to emphasize the core

features, especially the cooperative operations like threading, cycling, and instantaneous execution in MSSL. Note that this example demonstrates all the reactive constructs supported by MSSL.

```
1  // Thread A:
2  fn tha(s1, s2, s3) ->1 unit{
3      emit(s1);
4      emit(s2);
5      cooperate;
6      emit(s3);
7  }
8
9  // Thread B:
10 fn thb(s1, s2, s3) ->1 unit{
11     when(s1){
12         watch(s2){
13             when(s3){
14                 cooperate;
15             }
16         }
17     }
18 }
19
20 //main
21 {
22     Sig s1; Sig s2; Sig s3; //signal declaration
23     spawn(tha(s1,s2,s3));
24     spawn(thb(s1,s2,s3));
25 }
```

Listing 5.1: An Illustrative Instance Written in MSSL Showcasing Reactivity.

Listing 5.1 provides an example of two threads, `A` and `B`, that synchronize using signals `s1, s2` and `s3`. Note that the effect of the type system is represented by (->1) in the `tha` and `thb` functions. On line 22, three signals, `s1, s2` and `s3`, are created and assigned to them respectively. Following this, on lines 23 and 24, two threads, `tha` and `thb`, are created using the `spawn` expression. Now, let us delve into the scheduling of these threads and explain when and how each thread executes, illustrated in the diagram for each logical instant in Figure 5.3.

In the Figure 5.3, we provide an overview of the execution of the two threads as follows:

- **During Instant 1** :
    1. `Thread A` emits `s1` and `s2` and then cooperates on line 5, explicitly completing its execution for the current instant.

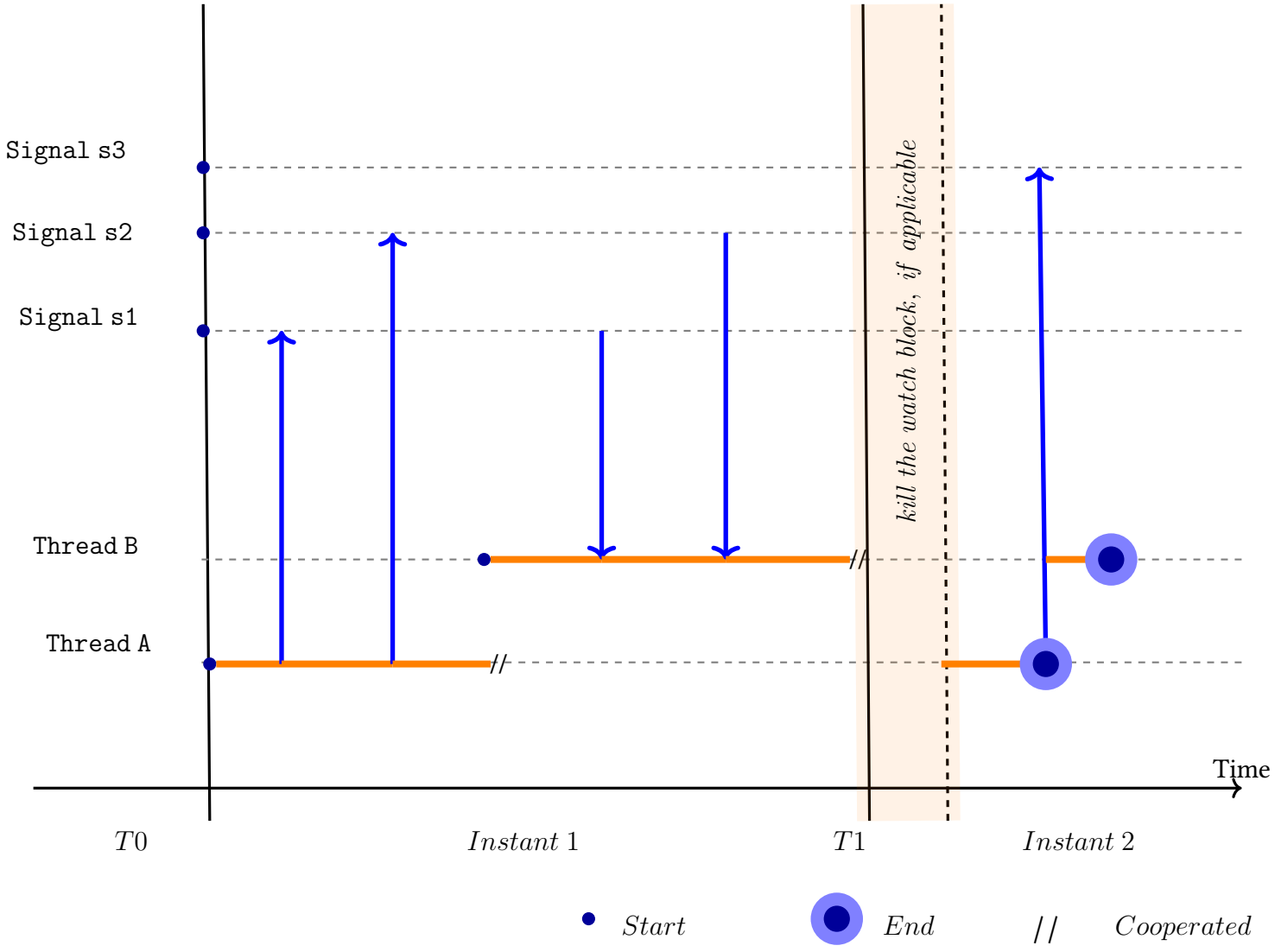

Figure 5.3: Functional Overview of Thread Execution During Instants.

2. `Thread B` waits for `s1`, which has already been emitted by `Thread A` during the instant. Therefore, `Thread B` does not cooperate and executes the body of the `when` expression. On line 9, `Thread B` *immediately* runs the body of the `watch` expression without verifying whether `s2` is emitted or not. Afterwards, it cooperates by waiting for `s3`

- When all threads are executed, the scheduler defines the end of the instant
- At this point, the *inter-instant* step occurs. Before the beginning of the next instant, the expression of the `watch` is reduced using the `Kill` function 5.7.

1. Using the `Kill` function, we terminate the non-terminating block in the `watch` expression of `Thread B` since the signal `s2` is emitted during the instant.

- At the start of **Instant 2**, all signals are *reset* using the `reset` function 5.8 and all ϵ_1 values are modified to ϵ_0.
- **During Instant 2** :

1. `Thread A` emits the signal `s3` and immediately terminates
2. `Thread B`, after killing the body of the `watch`, also ends promptly.

3 Discussion

In this chapter, we have extended MSSL with several constructs. Specifically, handling function calls outside the `spawn` expression requires more attention than function calls inside the `spawn` expression. However, MSSL already provides the necessary features to manage this aspect effectively. Going back to what was said before, for function calls, a new typing environment and execution environment are established to handle the function's duration. This enables us to extend the notion of safe abstraction (as defined in 4.24) with the support of the notion "$S_{|1}$" to specify a portion of memory for the function call. Furthermore, with regard to the extension of the control flow and as already discussed, its typing returns the *union* of the environments of the two branches, for example:

$$\{\texttt{let mut } x = 0;\ \texttt{let mut } y = 0;\ \texttt{let mut } z = 1;\ \texttt{let mut } a = \&x;\ \texttt{let mut } b = \&\texttt{mut } y;\ \texttt{if } \textbf{cond } \{a = \&\, z\}^{\texttt{n}}\texttt{else}\{b = \&\texttt{mut } z\}^{\texttt{m}}\}^{1} \tag{5.21}$$

After typing the `if/else` expression, the resulting typing environment is as follows: $\Gamma = [x \mapsto \langle \texttt{int} \rangle^1,\ y \mapsto \langle \texttt{int} \rangle^1,\ z \mapsto \langle \texttt{int} \rangle^1, a \mapsto \langle \&\, z \rangle^1, b \mapsto \langle \&\texttt{mut } z \rangle^1]$. At this stage and according to the Definition 4.25 this environment is *ill-formed* because z is borrowed both mutable and immutable. However, in practice, this example is safe because at runtime, only one branch will be executed, and z will be borrowed either as mutable or immutable. Similarly, for `Trc`, it is possible to handle cases where one branch moves an active `Trc` while the other branch creates a clone. In summary, there is no need to be overly concerned about the safety level; we can adapt Definition 4.25 to accommodate such cases or create a new Definition. Furthermore, MSSL does not include `loops` or `while` in its syntax to avoid issues of divergence. A reactive program that diverges would not make sense in this context (and would require adding a *co-induction* proof).

Chapter 6

Implementation

Patience is a key element of success

– Bill Gates

This chapter provides a comprehensive explanation of the entire MSSL language set implementation. It begins with an introduction to the cooperative set (FR_{FT}), emphasizing the implementation of our `Trc` contribution and its translation into Fairthreads. Additionally, two examples of reactive multithreading employing the cooperative construct are presented. The chapter then proceeds to the following section, which centers around the implementation of the extensions introduced in Chapter 5.

Dans ce chapitre, nous abordons la mise en place complète du langage MSSL. Nous débutons par la présentation de l'ensemble coopératif (FR_{FT}), en mettant l'accent sur la mise en œuvre de notre contribution `Trc` et la traduction de celle-ci en Fairthreads. Ensuite, nous concluons avec deux exemples d'utilisation du multithreading réactif en utilisant la construction coopérative. Par la suite, nous continuons ce chapitre avec une seconde section se concentrant sur la mise en œuvre des extensions présentées dans le chapitre 5.

1 Overview

The significance of MSSL lies in its ability to enhance the Fairthreads programming model. In earlier chapters, we delved into the core of the Rust type system and established a type system for a kernel programming language called FR_{FT}. The FR_{FT} type system incorporates two essential Rust features: ownership and borrowing, complemented by an innovative smart pointer design known as `Trc`. A notable advantage of FR_{FT} over Rust is its capability to enable data sharing between cooperative threads without requiring `Mutexes`, thanks to the encapsulation of shared data in `Trc`. Additionally, we introduced reactivity to Rust, introducing reactive constructs and the ability to create cooperative threads.

Once we have defined the semantics and typing rules for FR_{FT}, and have established the soundness of our type system, the next step involves incorporating the practical elements to evaluate our efforts. This chapter, therefore, presents an implementation of MSSL, which is built upon the operational semantics discussed in Chapter 3 and the typing rules from Chapter 4. To kickstart our implementation, we articulate the practical aspects of Rust (FR [73]) by introducing a new form of smart pointer called `Trc`, along with two multithreaded constructions to enforce our rules effectively.

The implementation of `Trc` is based on the reference counting approach, a memory management technique that automatically tracks and manages the lifetime of objects in MSSL. As we have previously discussed, a vital requirement for reactive systems is a programming language capable of automatic memory management without relying on a garbage collection system. Consequently, in our implementation, we have successfully integrated the drop 3.4 function to deallocate memory when needed. In practice, given that MSSL follows the lexical scope approach, the drop function is automatically inserted at the end of each block, leading to recursive deallocation of data based on its type.

Our implementation is realized in Java and is freely accessible on GitHub. Specifically, to generate the lexer and parser, we utilised ANTLR (ANother Tool for Language Recognition) [11], which proves highly effective for automatically generating these components based on the defined input language and grammar rules. ANTLR takes a formal grammar specification as input and generates a parser capable of recognizing and processing the input language according to the specified grammar rules.

As mentioned earlier, the primary goal of MSSL is to enhance the Fairthreads programming model. Consequently, once we have performed a safe verification of references and data shared between threads within an MSSL program, we provide an automatic translation into Reactive-C, especially targeting Fairthreads in C [26]. This translation offers both memory safety and automated data management, without requiring user intervention.

This implementation ensures robust local memory safety for each thread, as well as strong global memory safety between threads, leveraging the characteristics of Rust. Throughout the remainder of this chapter, we initially present the implementation of the type system without the extensions, inspired by the reference implementation provided by FR [73]. Subsequently, we go through the implementation of the extensions introduced in Chapter 5. Moreover, we have sugar coated our implementation by incorporating "tuples" into MSSL. The introduction of tuples allows us to test more meaningful and significant

examples, enabling a deeper exploration of our insights.

1.1 Implementation of MSSL

The implementation is derived from the semantic and typing rules detailed in Chapters 3 and 4. It comprises three key components: (1) an abstract syntax tree constructed using ANTLR, (2) a type system that incorporates a borrow checker, and (3) a translation mechanism to convert to C. To further illustrate, let's examine the reduction rule applied to the creation of a new `Trc`:

$$\frac{\ell_a \notin dom(S_1) \quad S_2 = S_1[\ell_a \mapsto \langle v \rangle^1]}{\langle S_1 \triangleright \texttt{trc}(v) \rightarrow_0 S_2 \triangleright \ell_a^{\blacklozenge} \rangle^1} \quad (\textit{R-Trc})$$

The *R-Trc* rule is responsible for generating a new location ℓ_a in the program store S_1 and initialising it to the value v. As `Trc` operates based on reference counting, its counter is initialised to 1 in accordance with the *R-Trc* rule. Thus, the implementation of this rule in MSSL is as follows:

```
1     //  R-Trc.
2     protected Pair<State, Expression> reduceTrc(State state,
3                                       Lifetime lifetime, Value value) {
4         // get the global lifetime
5         Lifetime global = lifetime.getRoot();
6         // allocate a new location with the global lifetime
7         // Set the trc counter to 1
8         return state.allocate(global, value, 1);
9     }
```

Listing 6.1: Implementing the *R-Trc* Rule.

As illustrated in Listing 6.1, to create a new `Trc`, first we require the current program store, S. The latter, in the semantic rule, is referred to: as `State` in Listing 6.1. Based on the definition of S in Chapter 3, the `State` is implemented as a `Map`, which serves to map variables to their respective locations (i.e., `StackFrame`) and also maps locations to their corresponding values (i.e., Store). Subsequently, to proceed with the creation of a new `Trc`, the value v needs to be retrieved and allocated within the `State` by calling the allocate function. This function takes one argument which represents the `Trc` counter

Additionally, we consider the typing rule *T-Trc* introduced in Chapter 4:

$$\frac{\Gamma_1 \vdash \langle e : \tau \rangle_\sigma^1 \dashv \Gamma_2 \quad \neg \textit{containsTrc}(\Gamma_2, \tau)}{\Gamma_1 \vdash \langle \texttt{trc}(e) : \blacklozenge\tau \rangle_\sigma^1 \dashv \Gamma_2} \quad (\textit{T-Trc})$$

As previously detailed, the *T-Trc* rule checks whether the type of expression e contains a `Trc` type. Consequently, the corresponding implementation of this rule in our system is as follows:

```
1      // T-Trc
2     protected Pair<Environment, Type> apply(Environment gam1,
3             Lifetime lifetime, Syntax.Expression.Trc expression) {
4         // first premise: typing the expression
5         Pair<Environment, Type> typing = apply(gam1, lifetime,
6                         expression.getOperand());
7         Environment gam2 = typing.first();
8         Type type = typing.second();
9         // second premise: applying the containsTrc function
10        try {
11            check(containsTrc(gam2, type),
12                        "This type contains a Trc type ");
13        } catch (ExceptionsMSG e) {
14            throw new RuntimeException(e);
15        }
16       return new Pair<>(gam2, new Type.Trc(type));
17    }
```

Listing 6.2: Implementation of the *T-Trc* Rule.

As shown in Listing 6.2, the process of typing the expression `trc`(e) begins with the requirement of the typing environment Γ, which is defined in Chapter 4 and associates variables with their corresponding types and lifetimes. In our implementation, we represent Γ as an `Environment`, where each variable name maps to a location that further maps to the type and lifetime. on Line 5, we proceed to type the expression provided as a parameter and subsequently apply the *containsTrc* function. This function examines the type of the expression and throws an exception if a `Trc` type is detected. If the result is successful, we return the resulting environment and a `Trc` type. Moreover, we have implemented the `Trc` structure in C to incorporate reference counting. Hence, when the expression `trc`(e) is verified by the function outlined in Listing 6.2, we automatically utilise our library to translate this new type of smart pointer into a C reference counting mechanism.

Given that we can also create an inactive `Trc` , let us dive into the implementation of this category in MSSL. As mentioned previously, we will now revisit the semantic rule that outlines the reduction of an inactive `Trc`:

$$\frac{\langle \ell_a^{\blacklozenge} \rangle^{\mathtt{m}} = \mathit{read}(S_1, \omega, \mathtt{1}) \quad S_1(\ell_a) = \langle v \rangle^{\mathtt{i}} \quad S_2 = S_1[\![\ell_a \mapsto \langle v \rangle^{\mathtt{i+1}}]\!]}{\langle S_1 \triangleright \omega.\mathit{clone} \rightarrow_0 S_2 \triangleright \ell_a^{\diamond} \rangle^{\mathtt{1}}} \quad (\textit{R-Clone})$$

To clone an active `Trc`, we retrieve its value, unlike mutable and immutable references, which access its location. Afterwards, we increment its counter to 1, in accordance with the *R-Clone* rule. Now, let us detail the implementation of *R-Clone* in MSSL:

```
1     // R-Clone.
2     protected Pair<State, Expression> reduceClone(State state,
3                         Lifetime l, Lval lval) {
4         // read the value of the lval
```

```
5           Value lx = state.read(lval, l);
6           // increments the Trc counter by 1
7           Pair<State, Value.Reference> S2 =
8                                       state.increment_counter(lval, l);
9           return new Pair<>(S2.first(), S2.second().toCloned());
10      }
```

Listing 6.3: Implementation of the *R-Clone* Rule.

In Listing 6.3, we can observe the implementation of the *R-Clone* rule in MSSL. We obtain the lval value from the program's store state by utilizing the *read* function. The read function's implementation aligns with the definition provided in 3.2. Subsequently, on Line 7, we increment the counter of the retrieved value using the *increment_counter* function. However, it is crucial to explore the implementation of the *T-Clone* rule in further detail:

$$\frac{\Gamma \vdash \omega : \langle \blacklozenge \tau \rangle^{\mathtt{m}}}{\Gamma \vdash \langle \omega.clone : \diamond \omega \rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma} \quad (\textit{T-Clone})$$

The *T-Clone* rule primarily ensures that ω is of an active `Trc` type. However, at the implementation level, to ensure optimal performance, we incorporate several checks as outlined below:

```
1       // T-Clone
2       protected Pair<Environment, Type> apply(Environment gam,
3               Lifetime lifetime, Syntax.Expression.Clone expression) {
4           Lval w = expression.getOperand();
5           // Determine type being read
6           Pair<Type,Lifetime> p = w.typeOf(gam);
7           // verify if w exists
8           if(p == null){
9               try {
10                  check(true, w.name()+"␣is␣invalid");
11              } catch (ExceptionsMSG e) {
12                  throw new RuntimeException(e);
13              }
14          }
15         // Extract w's type
16          Type T = p.first();
17          // verify if w is well-typed
18          try {
19              check(!T.defined(), w.name()+"␣is␣moved");
20          } catch (ExceptionsMSG e) {
21              throw new RuntimeException(e);
22          }
23          // verify if w has an active Trc type
```

```
        try {
            check(!(T instanceof Type.Trc), w.name()+
                        "does not have the Trc type");
        } catch (ExceptionsMSG e) {
            throw new RuntimeException(e);
        }
        //Done
        return new Pair<>(gam, new Type.Clone(w));
    }
```

Listing 6.4: Implementation of the *T-Clone* Rule.

As demonstrated in Listing 6.4, the first step involves obtaining ω, which is defined in the form (path | variable name). Subsequently, we conduct the following checks: (1) Verify whether ω exists in Γ (the typing environment). (2) Check if ω has already been moved. (3) Ensure that ω is of an active `Trc` type. In the event of failure for any of these requirements, we throw an exception. Upon successful verification of the typing for the $\omega.clone$ expression, similar to an active `Trc`, we automatically invoke our library to increment the `Trc` counter to 1.

Whenever a thread is created, several restrictions are enforced by the mechanism ($\Gamma_2 \vdash (\overline{\mathbb{S}}) \Longleftarrow (\overline{\tau})$) in the *T-Spawn* rule below:

$$\frac{\mathcal{D}(f) = (\overline{\mathbb{S}}) \quad \Gamma_1 \vdash \langle \overline{e : \tau} \rangle^{1}_{\sigma} \dashv \Gamma_2 \qquad \Gamma_2 \vdash (\overline{\mathbb{S}}) \Longleftarrow (\overline{\tau})}{\Gamma_1 \vdash \langle \texttt{spawn}(f(\overline{e})) : \epsilon \rangle^{1}_{\sigma} \dashv \Gamma_2} \qquad (\textit{T-Spawn})$$

As explained in Chapter 4, during spawn time, inactive `Trc`'s transform into active `Trc`'s. However, the mentioned mechanism needs to verify beforehand whether there exist two inactive `Trc`'s pointing to the same memory region. This constraint is crucial for preserving the uniqueness of `Trc` property enforced by our type system. Now, we present the implementation of the *T-Spawn* rule in MSSL, ensuring compliance with the requirements of our mechanism:

```
    // T-Spawn
    protected Pair<Environment, Type> apply(Environment gam,
            Lifetime lifetime, InvokeFunction expression) {
        // premise 1: identify the invoked function
        Function declaration = functions.get(expression.getName());
                    ...
        // premise (3): verify mechanism constraints
        Environment gam2 = typedarguments.first();
        Type[] args = typedarguments.second();
        // (3.1): the compatibilities between types and signatures
        try {
            check(!compatibleSigType(parameters, args, gam2),
```

```
13                      "Incompatible␣Argument(s)!");
14          } catch (ExceptionsMSG e) {
15              throw new RuntimeException(e);}
16          // (3.2): verify if there exists two inactive Trc
17                 // pointing to the same memory location
18          try {
19              check(!invariantTrcSpawn(args, gam2),
20   "Having␣two␣inactive␣Trcs␣pointing␣to␣the␣same␣memory␣location!");
21          } catch (ExceptionsMSG e) {
22              throw new RuntimeException(e);}
23          return new Pair<>(gam2, Type.Unit);
24      }
```

Listing 6.5: Implementation of the *T-Spawn* Rule.

Listing 6.5 outlines the constraints enforced by the *T-Spawn* rule. For instance, on Line 11, our mechanism verifies the compatibility between signatures and types, following Figure 4.15. Then, it proceeds to check if two inactive `Trc`'s share the same active `Trc` by utilizing the *invariantTrcSpawn* function.

A significant aspect of our type system is to guarantee that a thread's execution does not generate unintended data in other thread's memory. This assurance is verified during cooperation time through the *T-Cooperate* rule:

$$\frac{\mathit{safeTrc}\,(\Gamma)}{\Gamma \vdash \langle \texttt{cooperate} : \epsilon \rangle^{1}_{\sigma} \dashv \Gamma} \qquad (\textit{T-Cooperate})$$

In accordance with Definition 4.5 of the *safeTrc* function, we thoroughly inspect the typing environment of the current thread to ensure the absence of any reference to shared data with other threads. The implementation of the *T-Cooperate* rule is presented in Listing 6.6.

```
1       // T-Cooperate
2       protected Pair<Environment, Type> apply(Environment gam,
3           Lifetime lifetime, Syntax.Expression.Cooperate expression) {
4           try {
5               check(!SafeTrc(gam),
6           "Borrowed␣shared␣data␣exists,␣it's␣not␣safe␣to␣cooperate");
7           } catch (ExceptionsMSG e) {
8               throw new RuntimeException(e);}
9           return new Pair<>(gam, new Type.Unit());
10      }
```

Listing 6.6: Implementation of the *T-Cooperate* Rule.

Like the *T-Cooperate* rule, Listing 6.6 also utilises the *safeTrc* function to assess the presence of any active `Trc` data being borrowed. Throughout the discussion, we have introduced significant semantic and typing rules, elucidating their implementations. Consequently, we

now showcase a practical example that demonstrates the application of our MSSL syntax in a reactive context.

```
fn bar(mut x:trc<int>, mut y:trc<int>)->unit{
    {
    let mut a = &mut *x;
    *a = 5;
    // a is a reference, it is not responsible
    //for dropping its content
    }
    cooperate;
    print!(*x);
    //The ‘drop‘ function is automatically called
    //for dropping x and y
}

fn foo(mut x: trc<int>, mut y: box<trc<int>>)->unit{
    *x = 1;
    cooperate;
    spawn(bar(x.clone, *y.clone));
    cooperate;
    print!(*x);
    //The ‘drop‘ function is automatically called
    //to drop x, y recursively
}

fn foofoo(mut x: trc<int>)->unit{
   {
    let mut y =&mut x;
    **y = 2;
    }
    cooperate;
    //The ‘drop‘ function is automatically called
    // to drop x
}

//main
{
let mut x = trc(0);
let mut y = trc(1);
spawn(foo(x.clone,box(y.clone)));
spawn(foofoo(x.clone));
//The ‘drop‘ function is automatically called
```

```
41 //to drop x and y
42 }
```

Listing 6.7: A Reactive Example accepted in MSSL.

Listing 6.7 presents a reactive program written in MSSL that is successfully accepted by its type system. This program creates three cooperative threads that safely share the same memory regions without the need for `mutexes`. Notably, the thread executing the `foo` function creates a new thread on Line 17 and cooperates. Due to the round-robin scheduler, the new thread will take control in the next instant and execute the `bar` function.

Within the body of the `bar` function, a mutable reference is created, pointing to the contents of `x` (i.e., shared data), and its value is mutated. Then, the thread cooperates on Line 8. Despite the existence of a reference to `x` (of type active `Trc`) on Line 3, its lifetime ends on Line 7. As a result, the cooperate expression on Line 8 is executed safely.

Importantly, since no exceptions are relevant to the program in Listing 6.7, it undergoes an automatic translation to Reactive-C. As in Rust, the primary objective of MSSL is to manage memory automatically without burdening the user. Consequently, we automatically call the *drop* function for a type when the defined variables go out of scope. To highlight this aspect, we have added comments at the end of each block in Listing 6.7, indicating the *drop* call and the associated variable. As previously mentioned, our implementation is available on GitHub, where we provide a more detailed explanation of how to execute an MSSL program.

Next, we showcase another reactive MSSL program, but this time, it is rejected by its type system:

```
1  fn bar(mut x:trc<int>, mut y:box<trc<int>>)->unit{
2      let mut a = *y.clone;
3      cooperate;
4      let mut b = *y;
5      spawn(foofoo(b.clone));
6  }
7
8  fn foo(mut x:trc<int>)->unit{
9      let mut y = x.clone;
10     spawn(bar(x.clone,box(y)));
11     cooperate;
12     *x=7;
13 }
14
15 fn foofoo(mut x:trc<int>)->unit{
16     //add code
17 }
18
19 //main
```

```
20 {
21 let mut x = trc(0);
22 spawn(foo(x.clone));
23 }
```

Listing 6.8: A Reactive Example rejected in MSSL.

In contrast to Listing 6.7, Listing 6.8 demonstrates a rejected program for the following reasons:

1. In the body of the `bar` function, on Line 2, an inactive `Trc` (a clone of the active `Trc` in the `Box`) is created and linked to 'a'. Subsequently, on Line 4, the content of variable 'y' is moved into 'b'. However, this violates the *T-Move* rule, leading to the application of the *TrcMoveProhibited* function. The violation occurs because 'y' contains an active `Trc` that is cloned in the current typing environment.

2. In the body of the `foo` function, on Line 9, an inactive `Trc` (a clone of 'x') is created and linked to 'y'. A new thread is then created to execute the `bar` function. on Line 10, `bar` takes two arguments, the first of type (◇ x) and the second of type (■ ◇ x). This triggers the mechanism of the *T-Spawn* rule, revealing an error since the two arguments point to the same memory region. Consequently, we reject this program to preserve the well-formedness of inactive `Trc` types and safeguard memory against aliasing. As a result, the translation to Reactive-C is not performed.

Thus far, we have illustrated the behavior of MSSL type systems for two different reactive programs. The information presented in this section is rich and valuable, and we encourage readers to experiment with MSSL programs for testing purposes. In the following section, we will briefly discuss the implementation of the extensions presented in Chapter 5.

1.2 Implementation with Extensions

In this section, we proceed with the implementation of the MSSL language set in its entirety with a specific emphasis on the extensions introduced in Chapter 5. These extensions encompass control flow, function calls outside the `spawn` expressions, and the reactive constructs.

For "control flow" extensions, there are two reduction rules, as follows:

$$\frac{}{\langle \psi, S \triangleright \texttt{if (true)}\ \{\overline{e}\}^{\texttt{n}}\ \texttt{else}\ \{\overline{e}\}^{\texttt{m}} \rightarrow_0 \psi, S \triangleright \{\overline{e}\}^{\texttt{n}} \rangle^{\texttt{l}}} \quad \textit{(R-IfTrue)}$$

$$\frac{}{\langle \psi, S \triangleright \texttt{if (false)}\ \{\overline{e}\}^{\texttt{n}}\ \texttt{else}\ \{\overline{e}\}^{\texttt{m}} \rightarrow_0 \psi, S \triangleright \{\overline{e}\}^{\texttt{m}} \rangle^{\texttt{l}}} \quad \textit{(R-IfFalse)}$$

Upon evaluating the condition specified in the "`if`" expression, two scenarios emerge: if the condition is true, the code block within the "`if`" branch will be executed, and if the condition

is false, the code block within the "`else`" branch will be executed. In a straightforward manner, the implementation of the "`if\else`" extensions in MSSL is as follows, based on the *R-Cond* rule introduced in Chapter 3:

```
    // R-IfElse
    protected Pair<State, Expression> apply(State S,
                    Lifetime lifetime, IfElse expression, int k) {
        // R-Cond
        Expression cond = expression.getConditions();
        if(cond){
            if(cond instanceof Value.Boolean){
            // R-IfTrue
            Boolean b = cond.value();
                if(b){
                    return new Pair<>(S, expression.getIfblock());
                }
                // R-IfFalse
                else {
                    return new Pair<>(S, expression.getElseblock());
                }
            }
            else {
                //exception
                    ...
            }
        }
        else {
            Pair<State, Expression> r1 = apply(S, lifetime,
                                expression.getConditions(), k);
            return new Pair<>(r1.first(), new IfElse(r1.second(),
            expression.getIfblock(),expression.getElseblock()));
        }
        return  new Pair<>(S, expression);
    }
```

Listing 6.9: Implementation of the *R-IfTrue* and *R-IfFalse* Rules.

As demonstrated in Listing 6.9, our initial step involves reducing the "`cond`" expression by invoking the function that describes the *R-cond* rule. Subsequently, we return the corresponding block expression based on the value obtained from the reduction. On the other hand, the *T-IF* typing rule demands more effort and attention:

$$\frac{\Gamma_1 \vdash_\kappa \langle e : \texttt{bool} \rangle^{\texttt{l}}_\sigma \dashv \Gamma_2 \quad \Gamma_2 \vdash_\kappa \langle \{e_1\}^{\texttt{n}} : \tau_1 \rangle^{\texttt{l}}_\sigma \dashv \Gamma_3 \quad \Gamma_2 \vdash_\kappa \langle \{e_2\}^{\texttt{m}} : \tau_2 \rangle^{\texttt{l}}_\sigma \dashv \Gamma_4}{\Gamma_1 \vdash_\kappa \langle \texttt{if}(e)\ \{e_1\}^{\texttt{n}}\ \texttt{else}\ \{e_2\}^{\texttt{m}} : \mathit{union}(\tau_1, \tau_2) \rangle^{\texttt{l}}_\sigma \dashv \Gamma_3 \sqcup \Gamma_4} \text{(T-IF)}$$

As elaborated in Chapter 5, our type system lacks the knowledge of which branch to execute during runtime. Hence, we must account for all potential scenarios, meaning we consider the side effects of both blocks. To accomplish this, the *T-IF* rule individually types the two blocks of the "`if`" and "`else`" branches, and subsequently, it combines their respective types using the "*unions*" function and their environments. The implementation of this rule is provided below:

```
1     // T-IF
2     protected Pair<Environment, Type> apply(Environment gam,
3     Lifetime lifetime, Syntax.Expression.IfElse expression, int k) {
4         //type the condition
5         Pair<Environment, Type> r1 = apply(gam, lifetime,
6                                     expression.getConditions(), k);
7             Environment gam1 = r1.first();
8             Type _t = r1.second();
9         //type the "if" block
10        Pair<Environment, Type> r2 = apply(gam1, lifetime,
11                                    expression.getIfblock(), k);
12            Environment gam2 = r2.first();
13            Type _t1 = r2.second();
14        //type the "else" block
15        Pair<Environment, Type> r3 = apply(gam2, lifetime,
16                                     expression.getElseblock(), k);
17            Environment gam3 = r3.first();
18            Type _t2 = r3.second();
19
20        // ensure the compatibilities of _t1 and _t2
21        try {
22        check(!compatibleShape(gam2, _t1, _t2), "Incompatible Type");
23        } catch (ExceptionsMSG e) {
24            throw new RuntimeException(e);
25        }
26        //join the environment
27        Environment gam4 = join(gam2, gam3, expression, k);
28        //join the type
29        return new Pair<>(gam4, _t1.union(_t2));
30    }
```

Listing 6.10: Implementation of the *T-IF* Rule.

Listing 6.10 provides a detailed explanation of typing the "`if\else`" expression, as follows:

1. Line 5 types the condition passed as a parameter, which must have a "`bool`" type.
2. on Line 10, we type the block of the "`if`" branch, and then on Line 15, we type the

block of the "`else`" branch.

3. Line 22 checks the types of the two blocks to combine their respective types.
4. on Line 27, the two environments are merged.
5. Line 29 combines the types of the two branches.

It is important to note that the integer "k" provided as a parameter represents the "κ" effect required by our type system.

MSSL is distinguished by its signal management, wherein MSSL signals are treated as reference counting. Consequently, upon signal creation, the counter is set to 1, and the value is initialised to 0 (absent by default), as shown by the *R-Sig* rule:

$$\frac{\ell_a \notin dom(\psi_1) \quad \psi_2 = \psi_1[\ell_a \mapsto \langle 0 \rangle^1] \quad S_2 = S_1[\ell_{1::s} \mapsto \langle \ell_a^s \rangle^1]}{\langle \psi_1, S_1 \rhd \texttt{Sig}\ s \rightarrow_0 \psi_2, S_2 \rhd \epsilon_0 \rangle^1} \qquad (\textit{R-Sig})$$

This rule demonstrates that the value of a signal is stored in a signal environment ψ_1. However, in the implementation, there is no need to separate ψ from S. The *R-Sig* rule is implemented as follows:

```
1     // R-Sig
2     protected Pair<State, Expression> apply(State state,
3                 Lifetime lifetime, Sig expression, int k) {
4         /** get the global lifetime */
5         Lifetime global = lifetime.getRoot();
6        /**
7          * initialise the counter of signal to 1 and its value to 0
8          */
9         Value.Integer v = new Value.Integer(0);
10        Pair<State, Value.Reference> pl = state.allocate(global,v,1);
11
12        State S2 = pl.first();
13        Value.Reference ls = pl.second();
14
15        /** Bind signal to location
16         * and bind signal to the lifetime
17         */
18        String s = expression.getVariable();
19        return reducedDeclare(S2,lifetime,s,ls);
20    }
```

Listing 6.11: Implementation of the *R-Sig* Rule.

Listing 6.11 illustrates the process of creating a signal. on Line 9, a value containing 0 is created, and on Line 10, a new location for the new signal is established in the program

store "S" by initialising the counter to 1. Alternatively, when encountering the expression "`Sig s`", it only necessitates verifying whether the signal "`s`" already exists in the $\mathscr{L}$ signal declaration context, as demonstrated by the *T-Sig* rule:

$$\frac{s \notin \mathscr{L}_1 \quad \mathscr{L}_2 = \mathscr{L}_1 \sqcup \{s_1\}}{\mathscr{L}_1, \Gamma_1 \vdash_\kappa \langle \texttt{Sig}\ s : \epsilon \rangle^1_\sigma \dashv \mathscr{L}_2, \Gamma_1} \quad (\textit{T-Sig})$$

The implementation of the *T-Sig* rule is straightforward and can be summarized as follows:

```
1     // T-Sig
2     protected Pair<Environment, Type> apply(Environment gam1,
3         Lifetime lifetime, Syntax.Expression.Sig expression, int k) {
4         String s = expression.getVariable();
5         Location ls = gam1.get(s);
6         // an exception is raised if the signal already exists.
7         if(ls!=null){
8             try {
9                 check(true, "Signal␣already␣declared␣");
10            } catch (ExceptionsMSG e) {
11                throw new RuntimeException(e);
12            }
13        }
14        // update
15        Environment gam2 = gam1.put(s,Type.Sig,lifetime);
16        // done
17        return new Pair<>(gam2, Type.Unit);
18    }
```

Listing 6.12: Implementation of the *T-Sig* Rule.

In the typing rules, we have introduced a dedicated environment for managing the declared signals. However, for simplification in the implementation, we directly store the signals in the `Environment`. In Listing 6.12, we retrieve the signal name and verify its existence in the `Environment` "`gam1`" on Line 9. If the signal does not exist, we create it in the "`gam2`" `Environment`.

To wait for a signal, MSSL provides the "`when`" expression. For this expression, three reduction rules are present, as explained in Chapter 5. Here, we focus on the rule where the signal is emitted (the other cases are straightforward), which is denoted by the *R-WhenTrue* rule:

$$\frac{\mathit{read}(S_1, s, \mathtt{1}) = \langle \ell^s_a \rangle^{\mathtt{m}} \quad \psi_1(\ell_a) = \langle 1 \rangle^{\mathtt{i}} \quad \langle \psi_1, S_1 \triangleright \{e_1\}^{\mathtt{m}} \rightarrow_i \psi_2, S_2 \triangleright \{e_2\}^{\mathtt{m}} \rangle^{\mathtt{l}}}{\langle \psi_1, S_1 \triangleright \mathtt{when}(s)\ \{e_1\}^{\mathtt{m}} \rightarrow_i \psi_2, S_2 \triangleright \mathtt{when}(s)\ \{e_2\}^{\mathtt{m}} \rangle^{\mathtt{l}}} \quad (\textit{R-WhenTrue})$$

As mentioned earlier, the *R-WhenTrue* rule retrieves the signal value to verify if it has already been emitted. In such a scenario, we execute the block of the "`when`" expression. The implementation of this rule in MSSL is as follows:

```
1          // R-WhenTrue and R-WhenFalse
2          protected Pair<State, Expression> apply(State state,
3                  Lifetime lifetime, When expression, int k) {
4          /** read the value of s **/
5          String s = expression.getVariable();
6          Path.Element[] es = new Path.Element[1];
7          es[0]=Path.DEREF_ELEMENT;
8          Lval w = new Lval(s, new Path(es));
9          /** determine if w exists **/
10         Value v = state.read(w, lifetime);
11        /** R-whenFalse **/
12         if(Integer.valueOf(v.toString()) == 0){
13             return new Pair<>(state, expression);
14         }
15        /** R-whenTrue **/
16        else{
17            Pair<State,Expression> S = apply(state,lifetime,
18                                  expression.getOperand(), k);
19            return new Pair<>(S.first(), S.second());
20         }
21
22     }
```

Listing 6.13: Implementation of the *R-WhenFalse* and the *R-WhenTrue* Rules.

As observed, once we retrieve the signal value, we examine whether the signal has already been emitted. If the signal has not been sent, on Line 12, we simply return the "`when`" expression itself. On the other hand, if the signal is already emitted, on Line 17, we execute the block of the "`when`" expression using the "`apply`" function.

Concerning the typing of the "`when`" expression, three conditions must be fulfilled: (1) The effect "κ" should be equal to 1, (2) the signal must have been previously declared, and (3) there should be no shared data borrowed in the current thread environment, as outlined below:

$$\frac{s \in \mathscr{L}_1 \quad \mathit{safeTrc}(\Gamma_1) \quad \mathscr{L}_1, \Gamma_1 \vdash_1 \langle \{e_1\}^{\mathtt{m}} : \tau \rangle^{\mathtt{l}}_{\sigma} \dashv \mathscr{L}_2, \Gamma_2}{\mathscr{L}_1, \Gamma_1 \vdash_1 \langle \mathtt{when}(s)\ \{e_1\}^{\mathtt{m}} : \tau \rangle^{\mathtt{l}}_{\sigma} \dashv \mathscr{L}_2, \Gamma_2} \quad (\textit{T-When})$$

Unlike the "`when`" expression, the "`watch`" expression defers the signal emission check until the end of the current instant. As depicted in the below rule, we directly execute the block of the "`watch`" expression:

$$\frac{\langle \psi_1, S_1 \triangleright \{e_1\}^{\mathtt{m}} \rightarrow_i \psi_2, S_2 \triangleright \{e_2\}^{\mathtt{m}} \rangle^{\mathtt{l}}}{\langle \psi_1, S_1 \triangleright \mathtt{watch}(s)\ \{e_1\}^{\mathtt{m}} \rightarrow_i \psi_2, S_2 \triangleright \mathtt{watch}(s)\ \{e_2\}^{\mathtt{m}} \rangle^{\mathtt{l}}} \quad (\textit{R-Watch})$$

The implementation of the *R-Watch* rule is straightforward and concise as follows:

```
        // R-Watch
        protected Pair<State, Expression> apply(State state,
                Lifetime lifetime, Watch expression, int k) {
  /** determine whether the signal given as a parameter exists **/
        Pair<State,Expression> S = apply(state,lifetime,
                              expression.getOperand(), k);
        return new Pair<>(S.first(), S.second());
    }
```

Listing 6.14: Implementation of the *R-Watch* Rule.

To type the "watch" expression, as illustrated by the *T-Watch* rule, it is necessary that the signal "s" already exists in the context $\mathscr{L}_1$:

$$\frac{s \in \mathscr{L}_1 \quad \mathscr{L}_1, \Gamma_1 \vdash_1 \langle \{e_1\}^{\mathtt{m}} : \tau \rangle_{\sigma}^{\mathtt{l}} \dashv \mathscr{L}_2, \Gamma_2}{\mathscr{L}_1, \Gamma_1 \vdash_{\kappa} \langle \mathtt{watch}(s)\ \{e_1\}^{\mathtt{m}} : \tau \rangle_{\sigma}^{\mathtt{l}} \dashv \mathscr{L}_2, \Gamma_2} \quad (\textit{T-Watch})$$

The implementation of the *T-Watch* rule is as follows:

```
    // T-Watch
    protected Pair<Environment, Type> apply(Environment gam,
    Lifetime lifetime, Syntax.Expression.Watch expression, int k) {
        // retrieve the name of signal
        String s = expression.getVariable();
        Location ls = gam.get(s);
        // check if the signal already exists.
            ....
        if(ls==null){
            try {
                check(true, "The signal "+ s+" does not exist ");
            } catch (ExceptionsMSG e) {
                throw new RuntimeException(e);
            }
        }
        // done
        return apply(gam, lifetime, expression.getOperand(), k);
    }
```

Listing 6.15: Implementation of the *T-Watch* Rule.

In Listing 6.15, we simply check whether the signal exists and then proceed to type block of the "watch" expression. Following a detailed explanation of several extensions introduced in Chapter 5, this section concludes with a reactive example that incorporates these extensions. For further insights, please refer to our implementation on GitHub.

```
    // th1
   fn f1(mut x : trc<int>; s1,s2)->1 unit{
```

```
        watch(s2){
            when(s1){
                *x=1;
            }
        cooperate;
        let mut y = x.clone;
        spawn(f3(box(y); s1,s2));
        emit(s1);
        }
    }
    // th2
    fn f2(mut x : trc<int>; s1,s2)->1 unit{
        let mut y = 1;
        if(&*x==&y){
            emit(s2);
        }else{
            emit(s1);
            cooperate;
        }
    }
    // th3
    fn f3(mut x :box<trc<int>>; s1,s2)->1 unit{
        watch(s1){
            when(s2){
                let mut a = &mut *x;
                **a = 1;
                cooperate;
            }
        }
    }

    // main
    {
        let mut x = trc(0);
        Sig s1; Sig s2;
        spawn(f1(x.clone; s1,s2));
        spawn(f2(x.clone; s1,s2));
    }
```

Listing 6.16: A Reactive Example illustrated in MSSL.

The example presented in Listing 6.16 is accepted by our type system. During the execution of this example, we initiate two threads at the beginning, with the first thread executing

"f1" and the second executing "f2". We suppose that the first thread assumes control and executes the "watch" expression block on Line 3. Subsequently, this thread checks whether signal "s1" is being emitted, which is not the case. Then it cooperates, and the second thread takes over. The second thread examines whether the content of "x" is 1 (which it is not), and executes the "else" branch, where it emits the "s1" signal and cooperates. According to the reduction rules, the current instant is not over, and the scheduler initiates a new cycle without resetting the signals. Consequently, the first thread resumes execution at line 4, and since the signal is emitted, the thread continues its execution. As for the second thread, in this new cycle, when it resumes execution at line 20, it completes its execution, and so forth.

1.3 MSSL to Fairthreads

As previously discussed, once the MSSL type system accepts an MSSL program, it is translated into Fairthreads. This section provides an overview of the translation from MSSL to Fairthreads.

Let us examine the example presented in Listing 6.16, which represents a program accepted by the MSSL system. Once this example has been accepted, the translation from MSSL to C is automatically performed by invoking the Fairthreads library. According to this example, the syntax of Fairthreads (see examples in Chapter 2) is as follows:

- `watch(s2){...}` will be converted to `ft_thread_set_event_watch(s2) {...}`
- `when(s1)` will be rendered as `ft_thread_when_event(s1)`
- `cooperate` will result in `ft_thread_cooperate()`
- `spawn(...)` will be transformed into `ft_thread_create(...)`
- `emit(s1)` will be translated as `ft_thread_generate(s1)`

Furthermore, at the end of each block, the `free` function will be added to facilitate automatic memory deallocation corresponding to the data type.

Chapter 7

Conclusion

The best way to get a better answer is to start asking a better question

– Anthony Robbins

Synchronous reactive programming is a programming paradigm grounded in logical instants, enabling the synchronisation of program threads and signal emissions and receptions at regular intervals. This greatly streamlines the development of programs engaging extensively with external environments. Typically, synchronous reactive programs are compiled into real-time programs scripted in lower-level abstraction languages, particularly C. Consequently, they encounter analogous challenges, with a key concern revolving around memory safety: how to guarantee that distinct lightweight processes refrain from making concurrent accesses to the same shared data at the same logical instant?

Memory safety is a well-known problem in programming, for which numerous solutions have been proposed in the literature. In a multi-process context, a classic solution involves using a mutual exclusion mechanism (mutex), which must be used judiciously by the programmer to avoid incorrect concurrent memory accesses or deadlocks. Another solution is to employ a type system that statically guarantees that a program does not perform illegal memory operations, such as accessing a deallocated memory block. The complex type system of the Rust language contributes to its growing popularity by ensuring memory safety while enabling the writing of low-level programs compiled into highly efficient code.

The original approach of this thesis involves proposing a new language called MSSL (Memory Safe Synchronous Language), which adapts and extends Rust's type system for synchronous reactive programs. The chosen reactive model employs lightweight processes executing in a round-robin fashion, in a collaborative manner. This means that each process decides when to yield control to the scheduler using a specific language instruction. This language ensures the validity of memory accesses performed by different processes without the need for developers to use mutexes.

To substantiate this assertion, we concentrated on extending FR [73] with new extensions tailored for cooperative concurrent programming. Consequently, we presented MSSL, a memory-safe synchronous language that operates as a reactive programming language based on cooperative threads and synchronous execution. MSSL enables data sharing among multiple cooperative threads while ensuring memory integrity. This outcome is attained by introducing a new abstraction called `Trc`, which combines the ownership safety of Rust references with the reference counting mechanism of Rust smart pointers.

MSSL draws inspiration from the family of synchronous reactive languages (e.g., Esterel, Reactive-ML, Reactive-C, Fairthreads) and incorporates elements from the popular Rust language. In MSSL, threads execute in a round-robin manner under the notion of logical time, called "instants," ensuring synchronous progress with a consistent view of signal states at each instant. Notably, MSSL's type system ensures safe memory management between threads without the need for locking primitives or garbage collection mechanisms. The core of MSSL is FR_{FT}, a calculus language. Compared to the FR language, FR_{FT} introduces a new type of smart pointers called `Trc`, enabling threads to communicate without requiring locking primitives. Since thread lifetimes cannot be statically inferred, our contribution focuses on reference counting, allowing us to bypass lexical scope constraints and to manage memory safely. Additionally, FR_{FT} models key aspects of Rust's type system, such as ownership, borrowing, lifetimes, copy and move semantics, as well as dynamic allocation via `Box`. To ensure memory safety for well-typed programs, we provided an operational semantics and a type system for MSSL. Our type, borrowing, and concurrency safety theorem guarantees that, from an initial state, the execution of an FR_{FT} program can be an infinite sequence of logical instances while preserving the well-typed state of each thread, memory safety between threads, the borrowing invariants specified by Rust, and the *uniqueness* of `Trc`. Building on FR_{FT}, we introduced several reactive extensions like `cooperate` for explicit cooperation, `when` for implicit cooperation, `watch`, and the concept of signals. Furthermore, we proposed an implementation of the type system for the complete MSSL syntax and a translation for well-typed programs into Fairthreads. In this context, we modified the behavior of the `cooperate` expression and added the when and watch expressions (representing weak preemption), which were not previously available in Fairthreads [26].

Future Work

Our future work encompasses further experiments and proposed research directions aimed at validating the results and gaining a more comprehensive understanding of the topic. The thesis focuses on two main aspects: memory safety between threads and the reactive synchronous model. For each aspect, we envision several directions to build upon the groundwork laid by this thesis.

In the first aspect, our objective is to demonstrate the ability to statically preserve memory safety between cooperating threads. Having achieved this goal, an intriguing next step would be to provide a Rust crate based on our results. This extension will require the use of Rust's unsafe feature. To deal with the Rust type system, cloning and sharing `Trc`, could be achieved through pack and unpack primitives. We intend to ensure the preservation of

safety by leveraging the techniques outlined in the RustBelt project [52]. Moreover, drawing inspiration from the effort in [36], an ownership-based language analyzing information flow in Rust, we could explore function invocation and further enrich our type system. On the other hand, MSSL is considerably smaller than full Rust, leaving room for introducing missing features like data structures, traits, pattern matching, immutable variables, etc. Another track for future work involves the implementation of NLL (Non-Lexical Lifetimes). As we have previously discussed, MSSL closely aligns with Rust version 1.39.0, where lifetimes are based on the LL (lexical lifetime) concept. To enhance our type system and accommodate more programs, we envision expanding it based on the NLL concept. The NLL approach treats a reference type such as ($\&^{\prime}\mathtt{a}\ \omega$) as a pair of values: a lifetime (denoted by `'a`) and a referent type ω. This type is interpreted as "ω *which lives for at least* `'a`". In other words, in the new version of Rust, a lifetime in NLL represents a set of lines of code for which a value lives (as a loan). Conversely, Polonius [78] introduces a different view of loans, reversing the relationship between lives and loans. According to this perspective, a lifetime is now referred to as a "*provenance*" of the loan.

The implementation section discussed in Chapter 6 requires further refinement. Thus, it's necessary to establish metrics to evaluate the implementation effort, particularly in terms of processing time. Additionally, incorporating real-world examples written in MSSL is essential to assess the language's capabilities and limitations. This process will enable a comprehensive exploration of the primary design and implementation decisions.

Furthermore, MSSL incorporates a reactive aspect that enhances the Fairthreads programming model. In addition to this, MSSL's threads are designed to be cooperative, but it also proves beneficial to introduce the concept of preemptive threads. The objective is to combine cooperative and preemptive threads while ensuring memory safety between them. The approach involves identifying in advance which threads share the same memory region and which do not. As a result, threads that share the same memory region will be executed cooperatively, while those that operate on distinct memory regions will run in parallel. This approach significantly boosts MSSL's efficiency, usefulness, and applicability across a wide range of applications.

Bibliography

[1] Amal Ahmed, Matthew Fluet, and Greg Morrisett. *L3: A Linear Language with Locations*. July 2004. DOI: 10.1007/b135690.

[2] Amal Ahmed et al. "Semantic Foundations for Typed Assembly Languages". In: (). DOI: 10.1145/1709093.1709094.

[3] Amal Jamil Ahmed. "Semantics of Types for Mutable State". PhD thesis.

[4] Roberto M. Amadio. "A synchronous pi-calculus". In: *Inf. Comput.* 205.9 (2007), pp. 1470–1490.

[5] Roberto M. Amadio. "The SL synchronous language, revisited". In: *J. Log. Algebraic Methods Program.* (2007).

[6] Roberto M. Amadio and Frédéric Dabrowski. "Feasible Reactivity for Synchronous Cooperative Threads". In: *Proceedings of the 12th Workshop on Expressiveness on Concurrency*.

[7] Roberto M. Amadio and Frédéric Dabrowski. "Feasible reactivity in a synchronous Pi-calculus". In: *Proceedings of the 9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 2007*.

[8] Pascalin Amagbegnon, Loc Besnard, and Paul Guernic. "Implementation of the Data-flow Synchronous Language SIGNAL". In: *ACM SIGPLAN Notices* (1998). DOI: 10.1145/223428.207134.

[9] Ch Andre. "Syncharts: A visual representation of reactive behaviors". In: (Oct. 2022).

[10] Charles André. "Semantics of S.S.M. (Safe State Machine)". In: (Jan. 2003).

[11] *ANTLR*. URL: https://www.antlr.org/.

[12] V. Astrauskas et al. "The Prusti Project: Formal Verification for Rust (invited)". In: *NASA Formal Methods (14th International Symposium)*. Springer, 2022.

[13] Todd Austin, Scott Breach, and Gurindar Sohi. "Efficient Detection of All Pointer and Array Access Errors". In: (2002). DOI: 10.1145/773473.178446.

[14] Henry Baker. "Use-Once Variables and Linear Objects - Storage Management, Reflection and Multi-Threading." In: (1995).

[15] Henry G. Baker. "Minimizing Reference Count Updating with Deferred and Anchored Pointers for Functional Data Structures". In: (1994). DOI: 10.1145/185009.185016.

[16] Sergio Benitez. "Short Paper: Rusty Types for Solid Safety". In: Oct. 2016, pp. 69–75. DOI: 10.1145/2993600.2993604.

[17] Albert Benveniste and Gérard Berry. "The Synchronous Approach to Reactive and Real-Time Systems". In: (1991). DOI: 10.1109/5.97297.

[18] Albert Benveniste and Paul Guernic. “Hybrid dynamical systems theory and the language "SIGNAL"”. In: *Automatic Control, IEEE Transactions on* (1990). DOI: 10.1109/9.53519.

[19] Albert Benveniste, Paul Le Guernic, and C. Jacquemot. “Programming with events and relations: The SIGNAL language and its semantics”. In: (1991).

[20] Caspi P. Edwards Stephen Halbwachs Nicolas Guernic P. Simone Benveniste Albert and Robert. “The Synchronous Languages 12 Years Later”. In: (2003). DOI: 10.1109/JPROC.2002.805826.

[21] Gérard Berry. “The Constructive Semantics of Pure Esterel”. In: (June 1996).

[22] Gérard Berry, Philippe Couronne, and Georges Gonthier. “Synchronous programming of reactive systems: an introduction to ESTEREL”. In: (1988).

[23] Gérard Berry and Georges Gonthier. “The Esterel Synchronous Programming Language: Design, Semantics, Implementation”. In: (1988). DOI: 10.1016/0167-6423(92)90005-V.

[24] Timothy Bourke and Marc Pouzet. “Zélus: A Synchronous Language with ODEs”. In: 2013. DOI: 10.1145/2461328.2461348.

[25] F. Boussinot and J-F. Susini. ““Java Threads and SugarCubes””. In: *Software Practice and Experience* (2000.).

[26] Frédéric Boussinot. “FairThreads: Mixing cooperative and preemptive threads in C”. In: *Concurrency and Computation: Practice and Experience* 18 (Apr. 2006), pp. 445–469. DOI: 10.1002/cpe.919.

[27] Frédéric Boussinot. “Reactive C: An Extension of C to Program Reactive Systems.” In: (1991). DOI: 10.1002/spe.4380210406.

[28] Frédéric Boussinot and Robert Simone. “The Esterel language”. In: (1991). DOI: 10.1109/5.97299.

[29] Frédéric Boussinot and Robert Simone. “The SL Synchronous Language”. In: *Software Engineering, IEEE Transactions on* (1996). DOI: 10.1109/32.491649.

[30] John Boyland. “Alias burying: Unique variables without destructive reads”. In: (2001). DOI: 10.1002/spe.370.

[31] Juan Caballero et al. “Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities”. In: (2012). DOI: 10.1145/2338965.2336769.

[32] Paul Caspi. “Lucid Synchrone”. In: *Proc. OPOPAC* (Jan. 1993).

[33] Dave Clarke, John Potter, and James Noble. “Ownership Types for Flexible Alias Protection”. In: (1998). DOI: 10.1145/286942.286947.

[34] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. “SCADE 6: A formal language for embedded critical software development”. In: 2017. DOI: 10.1109/TASE.2017.8285623.

[35] Karl Crary, David Walker, and Greg Morrisett. “Typed Memory Management in a Calculus of Capabilities”. In: (1999).

[36] Marco Agrawala Maneesh Hanrahan Crichton Will Patrignani and Pat. “Modular Information Flow through Ownership”. In: (2022). DOI: 10.1145/3519939.3523445.

[37] Trevor Jim Michael Hicks Yanling Wang Dan Grossman Greg Morrisett and James Cheney. “Region-Based Memory Management in Cyclone.” In: *In ACM SIGPLAN Conference on Programming Language Design and Implementation* (2002).

[38] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. “Creusot: A Foundry For The Deductive Verification Of Rust Programs”. In: Springer-Verlag, 2022.

[39] Igor Dobrovitski. “Exploit for CVS double free() for Linux pserver”. In: (June 2022).

[40] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. “Statically detecting use after free on binary code”. In: *Journal of Computer Virology and Hacking Techniques* (2014). DOI: 10.1007/s11416-014-0203-1.

[41] Matthew Fluet, Greg Morrisett, and Amal Ahmed. “Linear Regions Are All You Need”. In: 2006. DOI: 10.1007/11693024_2.

[42] Georges Gonthier. *Sémantique et modèles d’exécution des langages réactifs synchrones : application à Esterel*. PhD thesis, Université d’Orsay, Paris, France. 1988.

[43] Caspi P. Raymond Halbwachs Nicolas and Pascal Pilaud. “The synchronous dataflow programming language LUSTRE”. In: (2000).

[44] Nicolas Halbwachs. “A Synchronous Language at Work: The Story of Lustre”. In: 2005. DOI: 10.1109/MEMCOD.2005.1487884.

[45] Nicolas Halbwachs. “Synchronous Programming of Reactive Systems”. In: 1427 (Dec. 1999). DOI: 10.1007/BFb0028726.

[46] Björn Motika Christian Smyth Steven Mendler Michael Aguado Joaquín Loftus-Mercer Stephen O’Brien von Hanxleden Reinhard Duderstadt and Owen. “SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications”. In: 2014. DOI: 10.1145/2594291.2594310.

[47] David Harel and A. Pnueli. “On the Development of Reactive Systems”. In: (1989). DOI: 10.1007/978-3-642-82453-1_17.

[48] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. “Featherweight Java - A Minimal Core Calculus for Java and GJ”. In: (1999). DOI: 10.1145/320384.320395.

[49] *Implications of Rewriting a Browser Component in Rust – Mozilla Hacks - the Web developer blog*. URL: https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust.

[50] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. 1996.

[51] Hoang-Hai Kang Jeehoon Dreyer Jung Ralf Dang and Derek. “Stacked Borrows: An Aliasing Model for Rust”. In: (2021). DOI: 10.1145/3371109.

[52] Jacques-Henri Krebbers Robbert Dreyer Jung Ralf Jourdan and Derek. “RustBelt: Securing the Foundations of the Rust Programming Language”. In: (2017). DOI: 10.1145/3158154.

[53] Jacques-Henri Krebbers Robbert Dreyer Jung Ralf Jourdan and Derek. “Safe Systems Programming in Rust”. In: (2021). DOI: 10.1145/3418295.

[54] Gilles Kahn. “The Semantics of Simple Language for Parallel Programming.” In: 1974.

[55] David Lin Shang-Wei Liu Kan Shuanglong Sanán and Yang. “K-Rust: An Executable Formal Semantics for Rust”. In: (2018).

[56] Mats Kindahl. “Review of "Types and Programming Languages by Benjamin C. Pierce", MIT Press, 2002.” In: (2006).

[57] Jagun Kwon, Andy Wellings, and Steve King. “Ravenscar-Java: A High Integrity Profile for Real-Time Java”. In: (2003). DOI: 10.1145/583810.583825.

[58] Yves Lafont. “The Linear Abstract Machine.” In: (1988). DOI: 10.1016/0304-3975(88)90100-4.

[59] Travis Cho Chanhee Brun Matthias Subasinghe Isitha Zhou Yi Howell Jon Parno Bryan Hawblitzel Lattuada Andrea Hance and Chris. “Verus: Verifying Rust Programs Using Linear Ghost Types”. In: (2023). DOI: 10.1145/3586037.

[60] Thierry Borgne Michel Maire Claude Le Guernic Paul Gautier. “Programming real-time applications with SIGNAL”. In: (1991). DOI: 10.1109/5.97301.

[61] Nico Lehmann et al. *Flux: Liquid Types for Rust*. July 2022. DOI: 10.48550/arXiv.2207.04034.

[62] F. Boussinot M. Serrano and B. Serpette. ““Scheme Fair Threads.””. In: (2004).

[63] Louis Mandel and Marc Pouzet. “ReactiveML, a reactive extension to ML”. In: 2005. DOI: 10.1145/1069774.1069782.

[64] F. Maraninchi. “The Argos Language: Graphical Representation of Automata and Description of Reactive Systems”. In: (1992).

[65] Nicholas D. Matsakis. *An alias-based formulation of the borrow checker(2018)*. URL: http://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/.

[66] Nicholas D. Matsakis. *Introducing MIR*. 2016. URL: https://blog.rust-lang.org/2016/04/19/MIR.html..

[67] Nicholas D. Matsakis. “Non-lexical lifetimes: introduction”. In: 2016.

[68] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. “RustHorn: CHC-Based Verification for Rust Programs”. In: (2021). DOI: 10.1145/3462205.

[69] *Meet Safe and Unsafe*. https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html.

[70] Robin Milner. "A Theory of Type Polymorphism in Programming". In: (1978).

[71] Vitek J. Potter J. Noble J. "Flexible Alias Protection." In: *LNCS, vol. 1445, pp. 158–185. Springer, Heidelberg (1998)* (1998).

[72] N. Halbwachs P. Caspi D. Pilaud and J. Plaice. "Lustre : a declarative language for programming synchronous systems." In: (1987.).

[73] David Pearce. "A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust". In: (2021). DOI: 10.1145/3443420.

[74] J. Peng, M. Zhang, and Q. Wang. "Deduplication and Exploitability Determination of UAF Vulnerability Samples by Fast Clustering". In: (2016). DOI: 10.3837/tiis.2016.10.016.

[75] Benjamin Pierce. "Advanced Topics in Types and Programming Languages". In: (Jan. 2004).

[76] A. Pnueli. "Logics and Models of Concurrent Systems". In: (1984).

[77] "Pointer analysis". In: (). DOI: https://en.wikipedia.org/wiki/Pointer_analysis.

[78] *Polonius and the case of the hereditary harrop predicate*. URL: http://smallcultfollowing.com/babysteps/blog/2019/01/21/hereditary-harrop-region-constraints/.

[79] Alex Potanin et al. "A High Integrity Profile for Memory Safe Programming in Real-time Java". In: (Jan. 2005).

[80] M. Pouzet. "Lucid Synchrone: tutorial and reference manual". In: (Oct. 2022).

[81] R. Pucella. "Reactive Programming in Standard ML". In: *Proceedings of the IEEE International Conference on Computer Languages* (1998.).

[82] Eric Reed. "Patina: A Formalization of the Rust Programming Language. Separation Logic: A Logic for Shared Mutable Data Structures." In: (2015).

[83] G. Roşu and T. F. Şerbănuţă. "An overview of the k semantic framework." In: (2010.).

[84] *Rust Programming Language*. https://www.rust-lang.org/.

[85] J.A. Stankovic. "Real-time and embedded systems". In: Jan. 2004, pp. 83–1.

[86] Evgeniy Stepanov and Konstantin Serebryany. "MemorySanitizer: Fast detector of uninitialized memory use in C++". In: 2015. DOI: 10.1109/CGO.2015.7054186.

[87] J.-P. Talpin and P. Jouvelot. "The type and effect discipline". In: 1992. DOI: 10.1109/LICS.1992.185530.

[88] Jean-pierre Talpin and Pierre Jouvelot. "Polymorphic Type, Region and Effect Inference". In: (1998).

[89] MSRC Team. *We need a safer systems programming language – Microsoft Security Response Center*. URL: https://msrc-blog.microsoft.com/2019/07/18/we-need-a-safer-systems-programming-language/.

[90] Mark Theng Ralf Jung M. Frans Kaashoek Tej Chajed Joseph Tassarotti and Nickolai Zeldovich. "GoJournal: a verified, concurrent, crash-safe journaling system". In: *15th USENIX Symposium on Operating Systems Design and Implementation)*. 2021.

[91] "The Coq team. 2017". In: *The Coq proof assistant*.

[92] Gavin Thomas. "A proactive approach to more secure code". In: (2019).

[93] Mads Tofte and Jean-Pierre Talpin. "Region-Based Memory Management". In: (1997). DOI: 10.1006/inco.1996.2613.

[94] Jesse Tov and Riccardo Pucella. "Practical Affine Types". In: vol. 46. Jan. 2011. DOI: 10.1145/1926385.1926436.

[95] D. Walker and K. Watkins. "On regions and linear types". In: (2001). DOI: 10.1145/507669.507658.

[96] Fu Zhang Min Zhu Xiaoran Zhang Jun Wang Feng Song. "KRust: A Formal Executable Semantics of Rust". In: 2018. DOI: 10.1109/TASE.2018.00014.

[97] Daniel Matsakis Nicholas Ahmed Amal Weiss Aaron Patterson. *Oxide: The Essence of Rust*. Mar. 2019.

[98] Paul Wilson. "Uniprocessor Garbage Collection Techniques". In: 2006. DOI: 10.1007/BFb0017182.

[99] A.K. Wright and Matthias Felleisen. "A Syntactic Approach to Type Soundness". In: (1994). DOI: 10.1006/inco.1994.1093.

[100] Alain Frisch Jacques Garrigue Didier Rémy Xavier Leroy Damien Doligez and Jérôme Vouillon. *The OCaml system*. 2014. URL: https://college-de-france.hal.science/hal-00930213/.

[101] Dang Hoang-Hai Jung Ralf Dreyer Yanovski Joshua and Derek. "GhostCell: Separating Permissions from Data in Rust". In: (2021). DOI: 10.1145/3473597.

Résumé de la thèse

Les langages réactifs synchrones sont un excellent choix pour la programmation de l'internet des objets (IoT), en raison de leur sémantique claire concernant l'interaction entre le système et l'environnement. Cependant, dans le contexte de systèmes nécessitant une sécurité rigoureuse ou opérant avec des ressources limitées, les nouvelles propositions telles que Fairthreads ou ReactiveML font face à un défi bien reconnu. L'approche de gestion manuelle de la mémoire, comme celle adoptée par Fairthreads, peut conduire à des erreurs, tandis que l'utilisation du ramasse-miettes (comme dans le cas de ReactiveML) garantit la sécurité de la mémoire, mais au prix d'une surcharge de performance. Pour résoudre ces problèmes, nous repensons l'approche de FairThreads en introduisant des éléments issus du système de types de Rust. Cette adaptation assure ainsi la sécurité de la mémoire sans engendrer de surcharge d'exécution. Notre proposition introduit un nouveau type de pointeur intelligent pour le partage des données. Ces pointeurs permettent un accès sécurisé sans nécessiter le recours à une gestion de verrouillage, les rendant ainsi parfaitement compatibles avec l'ordonnancement coopératif propre à Fairthreads.

Dans les langages de programmation synchrones, un ensemble de comportements est exécuté par cycles successifs appelés "instants". À chaque instant, ces comportements réagissent à des signaux, pouvant contenir des valeurs, générées soit par l'environnement, soit par les comportements eux-mêmes. Un instant se termine lorsque toutes les réactions sont achevées, ce qui résulte en une sortie en réponse à l'environnement. Au début de chaque instant, l'état des signaux est réinitialisé et ils reçoivent de nouvelles entrées provenant de l'environnement. Dans les langages synchrones tels que Esterel, les réactions à l'absence d'un signal peuvent se produire instantanément. Toutefois, ces réactions peuvent engendrer des comportements non causaux, qui sont identifiés et écartés par des vérifications statiques. Le compilateur veille également à ce que les programmes soient réactifs ou productifs. Néanmoins, ces contrôles dépendent grandement de contraintes strictes au sein du langage, telles que l'absence de manipulation de données dynamiques et l'ordonnancement statique. Fairthreads, qui repose sur le langage SL, adopte une approche différente en différant les réactions à l'absence d'un signal jusqu'au prochain instant. Cette méthode contourne les problèmes de causalité et permet l'utilisation de données dynamiques ainsi que la création de threads. Des solutions basées sur des méthodes formelles ont été élaborées pour garantir la productivité. Toutefois, ces solutions supposent l'absence d'erreurs de mémoire.

Dans cette thèse, nous revisitons Fairthreads en introduisant un système de type similaire à celui du language Rust dans un langage expérimental appelé MSSL (Memory Safe Synchronous Language). MSSL est un langage de programmation réactif qui se fonde sur des threads coopératifs et une exécution synchrone. Dans ce modèle de programmation, les threads s'exécutent de manière séquentielle, et le changement de contrôle entre eux est effectué sur demande à l'aide d'une commande "yield". L'introduction d'une notion de signal permet aux threads de se synchroniser. Par l'intermédiaire d'une unité temporelle appelée "instant", les threads progressent simultanément avec une vision uniforme de l'état des signaux (présents ou absents) à chaque instant. Du fait de l'utilisation de l'ordonnancement coopératif, MSSL ne nécessite pas l'utilisation de mécanismes de verrouillage. Cependant,

bien que l'exécution des threads soit coopérative, l'incorporation du système de types de Rust demande des ajustements pour permettre le partage de données entre les threads, car cela introduit une nouvelle forme d'aliasing qui doit être contrôlée. Le concept de comptage de références, comme le propose Rust tels que `Rc` et `Arc`, se présente comme une solution prometteuse, car il permet de dépasser les limites de portée lexicale. Cela s'avère nécessaire pour autoriser le partage de données entre différents threads. Néanmoins, cette approche est limitée par le fait que ces références sont en "read-only". Supprimer cette restriction remettrait en cause la sécurité de la mémoire au niveau des threads. Pour résoudre ce problème, nous proposons une combinaison entre l'approche du comptage de références et les contraintes d'aliasing des références standards de Rust. Cette contribution majeure de notre travail se matérialise sous la forme d'un nouveau type de pointeur intelligent appelé `Trc` (Thread Reference Counting), spécifiquement conçu pour gérer le multithreading coopératif. À l'instar des pointeurs `Rc` de Rust, les pointeurs `Trc` reposent sur un mécanisme de comptage de références pour leur gestion. De manière intuitive, les pointeurs `Trc` encapsulent des données partagées, rappelant en quelque sorte les `Mutexes` de Rust. Cependant, contrairement aux `Mutexes`, les données à l'intérieur des pointeurs `Trc` ne nécessitent pas d'être verrouillées à chaque accès. L'une des distinctions clés entre les pointeurs `Trc` et les pointeurs `Rc` de Rust réside dans la mutabilité de leur contenu. Notre système de types interdit: (1) L'aliasing d'un pointeur `Trc` par des threads uniques et (2) la détention d'une référence au contenu d'un pointeur `Trc` aux moments de coopération.

Aperçu de Rust

Rust est un langage de programmation à typage statique axé sur la performance, la fiabilité et la sécurité. Il utilise un système de types robuste basé sur les principes de propriété et d'emprunt pour contrôler efficacement l'aliasing des données, en utilisant les concepts fondamentaux de la sémantique de "copy" et de "move", ainsi que de la mutation des données.

Rust gère également la gestion de la mémoire sans nécessiter de ramasse-miettes en associant chaque variable à une durée de vie et en insérant automatiquement des points de désallocation à la compilation. Le système de types protège contre les "dangling pointers" en suivant l'aliasing et en facilitant la désallocation automatique. En incorporant ces fonctionnalités, Rust vise à fournir les performances et la flexibilité du langage C tout en traitant les erreurs de gestion de mémoire. Cependant, une application stricte de ce système de types est trop restrictive. C'est là que la pragmatisme de Rust intervient. Des bibliothèques peuvent être développées dans lesquelles le système de types est désactivé en utilisant le mot-clé "`unsafe`". De nombreuses fonctionnalités de Rust, notamment `Rc`, `Arc` et `Mutex`, utilisent ce mécanisme. `Rc` et `Arc` sont des types de référence en lecture seule spéciaux qui permettent de partager la propriété de valeurs allouées sur le tas en utilisant le comptage de références. Cette approche est avantageuse en termes de partage de données entre les threads, franchissant les limites de la portée lexicale. Étant donné que ces références sont en lecture seule, leur contenu ne peut pas être modifié. Avec Rust, les `Mutex` sont utilisés pour muter des données partagées entre les threads. Cependant, dans les scénarios impliquant des threads coopératifs, la protection des données devient inutile, rendant l'utilisation des `Mutex` non recommandée. Pour résoudre ce problème, nous pro-

posons de combiner l'approche du comptage de références avec les contraintes d'aliasing de référence standard de Rust. Notre contribution introduit `Trc`, un nouveau type de pointeur intelligent spécialement conçu pour faciliter le partage de données entre des threads synchrones coopératifs.

En outre, cette thèse a apporté une contribution significative au développement de MSSL en tant que langage réactif synchrone, en tirant profit des aspects de sécurité de Rust. Les travaux exposés dans cette thèse sont organisés comme suit :

1. La première partie de la thèse se concentre sur la mise en évidence de FR_{FT}, un noyau coopératif de MSSL dépourvu d'opérations de synchronisation et qui étend FR [73]. Comparativement à FR, FR_{FT} introduit de nouvelles constructions, notamment deux instructions de multi-threading (création de threads et coopération explicite), ainsi qu'un nouveau type de pointeurs intelligents nommé `Trc` (Thread Reference Counting) conçu spécifiquement pour la communication entre les threads. Intuitivement, toutes les données partagées doivent être encapsulées dans un `Trc`. Le système de types garantira cette propriété et protégera de tels données contre toute corruption concurrente. Concevoir ce nouveau type de pointeurs intelligents a représenté un défi, car il devait permettre à la fois le partage et la mutabilité sans nécessiter une discipline de verrouillage. Plus précisément, `Trc` (1) autorise le partage entre les threads, (2) garantit l'unicité par thread des `Trc` et (3) assure qu'au moment de la coopération, les threads ne possèdent pas de références vers les données partagées. Les pointeurs `Trc` sont divisés en deux catégories distinctes : les `Trc` actifs, qui servent de points d'accès à la partie partagée du tas, et les `Trc` inactifs, qui sont des copies de `Trc` actifs destinées à être communiquées à d'autres threads. Lorsqu'un `Trc` inactif est communiqué à un autre thread, il devient actif. Seuls les `Trc` actifs peuvent être accédés. Cette partie de la thèse comprend également une description détaillée de la sémantique opérationnelle du langage, expliquant comment il fonctionne en pratique.

2. La deuxième partie de la thèse aborde le système de types de FR_{FT}, qui vise à garantir la sécurité du typage et de l'emprunt entre les threads. Ce système de types vise à assurer trois propriétés essentielles pour les programmes correctement typés :

   - La validité des références à chaque utilisation.
   - L'unicité des `Trc`.
   - La prévention de la corruption des données dans la mémoire des threads lors du partage de données entre eux.

   Parallèlement à Rust, nous introduisons des sémantiques de "copy" et de "move", incluant des références mutables et immutables, des "reborrowing", des durées de vie, ainsi qu'une allocation dynamique via Box, qui permet des déplacements partiels. Le système de types que nous avons développé est en mesure de prendre en compte tous ces aspects. De plus, dans cette partie, nous démontrons la solidité du système de types FR_{FT}. Cette solidité garantit que les programmes correctement typés ne se terminent pas par un plantage. En d'autres termes, ils atteignent soit un état final où toutes les expressions de thread deviennent des valeurs, soit ils produisent un nombre infini d'instants. Plus précisement, nous prouvons que les programmes FR_{FT} bien

types, qui permettent le partage de données entre des threads, s'exécutent en toute sécurité sans rencontrer de problèmes. De plus, nous revisitons le concept d'instant logique, qui indique que l'exécution d'un programme sûr peut être vue comme une séquence infinie d'instants tout en préservant la sécurité de la mémoire entre les threads. Cette notion garantit qu'un thread, lors de son exécution, maintient son type afin d'empêcher toute référence incorrecte. En outre, son exécution préserve le type des autres threads sans générer de données indésirables dans leurs espaces mémoire. Tout cela est réalisé tout en respectant les règles définies pour `Trc`, les durées de vie et les emprunts.

3. La troisième partie de la thèse présente la syntaxe complète de MSSL, tout en l'étendant au-delà de FR_{FT} en y ajoutant des extensions à la fois réactives et non réactives. Nous enrichissons considérablement MSSL en introduisant la notion de signaux, qui se révèlent être des éléments puissants, flexibles, et dotés de capacités de communication très pertinentes. À ce stade, les threads ont la possibilité de créer des signaux, émettre des signaux et attendre des signaux. De plus, nous offrons une explication détaillée de la manière dont l'invocation de méthode peut être implémentée dans ce contexte.

4. La dernière partie de la thèse se concentre sur l'implémentation complète du langage MSSL en Java. L'accent est mis sur la gestion de `Trc` et des emprunts, tout en intégrant d'autres extensions de MSSL. Cette mise en oeuvre respecte les règles sémantiques et de typage exposées dans la thèse, offrant ainsi une compréhension approfondie de la manière dont la solidité de nos règles se vérifie dans une implémentation concrète. Cette implémentation est réalisée en Java et est disponible gratuitement sur GitHub.

Darine RAMMAL

Sécurité de la mémoire pour la programmation réactive synchrone

Résumé:
Les langages réactifs synchrones constituent un excellent choix pour la programmation de l'IoT en raison de leur sémantique claire pour l'interaction entre le système et l'environnement. Cependant, en ce qui concerne les systèmes critiques en termes de sécurité ou contraints en ressources, des propositions récentes telles que Fairthreads ou ReactiveML font face à un problème bien connu. La gestion manuelle de la mémoire de Fairthreads peut entraîner des erreurs, tandis que la collecte des déchets de ReactiveML assure la sécurité de la mémoire mais introduit une surcharge d'exécution.
Cette thèse vise à résoudre le problème de la sécurité de la mémoire en développant un langage de programmation réactif spécifiquement conçu pour les systèmes en temps réel, intégrant des threads coopératifs et une exécution synchrone. En nous appuyant sur le solide système de types du langage de programmation Rust, nous proposons un système de types similaire à Rust pour un langage de programmation réactif de noyau appelé MSSL. MSSL offre un modèle de threads coopératifs et facilite le partage de données mutables entre les threads tout en préservant la sécurité des types et des emprunts. Pour y parvenir, nous introduisons une nouvelle abstraction appelée Trc (Thread Reference Counting), qui combine la sécurité de propriété des références de Rust avec le mécanisme de comptage de références des pointeurs intelligents de Rust. Nous présentons ensuite la sémantique et le système de types de MSSL pour démontrer ses capacités à maintenir la sécurité des types, la sécurité des emprunts et la sécurité de la concurrence. De plus, nous étendons MSSL en introduisant des extensions réactives, en intégrant le concept de signaux qui offrent des moyens de communication puissants, flexibles et fiables. Enfin, nous fournissons une implémentation en Java de l'ensemble complet de MSSL, en basant sur la sémantique et les règles de typage de son système de types.

Mots-clés : Ordonnancement coopératif, programmation réactive, langages synchrones, sécurité de la mémoire, Rust

Memory Safety for Synchronous Reactive Programming

Abstract:
Synchronous Reactive Languages are an excellent choice for IoT programming due to their clear system-environment interaction semantics. However, when it comes to safety- critial or resource-constrained systems, recent proposals like Fairthreads or ReactiveML face a well-known issue. Fairthreads' manual memory management can lead to errors, while ReactiveML's garbage collection ensures memory safety but introduces execution overhead.
This thesis aims to address the memory safety issue by developing a reactive programming language specifically designed for real-time systems, incorporating cooperative threads and synchronous execution. Drawing from the robust type system of the Rust programming language, we propose a Rust-like type system for a kernel reactive programming language named MSSL. MSSL features a cooperative threading model and facilitates mutable data sharing between threads while preserving type and borrowing safety. To achieve this, we introduce a novel abstraction called Trc (Thread Reference Counting), which combines the ownership safety of Rust references with the reference counting mechanism of Rust smart pointers. Then, we present the semantics and type system of MSSL to demonstrate its capabilities in maintaining type safety, borrowing safety, and concurrency safety.
Furthermore, we extend MSSL by introducing reactive extensions, incorporating the concept of signals that offer powerful, flexible, and reliable means of communication. Finally, we provide a Java implementation of the complete MSSL set, based on the semantic and typing rules of its type system.

keywords: Cooperative scheduling, reactive programming, synchronous languages, memory safety, Rust

Université d'ORLÉANS

LIFO - Bâtiment IIIA
Rue Léonard de Vinci
B.P. 6759 F-45067 ORLEANS Cedex 2

LABORATOIRE D'INFORMATIQUE FONDAMENTALE D'ORLEANS