IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Safety for Concurrent Systems via Model Checking on Behavioural and Session Types

Julia, Rebecca, Ougi, Homura GABET

Submitted in part fulfillment of the requirements for the degree of Doctor of Philosophy in Computing at Imperial College London, 2023.

Copyright Declaration

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution-Non Commercial 4.0 International Licence (CC BY-NC).

Under this licence, you may copy and redistribute the material in any medium or format. You may also create and distribute modified versions of the work. This is on the condition that: you credit the author and do not use it, or any derivative works, for a commercial purpose.

When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes.

Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

Statement of Originality

I hereby declare this submission to be my own work, and that to the best of my knowledge it contains no materials previously published or written by another person, nor substantial portions of other material which have been accepted for the award of any other degree or diploma at any educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked anywhere, is explicitly acknowledged in the thesis, especially any work summarised and explained in the context chapters. I also declare the intellectual content of this thesis to be the product of my own work, except to the extent that assistance from others in presentation, style or conception of the project's design, and linguistic expression, is acknowledged.

Abstract

Concurrent systems are ubiquitous in our modern world, from usual online interactions between our devices and servers to distributed systems and automated production chains. Safety of communication protocols in concurrent systems is therefore a key point to ensure the reliability of systems we rely on a daily basis. Several models exist to represent concurrent systems; and several works build around those models to provide verification toolchains for their protocols. In this thesis, we provide extensions of safety properties for two such frameworks.

One is a framework based on behavioural types theory for the popular programming language `Go`. Since a broad range of errors arise from mixed use of shared memory and channel-based communication, we develop an extension to both the type system and the static verification tool to account for *explicit memory locking mechanisms* and *direct-access shared memory primitives*, in addition to channel-based communication primitives that were accounted for in previous works. The other is a newly reworked multiparty session types (MPST) framework for general concurrency use, for which we propose an extension to account for *non-deterministic communications* and to make use of the *witness extraction capabilities* of model checking tools. The tools we provide with our theories are, to the best of our knowledge, respectively the first static verification framework for the `Go` language to uniformly analyse concurrency errors caused by a mix of asynchronous message-passing communications and shared memory accesses; and the first MPST toolchain to make use of witness extraction capabilities of model checking tools to link detected communication liveness or safety errors directly to the analysed program.

Extended Abstract

Concurrent systems are ubiquitous in our modern world, from usual online interactions between our devices and servers to distributed systems and automated production chains. Safety of communication protocols in concurrent systems is therefore a key point to ensure the reliability of systems we rely on a daily basis. Several models exist to represent concurrent systems; and several works build around those models to provide verification toolchains for their protocols. In this thesis, we provide extensions of safety properties for two such frameworks.

Thanks to its ability to efficiently combine concurrency and systems programming, `Go` is a popular programming language. However, a mixture of data races and communication problems creates the setting for a wide range of concurrency bugs. The first part of our work develops a theory based on behavioural types to statically detect data races, safety issues and deadlocks in `Go` programs. We extend `MiGo`, an existing framework that provides verification for channel-based communication in `Go` systems. Our extension provides a unified framework accounting for channel primitives as well as shared memory and lock primitives. We represent lock and shared-memory properties of programs in a μ-calculus model of types, and validate them using type-level model checking. We also implement a static verification tool that can uniformly detect concurrency errors caused by a mix of shared memory accesses and asynchronous message-passing communications. This is, to the best of our knowledge, the first static verification framework of this kind for the `Go` language.

Multiparty session types (MPST) provide a type-based methodology to statically ensure that a set of concurrent processes implements a desired protocol free of deadlocks and communication errors. While some MPST-based frameworks incorporate model checking tools, they have not yet made use of *witness extraction* to extract counterexamples from the target system when a verified property is not satisfied. In the second part of our work, we propose an integration of these capabilities of model checking with the MPST theory. To enable this integration, we establish a strong characterisation linking properties between three formalisms: the π-calculus, session types and the models. We show on the one hand *soundness and completeness* for safety, and on the other hand *session fidelity*, i.e. correspondence between safe processes, their types and their models for other desired properties. We implement a toolchain that verifies processes using a counterexample guided approach, offering precise information to the user about detected errors and their location to streamline the fixing process. To the best of our knowledge, this is the first MPST framework to make use of the witness extraction capabilities of model checking tools, and by extension the first to do so for error-fixing purposes.

Acknowledgements

I would like to thank the people without whom this work would not have been the same, or even not have been at all. Particular thanks go to my colleagues of the extended MRG, including Nicolas and Fangyi who were doing their own PhD at the same time and who helped a lot with writing ideas and proofreading; David, Lorenzo, Martin, Stephen, Shahin, Eva, Nick, Ping, Ray, Rumy, Alceste, Julien, Simon, Sung and Adam for their presence at various times, in particular external events, and their availability to answer questions relative to their own works; and Francisco who was particularly present with Fangyi and Nicolas when times were hard.

Thanks go to my supervisors, Nobuko Yoshida and Paul Kelly, for their presence and help in making relevant academic contacts throughout my PhD and allowing those years to happen, as well as Teresa and Amani who were always present when I had administrative questions or needed help with anything relative to work. Other collaborators met through them of through colleagues mentioned prior helped with several ideas, including Simon Gay and Nicolas Dilley. Many thanks also to my examiners, Iain Phillips and Mohammad Mousavi, for the discussion we had during the viva and the thorough remarks and questions that were asked both at the time and in the annotations provided for my corrections.

My family was a source of support in hard times, and especially in the hardest of times (in arbitrary order) Antoine, Manon, my grandmother and my father, but also Jérôme, Lucie, Emmanuel and Benjamin for being here as my siblings when I needed them. The rest of my close family in general for being present in the few moments I could travel back home, with the international restrictions we know during those years.

My colleagues at IRISA in 2023 when I was both working on a different project with team SPICY and finishing this present manuscript, including my supervisors Stéphanie and David, but also in no particular order Clément, Joseph, Claire, Louis, Gwendal, Pierrick, Clémence, Agathe, André, Damien, Léo, Arthur, Santiago, Joan, Hugo, Camille, Victoire, Ambre, Caroline and so many others I probably forgot while making this list.

Finally, my friends, among those a few colleagues cited previously, but additionally and in no particular order Emily, Lily, Charlotte, Manon, Xavier, Marie, Dayson, Nicolas, Geoffrey, Armand, Léa and Laura and many others who were kind enough to bear with me when things were hard and to laugh together in various games and stories for all those years; and Camille for the unwavering support and love you have given me these past few years and for simply being you.

Contents

List of Figures iv

List of Tables vi

List of Theorems vii

I Introduction viii

Motivation ix

Safety for Go programs using Behavioural Types ix

Extending Safety properties for Multiparty Session π-calculus xi

Contributions xiv

Structure of the Thesis xvi

II Behavioural Types for Go Verification 1

1 Background: Automated Go program verification with MiGo & Behavioural Types 3

1.1 MiGo: a π-calculus-based Preliminary Abstraction of Go Channels. 3

1.2 Typing MiGo with Behavioural Types: Preliminary Definitions 8

2 An Extended π-calculus-inspired Abstraction of Go: MiGo$^+$ 13

2.1 Extending MiGo with Memory Primitives: MiGo$^+$ 13

2.2 Desirable Properties of MiGo$^+$ with the Go Memory Model 21

3 Behavioural Types for MiGo$^+$ 27

3.1 Behavioural Types for MiGo$^+$ 27

3.2 Properties of Typed MiGo$^+$ 32

4 Model Checking Typed MiGo$^+$ Programs 44

4.1 Necessary Definitions on the Modal μ-calculus 44

4.2 Expressing our Properties with the Modal μ-calculus 46

4.3 Implementation Details: Godel 2 . . . 48
4.3.1 The toolchain . . . 48
4.3.2 Evaluation . . . 50

5 Related Works **52**

III Multiparty Session π-calculus 55

6 Background: Less is More: Multiparty Sessions Types Revisited **57**
6.1 Introduction to a New MPST Framework . . . 57
6.2 Our contributions to extend the new framework . . . 59

7 The Multiparty Session π-calculus **61**
7.1 Syntax of the Multiparty Session π-calculus . . . 61
7.2 Labelled Transition System for the Multiparty Session π-calculus . . . 64
7.3 Properties of Multiparty Session Processes . . . 65

8 Multiparty Session Types for the Multiparty Session π-calculus **67**
8.1 Syntax of the Types . . . 67
8.2 LTS of the Types and Type Safety . . . 68
8.3 Typing Rules . . . 70
8.4 Properties of Typed Multiparty Session Processes . . . 73
8.5 Relating Properties Between the π-calculus and the Types . . . 73

9 Model Checking the Session Types **90**
9.1 Model Checking and Counterexample Extraction . . . 90
9.2 Implementation and Counterexample Extraction . . . 93
9.2.1 Phase 1: Type Extraction . . . 94
9.2.2 Phase 2: Model Generation . . . 95
9.2.3 Phase 3: Model Checking with `mCRL2` . . . 96
9.2.4 Phase 4: Witness Extraction . . . 96

10 Related Works **98**

IV Conclusion & Future Works 100

Bibliography **103**

List of Figures

1 Go program with RWMutex: unsafe (left) and safe (right) x

2 Communication diagram of an unsafe implementation of the OAuth protocol xii

1.1 Syntax of MiGo processes and programs 3

1.2 Structural congruence rules for MiGo processes 5

1.3 Reduction rules for MiGo 5

1.4 Syntax of the behavioural types for MiGo, and their structural congruence 9

1.5 Rules for the LTS of the MiGo behavioural types 10

1.6 Typing rules for MiGo 12

2.1 Syntax of MiGo$^+$ processes and programs 14

2.2 Go program with RWMutex: unsafe (left) and safe (right) 15

2.3 Definition of the set of free names for MiGo$^+$ processes 16

2.4 Structural congruence rules for MiGo$^+$ 17

2.5 Labels used by the LTS of MiGo$^+$, barbs and happens-before relation 17

2.6 LTS rules for atomic actions (top), stores (middle) & store creation (bottom) 18

2.7 LTS rules for synchronisation (top) & structural constructs (bottom) 19

2.8 Rules for the happens-before relation 23

3.1 Syntax of the behavioural types for MiGo$^+$, and their structural congruence 28

3.2 The set of free names for behavioural types 28

3.3 Labels for the LTS of the behavioural types 28

3.4 Types LTS Rules for atomic actions (top), stores (middle) & stores creation (bottom) 29

3.5 Types LTS rules for synchronisation (top) & constructs (bottom) 29

3.6 Typing rules for MiGo$^+$ 33

3.7 Go implementation of a concurrent Prime Sieve algorithm [LNTY17, NY16] 42

4.1 Syntax of the μ-calculus formulae 45

4.2 Properties of the behavioural types as μ-calculus formulae 46

4.3 Workflow of the verification toolchain 49

6.1 Communication diagram of an unsafe implementation of the OAuth protocol 59

7.1 Structural congruence of the Multiparty session π-calculus 62
7.2 Subject channels of the multiparty session π-calculus 62
7.3 LTS of the multiparty session π-calculus (Definition 7.5) 64

8.1 Typing rules for the multiparty session π-calculus 71
8.2 Properties of typing contexts 73
8.3 Relation of the π-calculus properties, types properties and model checking 74

9.1 Properties of typing contexts as μ-calculus formulae 91
9.2 Workflow of the tool 94
9.3 Top: syntax of the processes in the tool; Bottom: extracted types output 94
9.4 The extracted `mCRL2` model 95
9.5 Verification results with witness 96
9.6 Full verification results 96

List of Tables

4.1 Verification results for Godel 2 51

7.1 Free/declared process variables and free channels/variables in the multiparty session π-calculus 63

9.1 Verification results of our recursive OAuth2 example, and other protocols from [SY19a] . . . 97

List of Theorems

2.11 Theorem (Equivalence of Data Race characterisations) . . . 24

3.8 Proposition (Subject Congruence) . . . 32
3.9 Theorem (Subject Reduction) . . . 34
3.10 Theorem (Subject Properties) . . . 35
3.11 Theorem (Progress) . . . 35
3.14 Theorem (Process Safety and Data Race Freedom) . . . 37
3.18 Proposition (May-Converging Liveness) . . . 39
3.20 Proposition (Strong Termination Liveness) . . . 40
3.26 Proposition (Liveness for Finite Branching) . . . 41
3.32 Theorem (Liveness for Alternating Conditionals) . . . 42

4.3 Theorem (Model Checking of MiGo$^+$ processes) . . . 48

8.11 Proposition (Subject Reduction) . . . 73
8.16 Theorem (Type Safety Soundness) . . . 77
8.22 Theorem (Type Safety Completeness for Unique-Role Processes) . . . 79
8.25 Proposition (Subject LTS Step) . . . 81
8.27 Proposition (Type Minimality Conservation) . . . 85
8.28 Proposition (Subject LTS Reduction) . . . 86
8.29 Theorem (Session Fidelity) . . . 86

9.1 Theorem (Precise Characterisation) . . . 92

Part I

Introduction

Motivation

Safety for Go programs using Behavioural Types

Go is a concurrent programming language designed by Google for *programming at scale* [Pik12]. Over the last few years, it has seen rapid growth and adoption: for instance in 2018, when we started working on the project, major developer surveys [Fra18] showed that StackOverflow placed Go in the top 5 most loved and the top 5 most wanted languages; and GitHub has reported [Git18] that Go was the 7th fastest growing language. More recently in 2022, StackOverflow's survey [Sta22] placed Go as 4th most wanted and 8th most loved language, and GitHub [Git22] shows Go as the 4th fastest growing language still; both showing that work on the language and verification of programs written with it is increasingly relevant.

One of the core pillars of Go is its concurrent programming features, including the locking of shared memory for thread synchronisation, and the use of explicit message passing through channels, inspired by process calculi concurrency models [Hoa85, Mil80]. In practice, shared accesses to memory using locking mechanisms are unavoidable, and could be error-prone. It is also of note that both shared memory and message passing operations provide a substantial part of the concurrency features of Go, and are the ones that are more prone to misuse-induced bugs. These unsafe memory accesses may lead to *data races*, where programs silently enter an inconsistent execution state leading to hard-to-debug failures. The left side of Figure 1 illustrates a Go program, which makes use of lock `m` to synchronise the `main` and `f` functions updating the content of variable `x`. On line 3, the statement `m := new(sync.RWMutex)` creates a new read-write lock `m`, called **`RWMutex`** in Go, used to guard memory accesses based on them being read or write accesses. The **`RWMutex`** object can then be passed around directly as a reference, as on line 4, circumventing the issue that could arise if we copied the mutex structure instead by passing it as a value. Indeed, copying a mutex structure makes it a distinct object, decorrelated from its original, and thus doing so would defeat the purpose of using a mutex. A mutex can be locked for writing by calling its `Lock()` method, unlocked from writing handle with its `Unlock()` method, and locked and unlocked for reading with the `RLock()` and `RUnlock()` methods. Readers and writers are mutually exclusive, and writers are mutually exclusive to each other too (hence the name **`Mutex`**, for *mutual exclusion* lock), but an arbitrary number of readers can hold the lock at the same time. The **`go`** keyword in front of a function call on line line 4 spawns a lightweight thread (called a *goroutine*) to execute the body of function `f`. The two parameters of function `f` – a rwmutex `m`, and an **`int`** pointer `ptr` – are shared between the caller and callee goroutines, `main` and

```go
1  func main() {
2    var x int
3    m := new(sync.RWMutex)
4    go f(m, &x)
5    m.RLock()   // acquire the lock for reading
6    x += 10     // write not protected by the lock
7    m.RUnlock() // release the read-lock
8    m.Lock()    // acquire the lock for writing
9    fmt.Println("x is", x)
10   m.Unlock()  // release the write-lock
11 }
12
13 func f(m *sync.RWMutex, ptr *int) {
14   m.RLock()
15   *ptr += 20  // write not protected by the lock
16   m.RUnlock()
17 }
```

```go
1  func main() {
2    var x int
3    m := new(sync.RWMutex)
4    go f(m, &x)
5    m.Lock()    // acquire the lock for writing
6    x += 10     // protected by the lock
7    m.Unlock()  // release the write-lock
8    m.RLock()   // acquire the lock for reading
9    fmt.Println("x is", x)
10   m.RUnlock() // release the read-lock
11 }
12
13 func f(m *sync.RWMutex, ptr *int) {
14   m.Lock()
15   *ptr += 20  // protected by the lock
16   m.Unlock()
17 }
```

Figure 1: Go program with RWMutex: unsafe (left) and safe (right)

`f`. Since concurrent access to the shared pointer `ptr` may introduce a data race, the developer tries to ensure serialised, mutually exclusive access to `ptr` in `f` and `x` in `main` by using read-locks. Using read-locks is unsafe in this case, allowing simultaneous write requests to `x` on lines 6 and 15, the program could then output "`x is 20`" with a bad scheduling, dropping the increase of 10 in the same thread as the print statement.

The right side of Figure 1 illustrates the same Go program, using the **`RWMutex`** feature correctly by putting writer sections of the code under writer locks. This alone prevents the data race seen in the first version of the program.

Go provides an optional *runtime* data race detector [VG13, Go13] as a part of the Go compiler toolchain. The race detector is based on LLVM's ThreadSanitizer [SI09, The15, SPIV12] library, which detects races that manifest during execution. It can be enabled by building a program using the "`-race`" flag. During the program execution, the race detector creates up to four shadow words for every memory object to store historical accesses of the object. It compares every new access with the stored shadow word values to detect possible races. These runtime operations cause high overheads of the runtime detector (5–10 times overhead in memory usage and 2–20 times in execution time on average [Go13]), hence it is unrealistic to run it with race detection turned on in production code; and because of that, race detection relies on extensive testing or fuzzing techniques [Vyu15, Syz15]. Moreover, as reported by Tu, Liu, Song and Zhang [TLSZ19], the detector fails to find many non-blocking bugs as it cannot keep a sufficiently long history; and its semantics does not capture Go specific non-blocking bugs.

The Go memory model [Go 22c] defines the behaviour of memory access in Go as a *happens-before* relation by a *combination* of shared memory and channel communications. It is also reported that the most difficult bugs to detect are caused when synchronisation mechanisms are used together with message passing operations [TLSZ19]. For instance, Go can use message passing for sharing memory (channel-as-lock) or passing pointers through channels (pointer-through-channel), which might lead to a serious non-blocking bug, i.e. the program may continue to execute in unwanted and incorrect states or corrupt data in its computations [TLSZ19], due to subtle interplays with buffered asynchronous communications.

Those issues motivate us to *uniformly* model, statically analyse and detect concurrent non-blocking/blocking shared memory/channel-communications bugs in Go, using a formal model based on a process calculus [Hoa85, Mil80].

Extending Safety properties for Multiparty Session π-calculus

The *session types systems* [Hon93, HVK98, THK94] are a set of *concurrent behavioural type system* [HLV+16] that can ensure protocol compliance for communicating processes. Many variants of session types consist of not only simple actions for sending and receiving values, but also channel passing (delegation), choices, sequencing and recursion. They can abstract the complex *dynamism* of process behaviours, despite their simple syntax: when a session typed process communicates, its session type also evolves, reflecting the new state of the process. This rich behavioural aspect has opened a new research area to connect session type-based verification to *model checking*, by linking the labelled transition systems of session types to those of the μ-calculus [LY17].

The idea – model checking concurrent behavioural types – had been first proposed in 2001 [RR01], but the method was only explored recently thanks to the significant advance in scalability and usability of off-the-shelf model checkers. The initial work from Lange and Yoshida [LY17] uses the mCRL2 model checking tool [Tec18, GM14], for a scalable verification of subtyping relations of *binary* session types. Their idea is further extended for *multiparty session types (MPST)* [HYC16, HY16] by Scalas and Yoshida [SY19a], where more than two participants interact together, complying with a *choreographic* protocol. Scalas and Yoshida [SY19a] propose a general framework for type-checking multiparty session processes by verifying *type-level behaviours* (without using global types), written as linear-time μ-calculus properties. The generality of this framework enables not only *type-safety*, but also other rich properties such as *deadlock-freedom*, *liveness* and *termination* of multiparty session processes, to be verified by applying mCRL2 *to session types only*. This framework is integrated in a recent work on process and communication reliability by including crash-stop failures, verifying whether processes can handle unexpected crash scenarios properly [BSYZ22]. The idea of model checking concurrent behavioural types is extended to a static verification of real-world message-

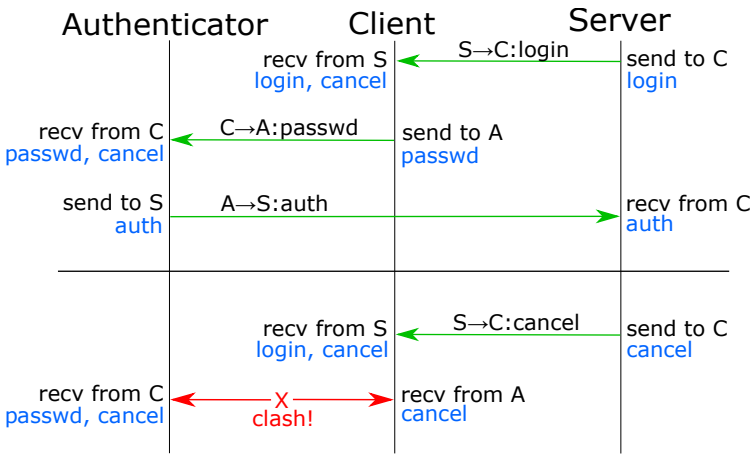


Figure 2: Communication diagram of an unsafe implementation of the OAuth protocol

passing programming languages, such as Go [LNTY17, LNTY18, GY20a] and Scala [SYB19], also using `mCRL2`.

This work takes a step further in the exploration of the use of a model checking framework for the verification of *multiparty* session processes, applying witness extraction to extract a *counterexample* on representative properties when they are determined to be false. This enables us to notify the developer what the problems in their input system are and where they are situated.

In the context of MPST where, due to complexity of interactive behaviours, the typing systems are notoriously error-prone [SY19a], the counterexample guided approach is especially useful to guide the user to determine which session breaks safety or liveness, and at what point of the protocol, easing the fixing process. We implement a toolchain applying `mCRL2` to verify various properties of (untyped) session processes, and identifying the location and nature of detected errors.

We also discovered an error in the version of *session fidelity* (a core theorem in MPST based on which deadlock-freedom/liveness are guaranteed) [SY19a], and show both an example breaking their original version and a fix.

To showcase this work, we use the *OAuth protocol* [OAu12] as a running example throughout Part III. The protocol is illustrated in Figure 2. There are three *roles* (or *participants*), the Authenticator, the Client and the Server. The part above the horizontal line in Figure 2 shows a correct interaction: first, Server sends a `login` request to Client; next Client sends its `password` to Authenticator; then Authenticator confirms authorisation by sending `auth` to Server. Note the second option in the first two interactions, `cancel`, which can be selected by Server and forwarded by Client, respectively. The part below the horizontal line shows an incorrect interaction, where the Client does not properly forward the `cancel` request to the Authenticator, and instead waits for it to send `cancel`, where it is waiting for the Client to send either a

`password` or a `cancel` request.

Albeit looking simple for a multiparty protocol, the recent work by Scalas and Yoshida [SY19a] has uncovered that none of existing MPST frameworks which use *end-point projections* from *global types* work to provide *type-safety* (i.e., Subject Reduction Theorem) of this example. This is because the typing system in those works either is too restrictive [HYC16, HY16]; or fails to satisfy the consistency condition [YDBH10]. Thus, one needs to directly verify a *global type* to type-check this example in these frameworks, which does not enable for type-safety verification of the local implementations of each endpoint.

Although it is general, the framework proposed by Scalas and Yoshida [SY19a] has one serious drawback: without any guidance by a global protocol, one can easily implement an erroneous process in a number of ways. The simplest such case is drawn below the horizontal line in Figure 2: Client and Authenticator have a clash in communication in case the interaction is cancelled by Server. In this case, Client tries to receive a `cancel` message from Authenticator, instead of forwarding the `cancel` message to Authenticator: both Authenticator and Client are waiting for a message from each other, and the communication can never occur (the process is ***deadlocked***). To fix this, we can change the blocking action in Client from an input to an output, sending a `cancel` message to Authenticator instead. All participants then know Server aborted the session, and the system can loop properly. This is but one possible mistake – a ***mismatch error***, where Client sends its `address` which Authenticator does not expect; or a ***liveness violation***, where Authenticator sends a `cancel` message that is never consumed, are easily introduced.

Contributions

Our first set of contributions proposes a uniform model for verifying Go programs that handles both message-passing concurrency — as described by the authors of the original works [LNTY17, LNTY18] — and shared memory concurrency — our own addition to the framework. We present the theory, design and implementation for a verification tool able to detect a set of concurrency-related bugs in Go: liveness and safety of channel use from the original works are included [LNTY17, LNTY18], and we add liveness and safety for lock use, as well as data race detection for shared memory. This is done by first extending their abstraction language, MiGo, to account for more Go primitives. We dub this extension MiGo$^+$, and formalise a happens-before relation following the official Go memory model, as well as the desired liveness and safety properties. We looked through the Go specification documents to find their description of the memory model used by the language (found at [Go 22c]), and used it to formalise our relation on our language model. Then, we introduce an extended behavioural type system from the one presented in the aforementioned works, to account for the new primitives. This enables us to use static analysis techniques on the processes to detect the bugs we target, in this case through the use of a model checking tool, mCRL2.

We prove the soundness of our type system, as well as a progress theorem, to show the formal relationship between type-level properties and process-level properties. In particular, we establish equivalences of the desired properties between the two sides for specific subsets of the processes. These subsets have some overlap, in particular terminatinng processes are entirely included in a subset we call May-Converging that satisfies equivalence of our properties. Another subset that partly intersects this May-Converging subset satisfies this equivalence as well, but is contained in a bigger set o processes that generally do not — unless they are in the intersections, of course. This allows us to only run the verification on the types, which is more efficient as an overapproximation of the execution, to obtain a result that is applicable to the processes we wish to verify — as long as they are in the subsets we establish equivalence for.

The mCRL2 tool enables a proof-of-concept tool for our framework. We infer the MiGo$^+$ primitives from Go code, then run the corresponding types through the model checker with generated properties corresponding to the ones presented in our theory. Our properties include both blocking and non-blocking errors: a liveness error or a data race is usually not blocking, but a safety error on channels or locks will usually cause a crash. The tool also detects deadlocks, a legacy from the original works [LNTY17, LNTY18].

The following publication presents the above contributions in conference format, and served as the base material along our notes to write the corresponding part of this thesis:

- Julia Gabet and Nobuko Yoshida. Static Race Detection and Mutex Safety and Liveness for Go Programs. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:30, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2020.4

- Julia Gabet and Nobuko Yoshida. Static Race Detection and Mutex Safety and Liveness for Go Programs (Artifact). *Dagstuhl Artifacts Series*, 6(2):12:1–12:3, 2020. URL: https://drops.dagstuhl.de/opus/volltexte/2020/13209, doi:10.4230/DARTS.6.2.12

Our second set of contributions offers an extension for the revisited Multiparty Session Types (MPST) framework, introduced by Scalas and Yoshida [SY19a] and detailed in Chapter 6. We achieve this by first formalising a labelled transition system (LTS) for the untyped multiparty session π-calculus, and introducing more error reductions, in order to define more precise liveness and safety properties directly for processes. Similar properties are defined on the type system, matching the properties of the processes and with the aim to verify the properties on the types to ensure the behaviour of the processes.

By proving a type preservation property for our LTS, and proving type soundness, we show that safe processes (i.e. processes that are typed with a safe type) only ever reduce to safe processes. We also prove that for a subset of the processes, if the process is safe, then a safe type for it exists. This subset is relatively constrained, as it enforces each process to not play more than one role, but encompasses several real-life protocols such as the OAuth authentication protocol [OAu12]. Moreover, we show that same subset of processes follows the properties of their types, if they are type-safe, with the *session fidelity* theorem we port from the original work [SY19a]. We also fix an error we found in the original version of that theorem in their paper, and provide a counterexample to their original version.

We finally show that the properties in the processes and that of the types, written with the types framework or in a modal μ-calculus abstraction, coincide with each other for the subset of processes mentioned above. This allows our second contribution for this part, a proof-of-concept verification tool for the multiparty session processes. Built on the tool from Scalas and Yoshida's work [SY19c], it adds support for the processes with a type extraction capability, extends the safety property to our broader set of errors, and implements a counterexample extraction capability. That last point is thanks to the then-recently overhauled witness extraction capability in `mCRL2`, and enables the user to locate the errors more easily when running their translated processes through the tool. A preliminary version of our tool is available on GitHub [Gab22].

Structure of the Thesis

In Part I, we introduced the motivations and contributions of the thesis, and now its structure.

In Part II, we detail our work on extending behavioural types verification for `Go` programs. Chapter 1 lays the context and state of the art for this contribution, especially presenting the original system before our extensions and rework. Then, Chapter 2 shows how we extended the process calculus to match our needs, laying the base for our work on verifying more `Go` constructs. Chapter 3 follows in the same fashion, extending the type system and showing its properties, both new and revisited ones. Finally, Chapter 4 links the verification of the types from Chapter 3 with that of the processes of Chapter 2, with the use of the model checking tool `mCRL2`. Related works for this contribution are laid out in Chapter 5.

In Part III, we detail our work on extending the recent MPST framework from Saclas and Yoshida [SY19a] to account for more errors and allow counterexample-based verification of processes. Chapter 6 introduces the context of this new framework and the limitations we wanted to overcome. Chapter 7 shows our extension to their process calculus to account for more errors and a more general communication behaviour. We extend the type system in the same fashion in Chapter 8, and show its properties along with a fix to an error we spotted in the original system. Finally, Chapter 9 links the verification of processes and types, again with the use of the `mCRL2` model checking tool, and showcases its witness extraction capabilities for our counterexample extraction mechanism for process verification. We close this part of our contribution with relative works in Chapter 10.

In Part IV, we conclude the thesis, and also list future works opened by our contributions.

Part II

Behavioural Types for Go Verification

In this part, we present a static verification framework for Go programs using behavioural types and the mCRL2 model checker. We first give context in Chapter 1 by going over the previous framework outlined in the original works we expend upon [LNTY17, LNTY18], with the same objective as ours, yet limited to the channel fragment of Go programs. Our contribution is then split in Chapters 2 to 4 as follows:

- First, Chapter 2 describes an abstracting fragment of the Go language that encapsulates the fragment from the previous works, MiGo$^+$, thus containing both Go channels (the MiGo fragment [LNTY17, LNTY18]) and Go memory primitives (our own contribution [GY20a]). Specifically, we abstract the types of shared data in channels and memory primitives. We also give some desirable properties and show a correspondence between our definition of a data race in MiGo$^+$ with the definition that can be inferred from the official Go memory model.
- Chapter 3 introduces the behavioural type system for our MiGo$^+$ processes, abstracting the fragment of the language we wish to verify, and then shows properties of typed processes. In particular, this is where we define precisely fragments of the typed MiGo$^+$ programs that can be verified precisely using our framework.
- Finally, Chapter 4 concentrates on automating the verification of typed MiGo$^+$ processes with the help of a model checking tool, mCRL2. This is where we introduce the necessary third layer of abstraction to use this automation tool, the modal μ-calculus, and showcase the tool built from our framework.

1 Background: Automated Go program verification with MiGo & Behavioural Types

This chapter describes MiGo as presented by Lange, Ng, Toninho and Yoshida in 2017 [LNTY17], and its associated behavioural types, introduced in that same work and extended by the same authors the following year [LNTY18]. Understanding their system is necessary to understand the scope of our own work. Unless explicitly stated otherwise, the content in this chapter is a direct adaptation of these two works, and should be considered one giant citation, as we will omit citing them in many places of this chapter for the sake of readability. Section 1.1 presents the base process calculus, called MiGo, and its reduction rules that we do not use in our own approach, but are emulated by our labelled transition system (LTS) in Chapter 2. Section 1.2 presents the associated behavioural types with their own LTS and the typing rules, as those are the main parts that will need adjustment beyond simple additions.

1.1 MiGo: a π-calculus-based Preliminary Abstraction of Go Channels.

This section presents MiGo as introduced in 2017 by Lange, Ng, Toninho and Yoshida [LNTY17].

The syntax for the MiGo processes is shown in Figure 1.1. This process calculus aims to abstract the channel fragment of Go programs, and is thus quite simple. It only accounts for channels and simple related constructs, foregoing anything that is not related — including more standard shared memory and exclusion-lock constructs. It is inspired by the π-calculus and designed to stay close to the Go syntax for the basic constructs included to be recognisable by Go developers.

$$
\begin{array}{rcl|rcl}
P, Q, R & := & \pi; P \mid (P \mid Q) \mid \mathbf{0} \mid (\nu c)P & \boldsymbol{P} & := & \{D_i\}_{i \in I} \,\mathsf{in}\, P \\
& \mid & \mathsf{close}\ u; P \mid \mathsf{select}\ \{\pi_i; P_i\}_{i \in I} & u & := & a \mid x \\
& \mid & \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q & \pi & := & \tau \mid u!\langle e \rangle \mid u?(y) \\
& \mid & X\langle \tilde{e}, \tilde{u} \rangle \mid \mathsf{newchan}(y{:}\sigma, n); P & v & := & \mathsf{n} \mid \mathsf{true} \mid \mathsf{false} \mid x \\
& \mid & c\langle \sigma, n \rangle :: \tilde{v} \mid c^{\oslash}\langle \sigma, n \rangle :: \tilde{v} & e & := & v \mid \mathsf{not}(e) \mid \mathsf{succ}(e) \\
D & := & X(\tilde{x}) = P & \sigma & := & \mathsf{bool} \mid \mathsf{int} \mid \ldots
\end{array}
$$

Figure 1.1: Syntax of MiGo processes and programs

The channels are ranged over by a, b, c; named variables are ranged over by x, y, z. Named variables can be channels in named processes, or otherwise ground-typing expressions able to be passed around in channels. Expressions e are constructed from ground-typing values, named variables, and operations on ground types such as negation for booleans and successor for integers. Programs $\boldsymbol{P}$ are a collection of process definitions D that can each depend on their own variable arguments — channels and expressions alike. Processes P, Q can be, in order:

- atomic actions of sending $u!\langle e \rangle$ or receiving $u?(y)$ through a channel u continued with a process;
- silent action τ continued with a process;
- parallel compositions $P \mid Q$;
- inactive processes ($\mathbf{0}$);
- a restricted process $(\nu c)P$ where channel c is invisible to exterior processes;
- closing of an existing channel;
- a select construct consisting of branches guarded by an atomic or silent action — a branch is chosen non-deterministically at runtime if its guard can be executed;
- an if-then-else construct;
- a process call to a predefined (possibly recursive) process $X\langle \tilde{e}, \tilde{u} \rangle$;
- the creation of a new channel of payload type σ and size n.

The syntax also contains the runtime constructs of open and closed channel buffers, the last two constructs in the list for P, Q, R. The buffers contain a first-in first-out (FIFO) list of messages of the channel payload type currently on the channel and waiting to be received, noted $\tilde{v}$. Goroutines, which are lightweight threads in Go, are abstracted by the use of parallel composition and process calls.

Remark 1.1. Note that if $n = 0$, then necessarily $\tilde{v} =$ 🤦 at all times; where 🤦 denotes the empty queue (we explicitly change the notation from $\emptyset$ in the cited work to be consistent with the notations of our contribution throughout the thesis). Another notation we changed is for closed channels, denoted with a 🚫 symbol in this thesis, instead of a $\star$ in the original work. This is again to have consistent notations throughout the thesis that we change it in this chapter, the meaning remains strictly the same. ⌟

Remark 1.2. Note that σ can **only** be a ground type, and not a channel, in order for subject reduction later to hold in the way it is stated by Lange, Ng, Toninho and Yoshida [LNTY17], and that we will be using as it extends nicely to our own framework later on. While not cited explicitly in their work, it is mandatory, and

$$P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad P \mid \mathbf{0} \equiv P$$
$$(\nu c)(\nu d)P \equiv (\nu d)(\nu c)P \qquad (\nu c)\mathbf{0} \equiv \mathbf{0} \qquad (\nu c)c\langle\sigma, n\rangle :: \tilde{v} \equiv \mathbf{0}$$
$$P \mid (\nu c)Q \equiv (\nu c)\,(P \mid Q)\ \ (c \notin \mathrm{fn}(P)) \qquad (\nu c)c^{\oslash}\langle\sigma, n\rangle :: \tilde{v} \equiv \mathbf{0}$$

Figure 1.2: Structural congruence rules for MiGo processes

$$[\text{SCOM}]\ \frac{e \downarrow v}{c!\langle e\rangle; P \mid c?(y); Q \mid c\langle\sigma, 0\rangle :: \text{👨} \to P \mid Q\{^{v}/_{y}\} \mid c\langle\sigma, 0\rangle :: \text{👨}}$$

$$[\text{SCLOSE}]\ c?(y); P \mid c^{\oslash}\langle\sigma, n\rangle :: \text{👨} \to P\left\{^{\sigma}/_{y}\right\} \mid c^{\oslash}\langle\sigma, n\rangle :: \text{👨}$$

$$[\text{CLOSE}]\ \ \text{close}\ c; P \mid c\langle\sigma, n\rangle :: \tilde{v} \to P \mid c^{\oslash}\langle\sigma, n\rangle :: \tilde{v}$$

$$[\text{NEWC}]\ \frac{c \notin \mathrm{fn}(P)}{\text{newchan}(y : \sigma, n); P \to (\nu c)\,(P\{^{c}/_{y}\} \mid c\langle\sigma, n\rangle :: \text{👨})}$$

$$[\text{IFT}]\ \frac{e \downarrow \text{true}}{\text{if}\ e\ \text{then}\ P\ \text{else}\ Q \to P} \qquad [\text{IFF}]\ \frac{e \downarrow \text{false}}{\text{if}\ e\ \text{then}\ P\ \text{else}\ Q \to Q}$$

$$[\text{DEF}]\ \frac{P\{^{\tilde{v},\tilde{c}}/_{\tilde{x}}\} \mid Q \to R \quad e_i \downarrow v_i \quad X(\tilde{x}) = P \in \{D_i\}_{i \in I}}{X\langle\tilde{e}, \tilde{c}\rangle \mid Q \to R}$$

$$[\text{TAU}]\ \tau; P \to P$$

$$[\text{PAR}]\ \frac{P \to P'}{P \mid Q \to P' \mid Q}$$

$$[\text{RES}]\ \frac{P \to P'}{(\nu c)P \to (\nu c)P'}$$

$$[\text{STR}]\ \frac{P \equiv Q \to Q' \equiv P'}{P \to P'}$$

$$[\text{SEL}]\ \frac{\pi_j; P_j \mid P \to R \quad j \in I}{\text{select}\ \{\pi_i; P_i\}_{i \in I} \mid P \to R}$$

$$[\text{OUT}]\ \frac{|\tilde{v}| < n \quad e \downarrow v}{c!\langle e\rangle; P \mid c\langle\sigma, n\rangle :: \tilde{v} \to P \mid c\langle\sigma, n\rangle :: v \cdot \tilde{v}}$$

$$[\text{INA}]\ c?(y); P \mid c\langle\sigma, n\rangle :: \tilde{v} \cdot v \to P\{^{v}/_{y}\} \mid c\langle\sigma, n\rangle :: \tilde{v}$$

$$[\text{INC}]\ c?(y); P \mid c^{\oslash}\langle\sigma, n\rangle :: \tilde{v} \cdot v \to P\{^{v}/_{y}\} \mid c^{\oslash}\langle\sigma, n\rangle :: \tilde{v}$$

Figure 1.3: Reduction rules for MiGo

we will work the entirety of this part of the thesis under this assumption. In particular, we do not allow for channel passing, which is a possibility in Go, though limited by the strong typing and lack of a reliable way to handle generics and polymorphism in the language at the time the work we cite here was produced — this is also true of our contribution. This limitation, along with the ongoing work on polymorphism in Go, will be discussed in the conclusion of this part of the thesis. ⌟

The structural congruence rules for MiGo are shown in Figure 1.2. They include commutativity and associativity of parallel composition, commutativity of restrictions on different names, commutativity of restriction with parallel composition if it is capture-avoiding, and what can be assimilated to garbage collection of unused channels and terminated processes.

The reduction rules for MiGo are given as a standard reduction system for processes, as outlined in Figure 1.3.

A synchronous communication happens when a channel of size 0 (synchronous channel) has a sender and a receiver available, computing the expression from the sender side to a value for the receiving process; or when a receiver is available on a closed channel with empty queue, receiving the default value for that data type. Closing a channel transforms the runtime construct from open to closed, marking it unable to process send actions — a Go program that tries to send to a closed channel will crash. Channel creation creates the (open) runtime construct with proper type and size ad an empty queue, in parallel of the current process.

If-then-else constructs reduce to the left process if the condition evaluates to true, or the right process otherwise. Process calls reduce in parallel of other processes as if they were the defined process with substituted parameters, as shown in [DEF]. Silent actions τ reduce without changing values in the rest of the process, and structural reductions in parallel compositions, restrictions and other structural congruence cases are standard. Reduction of a select construct happens non-deterministically between available branches, as explained earlier, as shown in [SEL].

Asynchronous rules for sending and receiving values over an asynchronous channel are standard, adding the value to the queue when sending and removing the received value from the queue upon reception.

Remark 1.3. Note that rule [INC] is actually an *addition*, not present in the original work [LNTY17], but needed for parity with the asynchronous semantics of Go. The most likely reason for this rule being forgotten in the original work is that asynchrony was a late addition to open the scope of their work. Adding this rule fills in the gap that exists with the Go semantics for channels, which is why we add this rule here. Note that we change the notation for the default value of type σ to $\mathbb{1}\sigma$, instead of some combination of σ and $\perp$ as found in the original work. This change is for the same reason as all previous notation changes, to keep things consistent and visual throughout the thesis. ⌟

The top set of rules works on its own and makes the case of a fully synchronous calculus — when all channels are size 0, in which case channel size can be omitted from the syntax, along with the buffer part at the end of the channel runtime construct if we want to really shorten the notations. Meanwhile, adding the bottom set of rules allows for asynchrony, i.e. channels of positive buffer size.

We define barbs to denote the availability of an action to the process. The doubled-arrow version is a transitive closure, showing the potential future availability of an action. Barbs are useful to define desirable properties of MiGo programs, such as liveness and safety.

Remark 1.4. Lange et al. use the notation $\mathrm{dom}(o)$ without defining it explicitly in their work. While we delay the formal definition for our own work in Chapter 2, the intuition for this notation is the set of all names appearing in the label of the predicate.

We use the occasion again to change the symbols for a few notations to make them more self-explanatory, in particular synchronisations are now noted with 🤝 instead of a τ, push and pop use 📤 and 📥 instead of a $\bullet$ in different places, and receiving any message from a closed channel now uses 📨 instead of the closed channel symbol (which will be used for the channel closing action in the LTS later). ⌟

Definition 1.5 (Barbs). We define predicates $\pi \downarrow_o$, $P \downarrow_o$ and $P \downarrow_{\tilde{o}}$ with $o, o_i \in \{a, \overline{a}, 🤝_a, \mathsf{end}[a], a^{📨}, a^{📤}, a^{📥}\}$.

$$c?(y) \downarrow_c \qquad c!\langle e\rangle \downarrow_{\overline{c}} \qquad \frac{\pi \downarrow_o}{\pi; Q \downarrow_o} \qquad \mathsf{close}\ c; Q \downarrow_{\mathsf{end}[c]} \qquad c^{🚫}\langle \sigma, n\rangle :: \tilde{v} \downarrow_{c^{📨}} \qquad \frac{|\tilde{v}| < n}{c\langle \sigma, n\rangle :: \tilde{v} \downarrow_{c^{📤}}} \qquad \frac{|\tilde{v}| \geq 1}{c\langle \sigma, n\rangle :: \tilde{v} \downarrow_{c^{📥}}}$$

$$\frac{P \downarrow_o}{P \mid Q \downarrow_o} \qquad \frac{P \downarrow_o \quad a \notin \mathrm{dom}(o)}{(\nu a)P \downarrow_o} \qquad \frac{P \downarrow_o \quad P \equiv Q}{Q \downarrow_o} \qquad \frac{Q\{\tilde{v}, \tilde{a}/\tilde{x}\} \downarrow_o \quad X(\tilde{x}) = Q}{X\langle \tilde{e}, \tilde{a}\rangle \downarrow_o}$$

$$\frac{\forall i \in \{1, \ldots, n\} : \pi_i \downarrow_{o_i}}{\mathsf{select}\ \{\pi_i; P_i\}_{i \in \{1,\ldots,n\}} \downarrow_{\{o_1,\ldots,o_n\}}} \qquad \frac{P \downarrow_a \quad Q \downarrow_o \quad o \in \{\overline{a}, a^{📨}, a^{📥}\}}{P \mid Q \downarrow_{🤝_a}} \qquad \frac{P \downarrow_{\overline{a}} \quad Q \downarrow_{a^{📤}}}{P \mid Q \downarrow_{🤝_a}}$$

$$\frac{P \downarrow_o \quad \pi_i \downarrow_{\overline{a}} \quad o \in \{a, a^{📤}\}}{P \mid \mathsf{select}\ \{\pi_i; Q_i\}_{i \in I} \downarrow_{🤝_a}} \qquad \frac{P \downarrow_o \quad \pi_i \downarrow_a \quad o \in \{\overline{a}, a^{📨}, a^{📥}\}}{P \mid \mathsf{select}\ \{\pi_i; Q_i\}_{i \in I} \downarrow_{🤝_a}}$$

$P \Downarrow_o$ if $P \to^* P'$ and $P' \downarrow_o$ with $o \in \{c, \overline{c}, 🤝_c, \mathsf{end}[c], c^{📨}, c^{📤}, c^{📥}\}$. ⌟

The first line of rules shows the barbs for atomic actions and runtime constructs. Then, a parallel composition gives the barbs of both sides, as we have both a rule for parallel composition and one for structural congruence. Restriction on a name that is not in a barb does not affect the barbs, and a process call has the same capabilities as its defined process substituted with its parameters. Select constructs provide a set of capabilities in a single barb if all of the guards are channel actions, we do not provide capabilities to select constructs containing at least one branch with a τ-action as guard. The last four rules show synchronisation barbs, when compatible channel capabilities are found on parallel processes — the first two are for single capabilities, and the last two are for select constructs.

From barbs, we define desired properties of MiGo programs for channel use, such as liveness and channel safety, which we show below. Liveness claims that if a program can send or receive a message through a channel at a given point, it should synchronise over that channel in the future; and if it can act on a set of channels through a select construct at a given point, it should synchronise over one of those channels in the future.

Definition 1.6 (Liveness). The program $\boldsymbol{P}$ satisfies liveness if for all Q such that $\boldsymbol{P} \to^* (\nu \tilde{c})Q$:

(a) if $Q \downarrow_a$ or $Q \downarrow_{\overline{a}}$ then $Q \Downarrow_{🤝_a}$;

(b) if $Q \downarrow_{\tilde{o}}$ then $Q \Downarrow_{🤝_{a_i}}$ for some $a_i \in \mathrm{dom}(\tilde{o})$. ⌟

Note that we denote $\mathrm{dom}(\tilde{o})$, instead of $\{\tilde{o}\}$ used by Lange et al. [LNTY17], since using set operations like "names", "free names" or "domain" is a clearer standard for cases like this. The same applies for the barb definition above with $\mathrm{dom}(o)$ instead of $\mathrm{fn}(o)$ in order to make the notation consistent, as well as to keep the free names notation to processes and types, i.e. abstractions of the programs.

The notion of safety, defined below, claims that a closed channel cannot be closed again or used to add new messages to the queue.

Definition 1.7 (Channel Safety)**.** The program $\boldsymbol{P}$ is *channel safe* if for all Q such that $\boldsymbol{P} \rightarrow^* (\nu\tilde{c})Q$, if $Q \downarrow_{a^{\text{✉}}}$ then $\neg(Q \Downarrow_{\mathsf{end}[a]})$ and $\neg(Q \Downarrow_{\overline{a}})$. ⌟

Note that the form that is given in those definitions can be seen as a normal form. While the existence of a normal form is never explicitly proven, it is a consequence of the structural congruence rules and the reduction system. In particular, the congruence rule to expand restrictions can be used as much as needed up to α-conversion, and all restrictions on the same level are interchangeable with the rule above the former. The fact that primitives only reduce when at the top of a parallel process secures that restrictions only happen at the topmost level in normal execution, unless one was intently introduced manually by the user when writing the MiGo process — thus the user would also have crafted the corresponding runtime construct by hand to get a working process.

1.2 Typing MiGo with Behavioural Types: Preliminary Definitions

This section presents the behavioural types for MiGo as presented by Lange et al. in 2017 [LNTY17] and extended by the same authors in 2018 [LNTY18]. We do not show *all* the additions from the latter, as some of them were not desirable for our own choice of additions to the framework, so the types presented here will be somewhere between the two works.

The syntax of the behavioural types matches and abstracts that of the processes, and is given in Figure 1.4 along with the structural congruence for the types. In particular, $\oplus\{T_i\}_{i\in I}$ represents an internal choice, the pendant of the conditional construct from Go and MiGo, while $\&\{\kappa_i;T_i\}_{i\in I}$ represents an external choice, also known as branching in concurrent process literature, and is the pendant of the select construct from above. A behavioural type can also send $\overline{u}$ or receive u on a channel u, be a parallel composition, be inactive, create a new channel a of size n before continuing to T, close a channel u before continuing to T, call a type definition with channel parameters $\tilde{u}$, or restrict channel name a on type T.

Types also contain runtime constructs for an open channel a of size n with k items currently in the buffer, and a closed channel a (without information of maximum size or current buffer size). A type equation $\boldsymbol{T}$ is a collection of type definitions applied to a behavioural type S. Structural congruence rules are the same

$$
\begin{array}{l}
T, S ::= \kappa;T \mid \oplus\{T_i\}_{i\in I} \mid \&\{\kappa_i;T_i\}_{i\in I} \mid (T \mid S) \mid \mathbf{0} \\
\qquad \mid (\mathsf{new}^n\ a);T \mid \mathsf{end}[u];T \mid \boldsymbol{t}_X\langle\tilde{u}\rangle \mid (\nu a)T \mid \lfloor a \rfloor_k^n \mid a^{\oslash} \\
\boldsymbol{T} ::= \{\boldsymbol{t}_{X_i}(\tilde{y}_i) = T_i\}_{i\in I}\ \mathsf{in}\ S \qquad \kappa ::= \overline{u} \mid u \mid \tau \\
\hline
T \mid S \equiv S \mid T \qquad T \mid (T' \mid S) \equiv (T \mid T') \mid S \qquad T \mid \mathbf{0} \equiv T \\
(\nu a)(\nu b)T \equiv (\nu b)(\nu a)T \quad (\nu a)\mathbf{0} \equiv \mathbf{0} \quad (\nu a)\lfloor a \rfloor_k^n \equiv \mathbf{0} \quad (\nu a)a^{\oslash} \equiv \mathbf{0} \\
T \mid (\nu a)S \equiv (\nu a)(T \mid S)\ (a \notin \mathrm{fn}(T)) \qquad T \equiv_\alpha T' \implies T \equiv T'
\end{array}
$$

Figure 1.4: Syntax of the behavioural types for MiGo, and their structural congruence

as for the processes, giving commutativity and associativity of parallel composition, commutativity of name restriction, commutativity of restriction and parallel composition if capture is avoided, and garbage collection rules for inactive processes and unused (closed) channels. Structural congruence contains α-equivalence.

The semantics of the behavioural types is given by a labelled transition system (LTS) shown in Figure 1.5. The labels for the LTS are ranged over by α, β and have the following form:

$$\alpha, \beta ::= \overline{a} \mid a \mid \tau \mid \text{🤝}_a \mid \mathsf{end}[a] \mid a^{\oslash} \mid a^{\text{📨}} \mid a^{\text{📤}} \mid a^{\text{📥}}$$

In order, they represent send and receive on channel a, silent action, synchronisation over channel a, request to close and closing of channel a, output of closed message from a closed channel $a^{\oslash}$, and push and pop of messages on open asynchronous channel a.

We write $\rightarrow$ for $\equiv \cdot \xrightarrow{\tau} \cdot \equiv$ and $T \rightarrow^* \xrightarrow{\alpha}$ if there exists T' and T'' such that $T \rightarrow^* T' \xrightarrow{\alpha} T''$. The reductions are mostly self-explanatory: the difference with the processes is that we have partial actions represented in the LTS of types along with the expected interactions that produce synchronisations between threads. Note that "naked" synchronisations are not included in $\rightarrow$ in this system, in line with the expected use of types as an abstraction of closed processes, where no free channel name happens in the process.

We make an explicit change to use $a \notin \mathrm{dom}(\alpha)$ instead of $\mathrm{fn}(\alpha) \neq \{a\}$ in rule |RES-1|, for the same reason as previously to keep notations consistent on labels and limit the free names notation to abstractions of programs — in this case processes and types. This will be reused in the definitions for type barbs and liveness coming up next.

In the rest of this thesis, we work in a context where all types (and associated processes) are *finite control*, i.e. no parallel composition appears in the context of a recursion: the program does not spawn unboundedly many new threads. This will be stated again when it has relevance in Chapter 3. This

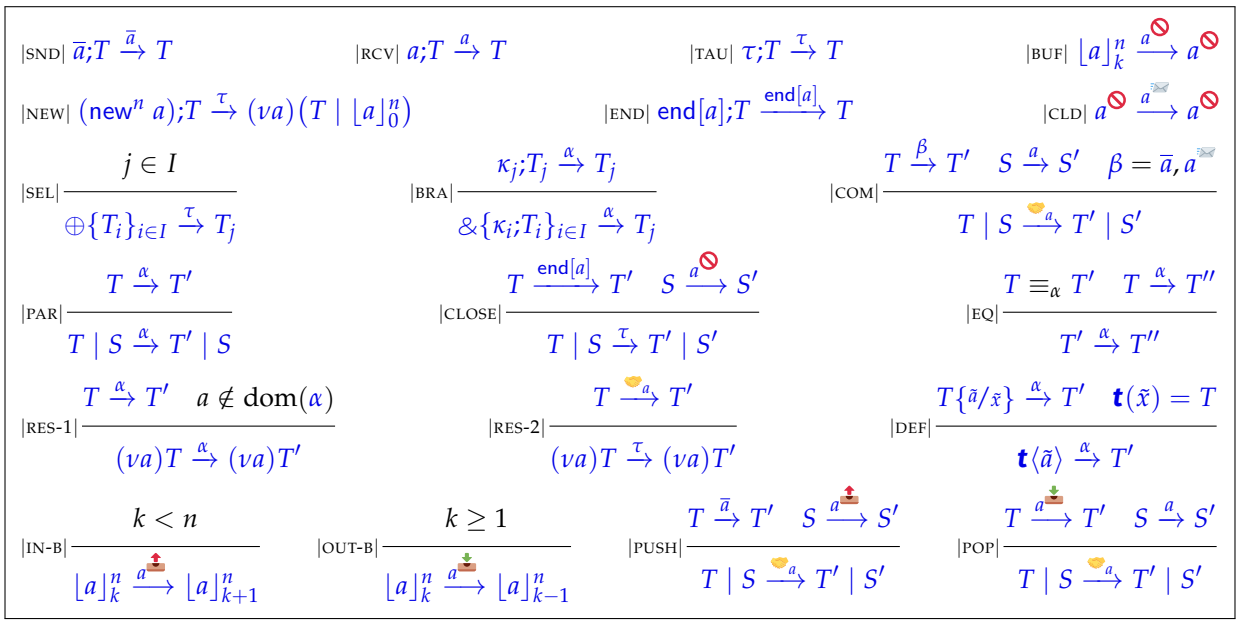


Figure 1.5: Rules for the LTS of the MiGo behavioural types

hypothesis is stated explicitly in the cited work, but we make it a general constraint to simplify a few definitions in the rest of this section. It is a necessary hypothesis for the automated verification of the types to work.

We define barbs for the types like we did for the processes. Those barbs are used for much the same reason — and to match the desired properties between types and processes. We use a different definition for the barbs in our own version of the framework in the following chapters, though with pretty much the same use as well as extended matching of verified properties between the formalisms.

Definition 1.8 (Barbs). We define the predicates $\kappa\downarrow_o$, $T\downarrow_o$ and $T\downarrow_{\tilde{o}}$ with $o, o_i \in \{a, \overline{a}, \tau_a, \text{end}[a], a^{\text{✉}}, a^{\text{📤}}, a^{\text{📥}}\}$.

$$a\downarrow_a \quad \overline{a}\downarrow_{\overline{a}} \quad \frac{\kappa\downarrow_o}{\kappa;T\downarrow_o} \quad \text{end}[a];T\downarrow_{\text{end}[a]} \quad a^\oslash\downarrow_{a^{\text{✉}}} \quad \frac{k < n}{\lfloor a \rfloor_k^n\downarrow_{a^{\text{📤}}}} \quad \frac{k \geq 1}{\lfloor a \rfloor_k^n\downarrow_{a^{\text{📥}}}}$$

$$\frac{T\downarrow_o}{T \mid T'\downarrow_o} \quad \frac{T\downarrow_o \quad a \notin \text{dom}(o)}{(\nu a)T\downarrow_o} \quad \frac{T\downarrow_o \quad T \equiv T'}{T'\downarrow_o} \quad \frac{T\{\tilde{a}/\tilde{x}\}\downarrow_o \quad \mathbf{t}_\mathbf{t}(\tilde{x}) = T}{\mathbf{t}\langle\tilde{a}\rangle\downarrow_o}$$

$$\frac{\forall i \in \{1,\ldots,n\} : \kappa_i\downarrow_{o_i}}{\&\{\kappa_i;T_i\}_{i\in\{1,\ldots,n\}}\downarrow_{\{o_1,\ldots,o_n\}}} \quad \frac{T\downarrow_a \quad T'\downarrow_o \quad o \in \{\overline{a}, a^{\text{✉}}, a^{\text{📥}}\}}{T \mid T'\downarrow_{\tau_a}} \quad \frac{T\downarrow_{\overline{a}} \quad T'\downarrow_{a^{\text{📤}}}}{T \mid T'\downarrow_{\tau_a}}$$

$$\frac{T\downarrow_o \quad \kappa_i\downarrow_{\overline{a}} \quad o \in \{a, a^{\text{📤}}\}}{T \mid \&\{\kappa_i;T'_i\}_{i\in I}\downarrow_{\tau_a}} \quad \frac{T\downarrow_o \quad \kappa_i\downarrow_a \quad o \in \{\overline{a}, a^{\text{✉}}, a^{\text{📥}}\}}{T \mid \&\{\kappa_i;T'_i\}_{i\in I}\downarrow_{\tau_a}}$$

We write $T\Downarrow_o$ if $T \to^* T'$ and $T'\downarrow_o$ with $o \in \{a, \overline{a}, \tau_a, \text{end}[a], a^{\text{✉}}, a^{\text{📤}}, a^{\text{📥}}\}$. ⌟

The barbs give the same capabilities as for the processes before, and in the same order. The first line defines capabilities of atomic actions and runtime constructs, and we then define capabilities for parallel compositions, restrictions and structurally congruent types as well as the type equivalent of a process call. Then, the external choice is given a set of capabilities like the select construct it represents, and we show synchronisation capabilities first between types that do not involve external choice, and then when an external choice is involved.

We give simplified versions of type liveness and channel safety from what appears in the 2017 work, as we work under the hypothesis that all programs and types are finite control. Their meaning is the same as that of liveness and channel safety for the processes. Note that we distinguish the entrypoint S of a type system $\boldsymbol{T}$ by naming it $\mathbf{t}_0()$. Similarly, X_0 is sometimes used to refer to the process variable of the program entrypoint.

Definition 1.9 (Liveness)**.** The system $\boldsymbol{T}$ is *live* if, for all T such that $\mathbf{t}_0\langle\rangle \to^* (\nu\tilde{a})T$,

(a) if $T\downarrow_a$ or $T\downarrow_{\bar{a}}$ then $T\Downarrow_{a}$;

(b) if $T\downarrow_{\tilde{o}}$ then $T\Downarrow_{a_i}$ for some $a_i \in \mathrm{dom}(\tilde{o})$. ⌟

Definition 1.10 (Channel Safety)**.** The system $\boldsymbol{T}$ is *channel safe* if, for all T such that $\mathbf{t}_0\langle\rangle \to^* (\nu\tilde{a})T$, if $T\downarrow_{a}$ then $\neg(T\Downarrow_{\mathsf{end}[a]})$ and $\neg(T\Downarrow_{\bar{a}})$. ⌟

The typing rules are shown in Figure 1.6 and are mostly untouched, barring minor corrections like the forgotten universal quantification in ⟨DEF⟩. In the typing judgements, Γ is a typing environment that keeps track of information on payload types of channels, types of bound communication variables as well as recursion variables — really process call variables, as there is no formal recursion notation in the system, but recursion is easily achievable by doing a recursive call to a named process within itself.

The judgements match expected typing rules given the process calculus that is MiGo and the types defined here, and come in three varieties:

$\Gamma \vdash P \blacktriangleright T$ types basic constructs outside of runtime, including process call and channel creation, the latter excluding the channel name from the typing context Γ as a restriction would;

$\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}$ types a complete program, giving proper typing to each named process and verifying the entrypoint has no undefined process call; and

$\Gamma \vdash_B P \blacktriangleright T$ types the processes during runtime, keeping track of free names of runtime channels (annotated in purple) during the construction of the judgement tree: names are added when channel buffers are added to the composition, and removed from the annotation when they are restricted. Note the removal shows as $B\backslash c$ instead of $B\backslash\{c\}$ in the cited material, which we fix in our contribution.

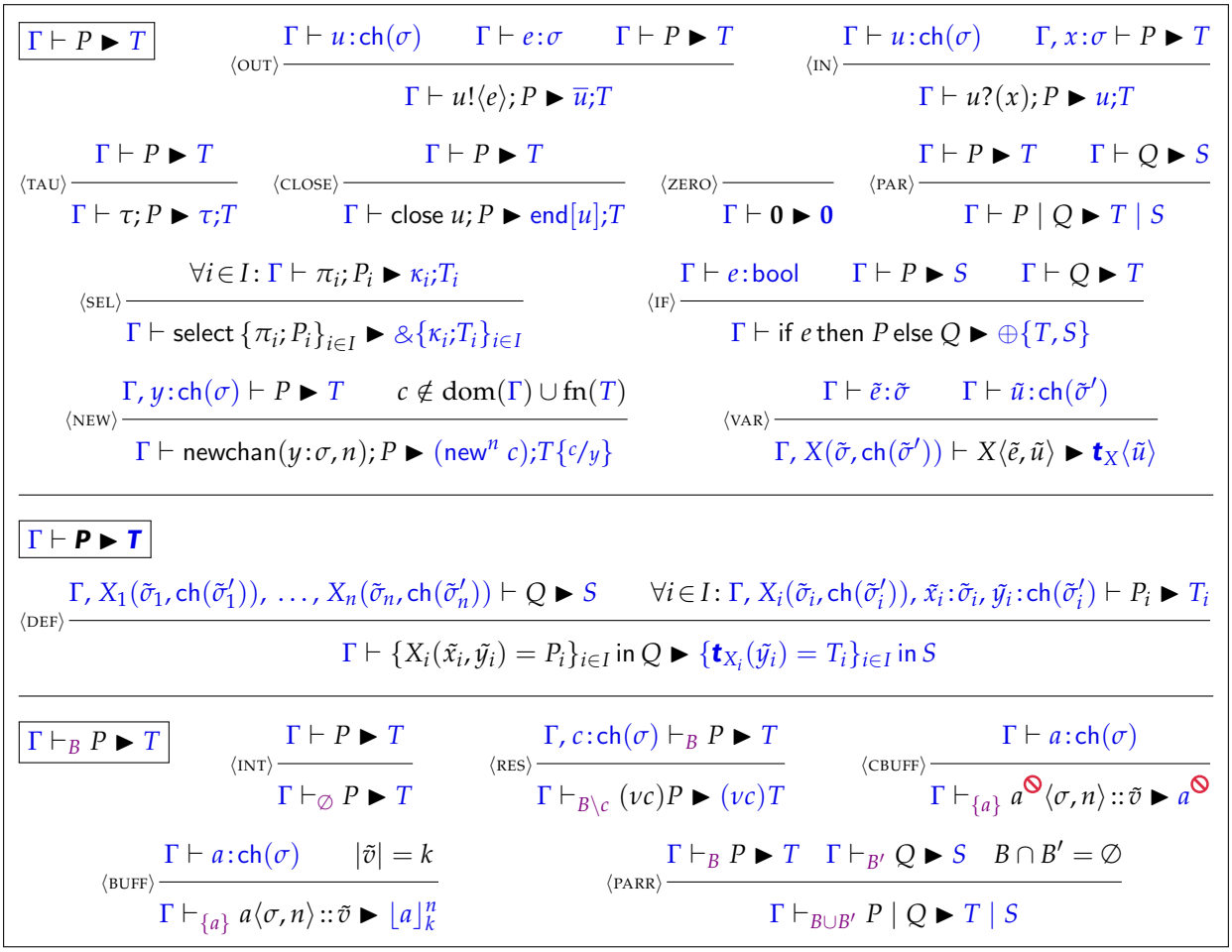


Figure 1.6: Typing rules for MiGo

Many more properties from the works cited in this chapter have inspired and served as a base for our own work, including a subject reduction theorem, convergence of properties of liveness and safety between types and processes, and restrictions on the subset of processes where these convergences apply. However, as the changes to them are minimal, we do not include them in this context chapter. They will instead be introduced duly in their own time in the following chapters, as well as a quick overview of the original tools in the penultimate chapter of this part — where we present our own tool.

2 An Extended π-calculus-inspired Abstraction of Go: MiGo$^+$

This chapter describes MiGo$^+$, an extension of MiGo shown in the previous chapter. MiGo$^+$ was created for the work we published in 2020 for ECOOP [GY20a], and we go over everything that was changed and added from MiGo for this framework, fixing some typos and errors from our publication in the process of this thesis. We also present unique extensions of our framework on the process side, particularly the characterisation of data races using a formalisation of the happens-before relation from Go in MiGo$^+$.

2.1 Extending MiGo with Memory Primitives: MiGo$^+$

We extend MiGo to account for more constructs of the Go language, namely grasp the notions of data races and memory safety. We dub our extended process calculus MiGo$^+$. The syntax for MiGo$^+$ is shown in Figure 2.1.

The parts in red are minor adaptations to the MiGo syntax to account for the new part of the syntax, while said new part is highlighted in purple. The new additions include shared memory variables, which are ranged over by m, m'; and (read-write) locks, also called mutexes (short for *mutual exclusion* locks), ranged over by l, l'; as well as their respective actions. In our contribution, we distinguish the names of channels between processes and types. We prefer the use of c for channels in untyped processes, and a for channels in types without a process — hence a was replaced by c in the process syntax for the sake of consistency. This is solely to distinguish between the definitions of processes and types initially, and the unification of those names happens as expected under a typing environment guided by the typing rules that we expand near the end of Section 3.1.

Locks and read-write locks are based on the Go constructs whose source is visible on the main repository for the language [Go 22a, Go 22b]. A *lock* in MiGo$^+$ is programmatically speaking a mutual exclusion lock in Go, i.e. a structure that must not be copied after first use — we instead pass a pointer to it to threads that must use it. While there is no specific mechanism to forbid copying a mutex, doing so will create an independent copy that is decoupled from its original, thus defeating the purpose of sharing the mutex between several processes. We work under the assumption that mutexes are passed by sharing a pointer to them in the rest of this thesis, in order to simplify this aspect and concentrate on other misuses that are common even under this assumption. We furthermore restrict to not send locks through channels to keep

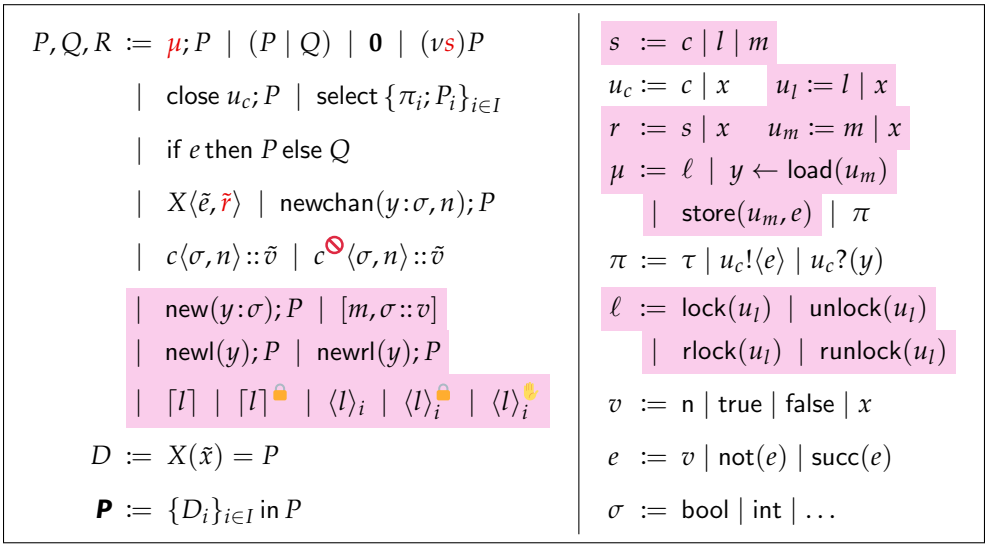


Figure 2.1: Syntax of MiGo$^+$ processes and programs

things simple, akin to the restriction on channel-passing we carry from the original works.

A lock has two operations: a thread can acquire it (lock action), and release it (unlock action). A thread is said to hold a lock when it acquired it and is yet to release it. Any specific lock can be held by at most one thread at a time. After creation using the newl construct, the runtime construct of a lock is represented in our model by its name between upwards brackets. If the lock is held by a process, a padlock is added in exponent to its runtime representation.

A read-write lock, similarly, should not be copied after use, and possesses the same properties as a lock plus the ability to be held in reading mode (rlock operation): the lock can be shared by several readers at the same time, excluding any writers from accessing it until all readers have released their hold on that lock (each one using runlock). After creation with the newrl construct, the runtime construct is represented in our model by its name between angled brackets, and the number of readers currently holding the read-write lock is annotated by index counter i. If a writer requested the read-write lock and readers still hold it, it is put in waiting mode, where new readers cannot take hold of it, denoted by a hand in exponent after the runtime representation. If a writer currently has hold of the read-write lock, we represent it with a padlock akin to a regular lock.

Shared variables are named areas in the store, holding values of predefined types and able to be read (using the load action) and written (using the store action) by any process within their scope. We do not aim at a specific package in the Go implementation for this construct, as it covers a wide range of ground types in our case and serves as an abstraction of the shared access to memory of those types in Go. After creation using the new construct, the runtime store for shared memory primitives is represented in our model by its name, type and value between square brackets. The name and type are separated by a comma, and the

```go
func main() {
  var x int
  m := new(sync.RWMutex)
  go f(m, &x)
  m.RLock()   // acquire the lock for reading
  x += 10     // write not protected by the lock
  m.RUnlock() // release the read-lock
  m.Lock()    // acquire the lock for writing
  fmt.Println("x is", x)
  m.Unlock()  // release the write-lock
}

func f(m *sync.RWMutex, ptr *int) {
  m.RLock()
  *ptr += 20  // write not protected by the lock
  m.RUnlock()
}
```

```go
func main() {
  var x int
  m := new(sync.RWMutex)
  go f(m, &x)
  m.Lock()    // acquire the lock for writing
  x += 10     // protected by the lock
  m.Unlock()  // release the write-lock
  m.RLock()   // acquire the lock for reading
  fmt.Println("x is", x)
  m.RUnlock() // release the read-lock
}

func f(m *sync.RWMutex, ptr *int) {
  m.Lock()
  *ptr += 20  // protected by the lock
  m.Unlock()
}
```

Figure 2.2: Go program with RWMutex: unsafe (left) and safe (right)

value is further separated from the type by a double colon.

Runtime constructs and variable names are bundled in several meta-names: s can be a channel, mutex or shared variable name; u_c (resp. u_l, u_m) can be a channel (resp. mutex, shared variable) name or a local variable; and r can be all the above (channel, mutex or shared variable name, or a local variable). Atomic actions on runtime constructs are also bundled in meta-names: π is a channel action (send or receive) or a tau-action; ℓ is a (read-write) lock action (lock, unlock, rlock or runlock); and μ is a memory action (load or store), or a channel or mutex action, or a tau-action. Values v can be integers, booleans or local variables; expressions e can be values, or negations of expressions, or successors of expressions; and we use σ to denote usual ground types for channel payloads and shared variables, including booleans and integers.

We remind the syntax of MiGo$^+$ processes contains: an action continued by a process, parallel composition, the inactive process, restriction, the closing of a channel, a selection between several processes guarded by a channel action or a tau-action, conditional constructs using if-then-else, (potentially recursive) process calls, the creation of new channels (resp. shared variables, lock, read-write locks), and the runtime constructs for (closed) channels (resp. shared variables, and locks in their different states detailed above). Definitions D are assignations from a process variable and argument names to a process; and finally a top-level process $\boldsymbol{P}$ is a set of definitions bound to a process.

Our running example is represented in Figure 2.2 (an exact copy of Figure 1), with a safe version on the right; and an unsafe version on the left that misuses the read-write lock in the subprocess.

Example 2.1 (Processes from Figure 2.2)**.** The following processes represent respectively the unsafe code

$$\mathrm{fn}(\,(P \mid Q)) = \mathrm{fn}(P) \cup \mathrm{fn}(Q) \qquad \mathrm{fn}(\mathbf{0}) = \emptyset \qquad \mathrm{fn}\big((\nu s)P\big) = \mathrm{fn}(P) \setminus \{s\}$$

$$\mathrm{fn}\big(X\langle \tilde{e}, \tilde{r}\rangle\big) = \mathrm{fn}(\tilde{e}) \cup \mathrm{fn}(\tilde{r}) \qquad \mathrm{fn}(\mathsf{close}\ u_c; P) = \mathrm{fn}(P) \cup \{u_c\}$$

$$\mathrm{fn}\big(\mathsf{select}\ \{\pi_i; P_i\}_{i\in I}\big) = \bigcup_{i\in I} \mathrm{fn}(\pi_i; P_i) \qquad \mathrm{fn}(\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q) = \mathrm{fn}(e) \cup \mathrm{fn}(P) \cup \mathrm{fn}(Q)$$

$$\mathrm{fn}\big(\mathsf{newchan}(y{:}\sigma, n); P\big) = \mathrm{fn}\big(\mathsf{new}(y{:}\sigma); P\big) = \mathrm{fn}\big(\mathsf{newl}(y); P\big) = \mathrm{fn}\big(\mathsf{newrl}(y); P\big) = \mathrm{fn}(P) \setminus \{y\}$$

$$\mathrm{fn}\big(c\langle \sigma, n\rangle :: \tilde{v}\big) = \mathrm{fn}\Big(c^{\oslash}\langle \sigma, n\rangle :: \tilde{v}\Big) = \{c\} \cup \mathrm{fn}(\tilde{v}) \qquad \mathrm{fn}(\,[m, \sigma :: v]) = \{m\} \cup \mathrm{fn}(v)$$

$$\mathrm{fn}(\lceil l \rceil) = \mathrm{fn}\Big(\lceil l \rceil^{\text{🔒}}\Big) = \mathrm{fn}(\langle l \rangle_i) = \mathrm{fn}\Big(\langle l \rangle_i^{\text{🔒}}\Big) = \mathrm{fn}\Big(\langle l \rangle_i^{\text{✋}}\Big) = \{l\}$$

$$\mathrm{fn}\big(y \leftarrow \mathsf{load}(u_m); P\big) = \big(\mathrm{fn}(P) \cup \{u_m\}\big) \setminus \{y\} \qquad \mathrm{fn}\big(\mathsf{store}(u_m, e); P\big) = \mathrm{fn}(P) \cup \mathrm{fn}(e) \cup \{u_m\}$$

$$\mathrm{fn}(\tau; P) = \mathrm{fn}(P) \qquad \mathrm{fn}\big(u_c!\langle e\rangle; P\big) = \mathrm{fn}(P) \cup \mathrm{fn}(e) \cup \{u_c\} \qquad \mathrm{fn}\big(u_c?(y); P\big) = \big(\mathrm{fn}(P) \cup \{u_c\}\big) \setminus \{y\}$$

$$\mathrm{fn}\big(\mathsf{lock}(u_l); P\big) = \mathrm{fn}\big(\mathsf{unlock}(u_l); P\big) = \mathrm{fn}\big(\mathsf{rlock}(u_l); P\big) = \mathrm{fn}\big(\mathsf{runlock}(u_l); P\big) = \mathrm{fn}(P) \cup \{u_l\}$$

$$\mathrm{fn}(\mathsf{n}) = \mathrm{fn}(\mathsf{true}) = \mathrm{fn}(\mathsf{false}) = \emptyset \qquad \mathrm{fn}(x) = \{x\} \qquad \mathrm{fn}\big(\mathsf{not}(e)\big) = \mathrm{fn}\big(\mathsf{succ}(e)\big) = \mathrm{fn}(e)$$

Figure 2.3: Definition of the set of free names for MiGo$^+$ processes

from the left side of Figure 2.2 and the safe code from the right side of that same figure. We separate the `main` function into two parts: the part instantiating the variable and lock and spawning the subroutine is what constitutes the entrypoint X_0, which then spawns both the continuation of `main`, which we name P, and the process representing the function `f`, which we call Q.

$$\boldsymbol{P}_{\mathsf{race}} \coloneqq \left\{ \begin{array}{ll} X_0 & = \mathsf{new}(x{:}\mathsf{int}); \mathsf{newrl}(l); (P\langle x, l\rangle \mid Q\langle x, l\rangle) \\ P(y,z) & = \mathsf{rlock}(z); t_1 \leftarrow \mathsf{load}(y); \mathsf{store}(y, t_1 + 10); \mathsf{runlock}(z); \\ & \quad\ \mathsf{rlock}(z); t_2 \leftarrow \mathsf{load}(y); \tau; \mathsf{runlock}(z); \mathbf{0} \\ Q(y,z) & = \mathsf{rlock}(z); t_0 \leftarrow \mathsf{load}(y); \mathsf{store}(y, t_0 + 20); \mathsf{runlock}(z); \mathbf{0} \end{array} \right\} \mathsf{in}\ X_0\langle\rangle$$

$$\boldsymbol{P}_{\mathsf{safe}} \coloneqq \left\{ \begin{array}{ll} X_0 & = \mathsf{new}(x{:}\mathsf{int}); \mathsf{newrl}(l); (P\langle x, l\rangle \mid Q\langle x, l\rangle) \\ P(y,z) & = \mathsf{lock}(z); t_1 \leftarrow \mathsf{load}(y); \mathsf{store}(y, t_1 + 10); \mathsf{unlock}(z); \\ & \quad\ \mathsf{rlock}(z); t_2 \leftarrow \mathsf{load}(y); \tau; \mathsf{runlock}(z); \mathbf{0} \\ Q(y,z) & = \mathsf{lock}(z); t_0 \leftarrow \mathsf{load}(y); \mathsf{store}(y, t_0 + 20); \mathsf{unlock}(z); \mathbf{0} \end{array} \right\} \mathsf{in}\ X_0\langle\rangle$$

⌟

We define the free names of processes, $\mathrm{fn}(P)$, in Figure 2.3. Their primary use is to make sure several actions are capture-avoiding, including the scope opening of a restriction and the creation of a new name.

The structural congruence for the extended syntax is in Figure 2.4, with the same colour-coding as above. This structural congruence includes α-equivalence, which is standard in process calculi.

The labels used by the LTS (α, β), those used by the happens-before relation and barbs (o, o'), and the occurrence indices ι, ι' for data writing actions, are defined in Figure 2.5.

$$P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad P \mid \mathbf{0} \equiv P \qquad (\nu s)\mathbf{0} \equiv \mathbf{0}$$
$$(\nu s)(\nu s')P \equiv (\nu s')(\nu s)P \qquad P \mid (\nu s)Q \equiv (\nu s)\,(P \mid Q)\ \ (s \notin \mathrm{fn}(P)) \qquad P \equiv_\alpha Q \implies P \equiv Q$$
$$(\nu c)c\langle \sigma, n\rangle :: \tilde{v} \equiv \mathbf{0} \qquad (\nu c)c^{\oslash}\langle \sigma, n\rangle :: \tilde{v} \equiv \mathbf{0} \qquad (\nu m)\,[m, \sigma :: v] \equiv \mathbf{0}$$
$$(\nu l)\lceil l \rceil \equiv \mathbf{0} \qquad (\nu l)\lceil l \rceil^{🔒} \equiv \mathbf{0} \qquad (\nu l)\langle l \rangle_i \equiv \mathbf{0} \qquad (\nu l)\langle l \rangle_i^{🔒} \equiv \mathbf{0} \qquad (\nu l)\langle l \rangle_i^{✋} \equiv \mathbf{0}$$

Figure 2.4: Structural congruence rules for MiGo$^+$

$$\alpha, \beta ::= o_\mathsf{p} \mid o_\mathsf{d}, e \qquad \iota ::= * \mid 1.\iota \mid 2.\iota \qquad o ::= o_\mathsf{d} \mid o_\mathsf{p}$$
$$o_\mathsf{d} ::= \overline{c} \mid c \mid c^{📤} \mid c^{📥} \mid c^{📨} \mid \mathsf{r}\langle m\rangle \mid \overline{\mathsf{r}\langle m\rangle} \mid (\mathsf{w}\langle m\rangle, \iota) \mid \overline{\mathsf{w}\langle m\rangle}$$
$$o_\mathsf{p} ::= \mathsf{end}[c] \mid c^{\oslash} \mid \mathsf{l}\langle l\rangle \mid \mathsf{ul}\langle l\rangle \mid \mathsf{rl}\langle l\rangle \mid \mathsf{rul}\langle l\rangle \mid l^{🔒} \mid l^{🔑} \mid l^{📖} \mid l^{📕} \mid l^{✋} \mid 🤝_s \mid \tau$$

Figure 2.5: Labels used by the LTS of MiGo$^+$, barbs and happens-before relation

In particular, the data-adjacent labels o_d correspond, in order, to: sending over a channel, receiving over a channel, pushing a value to a channel queue, popping a value from a channel queue, popping data from a closed channel's queue, reading a shared variable, the runtime construct answering a read request on a shared variable, writing a shared variable (along with the position of the write action), and the runtime construct answering a write request on a shared variable.

The data-independent labels o_p correspond, in order, to: requesting the closure of a channel, the runtime construct of a channel being closed, locking, unlocking read-locking and read-unlocking a (read-write) lock, a (read-write) lock being locked for writing, a locked (read-write) lock being unlocked, a read-write lock being locked for reading, a read-write lock being released from reading, a read-write lock being put to wait by a write request, a synchronisation over runtime construct s, and a tau-action.

We define the domain of a label (resp. action) $\mathrm{dom}(\alpha)$ (resp. $\mathrm{dom}(o)$), as follows:

Definition 2.2 (Label Domain)**.**

$$\mathrm{dom}((o_\mathsf{d}, e)) = \mathrm{dom}(o_\mathsf{d}) \cup \mathrm{fn}(e) \qquad \mathrm{dom}(\mathsf{r}\langle m\rangle) = \mathrm{dom}\left(\overline{\mathsf{r}\langle m\rangle}\right) = \mathrm{dom}((\mathsf{w}\langle m\rangle, \iota)) = \mathrm{dom}\left(\overline{\mathsf{w}\langle m\rangle}\right) = \{m\}$$
$$\mathrm{dom}(\overline{c}) = \mathrm{dom}(c) = \mathrm{dom}\left(c^{📤}\right) = \mathrm{dom}\left(c^{📥}\right) = \mathrm{dom}\left(c^{📨}\right) = \mathrm{dom}(\mathsf{end}[c]) = \mathrm{dom}\left(c^{\oslash}\right) = \{c\}$$
$$\mathrm{dom}(\mathsf{l}\langle l\rangle) = \mathrm{dom}(\mathsf{ul}\langle l\rangle) = \mathrm{dom}(\mathsf{rl}\langle l\rangle) = \mathrm{dom}(\mathsf{rul}\langle l\rangle) = \{l\} \qquad \mathrm{dom}(\tau) = \emptyset; \qquad \mathrm{dom}(🤝_s) = \{s\}$$
$$\text{and} \qquad \mathrm{dom}\left(l^{🔒}\right) = \mathrm{dom}\left(l^{🔒}\right) = \mathrm{dom}\left(l^{🔑}\right) = \mathrm{dom}\left(l^{📖}\right) = \mathrm{dom}\left(l^{📕}\right) = \mathrm{dom}\left(l^{✋}\right) = \{l\}$$

Occurrence indices ι are constructed with the use of left and right shifting operators, which we define in Definition 2.3 below. The intuition behind those operators and the occurrence indice itself is to represent a path through the parallel constructs in a process.

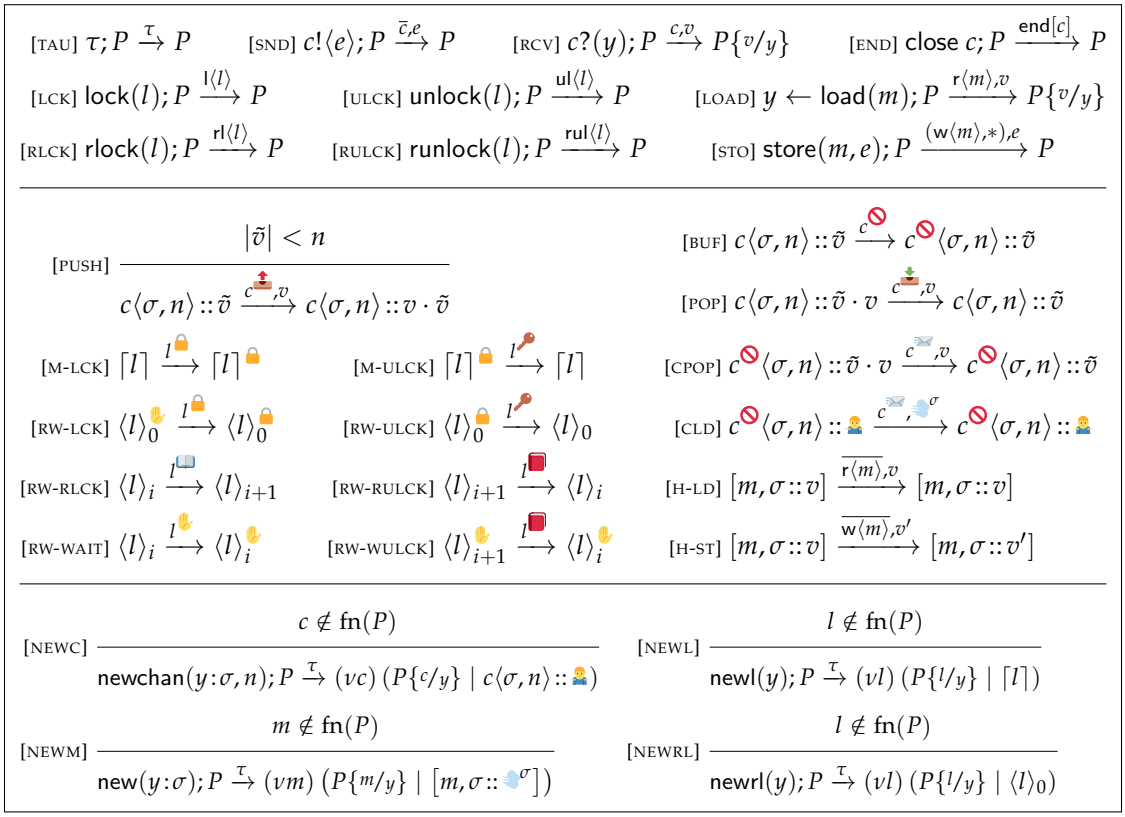


Figure 2.6: LTS rules for atomic actions (top), stores (middle) & store creation (bottom)

Definition 2.3. The left and right shifting operators on actions, left $(\cdot)$ and right $(\cdot)$, are defined as:

$$\mathsf{left}\left((\mathsf{w}\langle m\rangle,\iota)\right) = (\mathsf{w}\langle m\rangle, 1.\iota); \quad \mathsf{right}\left((\mathsf{w}\langle m\rangle,\iota)\right) = (\mathsf{w}\langle m\rangle, 2.\iota);$$

$$\text{and } \mathsf{left}\left(o_\mathsf{d}\right) = \mathsf{right}\left(o_\mathsf{d}\right) = o_\mathsf{d} \text{ if } o_d \neq (\mathsf{w}\langle m\rangle,\iota); \text{ with}$$

$$\mathsf{left}\left(o_\mathsf{d},e\right) = \mathsf{left}\left(o_\mathsf{d}\right),e; \quad \mathsf{right}\left(o_\mathsf{d},e\right) = \mathsf{right}\left(o_\mathsf{d}\right),e;$$

$$\text{and } \mathsf{left}\left(o_\mathsf{p}\right) = \mathsf{right}\left(o_\mathsf{p}\right) = o_\mathsf{p}.$$

⌟

By building occurrences with the two operators defined above in the construction of our LTS, we can track which process is writing to a shared variable. The occurrence in a write action represents the path in the syntactic tree of the program to the specific process containing the write action in question, which is useful to detect data races with the usual definition of two writes happening at the same time in different subprocesses. The LTS is shown in Figures 2.6 and 2.7, with Figure 2.6 showing the rules for the atoms of the MiGo$^+$ processes, while Figure 2.7 shows the rules for more advanced constructs and synchronisations of atomic rules. The LTS is shown in the form of processes P, Q only, and the set of definitions $\{D_i\}_{i\in I}$ of the top-level process definition $\boldsymbol{P}$ is assumed to be a given of the environment.

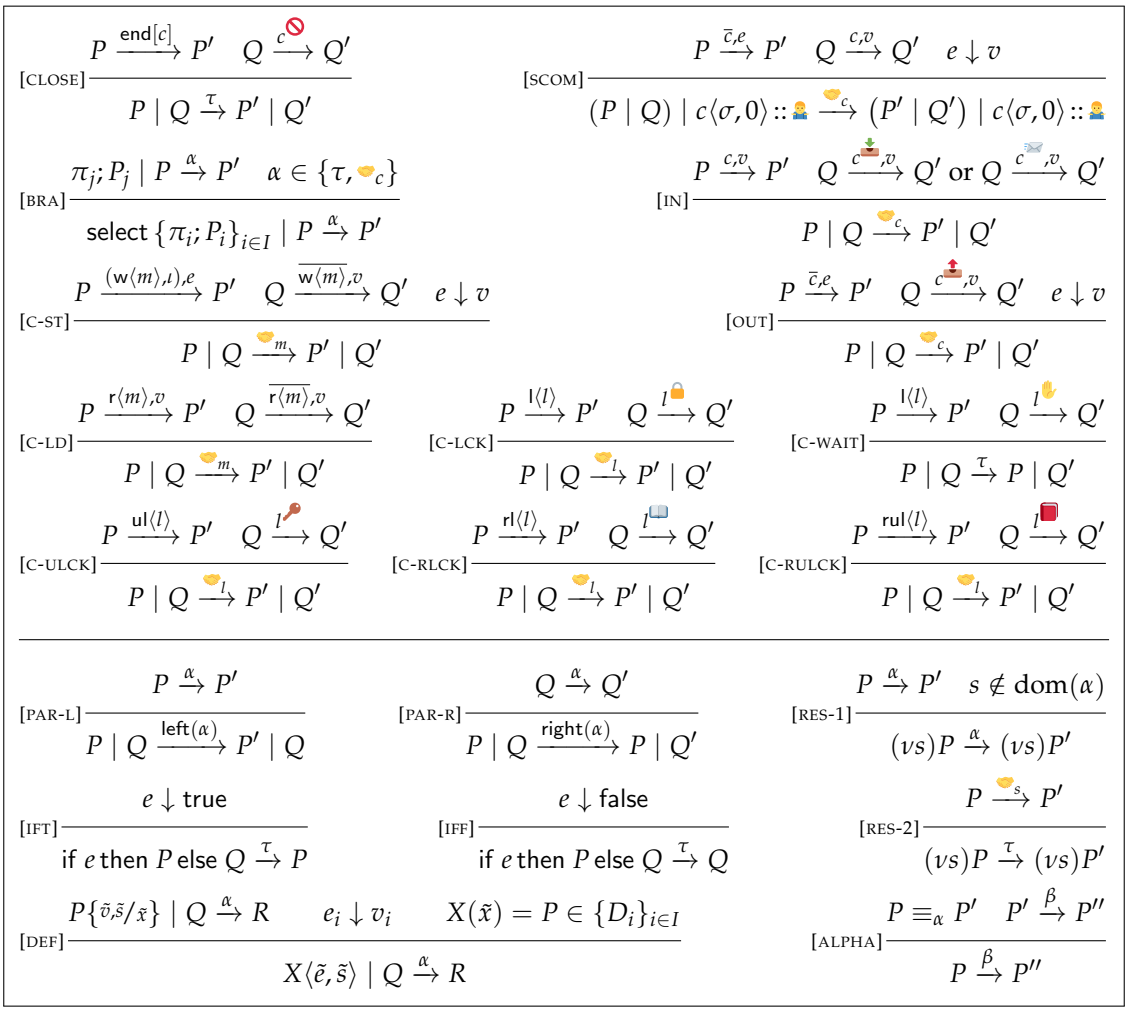


Figure 2.7: LTS rules for synchronisation (top) & structural constructs (bottom)

We use an LTS for our reduction in order to both simplify the definition of the barbs and allow to match the types more closely. We write $P \to P'$ for $P \xrightarrow{\tau} P'$ and $P \xrightarrow{s} P'$, up to structural congruence. This emulates the reduction shown in Figure 1.3, with extension for the added shared memory and mutual exclusion lock primitives. We also write $\to^*$ as the transitive and reflexive closure of $\to$, like in MiGo.

The atomic rules (top) in Figure 2.6 show the partial reduction of actions explicitly written by the developer and are mostly self-explanatory, with the actions that receive/load a value using substitution in the continuation in rules [RCV] and [LOAD]. Store rules (middle) show the partial reduction of runtime constructs and is also mostly self-explanatory: channels can push and pop values from their queue, become closed and send default values in case they are closed and have emptied their queue; locks can be locked and unlocked, read-write locks can do the same and also be locked for reading, and put to wait by a writer to stop accepting new readers until the writer can get hold of the lock and has subsequently released it; and shared memory can be written to or read by other threads.

Store creation rules (bottom) outline how those runtime constructs are created, with channels initially open and empty, locks initially unlocked, and shared memory initially holding the default value for its payload type. In particular, the process is subject to a substitution from the variable used to refer the name of the created construct to its reference name, and the runtime construct is spawned as a parallel process in its initial state.

Synchronisation actions (top) in Figure 2.7 show how different threads and runtime constructs interact with each other. In particular, channels work by pairs or sending and receiving side, at most one of which can be a buffer — the buffer does not do anything if the channel is of size 0, i.e. synchronous. Shared memory works by interaction between the store and a single thread, updating both according to the action taken. Lock interactions are between a single process and the runtime construct of the lock it talks with.

Remark 2.4 (Waiting stage for read-write locks). We put special emphasis on the notable case of rule [C-WAIT], which puts the read-write lock to wait *without* consuming the lock action, which is consumed when the actor actually gets hold of the lock later with [C-LCK]. ⌟

Construct rules (bottom) are here to handle (partial) actions with the rest of the MiGo$^+$ syntax. Specifically, it handles shifting the location index properly for observed write actions in parallel compositions, hiding synchronisations for restricted names, choosing the branch of conditional constructs, and the execution of process calls.

Remark 2.5. We give reduction up to α-equivalence because write operations are indexed to keep track of their location in the program. This is a bit more restrictive than in usual π-calculus-inspired frameworks, where it is standard to give reduction up to structural congruence, yet mandatory to handle the representation of data races later in our work.

We can however build a bisimulation-inspired transformation from our LTS to an LTS with the same labels except for the write action, for which we remove the occurrence index from the label. For example, if $P \xrightarrow{(\mathsf{w}\langle m\rangle,\iota),e} P'$ in the LTS presented in this work, then $P \xrightarrow{\mathsf{w}\langle m\rangle,e} P'$ in the occurrence-free LTS.

This LTS without occurrences is given up to structural congruence, by replacing the parallel rules by a single one and the alpha-equivalence rule by a structural congruence rule. Going from our LTS in this work to that occurrence-free LTS is a simple matter of erasing this information from the relevant labels; while going back is a matter of inserting the proper occurrence index to each respective write action, which can be determined by the structure of the process in the state before that transition.

In that sense, we can say that if $P \equiv Q$, P simulates Q in our system, using that occurrence-free LTS as an intermediate translation layer. This allows us to focus on our occurrence-aware system in order to keep track of write positions to determine data races, without having to dismiss the expressiveness of structural congruence altogether. ⌟

In the rest of this work, in order to simplify notations, we will often omit the definition part in front of redexes of top-level processes, e.g. we write $\boldsymbol{P} \rightarrow^* (\nu\tilde{s})Q$, denoting only the internal, evolving part of the redex, for $\boldsymbol{P} \rightarrow^* \{D_i\}_{i\in I}\,\mathsf{in}\,(\nu\tilde{s})Q$ — where $\boldsymbol{P} = \{D_i\}_{i\in I}\,\mathsf{in}\,P$ and $P \rightarrow^* (\nu\tilde{s})Q$ under the environment set by the definitions $\{D_i\}_{i\in I}$. The set of definitions $\{D_i\}_{i\in I}$ is kept implicit and assumed as a given of the working environment whenever we do this in a result or proof.

In order to have a better intuition on the occurrences, let us illustrate their use in process reduction:

Example 2.6 (Occurrences)**.** Let $P = \mathsf{store}(x,e); P'$, $Q = \mathsf{store}(x,e'); Q'$ and $R = z \leftarrow \mathsf{load}(x); R'$. It follows $P \xrightarrow{(\mathsf{w}\langle x\rangle,*),e} P'$, $Q \xrightarrow{(\mathsf{w}\langle x\rangle,*),e'} Q'$ and $R \xrightarrow{\mathsf{r}\langle x\rangle,v} R'\{v/z\}$.

If we compose P and Q, we use [PAR-L] and [PAR-R] to determine the new reductions:

$$P \mid Q \xrightarrow{(\mathsf{w}\langle x\rangle,1.*),e} P' \mid Q \qquad \mathsf{left}\,((\mathsf{w}\langle x\rangle,*),e) = (\mathsf{w}\langle x\rangle,1.*),e$$
$$P \mid Q \xrightarrow{(\mathsf{w}\langle x\rangle,2.*),e'} P \mid Q' \qquad \mathsf{right}\,((\mathsf{w}\langle x\rangle,*),e') = (\mathsf{w}\langle x\rangle,2.*),e'$$

Composing again, with R:

$$(P \mid Q) \mid R \xrightarrow{(\mathsf{w}\langle x\rangle,1.1.*),e} (P' \mid Q) \mid R \qquad \mathsf{left}\,((\mathsf{w}\langle x\rangle,1.*),e) = (\mathsf{w}\langle x\rangle,1.1.*),e$$
$$(P \mid Q) \mid R \xrightarrow{(\mathsf{w}\langle x\rangle,1.2.*),e'} (P \mid Q') \mid R \qquad \mathsf{left}\,((\mathsf{w}\langle x\rangle,2.*),e') = (\mathsf{w}\langle x\rangle,1.2.*),e'$$
$$(P \mid Q) \mid R \xrightarrow{\mathsf{r}\langle x\rangle,v} (P \mid Q) \mid R'\{v/z\} \qquad \mathsf{right}\,(\mathsf{r}\langle x\rangle,v) = \mathsf{r}\langle x\rangle,v$$

⌟

2.2 Desirable Properties of MiGo$^+$ with the Go Memory Model

We define barbed predicates for processes, covering the predicates introduced by the authors of the initial MiGo framework [LNTY17] (see Definition 1.5) and extending them to our new primitives. The definition is however shorter, reusing the labels defined before for the LTS. The barbed predicates express the

availability of a given action at a given point in the process execution, or in the potential futures of this execution.

Definition 2.7 (Process barbs)**.** If $P \xrightarrow{o_\mathsf{d}, e} P'$ then $P \downarrow_{o_\mathsf{d}}$; if $P \xrightarrow{o_\mathsf{p}} P'$ and $o_\mathsf{p} \neq \tau$ then $P \downarrow_{o_\mathsf{p}}$.
Moreover, if none of the actions guarding a select construct are τ actions, the select construct offers a set of actions as its barb; and the structural construction rules apply in the following way:

$$\frac{\forall j \in I,\ \pi_j; P_j \downarrow_{o_j}}{\mathsf{select}\ \{\pi_i; P_i\}_{i\in I} \downarrow_{\{o_i\}_{i\in I}}} \qquad \frac{P \downarrow_o \quad s \notin \mathrm{dom}(o)}{(\nu s)P \downarrow_o} \qquad \frac{P \downarrow_o \quad P \equiv_\alpha Q}{Q \downarrow_o}$$

$$\frac{P \downarrow_o}{P \mid Q \downarrow_{\mathsf{left}(o)}} \qquad \frac{P \downarrow_o}{Q \mid P \downarrow_{\mathsf{right}(o)}} \qquad \frac{P\{\tilde{v},\tilde{s}/\tilde{x}\} \mid Q \downarrow_o \quad X(\tilde{x}) = P \in \{D_i\}_{i\in I}}{X\langle \tilde{e}, \tilde{s}\rangle \mid Q \downarrow_o}$$

We write $P \Downarrow_o$ if $P \to^* P'$ and $P' \downarrow_o$. ⌟

We do not cover lone primitives in the barbs since they are not necessary for defining liveness, safety and data races, as those are defined in formed processes; and the barbs over lone primitives were used only to kickstart the definition of barbed predicates over processes in the seminal MiGo work [LNTY17].

We define liveness and safety for processes by encompassing the definitions of liveness and channel safety from the original MiGo work [LNTY17] (see Definitions 1.6 and 1.7) and adding liveness and safety for locks. Memory safety will be dealt with later on in the form of data race detection.

Definition 2.8 (Liveness)**.** The program $\boldsymbol{P}$ satisfies liveness if for all Q such that $\boldsymbol{P} \to^* (\nu\tilde{s})Q$:

(a) if $Q \downarrow_c$ or $Q \downarrow_{\bar{c}}$ then $Q \Downarrow_{c}$;

(b) if $Q \downarrow_{\tilde{o}}$ then $Q \Downarrow_{c_i}$ for some $c_i \in \mathrm{dom}(\tilde{o})$;

(c) if $Q \downarrow_{\mathsf{l}\langle l\rangle}$ or $Q \downarrow_{\mathsf{rl}\langle l\rangle}$ then $Q \Downarrow_{l}$. ⌟

Definition 2.9 (Channel and Lock Safety)**.** The program $\boldsymbol{P}$ satisfies safety — for channels and (rw)locks — if for all Q such that $\boldsymbol{P} \to^* (\nu\tilde{s})Q$:

(a) if $Q \downarrow_{a}$ then $\neg(Q \Downarrow_{\mathsf{end}[a]})$ and $\neg(Q \Downarrow_{\bar{a}})$;

(b) if $Q \downarrow_{\mathsf{ul}\langle l\rangle}$ then $Q \downarrow_{l}$;

(c) if $Q \downarrow_{\mathsf{rul}\langle l\rangle}$ then $Q \downarrow_{l}$. ⌟

For our next desirable property, we want to express that a process is data race free. We must first define a data race in our system, and we do that using the specification of the Go language, specifically its memory model [Go 22c]. We thus define a happens-before relation linking our system to the official Go specification.

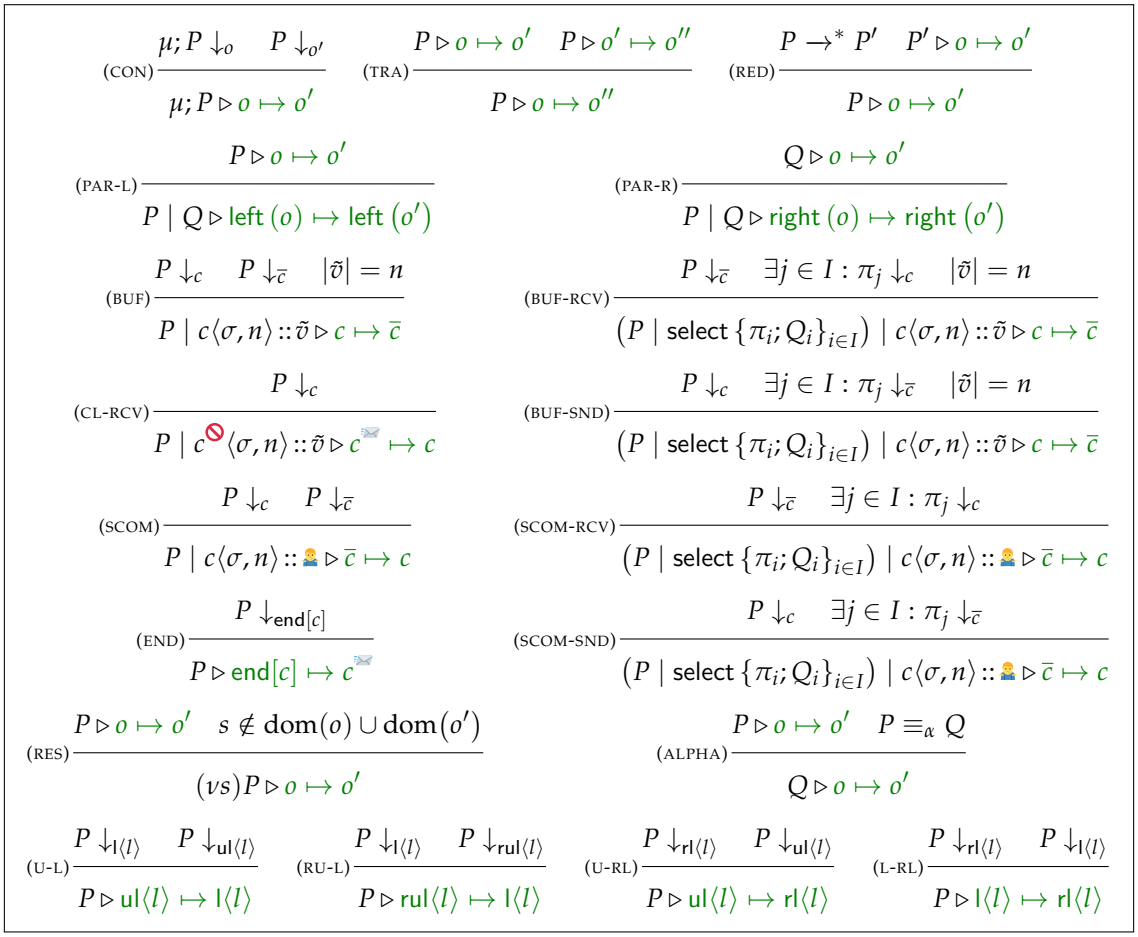


Figure 2.8: Rules for the happens-before relation

The happens-before relation is a binary relation between actions that is irreflexive, asymmetric and transitive. It is defined by the relation rules described in Figure 2.8. The statement is formed of a process on the left side, and the ordered actions on the right (in green in this work), linked by an arrow. These rules show the order in which available actions are set to happen when the processes execute, depending on the current state. The presence of the process in front of the relation serves to disambiguate contexts when needed. The left and right shifting operators are the same as defined previously on labels in Definition 2.3.

Rules (CON) and (TRA) express that the happens-before order in a linear thread is the construction order of that thread. Rule (RED) strengthens that by giving backwards inheritance, wherein happens-before order from possible reductions is inherited by the reducing program. There is however no forwards inheritance, in particular actions that are eliminated in the reduction $P \to^* P'$ are no longer in the relation for P', thus cannot be ordered in that relation.

Rules (PAR-L) and (PAR-R) serve to build the indexation for write actions on shared variables, much like the reduction rules from the LTS in Figure 2.7. Like in the LTS, this is needed to be able to distinguish between

different writes on the same shared variable, in order to characterise data races.

Rules (BUF), (BUF-RCV), (CL-RCV), (BUF-SND), (SCOM), (SCOM-RCV), (END) and (SCOM-SND) serve to express the channel synchronisation mechanisms described in the Go memory model [Go 22c]: a send must happen before the corresponding receive; if the channel has buffer size n, the k-th receive must happen before the $(k+n)$-th send; closing a channel happens before that channel is closed and default values can be received from it; and default values from a closed channel are emitted before the receive that reads them happens.

Rules (U-L), (RU-L), (U-RL) and (L-RL) specify the order between (read-)locks and (read-)unlocks in accordance with our reduction semantics and the Go memory model [Go 22c].

Finally, rules (RES) and (ALPHA) serve as structure rules to account for structural changes that do not affect visibility of the actions that are observed by the relation. Note that we could forego any form of relation on restriction completely, as the relation wants to look inside the process for ordering.

For the sake of simplicity, we consider MiGo$^+$ programs to be in the order compiled by the Go compiler, which *can* under some circumstances reorder some actions inside the same thread when they do not change interactions between different threads and calculation results — thus ignoring this potential reordering in our writing. This reordering in real-world Go programs ensures the inter-thread dependence relations are not affected, so the happens-before relation of actions in different threads stays identical, and those are the relations that we are interested in to detect blocking bugs and data races.

We define data races (and by extension data race freedom), from the happens-before relation, and prove that this definition coincides with the expected characterisation of data races with barbs — i.e. that two memory actions on the same shared variable including at least one write are available at the same time.

Definition 2.10 (Data Race)**.** Program $\boldsymbol{P}$ has a *data race* if there exist two distinct actions $o_1 \neq o_2$, two distinct occurrences $\iota \neq \iota'$, and $\boldsymbol{P} \rightarrow^* (\nu\tilde{s})P$, with $o_1 = (\mathsf{w}\langle m\rangle, \iota)$ and $o_2 \in \{(\mathsf{w}\langle m\rangle, \iota'),\ \mathsf{r}\langle m\rangle\}$, such that $P \Downarrow_{o_1}$, $P \Downarrow_{o_2}$, $\neg\,(P \triangleright o_1 \mapsto o_2)$ and $\neg\,(P \triangleright o_2 \mapsto o_1)$.
This states that at some point in the process, two memory actions are available without any clear ordering, thus are in a race. Program $\boldsymbol{P}$ is *data race free* if it has no data race. ⌟

Another way to characterise a data race is usually to check at every single point in the reduction if two (or more) distinct memory actions on the same shared variable can happen. We show that the two definitions are strictly equivalent, since we decided to follow the official Go specification to stay close to the language in our formalisation.

Theorem 2.11 (Equivalence of Data Race characterisations)**.** $\boldsymbol{P}$ has a data race if and only if there exists P such that $\boldsymbol{P} \rightarrow^* (\nu\tilde{s})P$ with $P \downarrow_{o_1}$, $P \downarrow_{o_2}$, $o_1 = (\mathsf{w}\langle m\rangle, \iota)$, $o_2 \in \{(\mathsf{w}\langle m\rangle, \iota'), \mathsf{r}\langle m\rangle\}$ and $\iota \neq \iota'$. ⌟

Proof. Suppose (1) $\boldsymbol{P} \rightarrow^* (\nu\tilde{s})P$; (2) $P \downarrow_{(\mathsf{w}\langle m\rangle, \iota)}$; (3) $P \downarrow_{o_2}$ with $o_2 \in \{(\mathsf{w}\langle m\rangle, \iota'), \mathsf{r}\langle m\rangle\}$; and (4) $\iota \neq \iota'$, and prove that $\neg(P \triangleright (\mathsf{w}\langle m\rangle, \iota) \mapsto o_2)$ and $\neg(P \triangleright o_2 \mapsto (\mathsf{w}\langle m\rangle, \iota))$.

By contradiction, suppose that $P \triangleright (\mathsf{w}\langle m\rangle, \iota) \mapsto o_2$ or $P \triangleright o_2 \mapsto (\mathsf{w}\langle m\rangle, \iota)$, and use induction on the happens-before relation. Ignoring α-equivalence, there are six rules that allow for read and write events to shared variables, so we only need to take these six into account. We can safely ignore α-equivalence since it does not change the structure of the process, only the names of bound names which cannot occur in the actions we are considering by construction. We remind that we do not work with structural congruence, since associativity and symmetry on parallel composition would change the positions of writes in the syntactic tree, and thus the occurrences referring to them.

We prove the case for $P \triangleright (\mathsf{w}\langle m\rangle, \iota) \mapsto o_2$, the other case is symmetric — contradicting (2) instead of (3).

- Rule (CON) is the base case, with $P = \mu; P'; \mu; P' \downarrow_{(\mathsf{w}\langle m\rangle, \iota)}$; and $P' \downarrow_{o_2}$. In this case, however, $\neg P \downarrow_{o_2}$, which contradicts hypothesis (3).
- Rule (RES) (resp. (RED)) supposes $P = (\nu r)P'$ (resp. $P \rightarrow^* P'$), with $P' \triangleright (\mathsf{w}\langle m\rangle, \iota) \mapsto o_2$, so by induction on P' we prove that $\neg P' \downarrow_{o_2}$, contradicting (3) again.
- Rule (PAR-L) (resp. (PAR-R)) supposes $P = P' \mid Q$ (resp. $P = Q \mid P'$) and $P' \triangleright (\mathsf{w}\langle m\rangle, \iota') \mapsto o_2'$ with $\iota = 1.\iota'$ and $o_2 = \mathsf{left}\,(o_2')$ (resp. $\iota = 2.\iota'$ and $o_2 = \mathsf{right}\,(o_2')$), so by induction on P' we prove that $\neg P' \downarrow_{o_2'}$, contradicting (3) again.
- Rule (TRA) supposes there is an intermediate action o, with $P \triangleright (\mathsf{w}\langle m\rangle, \iota) \mapsto o$ and $P \triangleright o \mapsto o_2$. We only need to prove $\neg P \downarrow_{o_2}$, which we get by induction on the second relation $P \triangleright o \mapsto o_2$, contradicting (3).

That concludes the first contradiction case, and the first direction of the proof.

Conversely, suppose (1) $\boldsymbol{P} \rightarrow^* (\nu \tilde{s})P$; (2) $P \Downarrow_{(\mathsf{w}\langle m\rangle, \iota)}$; (3) $P \Downarrow_{o_2}$ with $o_2 \in \{(\mathsf{w}\langle m\rangle, \iota'), \mathsf{r}\langle m\rangle\}$; (4) $\iota \neq \iota'$; (5) $\neg(P \triangleright (\mathsf{w}\langle m\rangle, \iota) \mapsto o_2)$; and (6) $\neg(P \triangleright o_2 \mapsto (\mathsf{w}\langle m\rangle, \iota))$, and prove that there exists $P \rightarrow^* P'$ such that $P' \downarrow_{(\mathsf{w}\langle m\rangle, \iota)}$ and $P' \downarrow_{o_2}$.

By contradiction, suppose $(*)$ that for all $P \rightarrow^* P'$ such that $P' \Downarrow_{(\mathsf{w}\langle m\rangle, \iota)}$ and $P' \Downarrow_{o_2}$, it holds that $\neg P' \downarrow_{(\mathsf{w}\langle m\rangle, \iota)}$ or $\neg P' \downarrow_{o_2}$; and prove there is a contradiction by induction on the structure of P' and the happens-before relation.

Having $P' \Downarrow_{(\mathsf{w}\langle m\rangle, \iota)}$ and $P' \Downarrow_{o_2}$ allows us to reduce until one of them becomes available, as by construction with Definition 2.7 each of those predicates mean that there exists a finite-length reduction path that leads to the action. We can thus safely assume that $P' \downarrow_{(\mathsf{w}\langle m\rangle, \iota)}$ and $\neg P' \downarrow_{o_2}$. The other case is symmetric — contradicting (6) instead of (5).

- If $P' = \mathsf{store}(m, e); P''$, with necessarily $P'' \Downarrow_{o_2}$, then by construction of the happens-before relation we have $P' \triangleright (\mathsf{w}\langle m\rangle, \iota) \mapsto o_2$, thus $P \triangleright (\mathsf{w}\langle m\rangle, \iota) \mapsto o_2$, which contradicts hypothesis (5).
- If $P' = P_1 \mid P_2$, then if both actions come from the same P_i, the induction on this P_i proves the contradiction; if not we can suppose $P_1 \downarrow_{(\mathsf{w}\langle m\rangle, \iota_1)}$ and $P_1 \mid P_2 \Downarrow_{o_2}$, with $\iota = 1.\iota_1$ and $\iota' = 2.\iota_2'$.

Then for all $P_2 \to^* P_2'$ such that $P_1 \mid P_2' \Downarrow_{o_2}$, we have $\neg P_2' \downarrow_{o_2'}$ with $o_2' \in \{(\mathsf{w}\langle m\rangle, \iota_2'), \mathsf{r}\langle m\rangle\}$ — otherwise the contradiction is on both actions happening at the same time.

Suppose $P_2 \to^* P_2' \not\to$ and $P_1 \mid P_2' \Downarrow_{o_2}$. Then $P_2' \downarrow_o$ with $o \in \{c, \bar{c}, \mathsf{l}\langle l\rangle, \mathsf{rl}\langle l\rangle\}$ one of the blocking actions. We can suppose it is the last blocking action before o_2 on this branch, as we can just induce on the chain of blocking actions if not. Because it is the last blocking action, it cannot be unlocked by P_1 unless it reduces the suspended $(\mathsf{w}\langle m\rangle, \iota_1)$ action — otherwise we can reach a state where both $(\mathsf{w}\langle m\rangle, \iota)$ and o_2 can execute at the same time.

By this, we have $P_1 \triangleright (\mathsf{w}\langle m\rangle, \iota_1) \mapsto o'$ and $P_2 \triangleright o \mapsto o_2'$, with o' the action that unblocks o in P_2: $o' = \bar{c}$ if $o = c$; $o' = c$ if $o = \bar{c}$; $o' = \mathsf{ul}\langle l\rangle$ if $o = \mathsf{rl}\langle l\rangle$; and $o' \in \{\mathsf{ul}\langle l\rangle, \mathsf{rul}\langle l\rangle\}$ if $o = \mathsf{l}\langle l\rangle$. If the blocking action was a channel action, it cannot be blocked by a non-empty buffer, as otherwise firing that buffer contradicts hypothesis $(*)$.

Since o' is unblocking o, we also have the corresponding rule from happens-before that holds at some point in the reduction, so $P_1 \mid P_2' \triangleright o' \mapsto o$, which in two applications of the transitivity rule gives us $P_1 \mid P_2' \triangleright (\mathsf{w}\langle m\rangle, \iota) \mapsto o_2$, thus $P' \triangleright (\mathsf{w}\langle m\rangle, \iota) \mapsto o_2$; and finally $P \triangleright (\mathsf{w}\langle m\rangle, \iota) \mapsto o_2$, contradicting hypothesis (5). □

We now illustrate the race in our running example:

Example 2.12 (Unsafe Process from Figure 2.2)**.** We show a possible reduction of $\boldsymbol{P}_{\mathsf{race}}$ in Example 2.1 that causes the unwanted race.

$$
\begin{aligned}
\boldsymbol{P}_{\mathsf{race}} \;=\; & \mathsf{new}(x:\mathsf{int}); \mathsf{newrl}(l); \left(\begin{array}{l} \mathsf{rlock}(l); t_1 \leftarrow \mathsf{load}(x); \mathsf{store}(x, t_1 + 10); \mathsf{runlock}(l); \\ \quad \mathsf{rlock}(l); t_2 \leftarrow \mathsf{load}(x); \tau; \mathsf{runlock}(l); \mathbf{0} \\ \mid \mathsf{rlock}(l); t_0 \leftarrow \mathsf{load}(x); \mathsf{store}(x, t_0 + 20); \mathsf{runlock}(l); \mathbf{0} \end{array}\right) \\
\to^2 (\nu x l) & \left(\begin{array}{l} \mathsf{rlock}(l); t_1 \leftarrow \mathsf{load}(x); \mathsf{store}(x, t_1 + 10); \mathsf{runlock}(l); \\ \quad \mathsf{rlock}(l); t_2 \leftarrow \mathsf{load}(x); \tau; \mathsf{runlock}(l); \mathbf{0} \\ \mid \mathsf{rlock}(l); t_0 \leftarrow \mathsf{load}(x); \mathsf{store}(x, t_0 + 20); \mathsf{runlock}(l); \mathbf{0} \mid [x, \mathsf{int} :: 0] \mid \langle l\rangle_0 \end{array}\right) \\
\to^6 (\nu x l) & \left(\begin{array}{l} \mathsf{store}(x, 10); \mathsf{runlock}(l); \mathsf{rlock}(l); t_2 \leftarrow \mathsf{load}(x); \tau; \mathsf{runlock}(l); \mathbf{0} \\ \mid \mathsf{store}(x, 20); \mathsf{runlock}(l); \mathbf{0} \mid [x, \mathsf{int} :: 0] \mid \langle l\rangle_2 \end{array}\right) = (\nu x l) P'
\end{aligned}
$$

The first line is obtained by rewriting $\boldsymbol{P}_{\mathsf{race}}$ using the process definition structure and rule [DEF], that tells us the rewritten program and the program with calls share the same reductions. Then we have $P' \downarrow_{(\mathsf{w}\langle x\rangle, 1.1.1.*)}$ and $P' \downarrow_{(\mathsf{w}\langle x\rangle, 1.1.2.*)}$, hence $\boldsymbol{P}_{\mathsf{race}}$ has a data race.

On the other hand, $\boldsymbol{P}_{\mathsf{safe}}$ is data race free, which is ensured by checking every reduction chain of the process for the absence of data race. ⌟

3 Behavioural Types for MiGo$^+$

This chapter presents the behavioural types for MiGo$^+$, going over the additions and changes from Section 1.2, then the remarkable properties and theorems that we have demonstrated on our framework on typed processes. This sets the stage for automated verification of desired properties over specific subsets of MiGo$^+$ in the next chapter.

3.1 Behavioural Types for MiGo$^+$

The syntax for the types and the associated structural congruence are defined in Figure 3.1.

This syntax makes an abstraction of MiGo$^+$, foregoing the value of most data, including the condition on which conditional constructs are branched, giving us the same selection operation $\oplus\{T_i\}_{i\in I}$ as in Section 1.2. Additions and changes are mostly mirror of those from the syntax of processes. Like for MiGo$^+$, we have $\equiv$ include $\equiv_\alpha$, which is standard in process calculi. We remind that m, m' are used to represent shared memory variables, and in the types we simplify the runtime construct even further, with $m^\blacksquare$ representing the cell used by name m.

We also define the set of free names for types, in the same way as the processes, in Figure 3.2.

Example 3.1. The unsafe program from the left side of Figure 2.2, modelled by process $\boldsymbol{P}_{\mathsf{race}}$ in Example 2.1, corresponds to the following type:

$$\boldsymbol{T}_{\mathsf{race}} := \left\{ \begin{array}{ll} \boldsymbol{t}_0() & = \mathsf{new}(x);\mathsf{newrl}(l);(\boldsymbol{t}_P\langle x,l\rangle \mid \boldsymbol{t}_Q\langle x,l\rangle) \\ \boldsymbol{t}_P(y,z) & = \mathsf{rl}(z);\mathsf{r}(y);\mathsf{w}(y);\mathsf{rul}(z);\mathsf{rl}(z);\mathsf{r}(y);\tau;\mathsf{rul}(z);\boldsymbol{0} \\ \boldsymbol{t}_Q(y,z) & = \mathsf{rl}(z);\mathsf{r}(y);\mathsf{w}(y);\mathsf{rul}(z);\boldsymbol{0} \end{array} \right\} \text{ in } \boldsymbol{t}_0\langle\rangle$$

The safe version from the right side of Figure 2.2, modelled by process $\boldsymbol{P}_{\mathsf{safe}}$ in Example 2.1, corresponds to the type:

$$\boldsymbol{T}_{\mathsf{safe}} := \left\{ \begin{array}{ll} \boldsymbol{t}_0() & = \mathsf{new}(x);\mathsf{newrl}(l);(\boldsymbol{t}_P\langle x,l\rangle \mid \boldsymbol{t}_Q\langle x,l\rangle) \\ \boldsymbol{t}_P(y,z) & = \mathsf{l}(z);\mathsf{r}(y);\mathsf{w}(y);\mathsf{ul}(z);\mathsf{rl}(z);\mathsf{r}(y);\tau;\mathsf{rul}(z);\boldsymbol{0} \\ \boldsymbol{t}_Q(y,z) & = \mathsf{l}(z);\mathsf{r}(y);\mathsf{w}(y);\mathsf{ul}(z);\boldsymbol{0} \end{array} \right\} \text{ in } \boldsymbol{t}_0\langle\rangle$$

⌟

As in Section 1.2 for the behavioural types of the MiGo framework [LNTY17], our behavioural types use a labelled transition system for reduction. The labels for this LTS are shown in Figure 3.3.

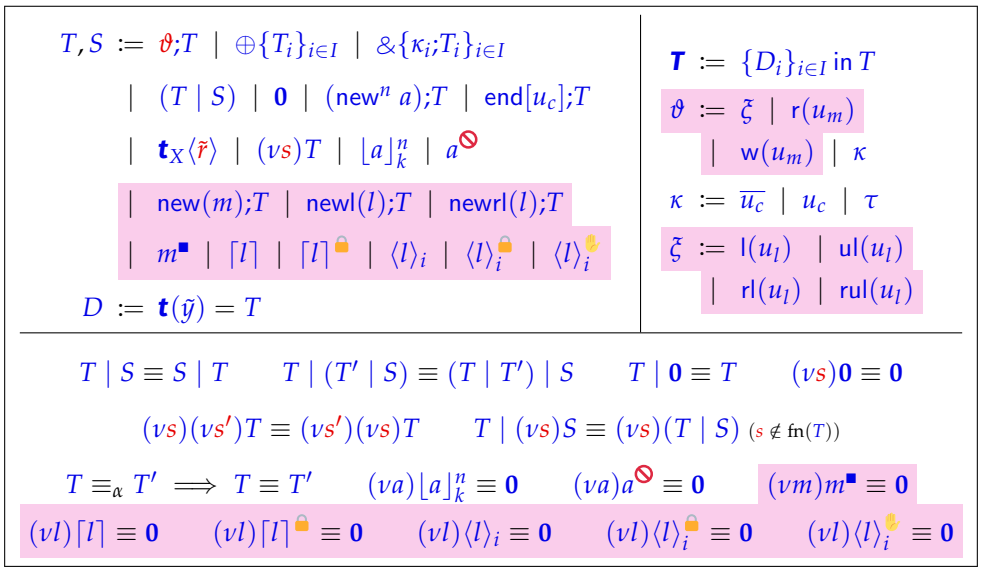


Figure 3.1: Syntax of the behavioural types for MiGo$^+$, and their structural congruence

$$\mathrm{fn}((T \mid S)) = \mathrm{fn}(T) \cup \mathrm{fn}(S) \qquad \mathrm{fn}(\mathbf{0}) = \emptyset \qquad \mathrm{fn}((\nu s)T) = \mathrm{fn}(T) \setminus \{s\} \qquad \mathrm{fn}(\mathbf{t}_X\langle\tilde{r}\rangle) = \mathrm{fn}(\tilde{r})$$
$$\mathrm{fn}(\mathsf{end}[u_c];T) = \mathrm{fn}(T) \cup \{u_c\} \qquad \mathrm{fn}(\&\{\kappa_i;T_i\}_{i\in I}) = \bigcup_{i\in I} \mathrm{fn}(\kappa_i;T_i) \qquad \mathrm{fn}(\oplus\{T_i\}_{i\in I}) = \bigcup_{i\in I} \mathrm{fn}(T_i)$$
$$\mathrm{fn}((\mathsf{new}^n\ a);T) = \mathrm{fn}(T) \setminus \{a\} \qquad \mathrm{fn}(\lfloor a \rfloor_k^n) = \{a\} \qquad \mathrm{fn}(a^{\oslash}) = \{a\} \qquad \mathrm{fn}(\mathsf{new}(m);T) = \mathrm{fn}(T) \setminus \{m\}$$
$$\mathrm{fn}(m^{\blacksquare}) = \{m\} \qquad \mathrm{fn}(\mathsf{newl}(l);T) = \mathrm{fn}(\mathsf{newrl}(l);T) = \mathrm{fn}(T) \setminus \{l\}$$
$$\mathrm{fn}(\lceil l \rceil) = \mathrm{fn}(\lceil l \rceil^{\text{🔒}}) = \mathrm{fn}(\langle l \rangle_i) = \mathrm{fn}(\langle l \rangle_i^{\text{🔒}}) = \mathrm{fn}(\langle l \rangle_i^{\text{✋}}) = \{l\}$$
$$\mathrm{fn}(\mathsf{r}(u_m);T) = \mathrm{fn}(\mathsf{w}(u_m);T) = \mathrm{fn}(T) \cup \{u_m\} \qquad \mathrm{fn}(\overline{u_c};T) = \mathrm{fn}(u_c;T) = \mathrm{fn}(T) \cup \{u_c\}$$
$$\mathrm{fn}(\mathsf{l}(u_l);T) = \mathrm{fn}(\mathsf{ul}(u_l);T) = \mathrm{fn}(\mathsf{rl}(u_l);T) = \mathrm{fn}(\mathsf{rul}(u_l);T) = \mathrm{fn}(T) \cup \{u_l\} \qquad \mathrm{fn}(\tau;T) = \mathrm{fn}(T)$$

Figure 3.2: The set of free names for behavioural types

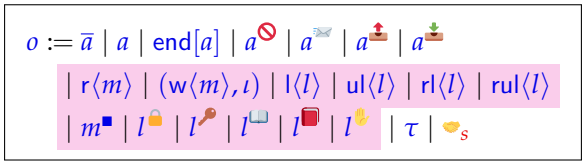


Figure 3.3: Labels for the LTS of the behavioural types

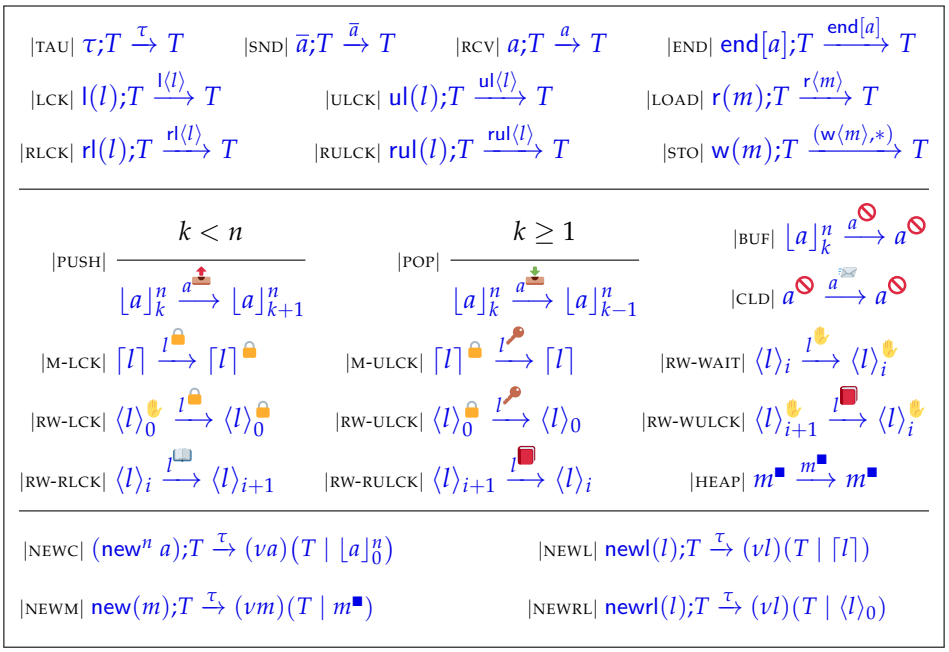


Figure 3.4: Types LTS Rules for atomic actions (top), stores (middle) & stores creation (bottom)

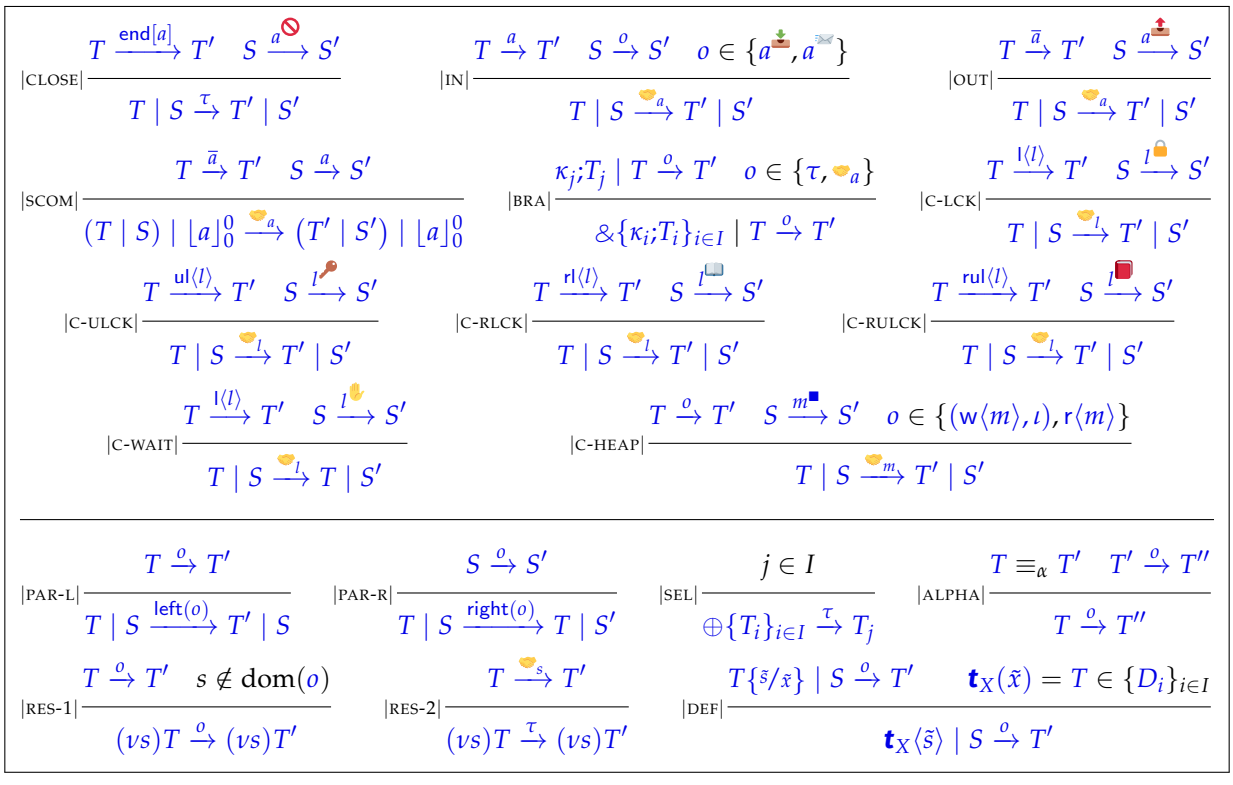


Figure 3.5: Types LTS rules for synchronisation (top) & constructs (bottom)

The LTS is described in Figures 3.4 and 3.5. The rules are separated and organised in the same way as the processes in the previous chapter. In particular, we find symmetry in the fact that |C-WAIT| does not consume the lock action, mirroring the LTS for MiGo$^+$ in Figure 2.7, as this synchronisation puts the runtime construct for the lock to wait for all current readers to be released in order to then allow the waiting writer to get the lock. We also define $\rightarrow$ and $\rightarrow^*$ in the same way as we defined $\rightarrow$ and $\rightarrow^*$ for the processes : $\rightarrow$ is used to denote $\xrightarrow{\tau}$ and $\xrightarrow{s}$ up to structural congruence, and $\rightarrow^*$ denotes the transitive and reflexive closure of $\rightarrow$. As for the processes, in order to simplify notations, we often omit the set of types definitions in the redex when writing the reduction of a top-level type, showing only the internal, evolving part of the types in the redex; and the set of types definitions is assumed as a given of the environment.

Example 3.2. Type $\boldsymbol{T}_{\mathsf{race}}$ in Example 3.1, matching process $\boldsymbol{P}_{\mathsf{race}}$ in Example 2.1 for the modelling of the unsafe version of Figure 2.2, can reduce in the following way (following the same reduction order as Example 2.12):

$$\begin{aligned}
\boldsymbol{T}_{\mathsf{race}} \quad &= \quad \mathsf{new}(x);\mathsf{newrl}(l);\left(\begin{array}{l}\mathsf{rl}(l);\mathsf{r}(x);\mathsf{w}(x);\mathsf{rul}(l);\mathsf{rl}(l);\mathsf{r}(x);\tau;\mathsf{rul}(l);\mathbf{0}\\ \mid \mathsf{rl}(l);\mathsf{r}(x);\mathsf{w}(x);\mathsf{rul}(l);\mathbf{0}\end{array}\right)\\
&\rightarrow^2 \quad (\nu x l)\left(\begin{array}{l}\mathsf{rl}(l);\mathsf{r}(x);\mathsf{w}(x);\mathsf{rul}(l);\mathsf{rl}(l);\mathsf{r}(x);\tau;\mathsf{rul}(l);\mathbf{0}\\ \mid \mathsf{rl}(l);\mathsf{r}(x);\mathsf{w}(x);\mathsf{rul}(l);\mathbf{0} \mid x^{\blacksquare} \mid \langle l\rangle_0\end{array}\right)\\
&\rightarrow^6 \quad (\nu x l)\left(\begin{array}{l}\mathsf{w}(x);\mathsf{rul}(l);\mathsf{rl}(l);\mathsf{r}(x);\tau;\mathsf{rul}(l);\mathbf{0}\\ \mid \mathsf{w}(x);\mathsf{rul}(l);\mathbf{0} \mid x^{\blacksquare} \mid \langle l\rangle_2\end{array}\right) = (\nu x l)T'
\end{aligned}$$

We note that T' corresponds with P', which has a data race in Example 2.12. ⌟

Like in the case of the processes, we redefine barbed predicates for types in a more straightforward way by reusing the labels for the LTS, and those predicates allow us to define the desired properties of types matching those of the processes.

Definition 3.3 (Types barbs)**.** If $T \xrightarrow{o} T'$ and $o \neq \tau$, then $T{\downarrow_o}$. Moreover, if none of the actions guarding a select construct are τ actions, the branch construct offers a set of actions as its barb; and the structural construction rules apply in the following way:

$$\frac{\forall j \in I,\ \kappa_j;T_j{\downarrow_{o_j}}}{\&\{\kappa_i;T_i\}_{i\in I}{\downarrow_{\{o_i\}_{i\in I}}}} \qquad \frac{T{\downarrow_o} \quad s \notin \mathrm{dom}(o)}{(\nu s)T{\downarrow_o}} \qquad \frac{T{\downarrow_o} \quad T \equiv_\alpha S}{S{\downarrow_o}}$$

$$\frac{T{\downarrow_o}}{T \mid S{\downarrow_{\mathsf{left}(o)}}} \qquad \frac{T{\downarrow_o}}{S \mid T{\downarrow_{\mathsf{right}(o)}}} \qquad \frac{T\{\tilde{s}/\tilde{x}\} \mid S{\downarrow_o} \quad \boldsymbol{t}_X(\tilde{x}) = T \in \{D_i\}_{i\in I}}{\boldsymbol{t}_X\langle\tilde{s}\rangle \mid S{\downarrow_o}}$$

We write $T{\Downarrow_o}$ if $T \rightarrow^* T'$ and $T'{\downarrow_o}$. ⌟

Desired properties are type liveness, type safety and data race freedom, like for processes, and have essentially the same meaning. We will use only the barb version for data races, as we do not define the happens-before relation on the types — more on that later when we reach the properties of typed MiGo$^+$.

Definition 3.4 (Liveness for Types)**.** Type $\boldsymbol{T}$ satisfies liveness if for all T such that $\boldsymbol{T} \to^* (\nu \tilde{s})T$:

(a) if $T\downarrow_c$ or $T\downarrow_{\bar{c}}$ then $T\Downarrow_{c}$;

(b) if $T\downarrow_{\tilde{o}}$ then $T\Downarrow_{c_i}$ for some $c_i \in \mathrm{dom}(\tilde{o})$;

(c) if $T\downarrow_{\mathsf{l}\langle l\rangle}$ or $T\downarrow_{\mathsf{rl}\langle l\rangle}$ then $T\Downarrow_{l}$. ⌟

Definition 3.5 (Channel and Lock Safety for Types)**.** Type $\boldsymbol{T}$ satisfies safety — for channels and (rw)locks — if for all T such that $\boldsymbol{T} \to^* (\nu \tilde{s})T$:

(a) if $T\downarrow_{a}$ then $\neg(T\Downarrow_{\mathsf{end}[a]})$ and $\neg(T\Downarrow_{\bar{a}})$;

(b) if $T\downarrow_{\mathsf{ul}\langle l\rangle}$ then $T\downarrow_{l}$;

(c) if $T\downarrow_{\mathsf{rul}\langle l\rangle}$ then $T\downarrow_{l}$. ⌟

Definition 3.6 (Data Race for Types)**.** Type $\boldsymbol{T}$ has a data race if there exists T such that $\boldsymbol{T} \to^* (\nu \tilde{s})T$ with $T\downarrow_{o_1}$ and $T\downarrow_{o_2}$, where $o_1 = (\mathsf{w}\langle m\rangle, \iota)$, $o_2 \in \{(\mathsf{w}\langle m\rangle, \iota'), \mathsf{r}\langle m\rangle\}$ and $\iota \neq \iota'$. $\boldsymbol{T}$ is data race free if it has no data race. ⌟

Figure 3.6 shows the typing rules, formally associating MiGo$^+$ constructs and their behavioural types counterparts in a contextualised environment. This type system forms the base for automated verification of the processes by verifying their abstraction as types, giving us information on the process with confidence under certain conditions described in the next section. As in the MiGo framework [LNTY17] and Section 1.2, there are three forms of typing judgement. $\Gamma \vdash P \blacktriangleright T$ types basic constructs outside of runtime with mostly simple matching, with the store creation typing rules excluding the type of the store from the context in the conclusion, as creation and restriction act the same, with one usually used for process definition and the other emulating name scopes at runtime after creation has happened. While we distinguish locks and read-write locks, the second variety must have access to basic lock actions too, hence the existence of the ⟨RL-SUB⟩ rule in our framework — that can be interpreted as a form of subtyping.

$\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}$ types a complete program, giving proper typing to each named process and verifying the entrypoint has no undefined process call.

Finally, $\Gamma \vdash_B P \blacktriangleright T$ types the processes during runtime, with the annotation of still open runtime names in purple on the judgement symbol to the left. This annotation is a set that grows as more runtime construct

are added to the context, and shrinks when those constructs are encapsulated in their due restriction. Well-formed typed processes during runtime will usually type with an empty set in this location: $\Gamma \vdash_\emptyset P \blacktriangleright T$.

Example 3.7. The processes and types from Examples 2.1 and 3.1, that represent our model and abstraction of Figure 2.2, can be matched by following their structure with the typing rules above. We respectively get that $\vdash \boldsymbol{P}_{\mathsf{safe}} \blacktriangleright \boldsymbol{T}_{\mathsf{safe}}$ for the safe version, and $\vdash \boldsymbol{P}_{\mathsf{race}} \blacktriangleright \boldsymbol{T}_{\mathsf{race}}$ for the unsafe version. ⌟

3.2 Properties of Typed MiGo$^+$

This section describes properties of typed MiGo$^+$ processes. We start with general properties that are frequently desired for type systems, including a subject reduction theorem. Following comes a progress theorem, that gives us the other direction of reduction between types and processes: a typed process with a type that reduces can also reduce, though not always according to the same path. We then finish with a series of theorems that relate desirable properties between types and processes, and the conditions needed for these relations to hold.

Note that, as stated earlier in this thesis, payload types for channels and shared variables can only be ground types, not other channels; this hypothesis is necessary for many theorems to hold, starting with *subject reduction*.

We state subject reduction first, for which we need to have a case for structural congruence of typed processes, that we prove with the subject congruence proposition.

Proposition 3.8 (Subject Congruence)**.** Suppose $\Gamma \vdash_B P \blacktriangleright T$ and $P \equiv P'$.
Then there exists $T' \equiv T$ such that $\Gamma \vdash_B P' \blacktriangleright T'$. ⌟

Proof. We prove the integrity case, regarding ⟨INT⟩, first. Suppose $\Gamma \vdash P \blacktriangleright T$ and $P \equiv P'$. Then from the typing judgments, either:

- $P = P' \mid \mathbf{0} \equiv P'$, then we can simply remove the last ⟨PAR⟩ rule along with its child ⟨ZERO⟩ rule: $T = T' \mid \mathbf{0} \equiv T'$.
- $P = P_1 \mid P_2 \equiv P_2 \mid P_1 = P'$. Then $T = T_1 \mid T_2$ and we can use $T' = T_2 \mid T_1 \equiv T$, the last rule of the typing judgment being ⟨PAR⟩ with the premises switched.
- $P = P_1 \mid (P_2 \mid P_3) \equiv (P_1 \mid P_2) \mid P_3 = P'$. Then $T = T_1 \mid (T_2 \mid T_3)$ and $T' = (T_1 \mid T_2) \mid T_3 \equiv T$; the

$\boxed{\Gamma \vdash P \blacktriangleright T}$

$$\langle \text{ZERO} \rangle \frac{}{\Gamma \vdash \mathbf{0} \blacktriangleright \mathbf{0}} \qquad \langle \text{BRA} \rangle \frac{\forall i \in I : \Gamma \vdash \pi_i ; P_i \blacktriangleright \kappa_i ; T_i}{\Gamma \vdash \mathsf{select}\, \{\pi_i ; P_i\}_{i \in I} \blacktriangleright \&\{\kappa_i ; T_i\}_{i \in I}} \qquad \langle \text{SEL} \rangle \frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash P \blacktriangleright S \quad \Gamma \vdash Q \blacktriangleright T}{\Gamma \vdash \mathsf{if}\, e\, \mathsf{then}\, P\, \mathsf{else}\, Q \blacktriangleright \oplus\{T, S\}}$$

$$\langle \text{NEWC} \rangle \frac{\Gamma, y : \mathsf{ch}(\sigma) \vdash P \blacktriangleright T \quad c \notin \mathrm{dom}(\Gamma) \cup \mathrm{fn}(T)}{\Gamma \vdash \mathsf{newchan}(y : \sigma, n); P \blacktriangleright (\mathsf{new}^n\, c); T\{^c/_y\}} \qquad \langle \text{OUT} \rangle \frac{\Gamma \vdash u_c : \mathsf{ch}(\sigma) \quad \Gamma \vdash e : \sigma \quad \Gamma \vdash P \blacktriangleright T}{\Gamma \vdash u_c!\langle e \rangle ; P \blacktriangleright \overline{u_c}; T}$$

$$\langle \text{IN} \rangle \frac{\Gamma \vdash u_c : \mathsf{ch}(\sigma) \quad \Gamma, x : \sigma \vdash P \blacktriangleright T}{\Gamma \vdash u_c?(x); P \blacktriangleright u_c; T} \qquad \langle \text{CLOSE} \rangle \frac{\Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \mathsf{close}\, u_c; P \blacktriangleright \mathsf{end}[u_c]; T} \qquad \langle \text{VAR} \rangle \frac{\Gamma \vdash \tilde{e} : \tilde{\sigma} \quad \Gamma \vdash \tilde{r} : \tilde{t}}{\Gamma, X(\tilde{\sigma}, \tilde{t}) \vdash X\langle \tilde{e}, \tilde{r} \rangle \blacktriangleright \mathbf{t}_X \langle \tilde{r} \rangle}$$

$$\langle \text{TAU} \rangle \frac{\Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \tau; P \blacktriangleright \tau; T} \qquad \langle \text{NEWM} \rangle \frac{\Gamma, m : \mathsf{var}(\sigma) \vdash P \blacktriangleright T}{\Gamma \vdash \mathsf{new}(m : \sigma); P \blacktriangleright \mathsf{new}(m); T} \qquad \langle \text{STO} \rangle \frac{\Gamma \vdash u_m : \mathsf{var}(\sigma) \quad \Gamma \vdash e : \sigma \quad \Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \mathsf{store}(u_m, e); P \blacktriangleright \mathsf{w}(u_m); T}$$

$$\langle \text{LOAD} \rangle \frac{\Gamma \vdash u_m : \mathsf{var}(\sigma) \quad \Gamma, y : \sigma \vdash P \blacktriangleright T}{\Gamma \vdash y \leftarrow \mathsf{load}(u_m); P \blacktriangleright \mathsf{r}(u_m); T} \qquad \langle \text{PAR} \rangle \frac{\Gamma \vdash P \blacktriangleright T \quad \Gamma \vdash Q \blacktriangleright S}{\Gamma \vdash P \mid Q \blacktriangleright T \mid S} \qquad \langle \text{RL-SUB} \rangle \frac{\Gamma \vdash u_l : \mathsf{RLock}}{\Gamma \vdash u_l : \mathsf{Lock}}$$

$$\langle \text{NEWL} \rangle \frac{\Gamma, l : \mathsf{Lock} \vdash P \blacktriangleright T}{\Gamma \vdash \mathsf{newl}(l); P \blacktriangleright \mathsf{newl}(l); T} \qquad \langle \text{LCK} \rangle \frac{\Gamma \vdash u_l : \mathsf{Lock} \quad \Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \mathsf{lock}(u_l); P \blacktriangleright \mathsf{l}(u_l); T} \qquad \langle \text{ULCK} \rangle \frac{\Gamma \vdash u_l : \mathsf{Lock} \quad \Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \mathsf{unlock}(u_l); P \blacktriangleright \mathsf{ul}(u_l); T}$$

$$\langle \text{NEWRL} \rangle \frac{\Gamma, l : \mathsf{RLock} \vdash P \blacktriangleright T}{\Gamma \vdash \mathsf{newrl}(l); P \blacktriangleright \mathsf{newrl}(l); T} \qquad \langle \text{RLCK} \rangle \frac{\Gamma \vdash u_l : \mathsf{RLock} \quad \Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \mathsf{rlock}(u_l); P \blacktriangleright \mathsf{rl}(u_l); T} \qquad \langle \text{RULCK} \rangle \frac{\Gamma \vdash u_l : \mathsf{RLock} \quad \Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \mathsf{runlock}(u_l); P \blacktriangleright \mathsf{rul}(u_l); T}$$

$\boxed{\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}}$

$$\langle \text{DEF} \rangle \frac{\forall i \in I : \Gamma, X_i(\tilde{\sigma}_i, \tilde{t}_i), \tilde{x}_i : \tilde{\sigma}_i, \tilde{y}_i : \tilde{t}_i \vdash P_i \blacktriangleright T_i \qquad \Gamma, X_1(\tilde{\sigma}_1, \tilde{t}_1), \ldots, X_n(\tilde{\sigma}_n, \tilde{t}_n) \vdash Q \blacktriangleright S}{\Gamma \vdash \{X_i(\tilde{x}_i, \tilde{y}_i) = P_i\}_{i \in I}\, \mathsf{in}\, Q \blacktriangleright \{\mathbf{t}_{X_i}(\tilde{y}_i) = T_i\}_{i \in I}\, \mathsf{in}\, S}$$

$\boxed{\Gamma \vdash_B P \blacktriangleright T}$

$$\langle \text{INT} \rangle \frac{\Gamma \vdash P \blacktriangleright T}{\Gamma \vdash_\emptyset P \blacktriangleright T} \qquad \langle \text{PARR} \rangle \frac{\Gamma \vdash_{B_1} P \blacktriangleright T \quad \Gamma \vdash_{B_2} Q \blacktriangleright S \quad B_1 \cap B_2 = \emptyset}{\Gamma \vdash_{B_1 \cup B_2} P \mid Q \blacktriangleright T \mid S}$$

$$\langle \text{RES} \rangle \frac{\Gamma, s : t \vdash_B P \blacktriangleright T}{\Gamma \vdash_{B \setminus \{s\}} (\nu s) P \blacktriangleright (\nu s) T} \qquad \langle \text{BUFF} \rangle \frac{\Gamma \vdash a : \mathsf{ch}(\sigma) \quad |\tilde{v}| = k}{\Gamma \vdash_{\{a\}} a\langle \sigma, n \rangle :: \tilde{v} \blacktriangleright \lfloor a \rfloor_k^n} \qquad \langle \text{CBUFF} \rangle \frac{\Gamma \vdash a : \mathsf{ch}(\sigma)}{\Gamma \vdash_{\{a\}} a^{\oslash}\langle \sigma, n \rangle :: \tilde{v} \blacktriangleright a^{\oslash}}$$

$$\langle \text{HEAP} \rangle \frac{\Gamma \vdash m : \mathsf{var}(\sigma)}{\Gamma \vdash_{\{m\}} [m, \sigma :: v] \blacktriangleright m^{\blacksquare}} \qquad \langle \text{MUT} \rangle \frac{\Gamma \vdash l : \mathsf{Lock}}{\Gamma \vdash_{\{l\}} \lceil l \rceil \blacktriangleright \lceil l \rceil} \qquad \langle \text{L-MUT} \rangle \frac{\Gamma \vdash l : \mathsf{Lock}}{\Gamma \vdash_{\{l\}} \lceil l \rceil^{\text{🔒}} \blacktriangleright \lceil l \rceil^{\text{🔒}}}$$

$$\langle \text{RMUT} \rangle \frac{\Gamma \vdash l : \mathsf{RLock}}{\Gamma \vdash_{\{l\}} \langle l \rangle_i \blacktriangleright \langle l \rangle_i} \qquad \langle \text{L-RMUT} \rangle \frac{\Gamma \vdash l : \mathsf{RLock}}{\Gamma \vdash_{\{l\}} \langle l \rangle_i^{\text{🔒}} \blacktriangleright \langle l \rangle_i^{\text{🔒}}} \qquad \langle \text{W-RMUT} \rangle \frac{\Gamma \vdash l : \mathsf{RLock}}{\Gamma \vdash_{\{l\}} \langle l \rangle_i^{\text{✋}} \blacktriangleright \langle l \rangle_i^{\text{✋}}}$$

Figure 3.6: Typing rules for MiGo$^+$

bottom of the tree then looks like:

$$\dfrac{\Gamma \vdash P_1 \blacktriangleright T_1 \quad \langle\text{PAR}\rangle \dfrac{\Gamma \vdash P_2 \blacktriangleright T_2 \quad \Gamma \vdash P_3 \blacktriangleright T_3}{\Gamma \vdash P_2 \mid P_3 \blacktriangleright (T_2 \mid T_3)}}{\langle\text{PAR}\rangle \quad \Gamma \vdash P_1 \mid (P_2 \mid P_3) \blacktriangleright T_1 \mid (T_2 \mid T_3)}$$

which turns into:

$$\dfrac{\dfrac{\Gamma \vdash P_1 \blacktriangleright T_1 \quad \Gamma \vdash P_2 \blacktriangleright T_2}{\langle\text{PAR}\rangle \quad \Gamma \vdash P_1 \mid P_2 \blacktriangleright (T_1 \mid T_2)} \quad \Gamma \vdash P_3 \blacktriangleright T_3}{\langle\text{PAR}\rangle \quad \Gamma \vdash (P_1 \mid P_2) \mid P_3 \blacktriangleright (T_1 \mid T_2) \mid T_3}$$

- if $P \equiv_\alpha P'$ then we can apply the same α-renaming to T and get $T \equiv_\alpha T'$ that with $\Gamma \vdash P' \blacktriangleright T'$ using the exact same tree, just renaming bound variables in the relevant rules.

For the general $\Gamma \vdash_B P \blacktriangleright T$ case, the proof is of the same form, only with congruence and typing rules including stores and name restriction. The congruence rules from processes are matched exactly with congruence rules from the typing system; the only typing rules whose application order we have to switch other than the ones above are $\langle$PARR$\rangle$ and $\langle$RES$\rangle$. The bookkeeping B does not incur any problem in the manipulations like the above, since the former only differs from $\langle$PAR$\rangle$ by enforcing we do not duplicate runtime constructs of stores, and the latter ensuring we keep track of free names only for this purpose — because bound names can always be alpha-converted freely to not clash with any free name. □

Theorem 3.9 (Subject Reduction)**.** Suppose $\Gamma \vdash_B P \blacktriangleright T$ and $P \to P'$.
Then there exists T' such that $\Gamma \vdash_B P' \blacktriangleright T'$ and $T \to T'$. ⌟

Proof. Suppose $\Gamma \vdash_B P \blacktriangleright T$ and $P \to P'$. We prove by induction on the reduction semantics from the processes that there exists T' such that $\Gamma \vdash_B P' \blacktriangleright T'$ and $T \to T'$. We first treat the base cases:

- Rules [SCOM], [OUT] and [IN] respectively correspond to |SCOM|, |OUT| and |IN|, giving the reduced type by way of matching the involved actions by looking them up in the typing deduction tree. The reduced subprocesses and types can be replaced in said deduction tree one-to-one to give the final typing.
- Similarly, rule [BRA] translates to |BRA|, and rule [CLOSE] translates to |CLOSE|, with one-to-one replacement.
- Rules [C-LD] and [C-ST] both fit with rule |C-HEAP|, as there is no need for matching data in the types; and the typing is matched in atomic actions again one-to-one.
- Rules [C-LCK], [C-ULCK], [C-RLCK] and [C-RULCK] respectively correspond to |C-LCK|, |C-ULCK|, |C-RLCK| and |C-RULCK|, and [C-WAIT] corresponds to |C-WAIT|, using the same matching procedure in the deduction tree.

- Runtime creation rules [NEWC], [NEWM], [NEWL] and [NEWRL] match rules |NEWC|, |NEWM|, |NEWL| and |NEWRL| respectively, but the modification to the typing tree is more involved: the one-to-last typing rule ⟨NEWC⟩, ⟨NEWM⟩, ⟨NEWL⟩ or ⟨NEWRL⟩ is replaced by a composition of ⟨PARR⟩ and the relevant runtime construct typing rule ⟨BUFF⟩, ⟨HEAP⟩, ⟨MUT⟩ or ⟨RMUT⟩ respectively, and the last typing rule ⟨INT⟩ is replaced by a restriction ⟨RES⟩.
- Finally, rules [IFT] and [IFF] both translate to rule |SEL|, reducing with τ in the same way as the simpler cases above, the typing simply being that of the subtree of the chosen path.

Next, we treat the induction cases:

- Rule [DEF] uses |DEF|, the induction falls on the leftmost hypotheses $P\{\tilde{v},\tilde{s}/\tilde{x}\} \mid Q$ and $T\{\tilde{s}/\tilde{x}\} \mid S$.
- Rules [PAR-L] and [PAR-R] correspond to |PAR-L| and |PAR-R| directly, induction happens on the relevant term, and is lifted back with the parallel typing rule.
- Rules [RES-1] and [RES-2] correspond to |RES-1| and |RES-2|, both cases calling induction on the non-restricted term with [RES-2] and |RES-2| pushing induction to the synchronisation case from the silent reduction case; and then the typing is lifted back within the restriction typing rule.
- Rule [ALPHA] uses |ALPHA| directly with induction falling on P' and T', the alpha-renaming being trivially reflected in the typing rules. □

Theorem 3.10 (Subject Properties)**.** Suppose $\Gamma \vdash P \blacktriangleright T$. If P is live (resp. has a data race), then T is live (resp. has a data race). ⌟

Proof. This follows directly from the typing rules and Theorem 3.9. Specifically, the typing rules ensure that if P has an immediate capability, then T has the same capability. Then, Theorem 3.9 ensures that reduction paths from P are inherited by T (though T can have *more* reduction paths, the extraneous ones are irrelevant here), and that the typing is preserved at each point in these paths, enforcing in turn the immediate capabilities from the process redexes to the type redexes. □

We now prove a progress theorem, giving us the ability to infer the existence of a process execution step from the ability of the type to reduce a step.

Theorem 3.11 (Progress)**.** Suppose $\Gamma \vdash P \blacktriangleright T$ and $T \to T_0$.
Then there exist P', T' such that $P \to P'$, $T \to T'$ and $\Gamma \vdash P' \blacktriangleright T'$. ⌟

To prove this theorem, we use the following lemma.

Lemma 3.12 (Correspondence of Barbs on Types and Processes)**.** Suppose $\Gamma \vdash P \blacktriangleright T$. If $T{\downarrow_o}$ with $o \neq m^{\blacksquare}$, then $P \downarrow_o$. If $T{\downarrow_{m^{\blacksquare}}}$ then $P \downarrow_{\overline{\mathbf{r}\langle m\rangle}}$ or $P \downarrow_{\mathbf{r}\langle m\rangle}$. ⌟

Proof. Suppose $\Gamma \vdash P \blacktriangleright T$, and $T{\downarrow_o}$. We do a case analysis on o, proving for each case the conclusion by induction on the structure of T.

- If $o = m^{\blacksquare}$, then T can be:
  - $T = m^{\blacksquare}$, then $P = [m, \sigma :: v]$ and it follows $P \downarrow_{\overline{\mathsf{w}\langle m\rangle}}$ and $P \downarrow_{\overline{\mathsf{r}\langle m\rangle}}$.
  - $T = T_1 \mid T_2$ and $T_1 \downarrow_o$, by the typing rules there exist P_1 and P_2 such that $\Gamma \vdash P_i \blacktriangleright T_i$ for $i = 1, 2$ and $P = P_1 \mid P_2$; then by induction on T_1, P_1 we get $P_1 \downarrow_{\overline{\mathsf{r}\langle m\rangle}}$ or $P_1 \downarrow_{\overline{\mathsf{w}\langle m\rangle}}$, and finally $P \downarrow_{\overline{\mathsf{r}\langle m\rangle}}$ or $P \downarrow_{\overline{\mathsf{w}\langle m\rangle}}$.
  - $T = (\nu s)T_0$ and $s \notin \mathrm{dom}(o)$, then there exists P_0 such that $\Gamma \vdash P_0 \blacktriangleright T_0$ and $P = (\nu s)P_0$; by induction on T_0, P_0 we get $P_0 \downarrow_{\overline{\mathsf{r}\langle m\rangle}}$ or $P_0 \downarrow_{\overline{\mathsf{w}\langle m\rangle}}$, and finally $P \downarrow_{\overline{\mathsf{r}\langle m\rangle}}$ or $P \downarrow_{\overline{\mathsf{w}\langle m\rangle}}$.
  - $T = \mathbf{t}_X\langle\tilde{s}\rangle$, with $\mathbf{t}_X(\tilde{x}) = T_0$ and $T_0\{\tilde{s}/\tilde{x}\}{\downarrow_o}$; then $P = X\langle\tilde{e}, \tilde{s}\rangle$, $X(\tilde{x}) = Q$, and so $\Gamma \vdash Q\{\tilde{e},\tilde{s}/\tilde{x}\} \blacktriangleright T_0\{\tilde{s}/\tilde{x}\}$; by induction $Q\{\tilde{e},\tilde{s}/\tilde{x}\} \downarrow_{\overline{\mathsf{r}\langle m\rangle}}$ or $Q\{\tilde{e},\tilde{s}/\tilde{x}\} \downarrow_{\overline{\mathsf{w}\langle m\rangle}}$, and finally $P \downarrow_{\overline{\mathsf{r}\langle m\rangle}}$ or $P \downarrow_{\overline{\mathsf{w}\langle m\rangle}}$.
  - $T \equiv_\alpha T_0$ and $T_0{\downarrow_o}$, then since α-conversion only renames bound variables, we get $\Gamma \vdash P \blacktriangleright T_0$ and we can use the former points on T_0, P, such that $P \downarrow_{\overline{\mathsf{r}\langle m\rangle}}$ or $P \downarrow_{\overline{\mathsf{w}\langle m\rangle}}$.
- If o is anything else, the barb construction rules from the types and the processes correspond exactly, with each types construct corresponding to the process that can fire actions the same way. □

We can now prove Theorem 3.11.

Proof. Suppose $\Gamma \vdash P \blacktriangleright T$ and $T \xrightarrow{o} T_0$ with $o \in \{\tau, \text{🤝}_s\}$.

- If $o = \text{🤝}_s$, then by structural congruence we can make sure the two actions that can sync are directly parallel to each other in a subprocess P_0 of a process $P' \equiv P$, providing the ability for P_0 to reduce, thus P' can reduce, and finally P can reduce.
- If $o = \tau$, then, T can be:
  - $T = \tau;T_0$, then $P = \tau; P' \to P'$.
  - $T = \&\{\kappa_i;T_i\}_{i\in I}$ and for a certain j, $\kappa_j = \tau$, then $P = \mathsf{select}\,\{\pi_i; P_i\}_{i\in I}$ and $\pi_j = \tau$, then $P \to P_j$. Note that in this case, T and P do not have a barb, because barbs for select constructs are only defined when no prefix is a τ prefix.
  - $T = \oplus\{T_1, T_2\} \xrightarrow{\tau} T_j$, then $P = \mathsf{if}\ e\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$ and, depending on the value of e, either $P \to P_1$ and $T_0 = T_1$; or $P \to P_2$ and $T_0 = T_2$.
  - $T = T_1 \mid T_2$ and $T_1 \xrightarrow{\tau} T_1'$, by the typing rules there exists P_1 and P_2 such that $\Gamma \vdash P_i \blacktriangleright T_i$ for $i = 1, 2$ and $P = P_1 \mid P_2$; then by induction on T_1, P_1 we get P_1' such that $P_1 \to P_1'$, and finally $P \to P' = P_1' \mid P_2$.

- Rules |NEWC|, |NEWM|, |NEWL| and |NEWRL| all correspond to the same constructs in the process world and reduce with the corresponding rules.
- Rule |CLOSE| corresponds to its similarly-named rule [CLOSE] as well.
- $T = (\nu s)T_1$ and $T_1 \xrightarrow{\tau} T_1'$, then there exists P_1 such that $\Gamma \vdash P_1 \blacktriangleright T_1$ and $P = (\nu s)P_1$; by induction on T_1, P_1 we get $P_1 \to P_1'$, and finally $P \to P' = (\nu s)P_1'$.
- $T = (\nu s)T_1$ and $T_1 \xrightarrow{s} T_1'$, then there exists P_1 such that $\Gamma \vdash P_1 \blacktriangleright T_1$ and $P = (\nu s)P_1$; by induction on T_1, P_1 using the sync case, we get $P_1 \to P_1'$, and finally $P \to P' = (\nu s)P_1'$.
- $T = \mathbf{t}_X\langle\tilde{s}\rangle \mid S$, with $\mathbf{t}_X(\tilde{x}) = T_1$ and $T_1\{\tilde{s}/\tilde{x}\} \mid S \xrightarrow{\tau} T'$; we have $P = X\langle\tilde{e},\tilde{s}\rangle \mid R$, $X(\tilde{x}) = Q$ and $\Gamma \vdash Q\{\tilde{e},\tilde{s}/\tilde{x}\} \blacktriangleright T_1\{\tilde{s}/\tilde{x}\}$, so by induction $Q\{\tilde{e},\tilde{s}/\tilde{x}\} \mid R \to Q'$ for some Q', and finally $P \to P'$ for some P' using rule [DEF].
- $T \equiv_\alpha T_1$ and $T_1 \xrightarrow{\tau} T'$, then as α-conversion only renames bound variables we have $\Gamma \vdash P_1 \blacktriangleright T_1$, where $P_1 \equiv_\alpha P$ has had the same renaming, and we can use the former points on T_1, P_1, getting $P_1 \to P'$ for some P' such that $\Gamma \vdash P' \blacktriangleright T'$. □

Remark 3.13. The Progress theorem does not give us liveness equivalence, as the process (and type) could contain a loop that allows it to reduce infinitely and never fire a synchronisation on a store for which an action awaits in a deadlocked parallel process. It however *does* give **termination and deadlock-freedom equivalence**, as the type being deadlock-free means it will only stop reducing if it reached an inactive type, and thus the process cannot be deadlocked since typing is preserved (and the converse is true thanks to the above theorem). ⌟

Similarly, progress allows us to prove process safety equivalence. We start with safety and data race freedom:

Theorem 3.14 (Process Safety and Data Race Freedom)**.** Suppose $\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}$ and $\boldsymbol{T}$ is safe (resp. data race free). Then $\boldsymbol{P}$ is safe (resp. data race free). ⌟

To prove this theorem, we need an inversion lemma.

Lemma 3.15 (Inversion)**.**

1. If $\Gamma \vdash_B P \blacktriangleright T$ and $P \equiv (\nu u)P'$ then $T \equiv (\nu u)T'$, with $\Gamma' \vdash_{B'} P' \blacktriangleright T'$ for some Γ' and B', with $\Gamma \subseteq \Gamma'$ and $B \subseteq B'$.
2. If $\Gamma \vdash_B P \blacktriangleright T$ and $P \equiv P_1 \mid P_2$ then $T \equiv T_1 \mid T_2$, with $\Gamma_1 \vdash_{B_1} P_1 \blacktriangleright T_1$ and $\Gamma_2 \vdash_{B_2} P_2 \blacktriangleright T_2$, with $B = B_1 \cup B_2$.
3. If $\Gamma \vdash_B P \blacktriangleright T$ and $P \Downarrow_o$ then:
   - if $o \notin \{\overline{\mathrm{r}\langle m\rangle}, \overline{\mathrm{w}\langle m\rangle}\}$ then $T \Downarrow_o$;

- if $o \in \{\overline{\mathsf{r}\langle m\rangle}, \overline{\mathsf{w}\langle m\rangle}\}$ then $T \Downarrow_{m\blacksquare}$.

4. If $\Gamma \vdash_B P \blacktriangleright T$ and $P \downarrow_o$ then:

   - if $o \notin \{\overline{\mathsf{r}\langle m\rangle}, \overline{\mathsf{w}\langle m\rangle}\}$ then $T \downarrow_o$;
   - if $o \in \{\overline{\mathsf{r}\langle m\rangle}, \overline{\mathsf{w}\langle m\rangle}\}$ then $T \downarrow_{m\blacksquare}$. ⌟

Proof. This is straight from the typing rules, much like Subject Congruence in Proposition 3.8. □

We go on with proving Theorem 3.14.

Proof. We decompose safety in its three parts. Suppose $X_0\langle\rangle \to^* (\nu\tilde{s})Q$ and

1. $Q \downarrow_{c✉}$. Then, by Lemma 3.15, there exists Γ', T such that $\Gamma' \vdash Q \blacktriangleright T$, and we have $T \downarrow_{c✉}$. Safety of $\boldsymbol{T}$ entails safety of T as a subterm of a reduced term from $\boldsymbol{T}$, thus $\neg(T \Downarrow_{\mathsf{end}[c]})$ and $\neg(T \Downarrow_{\overline{c}})$. This implies, by applying the third point of Lemma 3.15 again, $\neg(Q \Downarrow_{\mathsf{end}[c]})$ and $\neg(Q \Downarrow_{\overline{c}})$.

2. (a) $Q \downarrow_{\mathsf{ul}\langle l\rangle}$. Then, by Lemma 3.15, there exists Γ', T such that $\Gamma' \vdash Q \blacktriangleright T$, and we have $T \downarrow_{\mathsf{ul}\langle l\rangle}$. Safety of $\boldsymbol{T}$ entails safety of T as a subterm of a reduced term from $\boldsymbol{T}$, thus $T \downarrow_{l🔑}$, and by Lemma 3.12, $Q \downarrow_{l🔑}$.

   (b) $Q \downarrow_{\mathsf{rul}\langle l\rangle}$. Then, by Lemma 3.15, there exists Γ', T such that $\Gamma' \vdash Q \blacktriangleright T$, and we have $T \downarrow_{\mathsf{rul}\langle l\rangle}$. Safety of $\boldsymbol{T}$ entails safety of T as a subterm of a reduced term from $\boldsymbol{T}$, thus $T \downarrow_{l🟥}$, and by Lemma 3.12, $Q \downarrow_{l🟥}$.

3. $Q \downarrow_{(\mathsf{w}\langle m\rangle,\iota)}$. Then, by Lemma 3.15, there exists Γ', T such that $\Gamma' \vdash Q \blacktriangleright T$, and we have $T \downarrow_{(\mathsf{w}\langle m\rangle,\iota)}$. Data race freedom of $\boldsymbol{T}$ entails data race freedom of T as a subterm of a reduced term from $\boldsymbol{T}$, thus $\neg(T \downarrow_{(\mathsf{w}\langle m\rangle,\iota')})$ and $\neg(T \downarrow_{\mathsf{r}\langle m\rangle\iota'})$, for any $\iota' \neq \iota$, and by Lemma 3.15, $\neg(Q \downarrow_{(\mathsf{w}\langle m\rangle,\iota')})$ and $\neg(Q \downarrow_{\mathsf{r}\langle m\rangle\iota'})$, for any $\iota' \neq \iota$. □

We now prove liveness equivalence in several subsets of processes we deem interesting.

As liveness is not equivalent in the general case, per Remark 3.13, we define several subsets of programs for which we can prove liveness of the types implies liveness of the process. First, we look at programs that always have a terminating path. Then we will look at programs that are finite branching, and lastly we will look at programs simulating non-deterministic branching for recurring conditionals.

Our first category is called *may converging* programs, as per their definition in work [LNTY17] from which we follow the general path in the rest of this section.

Definition 3.16 (May Converging Program)**.** Let $\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}$. We write $\boldsymbol{P} \in \mathtt{May}_{\Downarrow}$ if, for all P' such that $\boldsymbol{P} \to^* P'$, then $P' \to^* \mathbf{0}$. ⌟

Example 3.17. An example of May Converging program is the following program, where process P loops and alternates x to values 1 and 0 until the *end* flag is set, and Q loops reading x until it reads a value 0, in which case it sets the *end* flag and returns:

$$\boldsymbol{P}_{\mathsf{mc}} := \left\{ \begin{array}{ll} X_0 & = \mathsf{new}(x:\mathsf{int});\mathsf{new}(end:\mathsf{bool});\mathsf{newrl}(l); \\ & \quad (P\langle x,end,l\rangle \mid Q\langle x,end,l\rangle) \\ P(x,end,l) & = \mathsf{lock}(l);y \leftarrow \mathsf{load}(x);\mathsf{store}(x,1-y);z \leftarrow \mathsf{load}(end); \\ & \quad \mathsf{unlock}(l);\mathsf{if}\ z\ \mathsf{then}\ \mathbf{0}\ \mathsf{else}\ P\langle x,end,l\rangle \\ Q(x,end,l) & = \mathsf{lock}(l);y \leftarrow \mathsf{load}(x);\mathsf{unlock}(l); \\ & \quad \mathsf{if}\ y=0\,\mathsf{then}\,\mathsf{lock}(l);\mathsf{store}(end,\mathsf{true});\mathsf{unlock}(l);\mathbf{0} \\ & \quad \mathsf{else}\ Q\langle x,end,l\rangle \end{array} \right\} \mathsf{in}\ X_0\langle\rangle$$

⌟

On those typed programs that always have a terminating path available, we prove that verifying liveness of the type is enough to conclude the program is live, and that in the case the program reaches a point of no reduction, then it has terminated and is not deadlocked.

Proposition 3.18 (May-Converging Liveness)**.** Assume $\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}$ and $\boldsymbol{T}$ is live.

1. Suppose there exists P' such that $\boldsymbol{P} \rightarrow^* P' \not\rightarrow$. Then $P' \equiv \mathbf{0}$.
2. If $\boldsymbol{P} \in \mathtt{May}_{\Downarrow}$, then $\boldsymbol{P}$ is live. ⌟

Proof. Assume $\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}$ and $\boldsymbol{T}$ is live.

1. Suppose by contradiction that $X_0\langle\rangle \rightarrow^* P \not\rightarrow$ but $P \not\equiv \mathbf{0}$. Then there exists Q such that $P \equiv (\nu\tilde{s})Q$ and either $Q \downarrow_o$ with $o \in \{c, \overline{c}, \mathsf{l}\langle l\rangle, \mathsf{rl}\langle l\rangle\}$, or $Q \downarrow_{\tilde{o}}$ for some $\tilde{o}$ (containing only blocking channel actions by definition). Then, by Lemma 3.15, this contradicts liveness for $\boldsymbol{T}$.
2. As there is always a path to term $\mathbf{0}$, which is live, then all blocking actions available at any given point can be fired on any available path to termination, hence $\boldsymbol{P}$ is live. □

Note that our tool, which we present in the next chapter, is only able to reliably verify liveness on programs that always terminate. Hence, we define the subset of strongly terminating programs next to encompass that. However, the tool can run on non-terminating programs, under the assumption they are finite control (not spawning an unbounded number of new threads), but may in rare instances return a false positive in this case, due to the approximations made by the model checking tool we use.

Definition 3.19 (Strongly Terminating Programs)**.** We say that $\boldsymbol{P}$ strongly terminates if there exists some non-negative integer n_0 such that, for all P and $n \geq n_0$ such that $\boldsymbol{P} \rightarrow^n P$, $P \not\rightarrow$. ⌟

Proposition 3.20 (Strong Termination Liveness)**.** Suppose $\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}$, $\boldsymbol{P}$ strongly terminates and $\boldsymbol{T}$ is live. Then $\boldsymbol{P} \in \mathtt{May}_{\Downarrow}$ and $\boldsymbol{P}$ is live. ⌟

Proof. Proposition 3.18 proves that $\boldsymbol{P}$ always reaches $\mathbf{0}$. Then, by definition of may converging programs, since $\boldsymbol{P}$ always reaches $\mathbf{0}$, it is may converging. Finally, Proposition 3.18 proves that $\boldsymbol{P}$ is live. □

Note there is an imprecision in our published work [GY20a] regarding strongly terminating programs (called simply *terminating programs* in that work), which states that all strongly terminating programs are may converging. This is false in general, as this work forgets the hypothesis of the program being typed with a live type, and this imprecision comes from a former (stronger) definition of strongly terminating programs, that stated the processes always reached $\mathbf{0}$ in the definition, before changing to a more general form of termination that can include deadlocks — and we forgot to change the proposition that follows this definition.

To look at finite branching programs and programs that emulate non-deterministic choice in their recurring conditional constructs, we need to define markings that distinguish between all conditionals in a program, to verify that one of them is indeed recurring.

Definition 3.21 (Marked Programs)**.** Given a program $\boldsymbol{P}$, we define its marking $\mathrm{Mark}(\boldsymbol{P})$ as the program obtained by deterministically labelling every occurrence of a conditional construct $\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q$ to $\mathsf{if}^n\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q$, such that n is a distinct integer for all conditionals in $\boldsymbol{P}$. ⌟

Definition 3.22 (Marked Reduction Semantics)**.** The LTS for the programs is modified and now denoted $P \xrightarrow{\sharp} P'$ to signify that process P reduces in a single step to P' by performing action $\sharp$. Action labels follow the grammar $\sharp ::= \alpha \mid n \cdot \mathrm{L} \mid n \cdot \mathrm{R}$, where we follow the previous notation α for non-conditional reducts, and $n \cdot \mathrm{L}$ (resp. $n \cdot \mathrm{R}$) denotes the reduction of a conditional to the left "then" branch (resp. the right "else" branch) by the following rules (replacing former [IFT] and [IFF]).

$$[\text{IFT}]\ \frac{e \downarrow \mathsf{true}}{\mathsf{if}^n\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{n\cdot\mathrm{L}} P} \qquad [\text{IFF}]\ \frac{e \downarrow \mathsf{false}}{\mathsf{if}^n\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{n\cdot\mathrm{R}} Q}$$

⌟

Note that because of this notation change, conditional branches are not considered standard reduction steps included in $\rightarrow$ for marked programs, with the marked reduction semantics replacing the two rules for conditional reduction.

Definition 3.23 (Trace)**.** We define an execution trace of a process P as the potentially infinite sequence of action labels $\overrightarrow{\sharp} = \{\sharp_1, \sharp_2, \ldots\}$ such that $P \xrightarrow{\sharp_1} P_1 \xrightarrow{\sharp_2} \ldots$; and we write $\mathbb{T}_P$ for the set of all possible traces of a process P. ⌟

Definition 3.24 (Reduction Contexts)**.** We define reduction contexts as processes with a hole, following the grammar: $\mathbb{C}_r ::= [] \mid (P \mid \mathbb{C}_r) \mid (\mathbb{C}_r \mid P) \mid (\nu s)\mathbb{C}_r$. ⌟

Definition 3.25 (Infinite Conditional)**.** Program $\boldsymbol{P}$ is said to have an infinite conditional, written $\boldsymbol{P} \in \texttt{Inf}$, if $\text{Mark}(\boldsymbol{P}) \to^* \mathbb{C}_r[\text{if}^n\ e\ \text{then}\ P\ \text{else}\ Q] = R$ for some n and R has an infinite trace where $n \cdot \text{L}$ or $n \cdot \text{R}$ appears infinitely often. Such n is called an *infinite conditional mark*, and we write $\text{InfCond}(\boldsymbol{P})$ for the set of all such marks of a program $\boldsymbol{P}$. ⌟

We call a program not in `Inf` "finite branching", and show that proving liveness of the type is enough to prove liveness of the typed process when it is finite branching.

Proposition 3.26 (Liveness for Finite Branching)**.** Suppose $\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}$, $\boldsymbol{T}$ is live and $\boldsymbol{P} \notin \texttt{Inf}$. Then $\boldsymbol{P}$ is live. ⌟

We prove the following lemma to ease the proof of the proposition:

Lemma 3.27. Suppose $\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}$, $\boldsymbol{T}$ is live and $\boldsymbol{P}$ is conditional-free, then $\boldsymbol{P}$ is live. ⌟

Proof. Since there is no conditional, all moves are strongly matched between types and processes, i.e. if $X_0\langle\rangle \to^* (\nu\tilde{r})P$, using Lemma 3.15 we get $\Gamma' \vdash P \blacktriangleright T$, and we have $P \Downarrow_o$ if and only if $T \Downarrow_o$. Thus, liveness of $\boldsymbol{T}$ induces liveness of $\boldsymbol{P}$. □

We now prove Proposition 3.26:

Proof. Suppose $\boldsymbol{P} \notin \texttt{Inf}$ and $X_0\langle\rangle \to^* (\nu\tilde{s})P$. By Lemma 3.15 there exists Γ', T such that $\Gamma' \vdash P \blacktriangleright T$. Since $\boldsymbol{P} \notin \texttt{Inf}$, we can always reduce to a term that is conditional-free, and along with the Inversion Lemma again, there is P', T', Γ'' such that $P \to^* P'$, $T \to^* T'$, $\Gamma'' \vdash P' \blacktriangleright T'$ and P' is conditional-free. We can then use Lemma 3.27 to conclude. □

Example 3.28. An example of finite branching program is the Dining Philosophers problem:

$$\boldsymbol{P}_{\text{dinephil}} ::= \left\{ \begin{array}{ll} X_0 & = \mathsf{new}(f_1:\mathsf{int});\mathsf{new}(f_2:\mathsf{int});\mathsf{new}(f_3:\mathsf{int}); \\ & \quad \mathsf{newl}(l_1);\mathsf{newl}(l_2);\mathsf{newl}(l_3); \\ & \quad (P\langle f_1,f_2,l_1,l_2,1\rangle \mid P\langle f_2,f_3,l_2,l_3,2\rangle \mid P\langle f_1,f_3,l_1,l_3,3\rangle) \\ P(f_l,f_r,l_l,l_r,id) & = \mathsf{lock}(l_l);y \leftarrow \mathsf{load}(f_l);\tau;\mathsf{store}(f_l,id); \\ & \quad \mathsf{lock}(l_r);z \leftarrow \mathsf{load}(f_r);\tau;\mathsf{store}(f_r,id+2); \\ & \quad \mathsf{unlock}(l_r);\mathsf{unlock}(l_l);P\langle f_l,f_r,l_l,l_r,id\rangle \end{array} \right\} \text{in } X_0\langle\rangle$$

Here, P defines the behaviour of a philosopher, trying to get a hold of both forks assigned to him, and them release them. ⌟

```
1  func Generate(ch chan<- int) {
2    for i :=2; ; i++ { ch <- i }
3  }
4
5  func Filter(in <-chan int, out chan<- int, prime int) {
6    for { i := <-in
7      if i%prime != 0 { out <- i }
8    }
9  }
```

```
11 func main() {
12   ch := make(chan int)
13   go Generate(ch)
14   for i := 0; ; i++ {
15     prime := <-ch
16     ch1 := make(chan int)
17     go Filter(ch, ch1, prime)
18     ch = ch1
19   }
20 }
```

Figure 3.7: Go implementation of a concurrent Prime Sieve algorithm [LNTY17, NY16]

Definition 3.29 (Conditional Mapping)**.** We define the conditional mapping $(\boldsymbol{P})^*$ of $\boldsymbol{P}$ by replacing all occurrences of marked conditionals $\mathsf{if}^n\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q$ such that $n \in \mathrm{InfCond}(\boldsymbol{P})$ with the non-deterministic variant $\mathsf{if}\ *\ \mathsf{then}\ P\ \mathsf{else}\ Q$.

The reduction semantics of this variant follows the reduction semantics of the types, choosing non-deterministically between P and Q, instead of using rules [IFT] and [IFF].

$$[\text{SEL}]\ \frac{j \in \{1,2\}}{\mathsf{if}\ *\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \xrightarrow{\tau} P_j}$$

⌟

We loosely apply the same mapping to processes P reduced from marked programs as well. From this last set of definitions, we can define the set of programs that emulate non-deterministic choice in its conditional constructs, which we will call the set of *alternating conditionals* programs, and prove that for this subset of MiGo$^+$ programs, liveness of the types is enough to prove liveness of the typed program.

Definition 3.30 (Alternating Conditionals)**.** We say that $\boldsymbol{P}$ has alternating conditional branches, written $\boldsymbol{P} \in \mathtt{AC}$, when $\boldsymbol{P} \in \mathtt{Inf}$ and if $\boldsymbol{P} \to^* (\nu\tilde{s})P$ then $(P)^* \Downarrow_o$ implies $P \Downarrow_o$. ⌟

Example 3.31. The concurrent version of the Prime Sieve algorithm [LNTY17, NY16] is an example of program that has alternating conditionals. A Go implementation of it is shown in Figure 3.7. Another simple example of alternating conditionals is as follows:

$$\boldsymbol{P}_{\mathsf{ac}} \coloneqq \left\{ \begin{array}{ll} X_0 & = \mathsf{new}(x:\mathsf{bool});\mathsf{new}(y:\mathsf{int});P\langle x,y\rangle \\ P(b,i) & = z \leftarrow \mathsf{load}(b);\mathsf{if}\ z\ \mathsf{then}\ t \leftarrow \mathsf{load}(i); \\ & \quad \mathsf{store}(i,t+1);\mathsf{store}(b,\mathsf{not}(z));P\langle b,i\rangle\ \mathsf{else}\ \mathsf{store}(b,\mathsf{not}(z));P\langle b,i\rangle \end{array} \right\} \mathsf{in}\ X_0\langle\rangle$$

⌟

Theorem 3.32 (Liveness for Alternating Conditionals)**.** If $\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}$, $\boldsymbol{T}$ is live and $\boldsymbol{P} \in \mathtt{AC}$, then $\boldsymbol{P}$ is live. ⌟

We first prove a simple lemma:

Lemma 3.33. If $\Gamma \vdash P \blacktriangleright T$, then for $o \in \{\tau, \text{🤝}_s\}$, $T \Downarrow_o$ if and only if $(P)^* \Downarrow_o$. ⌟

Proof. As for the conditional-free case in Lemma 3.27, we have a strong match for actions between the types and the conditional mapping, by removing the inherent determinism from $\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q$ constructs in MiGo$^+$. □

We now prove Theorem 3.32:

Proof. Suppose $X_0\langle\rangle \to^* (\nu\tilde{s})P$, Then by Lemma 3.15 and Theorem 3.9, we have T, Γ' such that $\Gamma' \vdash P \blacktriangleright T$ and $\boldsymbol{t}_0\langle\rangle \to^* (\nu\tilde{s})T$. By Lemma 3.33 we have $T \Downarrow_o$ if and only if $(P)^* \Downarrow_o$, and by $\boldsymbol{P} \in \mathtt{AC}$ we can conclude that $T \Downarrow_o$ implies $P \Downarrow_o$. Thus, $\boldsymbol{T}$ being live entails $\boldsymbol{P}$ is live. □

4 Model Checking Typed MiGo$^+$ Programs

In this chapter, we discuss how to automate the verification of properties on our typed abstraction of the Go language using a model checking tool. We first talk about the language used for formalising properties to be verified in our tool of choice: since we use mCRL2 for verification, we will be using a variant of μ-calculus to formalise our properties, which we introduce in Section 4.1. Then, Section 4.2 introduces our translations of the properties we want to check in that variant of the μ-calculus, and show how their validation holds together properties between the types and processes, thanks to the results of the previous two chapters. Finally, we have a look at our implementation in Section 4.3, explaining how it is built; and show selected benchmark with the verification of our defined properties and the time each example took to fully verify, giving an idea of how the proof-of-concept scales.

4.1 Necessary Definitions on the Modal μ-calculus

In this section, we introduce the modal μ-calculus [Koz83], a variant of μ-calculus that we use to translate our desired properties on the types to something that can be automatically verified on a model of the types by our tool of choice, mCRL2. We rely on course books [Sti01, Bra92] for the basics of the μ-calculus and give an overview of the variant we use in this chapter, including main properties and defining how it is tied to our type system.

The μ-calculus is generally used to describe temporal properties of concurrent systems such as our process calculus or its behavioural types, by defining logic formulae with temporal or modal operations. We use a modal variant in our work, and rely on fixpoint operators to express the temporalities on the reduction paths we explore. The specific syntax is described at the end of this section.

To describe the properties in such a system, we first define a pointed LTS for the types (Definition 4.1), basically an automaton constructed using the types we want to verify. We use the following notations throughout this chapter: $\mathcal{C}$ is the set of all channel names available at any given moment, and similarly $\mathcal{L}, \mathcal{M}, \mathcal{I}, \mathcal{S}$ are respectively the sets of all lock names, shared memory names, indexes for write actions, and store names (channels, locks and shared memory alike), respectively. We also use $\mathcal{P}(\mathcal{C})$ to denote the set of all possible selection barbs that may exist within the realm of the channel names.

Definition 4.1 (Pointed LTS for types)**.** The pointed LTS of a type $\boldsymbol{T}$, denoted $\mathcal{T}_{\boldsymbol{T}} = (\mathfrak{S}, \boldsymbol{t}_0\langle\rangle, \mathcal{A}, \mathcal{F})$, is defined by:

a set of states $\mathfrak{S}$ labelled by the behavioural types obtained from the possible reduction paths of $\boldsymbol{T}$ with $\rightarrow$, stripped of name restrictions wherever applicable, and with the entrypoint $\boldsymbol{t}_0\langle\rangle$ of $\boldsymbol{T}$ as the initial state (we identify states with their labels in the rest of this chapter);

a set of labelled transitions $\mathcal{A} \subseteq \mathfrak{S} \times \mathfrak{S} \times (\{\tau\} \cup \bigcup_{s \in \mathcal{S}} \{\text{🤝}_s\})$ defined by the possible transitions between the types labelling the states: $\mathcal{A} := \{(T, T', o) : T, T' \in \mathfrak{S} \text{ and } T \xrightarrow{o} T'\}$;

a set of flags on the states defined by the barbs of the type labelling the state they are attached to: for all $T \in \mathfrak{S}$, the flags of T are $\mathcal{F}(T) := \{\downarrow_o : T\downarrow_o\} \cup \{\downarrow_{\tilde{o}} : T\downarrow_{\tilde{o}}\}$. ⌟

Figure 4.1 shows the syntax for the formulae of the modal μ-calculus that we use in this chapter.

$$\phi := \top \mid \bot \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid [\alpha]\,\phi \mid \langle\alpha\rangle\,\phi \mid \nu Z.\phi \mid \mu Z.\phi \mid Z \qquad \alpha := \downarrow_o \mid \downarrow_{\tilde{o}} \mid \tau \mid \text{🤝}_s$$

Figure 4.1: Syntax of the μ-calculus formulae

The formula $\top$ holds for all states, and $\bot$ never holds for any state. Conjunction and disjunction are standard, holding respectively if all or any of the formulae linked by the operation are hold. Implication is also standard, holding if either the right side holds or the left side does not.

The box modality $[\alpha]\,\phi$ holds if ϕ holds in *all* states resulting from α: if α is a barb and the current state can play it, then it is the current state that must satisfy ϕ; and if α is a reduction action this is all states accessible from the current one by following transitions labelled by this action. It acts as a conjunction, thus being vacuously true if the set of states emanating from the action is empty. The diamond modality $\langle\alpha\rangle\,\phi$ holds if ϕ holds in *any* of the states resulting from α. It acts as a disjunction, thus being vacuously false if the set of resulting states is empty.

Formulae $\nu Z.\phi$ and $\mu Z.\phi$ are respectively the standard greatest and least fixpoint operators, binding Z in ϕ, and Z is used as a fixpoint variable, where the partial order is given by the labelled transitions and barbed flags of the pointed LTS. We remind the following properties of both fixpoint operators:

1. $\mu Z.Z{=}\bot$: the least fixpoint is trivially false on empty formulae;
2. $\nu Z.Z{=}\top$: the greatest fixpoint is trivially true on empty formulae;
3. if $\phi[Z := \psi] \Rightarrow \psi$, then $\mu Z.\phi \Rightarrow \psi$: the least fixpoint can be expanded on the left side of an implication;
4. if $\psi \Rightarrow \phi[Z := \psi]$, then $\psi \Rightarrow \nu Z.\phi$: the greatest fixpoint can be expanded on the right side of an implication.

To express that some μ-calculus formula ϕ holds on a state T in the LTS $\mathcal{T}$, we say that T satisfies ϕ in $\mathcal{T}$, and use the notation $T \models_{\mathcal{T}} \phi$.

1. Deadlock freedom: $\psi_g = \bigvee_{(c,l,\tilde{o}) \in \mathcal{C} \times \mathcal{L} \times \mathcal{P}(\mathcal{C})} \left(\langle \downarrow_{\bar{c}} \rangle \top \vee \langle \downarrow_{c} \rangle \top \vee \langle \downarrow_{\tilde{o}} \rangle \top \vee \left\langle \downarrow_{\mathsf{l}\langle l \rangle} \right\rangle \top \vee \left\langle \downarrow_{\mathsf{rl}\langle l \rangle} \right\rangle \top \right) \Rightarrow \left(\langle \tau \rangle \top \vee \bigvee_{s \in \mathcal{S}} \langle \text{🤝}_s \rangle \top \right)$

2. Channel safety: $\psi_s^{\pi} = \bigwedge_{c \in \mathcal{C}} \left(\left\langle \downarrow_{c\text{✉}} \right\rangle \top \Rightarrow \mathsf{AG} \left([\downarrow_{\bar{c}}] \bot \wedge \left[\downarrow_{\mathsf{end}[c]} \right] \bot \right) \right)$

3. Lock safety (a): $\psi_{s_a}^{\ell} = \bigwedge_{l \in \mathcal{L}} \left(\left\langle \downarrow_{\mathsf{ul}\langle l \rangle} \right\rangle \top \Rightarrow \left\langle \downarrow_{l\text{🔑}} \right\rangle \top \right)$

4. Lock safety (b): $\psi_{s_b}^{\ell} = \bigwedge_{l \in \mathcal{L}} \left(\left\langle \downarrow_{\mathsf{rul}\langle l \rangle} \right\rangle \top \Rightarrow \left\langle \downarrow_{l\text{🟥}} \right\rangle \top \right)$

5. Channel liveness (a): $\psi_{l_a}^{\pi} = \bigwedge_{c \in \mathcal{C}} \left(\left(\langle \downarrow_{\bar{c}} \rangle \top \vee \langle \downarrow_{c} \rangle \top \right) \Rightarrow \mathsf{AF} \left(\langle \text{🤝}_c \rangle \top \right) \right)$

6. Channel liveness (b): $\psi_{l_b}^{\pi} = \bigwedge_{\tilde{o} \in \mathcal{P}(\mathcal{C})} \left(\left(\langle \downarrow_{\tilde{o}} \rangle \top \right) \Rightarrow \mathsf{AF} \left(\bigvee_{c_i \in \mathsf{dom}(\tilde{o})} \langle \text{🤝}_{c_i} \rangle \top \right) \right)$

7. Lock liveness: $\psi_l^{\ell} = \bigwedge_{l \in \mathcal{L}} \left(\left(\left\langle \downarrow_{\mathsf{l}\langle l \rangle} \right\rangle \top \vee \left\langle \downarrow_{\mathsf{rl}\langle l \rangle} \right\rangle \top \right) \Rightarrow \mathsf{AF} \left(\langle \text{🤝}_l \rangle \top \right) \right)$

8. Data race freedom: $\psi_d = \bigwedge_{(m,\iota) \in \mathcal{M} \times \mathcal{I}} \left(\left\langle \downarrow_{(\mathsf{w}\langle m \rangle, \iota)} \right\rangle \top \Rightarrow \left(\left[\downarrow_{\mathsf{r}\langle m \rangle} \right] \bot \wedge \bigwedge_{\iota' \neq \iota} \left[\downarrow_{(\mathsf{w}\langle m \rangle, \iota')} \right] \bot \right) \right)$

Figure 4.2: Properties of the behavioural types as μ-calculus formulae

Finally, two key properties that we wish to express in the μ-calculus are that ϕ always holds, or that it eventually holds in a reachable state of the LTS. For that, we use the fixpoint operators with a fresh recursion variable:

Always ϕ : $\mathsf{AG}(\phi) = \nu Z. \left(\phi \wedge [\tau] Z \wedge \bigwedge_{s \in \mathcal{S}} [\text{🤝}_s] Z \right)$ Eventually ϕ : $\mathsf{AF}(\phi) = \mu Z. \left(\phi \vee \langle \tau \rangle Z \vee \bigvee_{s \in \mathcal{S}} \langle \text{🤝}_s \rangle Z \right)$

4.2 Expressing our Properties with the Modal μ-calculus

The properties we want to check, as defined for the types in Definitions 3.4 to 3.6, translate to our modal μ-calculus fragment in their *local* form as shown in Figure 4.2. The global form of the properties is "always ϕ", written $\mathsf{AG}(\phi)$ as above, for ϕ each of the properties we look at.

The deadlock freedom formula ψ_g states that, should any blocking action be attempted at any point (left-hand side of the implication), then a synchronisation on any runtime construct or a silent action must be possible at that same point of the execution (right-hand side). This is because there needs to be a reduction

path when a blocking action is in the queue, to ensure all blocking actions are eventually executed, and process reductions only happen via synchronisations and silent actions. This allows to detect if at any point, only blocking actions are available with none of them being able to synchronise with any of their counterparts.

The safety property is broken down to three formulae. Channel safety ψ_s^π corresponds to the inability for any channel that is already closed (left-hand side of the implication) to send to that channel or request its closure (right-hand side); the fact that the channel is closed is detected using the closed message barb $\downarrow_{c^{\text{✉}}}$, since a closed channel will always broadcast using this action. Lock safety has two parts: the first one $\psi_{s_a}^\ell$ detects attempts to unlock a mutual exclusion lock that is not currently locked: if an unlock attempt is made (left-hand side of the implication), then the locked runtime construct barb $\downarrow_{l^{\text{🔒}}}$ must be immediately available as well (right-hand side); and the second part $\psi_{s_b}^\ell$ detects attempts to read-unlock a mutual exclusion lock that has no reader currently assigned: if a read-unlock attempt is made (left-hand side of the implication), the read-locked runtime construct barb $\downarrow_{l^{\text{■}}}$ must be immediately available (right-hand side).

Liveness is also a three-part formula. Channel liveness is the two-parter in this case: the first part $\psi_{l_a}^\pi$ checks that, on any channel, any send or receive action (left-hand side of the implication) is eventually met by a synchronisation in the future (right-hand side); and the second part $\psi_{l_b}^\pi$ checks that any select construct with only non-silent actions that is attempted (left-hand side of the implication) has at least one of its action met by a future synchronisation on its channel (right-hand side). Similarly, lock liveness ψ_l^ℓ checks that, on any mutual exclusion lock, any lock or read-lock action (left-hand side of the implication) is met by a future synchronisation of its lock (right-hand side).

Finally, data race freedom ψ_d checks that, on any shared variable, whenever a write action is available (left-hand side of the implication), no other action can happen on the same shared memory variable at the same time, be it a read or a different write (right-hand side, with the conjunction on occurrence indices different from the one on the left covering any possible write that is not the one being checked against).

Example 4.2. We can check that the type T' from Example 3.2 does not verify ψ_d:

$$\psi_d = \left(\left\langle\downarrow_{(w\langle x\rangle,1.1.1.*)}\right\rangle\top \Rightarrow \left(\left[\downarrow_{(w\langle x\rangle,1.1.2.*)}\right]\bot \wedge \left[\downarrow_{r\langle x\rangle}\right]\bot\right)\right) \wedge \left(\left\langle\downarrow_{(w\langle x\rangle,1.1.2.*)}\right\rangle\top \Rightarrow \left(\left[\downarrow_{(w\langle x\rangle,1.1.1.*)}\right]\bot \wedge \left[\downarrow_{r\langle x\rangle}\right]\bot\right)\right)$$

which is false for T', hence $T' \not\models_{\mathcal{T}_{\mathsf{race}}} \psi_d$: locally, T' has a datarace. Then $\mathbf{t}_0() \not\models_{\mathcal{T}_{\mathsf{race}}} \mathsf{AG}\,(\psi_d)$, meaning $\boldsymbol{T}_{\mathsf{race}}$ has a data race, since its associated entrypoint in its LTS $\mathcal{T}_{\mathsf{race}}$ does not satisfy data race freedom property $\mathsf{AG}\,(\psi_d)$.

On the other hand, the type $\boldsymbol{T}_{\mathsf{safe}}$ from Example 3.2, modelling the safe version of our running example, verifies the data race freedom property, as well as safety and liveness, where ψ_l and ψ_s represent the conjunction of their respective three formulae:

$$\boldsymbol{T}_{\mathsf{safe}} \models_{\mathcal{T}_{\mathsf{safe}}} \mathsf{AG}\,(\psi_d) \wedge \mathsf{AG}\,(\psi_l) \wedge \mathsf{AG}\,(\psi_s)$$

The types corresponding to the other examples in Section 3.2 ($\boldsymbol{P}_{\mathsf{mc}}$, $\boldsymbol{P}_{\mathsf{dinephil}}$ and $\boldsymbol{P}_{\mathsf{ac}}$) are also safe, live and data race free. ⌟

The following theorem states the justification of type-level use of the model checking properties to justify the process properties, under the conditions determined before. We define the pointed LTS for programs and the satisfaction of a property for the program, $\mathcal{T}_{\boldsymbol{P}}$ and $\boldsymbol{P} \models_{\mathcal{T}_{\boldsymbol{P}}} \phi$, in the same way as they are defined for the types in Definition 4.1.

Theorem 4.3 (Model Checking of MiGo$^+$ processes)**.** *Suppose* $\Gamma \vdash \boldsymbol{P} \blacktriangleright \boldsymbol{T}$.

1. *If* $\boldsymbol{T} \models_{\mathcal{T}_{\boldsymbol{T}}} \mathsf{AG}\,(\phi)$ *for* $\phi \in \{\psi_g, \psi_s^{\pi}, \psi_{s_a}^{\ell}, \psi_{s_b}^{\ell}, \psi_d\}$, *then* $\boldsymbol{P} \models_{\mathcal{T}_{\boldsymbol{P}}} \mathsf{AG}\,(\phi)$.
2. *If* $\boldsymbol{T} \models_{\mathcal{T}_{\boldsymbol{T}}} \mathsf{AG}\,(\phi)$, *for* $\phi \in \{\psi_{l_a}^{\pi}, \psi_{l_b}^{\pi}, \psi_l^{\ell}\}$, *and either of the following is true: (a)* $\boldsymbol{P} \in \mathtt{May}_{\Downarrow}$ *or (b)* $\boldsymbol{P} \notin \mathtt{Inf}$ *or (c)* $\boldsymbol{P} \in \mathtt{AC}$; *then* $\boldsymbol{P} \models_{\mathcal{T}_{\boldsymbol{P}}} \mathsf{AG}\,(\phi)$. ⌟

Proof. The semantics of the μ-calculus given by the pointed LTS of types incurs that a type being deadlock-free, safe or data-race free is equivalent to that type satisfying the relevant properties in Figure 4.2, namely $\psi_g, \psi_s^{\pi}, \psi_{s_a}^{\ell}, \psi_{s_b}^{\ell}$ and ψ_d; and a type being live is equivalent to that type satisfying the liveness properties from the same figure, namely $\psi_{l_a}^{\pi}, \psi_{l_b}^{\pi}$ and ψ_l^{ℓ}. The same is true of processes. Hence, if the types of a typed process satisfy one of our μ-calculus properties, it is enough to show that the corresponding property of the types carry to the processes to have the corresponding process satisfy that same μ-calculus property.

The first item follows from two theorems. By Theorem 3.14, if a type is safe (resp. data-race free), then its process is also safe (resp. data-race free). Deadlock-freedom follows from Theorem 3.11, as the main requirement for that property is the availability of a reduction path under the specified conditions, and that theorem gives us that a reduction path being available for the types of a typed process implies the existence of a reduction path for the process as well.

The second item follows from the three results that we proved for the corresponding subsets of processes, since liveness equivalence is not generally true in our framework. Specifically, item (a) follows from the second item of Proposition 3.18; item (b) is given by Proposition 3.26; and item (c) follows from Theorem 3.32. □

4.3 Implementation Details: Godel 2

4.3.1 The toolchain

Our toolchain, summarised in Figure 4.3, is composed of a type inference tool and a type verification step. The inference tool, `migoinfer+`, is improved from the original `migoinfer` from the companion tool of MiGo [LNTY18], adding support for our extended set of primitives and fixing a few issues with edge cases in

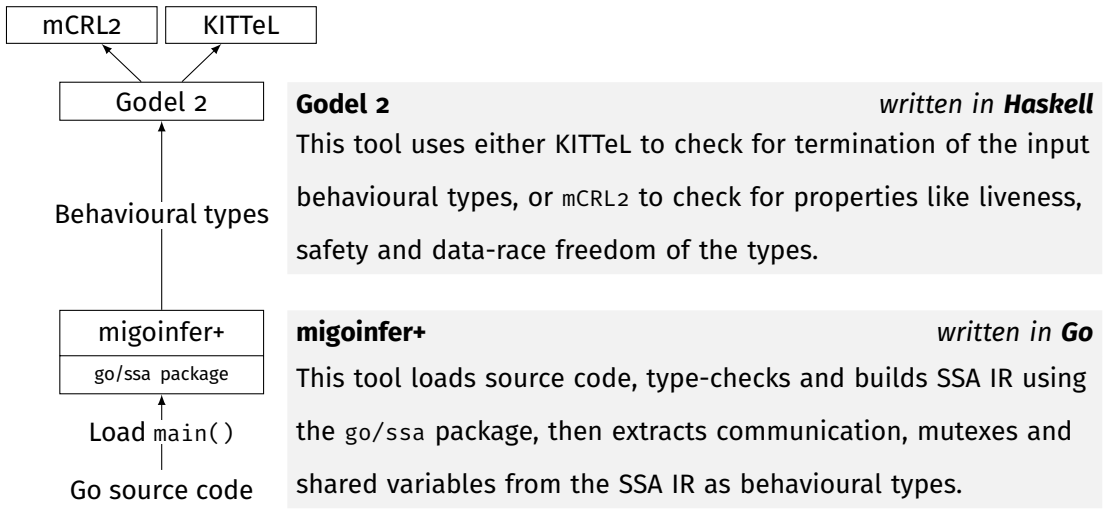


Figure 4.3: Workflow of the verification toolchain

dead call and empty functions clean-up. It is written in Go and is still part of the `gospal` framework, although we have not improved on the second tool from this framework, `ssaview`, in our fork.

The tool `migoinfer+`, as a fork of `migoinfer`, is open source and available on GitHub [Gab20c]. It extracts behavioural types from Go code, including both all previously available channel primitives — from the first version of the tool —, and our new additions in the form of eight new primitives — for shared memory and mutual exclusion locks. As a proof-of-concept, it has limitations of the same order as those in the original, channel-only, version: it can reliably extract only those supported structures that are explicitly defined in the body of an accessible function, and does not support embedding them in `struct` as a general rule, especially in the case of locks that are frequently embedded in more complex structures by developers. This usage pattern can however usually be easily converted to the flat representation of our p-o-c tool, as some examples in our (open source) benchmarks show. Undeclared passing of supported constructs to nameless children goroutines is not supported in general, as or tool may not extract the parameter passing properly, and is generally not advisable when not strictly necessary as it makes the code harder to read for external sources. Lastly, some less standard packages like the `net` package are known to break the extraction tool under certain conditions when used in programs that are to be extracted.

Note that `migoinfer+` can extract examples that are outside the verification scope of our framework, like the prime sieve example given in Figure 3.7, provided they respect the above limitations.

The type verification step, `Godel 2`, is an extension of `Godel` [LNTY18] as well, analysing our new primitives and implementing the theory presented in our work seamlessly with the channel-related properties from the earlier version. It is open source as well, and also available on GitHub [Gab20b], written in `Haskell`, and uses `mCRL2` [Tec18, GM14] as a backend model checker for the verification part of safety and liveness properties. It extracts a model and the desired properties from the MiGo output of `migoinfer+`

and then feeds them to either KITTeL [FB08] when verifying termination, or mCRL2 when verifying other properties. Since those properties are verified on the types, the conditions identified in Theorem 4.3 apply to identify whether the properties are also valid on the extracted program, especially regarding liveness properties. In particular, programs that terminate according to KITTeL should be part of the set where the properties coincide. We do not, however, have automated ways to determine whether a program has alternating conditionals in our toolchain, but we do provide a property that checks whether the program has a cycle.

Godel 2 also ensures programs are finite control before trying to verify other properties, as programs that are not finite control cannot be verified by mCRL2 because it will fail to generate a linear process specification for them.

4.3.2 Evaluation

We evaluated our proof-of-concept tool for reference on an 8-core Intel i7-7700K machine with 16 GiB of RAM, running a 64-bit Linux environment with Go 1.12.2 at the time of evaluation. The results are shown in Table 4.1 for a range of programs, inspired for part by those that were run against the first version of the toolchain by its original authors [LNTY18]. Like the other parts of our toolchain we authored, the benchmark files are available freely on GitHub [Gab20a]. The names of the programs are explicit, many of them referencing examples used throughout the previous chapters or well known examples in computer science research, like the Dining Philosopher problem used far and wide as an illustration for concurrent processes. The deposit programs are inspired by an example given in the book [DK15] I used early on in my first year as a PhD student for learning Go, referenced in the table.

All presented examples are finite control, as the properties we wish to verify could not be checked otherwise, which explains the lack of the prime sieve algorithm in the benchmark. For the sake of readability, we use the name of the local property to mark the columns representing the results of global deadlock-freedom, safety, liveness and data race freedom.

Timings are given for reference to show how the verification time by mCRL2 scales with the number of summands (possibly combined with action labels) in the linear process specification generated from the types, and the ordering of said summands. Those are directly dependent on the source of the analysed program, but are not a direct indication of what we usually envision as the program complexity, as the dining philosophers examples provided would all have the same order of complexity with usual definitions, but the channel versions have an order of magnitude more summands than the lock-based ones, and thus a far longer verification time through mCRL2.

Table 4.1: Verification results for Godel 2

Programs	LoC	Sum	ψ_g	ψ_s	ψ_l	ψ_d	Term	time (ms)
`no-race`	15	9	✓	✓	✓	✓	✓	691.45
`no-race-mutex`	24	33	✓	✓	✓	✓	✓	785.57
`no-race-mut-bad`	23	20	×	✓	×	✓	✓	721.77
`simple-race`	13	8	✓	✓	✓	×	✓	701.93
`simple-race-fix`	19	17	✓	✓	✓	✓	✓	731.73
`deposit-race`[DK15]	18	14	✓	✓	✓	×	✓	697.90
`deposit-fix`[DK15]	24	27	✓	✓	✓	✓	✓	727.43
`ch-as-lock-race`	19	20	✓	✓	✓	×	✓	753.99
`ch-as-lock-fix`	19	20	✓	✓	✓	✓	✓	745.64
`ch-as-lock-bad`	19	20	×	✓	×	✓	✓	749.97
`prod-cons-race`	38	156	✓	✓	✓	×	✓	1,903.52
`prod-cons-fix`	40	188	✓	✓	✓	✓	✓	1,971.26
`dine5-unsafe`	35	106	×	×	✓	✓	✓	6,996.27
`dine5-deadlock`	35	106	×	✓	×	✓	✓	12,278.33
`dine5-fix`	35	106	✓	✓	✓	✓	✓	8,998.04
`dine5-chan-race`	59	2672	✓	✓	✓	×	✓	$\sim$ 185min
`dine5-chan-fix`	59	2688	✓	✓	✓	✓	✓	$\sim$ 645min

LoC: Lines of Code, Sum: Summands, Term: Termination check,

✓/×: Formula is true/false

5 Related Works

The works we expanded [LNTY17, LNTY18] built behavioural types for verification of concurrency bugs for channel-based message passing. We integrate with their asynchronous calculus (a.k.a. AMiGo). These works, however, were lacking shared memory concurrency with locks and shared pointers, and did not tackle data races for shared pointers, which we do. They do not study happens before relations either (for channels). They furthermore were lacking complete proofs on their equivalence theorems for liveness, which is also addressed in our work. We also proved MiGo$^+$ satisfies the properties of the types characterised by the modal μ-calculus (Theorem 4.3). The second paper [LNTY18] has informally described them, but these have never been formalised nor proved.

Stadmüllwe, Sulzmann and Thiemann [SST16] define forkable behaviours (ie. regular expressions with a fork construct) to capture goroutine spawning in synchronous Go programs. They develop a tool based on this model to analyse directly Go programs. Their approach is sound, but suffers from several limitations, which were overcome by the works mentioned previously [LNTY17, LNTY18]: asynchrony and closing channels are not supported, and selective communication can only happen with trivial case bodies, i.e. there can not be a complex continuation in a selection. Additionally, their tool does not treat shared memory concurrency primitives and locks, and uses a tool named GURU to obtain aliasing information manually, assuming that all functions can be inlined: it thus cannot correctly analyse programs that cannot be trivially inlined, such as ones that repeatedly use the same function or make use of aliased channels.

Gleissenthall et al. [Kla19] observed that asynchronous distributed systems can be verified by only modelling *synchronisations* in the core protocol, and introduces a language IceT similar to MiGo$^+$ for specifying synchronisation in message-passing programs. Their focus was to verify functional correctness of the input protocol, and requires input programs to be synchronisable (i.e. neither deadlocks nor spurious sends in the input programs). Their approach allows for checking correctness of an implementation, given a reasonable amount of annotations. It is orthogonal to our work in which we only need to check for runtime sanity. Both approaches independently benefit the user, and should be run individually on testing code in order to check both for concurrency behavioural bugs and for implementation bugs.

Recent works [TLSZ19, DL19] provide empirical studies of Go programs, which show that almost half of concurrency bugs in Go are non-blocking bugs, mostly shared memory problems, and the remaining blocking bugs are mostly related to channel and lock misuse. That gives an incentive to make tools and implementations built on the concurrent behavioural theory, for easy detection of such bugs. Our work

is part of that effort. Furthermore, a 2022 study [CR22] shows a repartition of real-world data race bugs in Go programs by root causes, most — if not all — of which can be detected by tools built on concurrent behavioural theory, which reinforces the incentive to develop such theories and implement efficient tools built around them.

A large body of race detection tools targeting other languages such as Java are available. ThreadSanitizer (TSan) [SI09, The15, SPIV12] which is included in LLVM/Clang is one of the most widely deployed dynamic race detectors. The runtime race detector of Go [Go13] uses TSan's runtime library, which can only detect data races as they happen, while we strive to detect all potential data races before the program is run.

Midtgaard, Nielson and Nielson [MNN18] propose a subset of the Go language akin to MiGo$^+$, along with a modular approach to statically analyse processes. Their approach combines lattice-valued regular expressions and a shuffle operator allowing for separate analysis of single threads, and they prove their theory to be sound. They have a prototype implementation in OCaml to check deadlocks in synchronous message-passing programs. Meanwhile Casto et al. [CHJ$^+$19] use a protocol description language, Scribble [Scr17], which is a practical incarnation of multiparty session types [HYC08] to generate Go APIs, ensuring deadlock freedom and liveness of communications by construction. Neither of them [MNN18, CHJ$^+$19] treat communication error or data race detection, both handled in our work; nor do they treat shared variables, which our approach extends upon.

The idea to use the LTS of behavioural types for programming analysis dates back to 1994 [NN94] for Concurrent ML, and since then, it has been applied to many works [ABB$^+$17]. Some tackle mutual exclusion locks, but systematically lack support for read-write mutual exclusion locks [IK01, AFF06, HBK19]. In 2008, Kobayashi and Sandiorgi [KS08] aimed to guarantee liveness with termination of a typed π-calculus. We study wider classes in the theory, making use of the existing tool (KITTeL) for termination checking in order to integrate with reasonable scalability with our tool-chain – thus the main aim and the target (real Go programs in our case) differ from their work [KS08].

Type-level model-checking for message-passing programming was first addressed in 2002 by Chaki, Rajamani and Rehof [CRR02]. Recent applications using mCRL2 include verifications of multiparty session typed π-calculus [SY19a] and the Dotty programming language (the future Scala 3) [SYB19].

In addition, an orthogonal approach is taken by Shahare, Chabbi and Hegde [SCH23] in an attempt to solve bugs incurred by the misuse of mutual exclusion locking mechanisms in their various implementations — of which we only studied the one in the standard library of Go. They first present data showing different kinds of misuse in several locking algorithms, and the consequences these misuses can have on lock-sensitive applications. Their approach then consists in changing slightly the implementation of a number of these locking algorithms to leverage the bugs inherent to their respective known misuse

cases. They also show that their changes do not have significant performance impact for lock-sensitive applications using those implementations.

Part III

Multiparty Session π-calculus

In this part, we present a framework to verify safety of programs represented in a session types inspired π-calculus, using and extending the renewed multiparty session types (MPST) framework presented in 2019 by Scalas and Yoshida [SY19a]. Their work is a major overhaul of MPST theories that alleviated long-standing issues in many previous MPST theories — listed in Section 6.1 — by working from the ground up, instead of relying on a generalisation of binary session types. We present their original process calculus and their objectives in the first half of Chapter 6. The second half of Chapter 6 gives a detailed overview of our contributions, which are twofold:

- We first extend their calculus and type system to allow for new errors in Chapters 7 and 8, and go over an error that we found in their original framework and fixed.
- Then we extend their implementation in Chapter 9 by allowing for (simple) processes to be extracted to types — that are then checked using mostly the same method as their original tool extended to our new errors —; but also including counterexample extraction for user-friendly error reporting in the process.

6 Background: Less is More: Multiparty Sessions Types Revisited

This chapter presents the preliminary results from Scalas and Yoshida [SY19a], that we use as a base for our own. We present the multiparty session π-calculus they introduce and its semantics, and talk about the points we wish to extend and our methods for doing so while keeping the benefits of their new framework for session types.

6.1 Introduction to a New MPST Framework

Scalas and Yoshida [SY19a] revisit multiparty session types (MPST) from the ground up to address several limitations found in previous session types frameworks. In particular, previous session types frameworks tended to be incompatible with each other and the guarantees they provided were not fully satisfactory.

Namely, session consistency requirements restrained typability dramatically, rejecting even simple authentication protocols in the process. That meant rejecting many correct protocols in order to guarantee consistency in the framework, and Scalas and Yoshida address that restriction in their new framework.

More flexible frameworks using global types with full merging projections offered wider sets of protocols to be verified, but forewent consistency, making proofs dependent on subject reduction theorems not workable. Adding a consistency requirement to try and fix that looped back on the previous issue.

Those two sets of issues arose mainly from the ties of former MPST with binary session types, specifically from the generalisation of binary session types duality as described in seminal MPST work by Honda, Vasconcelos and Kubo [HVK98].

They start by introducing the following π-calculus for multiparty session protocols. The syntax is quite bare and focuses on main features of session types and their protocols. It contains inactive processes, parallel composition and process calls and definitions, as well as session-specific primitives that are selection and branching from an endpoint (i.e. channel with role) c towards role $\mathtt{q}$. Their framework also contains session restriction.

Definition 6.1 (Multiparty session π-calculus)**.** The syntax is defined as:

$$
\begin{array}{rcll}
c, d & ::= & x \mid s[\mathtt{p}] & \text{(variable, channel with role } \mathtt{p}\text{)} \\
P, Q & ::= & \mathbf{0} \mid P \mid Q \mid (\nu s)\, P & \text{(inaction, parallel, restriction)} \\
 & \mid & c[\mathtt{q}] \oplus \mathtt{m}\langle d \rangle.P \mid c[\mathtt{q}] \textstyle\sum_{i \in I} \mathtt{m}_i(x_i).P_i & \text{(selection, branching)} \\
 & \mid & \mathbf{def}\ D\ \mathbf{in}\ P \mid X\langle \widetilde{c} \rangle \mid \mathtt{err} & \text{(definition, call, error)} \\
D & ::= & X(\widetilde{x}) = P & \text{(process variable } X \text{ declaration)}
\end{array}
$$

In their syntax, the inactive process, parallel composition and restriction are standard. Selection is used to make the channel c select *deterministically* one label to communicate to role $\mathtt{q}$, along with some data, before continuing to P, while branching waits on channel c to receive a label from $\mathtt{q}$ among multiple possible choices, binding the potential data with a variable in the continuation of each branch. Process definition is standard, with a process variable being declared as a handle for a reusable (potentially recursive) process and made available for use in the process at the end of the definition structure — via the process call syntax. Finally, the error process is the result expected when the process fails, and is thus used as the result of defined inappropriate reductions in their semantics thereafter.

They give a semantics to their calculus in the form of a reduction relation. This reduction initially looks very simple, but it allows for a wide range of protocols to be expressed, and contains a built-in error detection in the form of mismatched communication labels, which is one of the most common communication errors in protocols expressed in that form.

We keep the definition of the structural congruence for Figure 7.1, since we define at the same time four sets of process objects that are used in that definition and were not explicitly defined by Scalas and Yoshida [SY19a]. Their structural congruence — only available in the technical report version of their work — is standard, including in particular commutativity and associativity for the parallel composition; simplification of inactive processes in parallel compositions, under restrictions and inside definitions; commutativity of restrictions; associativity of restriction with the parallel composition when one of the parallel processes does not involve the restricted name; associativity of definition and parallel composition if the defined process is not involved in one of the parallel processes; and commutativity of process definitions if the defined processes do not share process variables.

Definition 6.2 (Semantics)**.** The semantics of the multiparty session π-calculus is defined up-to reduction contexts and a standard structural congruence $\equiv$ that includes α-conversion. A reduction context $\mathbb{C}$ is $\mathbb{C} ::= \mathbb{C} \mid P \mid (\nu s)\, \mathbb{C} \mid \mathbf{def}\ D\ \mathbf{in}\ \mathbb{C} \mid [\,]$.

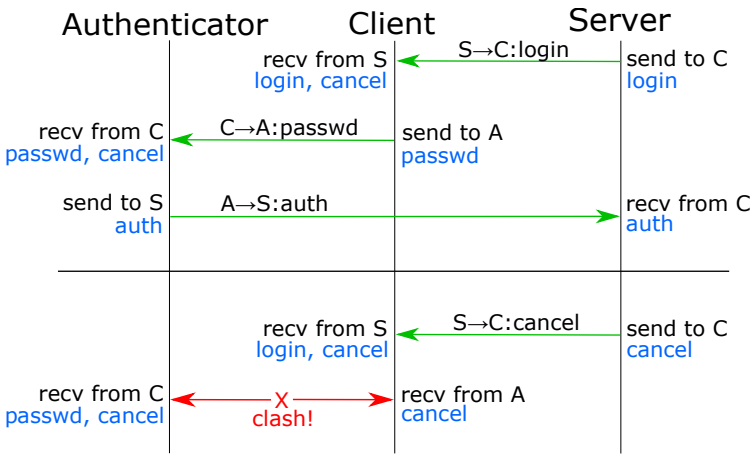


Figure 6.1: Communication diagram of an unsafe implementation of the OAuth protocol

Reduction $\rightarrow$ is then defined inductively as follows:

$$
\begin{aligned}
&\text{[R-Comm]} \quad s[\mathtt{p}][\mathtt{q}]\textstyle\sum_{i\in I}\mathtt{m}_i(x_i).P_i \mid s[\mathtt{q}][\mathtt{p}]\oplus\mathtt{m}_k\langle s'[\mathtt{r}]\rangle.Q \;\rightarrow\; P_k\{^{s'[\mathtt{r}]}/_{x_k}\} \mid Q \qquad \text{if } k \in I \\
&\text{[R-X]} \quad \mathbf{def}\ X(x_1,\ldots,x_n) = P\ \mathbf{in}\ (X\langle s_1[\mathtt{p}_1],\ldots,s_n[\mathtt{p}_n]\rangle \mid Q) \\
&\qquad\qquad \rightarrow\ \mathbf{def}\ X(x_1,\ldots,x_n) = P\ \mathbf{in}\ (P\{^{s_1[\mathtt{p}_1]}/_{x_1}\}\cdots\{^{s_n[\mathtt{p}_n]}/_{x_n}\} \mid Q) \\
&\text{[R-Ctx]} \quad P \rightarrow P' \quad \text{implies} \quad \mathbb{C}[P] \rightarrow \mathbb{C}[P'] \\
&\text{[R-Err]} \quad s[\mathtt{p}][\mathtt{q}]\textstyle\sum_{i\in I}\mathtt{m}_i(x_i).P_i \mid s[\mathtt{q}][\mathtt{p}]\oplus\mathtt{m}\langle s'[\mathtt{r}]\rangle.Q \;\rightarrow\; \mathtt{err} \qquad \text{if } \forall i \in I : \mathtt{m}_i \neq \mathtt{m} \\
&\text{[R-}\equiv\text{]} \quad P \equiv P' \quad P \rightarrow Q \quad Q \equiv Q' \quad \text{implies} \quad P' \rightarrow Q'
\end{aligned}
$$

We say that P has an error if and only if, for some $\mathbb{C}$, $P = \mathbb{C}[\mathtt{err}]$. ⌟

The rest of the system is similar enough to our own that we shall introduce it along our additions in the next few chapters. It is notable that their framework consists entirely of local types: there is no concept of "global types" in their work compared to most existing MPST works, which is both where they separate their framework from the historic tether to binary session types and how they handle the issues their work was set to tackle.

We follow their same path and take on their framework to add our own contributions. In the process, we show indirectly that this framework is fit to be extended for a wider range of errors and properties, and provides promising ground for verification of even more properties in multiparty protocols in a developer-friendly way.

6.2 Our contributions to extend the new framework

Our first contribution is to extend on their error definition by adding errors present in previous MPST frameworks that could not be represented in this new framework initially. An example is the protocol

represented in Figure 6.1 (an exact copy of Figure 2), an erroneous implementation of OAuth2, whose error could not be detected as a safety issue in the framework presented by Scalas and Yoshida [SY19a]. This is added as an actual error in our framework, along with the symmetric output-output case. This example will serve throughout our contribution to showcase the extension of the framework we propose. We also generalise the communication primitives by allowing for non-deterministic selection in the syntax of the processes.

Like in our previous work, we provide a labelled transition system (LTS) for the processes, and encapsulate the reduction from the framework we extend — in this case that of Definition 6.2 — into that LTS. Having an LTS for the processes allows us to define some of our desired properties on the processes more easily, and match them to the properties of the types. All the above is done in Chapter 7.

We then extend our types in the same way as the processes in Chapter 8. However, the change is lighter, so we don't present the original types in this context chapter and instead present it directly in our own framework, highlighting the changes and explaining their importance. We also prove several properties, some already presented in Scalas and Yoshida's work [SY19a] and proven anew to account for our changes, some new to our framework, including those relying on the new LTS of our processes.

In the process, we detected an error in the original version of their work [SY19a], for which we provide an explanation and a fix. This error happened in one of their main theorems, *session fidelity*, and was thus important to fix in order to get the full benefit of their new framework. Notified of the issue, they also provided a fix of their own: we talk about it as well, and compare to our own fix in terms of impact on the framework.

Finally, our contributions in the theory were all aimed towards easing counterexample extraction into the framework to provide an easier fixing process for users of the framework verifying their protocols. Hence, we finish by providing a proof-of-concept extension of the tool of the authors of [SY19a] — mpstk, available in its original version as a conference artifact [SY19c]. Our new version provides preliminary support for processes and type extraction, as well as counterexample extraction at the type-verification level, that are then lifted to the process when one was provided as input to the tool. Details are given in Chapter 9, with captures of the tool showing different steps in the process of automated verification.

7 The Multiparty Session π-calculus

This section describes our extended multiparty session types-inspired calculus.

7.1 Syntax of the Multiparty Session π-calculus

In the multiparty session π-calculus, let c, d represent channels, P, Q processes and D process definitions.

Definition 7.1 (Generalised multiparty session π-calculus)**.** The syntax is defined as:

$$
\begin{array}{rcll}
c, d & ::= & x \mid s[\mathtt{p}] & \text{(variable, channel with role } \mathtt{p}\text{)} \\
P, Q & ::= & \mathbf{0} \mid P \mid Q \mid (\nu s)\, P & \text{(inaction, parallel, restriction)} \\
 & \mid & c[\mathtt{q}] \oplus_{i \in I} \mathtt{m_i} \langle d_i \rangle . P_i \mid c[\mathtt{q}] \sum_{i \in I} \mathtt{m}_i (x_i) . P_i & \text{(selection, branching)} \\
 & \mid & \mathbf{def}\ D\ \mathbf{in}\ P \mid X \langle \widetilde{c} \rangle \mid \mathtt{err} & \text{(definition, call, error)} \\
D & ::= & X(\widetilde{x}) = P & \text{(process variable } X \text{ declaration)}
\end{array}
$$

⌟

Like in Scalas and Yoshida's work [SY19a], the notations are standard, where channels are represented by a session and endpoint, and can be replaced by a variable name in the process for the sake of channel-passing. We have the usual inactive process $\mathbf{0}$, parallel composition and restriction that are standard in π-calculus.

Selection is more general, extended to *non-deterministic* choices (in red), following the standard syntax [CDY14], while the calculus in the original work [SY19a] as presented in Definition 6.1 is deterministic. In our framework, when there are several branches with labels available, the process will choose one non-deterministically.

Branching is the same as in the previous work, offering all expected labels for selections to choose from and offering (potentially inactive) continuations for each of them. Process definition is as before as well, and is especially useful to write self-recursive processes, with the use of process calls. Finally, our calculus, like the one in the original work [SY19a], contains the error process, which is reached when a communication crashes.

Remark 7.2. We require that $I \neq \emptyset$ for both selection and branching. ⌟

We also use syntactic sugar for recursive processes following usual notations for recursion in process calculi and type systems, should process P contain calls to X:

$$\mu X(\widetilde{x}).P\langle \widetilde{c} \rangle ::= \mathbf{def}\ X(\widetilde{x}) = P\ \mathbf{in}\ X\langle \widetilde{c} \rangle$$

$$P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad P \mid \mathbf{0} \equiv P \qquad (\nu s)\,\mathbf{0} \equiv \mathbf{0} \qquad (\nu s)\,(\nu s')\,P \equiv (\nu s')\,(\nu s)\,P$$

$$(\nu s)\,(P \mid Q) \equiv P \mid (\nu s)\,Q \quad \text{if } s \notin \mathrm{fc}(P) \qquad \mathbf{def}\ D\ \mathbf{in}\ (\nu s)\,P \equiv (\nu s)\,(\mathbf{def}\ D\ \mathbf{in}\ P) \quad \text{if } s \notin \mathrm{fc}(D)$$

$$\mathbf{def}\ D\ \mathbf{in}\ \mathbf{0} \equiv \mathbf{0} \qquad \mathbf{def}\ D\ \mathbf{in}\ (P \mid Q) \equiv (\mathbf{def}\ D\ \mathbf{in}\ P) \mid Q \quad \text{if } \mathrm{dpv}(D) \cap \mathrm{fpv}(Q) = \emptyset$$

$$\mathbf{def}\ D\ \mathbf{in}\ (\mathbf{def}\ D'\ \mathbf{in}\ P) \equiv \mathbf{def}\ D'\ \mathbf{in}\ (\mathbf{def}\ D\ \mathbf{in}\ P)$$
$$\text{if } (\mathrm{dpv}(D) \cup \mathrm{fpv}(D)) \cap \mathrm{dpv}(D') = (\mathrm{dpv}(D') \cup \mathrm{fpv}(D')) \cap \mathrm{dpv}(D) = \emptyset$$

In the rules above, $\mathrm{fpv}(D)$ is the set of *free process variables* in D, and $\mathrm{dpv}(D)$ is the set of *declared process variables* in D, as defined in Table 7.1. While the two notations were used but not formally defined in the original work, we deemed useful to present a formal definition for them in our own work.

Figure 7.1: Structural congruence of the Multiparty session π-calculus

$$\mathrm{sc}(\mathbf{0}) = \emptyset \qquad \mathrm{sc}(P \mid Q) = \mathrm{sc}(P) \cup \mathrm{sc}(Q) \qquad \mathrm{sc}(c[\mathtt{q}] \oplus_{i \in I} \mathtt{m}_i\langle d_i \rangle.P_i) = \{c\} \cup \bigcup_{i \in I} \mathrm{sc}(P_i)$$
$$\mathrm{sc}((\nu s)\,P) = \mathrm{sc}(P) \setminus \{s[\mathtt{p}]\}_{\mathtt{p}\ \mathrm{roles}} \qquad \mathrm{sc}(c[\mathtt{q}] \textstyle\sum_{i \in I} \mathtt{m}_i(x_i).P_i) = \{c\} \cup \bigcup_{i \in I} (\mathrm{sc}(P_i) \setminus \{x_i\})$$
$$\mathrm{sc}(\mathbf{def}\ X(\tilde{x}) = P\ \mathbf{in}\ Q) = \mathrm{sc}(Q) \cup (\mathrm{sc}(P) \setminus \{\tilde{x}\}) \qquad \text{with } \mathrm{sc}(X\langle \tilde{c} \rangle) = \mathrm{sc}(P\{\tilde{c}/\tilde{x}\})$$

Figure 7.2: Subject channels of the multiparty session π-calculus

The *structural congruence*, $P \equiv Q$, is extended from Scalas and Yoshida [SY19a] and shown in Figure 7.1. Parallel composition properties and inactive process elimination are standard, specifically parallel composition is commutative and associative, and inactive processes can be eliminated in parallel compositions and simplify restrictions and definitions if they are the sole content of those structures.

As is also standard, restriction is commutative, and can commute with parallel composition if one of the parallel processes does not use the bound name, and with process definition if the definition part does not use the bound name. Process definition can commute with parallel composition if the defined process variables are not used in one of the parallel processes, as well as with other process definitions if they do not share process variables.

Remark 7.3. Since restriction, branching and declarations act as binders, following the Barendregt convention, alpha-conversion can be used to commute restriction and process definition with parallel composition to put the latter inside the binding construct. ⌟

The set of free channels with roles (resp. free variables) in P is written $\mathrm{fc}(P)$ (resp. $\mathrm{fv}(P)$), and defined in Table 7.1. Figure 7.2 defines the subject channels $\mathrm{sc}(P)$, which are the channels used for communications when $\mathrm{fpv}(P) = \emptyset$.

Table 7.1: Free/declared process variables and free channels/variables in the multiparty session π-calculus

Construct	$\mathrm{fpv}(\cdot)$	$\mathrm{dpv}(\cdot)$	$\mathrm{fc}(\cdot)$	$\mathrm{fv}(\cdot)$
x	—	—	$\emptyset$	$\{x\}$
$s[\mathtt{p}]$	—	—	$\{s[\mathtt{p}]\}$	$\emptyset$
$\mathbf{0}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$P \mid Q$	$\mathrm{fpv}(P) \cup \mathrm{fpv}(Q)$	$\mathrm{dpv}(P) \cup \mathrm{dpv}(Q)$	$\mathrm{fc}(P) \cup \mathrm{fc}(Q)$	$\mathrm{fv}(P) \cup \mathrm{fv}(Q)$
$(\nu s)\, P$	$\mathrm{fpv}(P)$	$\mathrm{dpv}(P)$	$\mathrm{fc}(P) \setminus \{s[\mathtt{p}]\}_{\mathtt{p}\ \text{roles}}$	$\mathrm{fv}(P)$
$c[\mathtt{q}] \oplus_{i \in I} \mathtt{m}_i\langle d_i \rangle.P_i$	$\bigcup_{i \in I} \mathrm{fpv}(P_i)$	$\bigcup_{i \in I} \mathrm{dpv}(P_i)$	$\mathrm{fc}(c) \cup \bigcup_{i \in I} (\mathrm{fc}(P_i) \cup \mathrm{fc}(d_i))$	$\mathrm{fv}(c) \cup \bigcup_{i \in I} (\mathrm{fv}(P_i) \cup \mathrm{fv}(d_i))$
$c[\mathtt{q}] \sum_{i \in I} \mathtt{m}_i(x_i).P_i$	$\bigcup_{i \in I} \mathrm{fpv}(P_i)$	$\bigcup_{i \in I} \mathrm{dpv}(P_i)$	$\mathrm{fc}(c) \cup \bigcup_{i \in I} \mathrm{fc}(P_i)$	$\mathrm{fv}(c) \cup \bigcup_{i \in I} (\mathrm{fv}(P_i) \setminus \{x_i\})$
$\mathbf{def}\ D\ \mathbf{in}\ P$	$\mathrm{fpv}(D) \cup \mathrm{fpv}(P) \setminus \mathrm{dpv}(D)$	$\mathrm{dpv}(P) \cup \mathrm{dpv}(D)$	$\mathrm{fc}(P) \cup \mathrm{fc}(D)$	$\mathrm{fv}(P) \cup \mathrm{fv}(D)$
$X\langle \tilde{c} \rangle$	$\{X\}$	$\emptyset$	$\mathrm{fc}(\tilde{c})$	$\mathrm{fv}(\tilde{c})$
$\mathtt{err}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$X(\tilde{x}) = P$	$\mathrm{fpv}(P) \setminus \{X\}$	$\mathrm{dpv}(P) \cup \{X\}$	$\mathrm{fc}(P)$	$\mathrm{fv}(P) \setminus \{\tilde{x}\}$

Example 7.4 (Processes)**.** We define the processes corresponding to the OAuth protocol in Figure 6.1 — both unsafe and safe—, where $P_{\mathtt{s}}$ plays the role of the server, $P_{\mathtt{c}}^{\mathsf{unsafe}}$ (resp. $P_{\mathtt{c}}^{\mathsf{safe}}$) that of the *unsafe* (resp. *safe*) client, and $P_{\mathtt{a}}$ that of the authenticator (in examples, we use constants for the payloads).

$$P_{\mathtt{a}} = \mu X(x).x[\mathtt{c}] \sum \left\{ \begin{array}{l} \mathtt{passwd}(y).x[\mathtt{s}] \oplus \mathtt{auth}\langle tt \rangle, \\ \mathtt{cancel}.X\langle x \rangle \end{array} \right\} \quad \langle s[\mathtt{a}] \rangle = s[\mathtt{a}][\mathtt{c}] \sum \left\{ \begin{array}{l} \mathtt{passwd}(y).s[\mathtt{a}][\mathtt{s}] \oplus \mathtt{auth}\langle tt \rangle, \\ \mathtt{cancel}.P_{\mathtt{a}} \end{array} \right\}$$

$$P_{\mathtt{s}} = \mu X(x).x[\mathtt{c}] \oplus \left\{ \begin{array}{l} \mathtt{login}.x[\mathtt{a}] \sum \mathtt{auth}(y), \\ \mathtt{cancel}.X\langle x \rangle \end{array} \right\} \langle s[\mathtt{s}] \rangle$$

$$P_{\mathtt{c}}^{\mathsf{unsafe}} = \mu X(x).x[\mathtt{s}] \sum \left\{ \begin{array}{l} \mathtt{login}.x[\mathtt{a}] \oplus \mathtt{passwd}\langle \texttt{"Password"} \rangle, \\ \mathtt{cancel}.x[\mathtt{a}] \sum \mathtt{cancel}.X\langle x \rangle \end{array} \right\} \langle s[\mathtt{c}] \rangle$$

$$P_{\mathtt{c}}^{\mathsf{safe}} = \mu X(x).x[\mathtt{s}] \sum \left\{ \begin{array}{l} \mathtt{login}.x[\mathtt{a}] \oplus \mathtt{passwd}\langle \texttt{"Password"} \rangle, \\ \mathtt{cancel}.x[\mathtt{a}] \oplus \mathtt{cancel}.X\langle x \rangle \end{array} \right\} \langle s[\mathtt{c}] \rangle$$

⌟

$$[\text{IN}]\ s[\mathtt{p}][\mathtt{q}]\textstyle\sum_{i\in I}\mathtt{m}_i(x_i).P_i \xrightarrow{s:\mathtt{p}\&\mathtt{q}:\mathtt{m}_k\langle s'[\mathtt{r}]\rangle} P_k\{s'[\mathtt{r}]/x_k\} \qquad [\text{OUT}]\ s[\mathtt{p}][\mathtt{q}]\oplus_{i\in I}\mathtt{m}_i\langle s_i'[\mathtt{r}_i]\rangle.P_i \xrightarrow{s:\mathtt{p}\triangleleft\mathtt{q}:\mathtt{m}_k\langle s_k'[\mathtt{r}_k]\rangle} P_k$$

$$[\text{OPEN}]\ \frac{P \xrightarrow{s:\mathtt{p}\triangleleft\mathtt{q}:\mathtt{m}\langle s'[\mathtt{r}]\rangle} P' \quad s' \neq s}{(\nu s')\,P \xrightarrow{s:\mathtt{p}\triangleleft\mathtt{q}:\mathtt{m}(s'[\mathtt{r}])} P'}$$

$$[\text{COM}]\ \frac{P \xrightarrow{s:\mathtt{p}\triangleleft\mathtt{q}:\mathtt{m}_k\langle s'[\mathtt{r}]\rangle} P' \quad Q \xrightarrow{s:\mathtt{q}\&\mathtt{p}:\mathtt{m}_k\langle s'[\mathtt{r}]\rangle} Q'}{P \mid Q \xrightarrow{s:\mathtt{p},\mathtt{q}:\mathtt{m}_k} P' \mid Q'} \qquad [\text{RES}]\ \frac{P \xrightarrow{\ell} P' \quad s \notin \mathrm{fn}(\ell)}{(\nu s)\,P \xrightarrow{\ell} (\nu s)\,P'}$$

$$[\text{R-COM}]\ \frac{P \xrightarrow{s:\mathtt{p}\triangleleft\mathtt{q}:\mathtt{m}_k\langle s'[\mathtt{r}]\rangle} P' \quad Q \xrightarrow{s:\mathtt{q}\&\mathtt{p}:\mathtt{m}_k\langle s'[\mathtt{r}]\rangle} Q'}{P \mid Q \xrightarrow{s:\mathtt{p},\mathtt{q}:\mathtt{m}_k} (\nu s')\,(P' \mid Q')} \qquad [\text{TAU}]\ \frac{P \xrightarrow{s:\mathtt{p},\mathtt{q}:\mathtt{m}} P'}{(\nu s)\,P \xrightarrow{\tau} (\nu s)\,P'}$$

$$[\text{DEF-1}]\quad \mathbf{def}\ X(x_1,\ldots,x_n) = P\ \mathbf{in}\ (X\langle s_1[\mathtt{p}_1],\ldots,s_n[\mathtt{p}_n]\rangle \mid Q) \xrightarrow{\tau} \mathbf{def}\ X(x_1,\ldots,x_n) = P\ \mathbf{in}\ (P\{s_1[\mathtt{p}_1]/x_1\}\cdots\{s_n[\mathtt{p}_n]/x_n\} \mid Q)$$

$$[\text{DEF-2}]\ \frac{P \xrightarrow{\ell} P' \quad \mathrm{bn}(D)\cap\mathrm{fn}(\ell) = \emptyset}{\mathbf{def}\ D\ \mathbf{in}\ P \xrightarrow{\ell} \mathbf{def}\ D\ \mathbf{in}\ P'} \qquad [\text{PAR}]\ \frac{P \xrightarrow{\ell} P' \quad \mathrm{fn}(Q)\cap\mathrm{bn}(\ell) = \emptyset}{P \mid Q \xrightarrow{\ell} P' \mid Q}$$

$$[\equiv]\ \frac{P' \equiv P \xrightarrow{\ell} Q \equiv Q'}{P' \xrightarrow{\ell} Q'} \qquad [\text{OO-ERR}]\ s[\mathtt{p}][\mathtt{q}]\oplus_{i\in I}\mathtt{m}_i\langle d_i\rangle.P_i \mid s[\mathtt{q}][\mathtt{p}]\oplus_{j\in J}\mathtt{m}_j'\langle d_j'\rangle.Q_j \xrightarrow{\tau} \mathtt{err}$$

$$[\text{II-ERR}]\ s[\mathtt{p}][\mathtt{q}]\textstyle\sum_{i\in I}\mathtt{m}_i(x_i).P_i \mid s[\mathtt{q}][\mathtt{p}]\sum_{j\in J}\mathtt{m}_j'(y_j).Q_j \xrightarrow{\tau} \mathtt{err}$$

$$[\text{MISM-ERR}]\ \frac{\exists j\in J, \forall i\in I, \mathtt{m}_i \neq \mathtt{m}_j}{s[\mathtt{p}][\mathtt{q}]\sum_{i\in I}\mathtt{m}_i(x_i).P_i \mid s[\mathtt{q}][\mathtt{p}]\oplus_{j\in J}\mathtt{m}_j\langle d_j\rangle.Q_j \xrightarrow{\tau} \mathtt{err}}$$

Figure 7.3: LTS of the multiparty session π-calculus (Definition 7.5)

7.2 Labelled Transition System for the Multiparty Session π-calculus

We introduce the LTS, which enables to define properties such as *liveness*, *safety* and *deadlock-freedom* on the untyped processes directly with the expressiveness of the transition system, instead of needing to define a capability set such as barbs, as usually done in behavioural typing systems like our previous work [GY20a].

Definition 7.5 (LTS)**.** We define labels for our LTS, respectively in the following order (1) τ-action; (2) explicit communication between $\mathtt{p}$ and $\mathtt{q}$ with label $\mathtt{m}$ on session s; (3) branching; (4) selection and (5) bound selection, the latter of which is a selection under a restriction (i.e. a scope opening):

$$\ell ::= \tau \mid s:\mathtt{p},\mathtt{q}:\mathtt{m} \mid s:\mathtt{p}\&\mathtt{q}:\mathtt{m}\langle s'[\mathtt{r}]\rangle \mid s:\mathtt{p}\triangleleft\mathtt{q}:\mathtt{m}\langle s'[\mathtt{r}]\rangle \mid s:\mathtt{p}\triangleleft\mathtt{q}:\mathtt{m}(s'[\mathtt{r}])$$

The reduction rules of the LTS on multiparty session processes are defined in Figure 7.3.

We use $\to$ to denote either $\xrightarrow{\tau}$ or $\xrightarrow{s:\mathtt{p},\mathtt{q}:\mathtt{m}}$ for all $s,\mathtt{p},\mathtt{q},\mathtt{m}$. We say $P \to$ (resp. $P \xrightarrow{\ell}$) if there exists P' such that $P \to P'$ (resp. $P \xrightarrow{\ell} P'$); and $P \not\to$ if there is no P' such that $P \to P'$. We define $\to^*$ as the transitive and reflexive closure of $\to$. ⌟

Rules [IN] and [OUT] respectively show the input and output capabilities over a session s as role $\mathtt{p}$, to or from $\mathtt{q}$ with label $\mathtt{m}$ and some message d. Rule [OPEN] shows scope opening when sending a bound session

name in an outgoing message. Rules [COM] and [R-COM] respectively show synchronisation of free and bound message passing, with [OPEN] being relative to the latter, and [OUT] to the former. Rule [RES] shows restriction of a name not used in an available action. Rule [DEF-1] shows the basic reduction of a definition construct, and rule [DEF-2] shows ability to reduce the context of such a definition. Rule [PAR] shows reduction for parallel composition, and rule [R-$\equiv$] shows conservation of available actions over structural congruence. Note that the last condition for rules [DEF-2] and [PAR] can always be made true up to α-conversion.

Rules [OO-ERR], [II-ERR] and [MISM-ERR] define the three kinds of error, which would happen at the same channel on communicating endpoints to perform the same action (send or receive), or on a channel with mismatching labels. These errors are present in a number of MPST works, including 2014 work by Chen, Dezani-Ciancaglini and Yoshida [CDY14].

Example 7.6 (LTS)**.** Continuing Example 7.4, let $P^{\mathsf{X}} = P_{\mathtt{s}} \mid P_{\mathtt{c}}^{\mathsf{X}} \mid P_{\mathtt{a}}$, where $\mathsf{X} \in \{\mathsf{safe}, \mathsf{unsafe}\}$:

$$P^{\mathsf{unsafe}} \to P_{\mathtt{s}} \mid s[\mathtt{c}][\mathtt{a}] \sum \mathtt{cancel}.P_{\mathtt{c}}^{\mathsf{unsafe}} \mid s[\mathtt{a}][\mathtt{c}] \sum \left\{ \begin{array}{l} \mathtt{passwd}(x).s[\mathtt{a}][\mathtt{s}] \oplus \mathtt{auth}\langle tt \rangle\,, \\ \mathtt{cancel}.P_{\mathtt{a}} \end{array} \right\} \to P_{\mathtt{s}} \mid \mathtt{err}$$

The unsafe version first reduces by synchronising between $\mathtt{c}$ and $\mathtt{s}$ here, where the server chooses to cancel in this case to showcase the error that can come up. Then, because both $\mathtt{a}$ and $\mathtt{c}$ try to receive from each other, said error happens, following rule [II-ERR]. This error does not occur in the safe version, and the process will simply start again if it chooses the cancel path:

$$P^{\mathsf{safe}} \to P_{\mathtt{s}} \mid s[\mathtt{c}][\mathtt{a}] \oplus \mathtt{cancel}.P_{\mathtt{c}}^{\mathsf{safe}} \mid s[\mathtt{a}][\mathtt{c}] \sum \left\{ \begin{array}{l} \mathtt{passwd}(x).s[\mathtt{a}][\mathtt{s}] \oplus \mathtt{auth}\langle tt \rangle\,, \\ \mathtt{cancel}.P_{\mathtt{a}} \end{array} \right\} \to P_{\mathtt{s}} \mid P_{\mathtt{c}}^{\mathsf{safe}} \mid P_{\mathtt{a}} = P^{\mathsf{safe}}$$

⌟

7.3 Properties of Multiparty Session Processes

With the untyped LTS, we can define several desired properties for the processes: a process is *safe* if it never reaches an error state in computation; *deadlock-free* if it ever only reaches a stale state when it is structurally equivalent to the inactive process; *fully terminating* if there is a numerical bound $j \in \mathbb{N}^0$ such that for all reduction paths of P longer than j, the result process is structurally equivalent to the inactive process; *never-terminating* if there is always a possible reduction in any reduction path from P; and *live* if the availability of a selection or branching action between $\mathtt{p}$ and $\mathtt{q}$ on session s implies the eventual synchronisation on s between $\mathtt{p}$ and $\mathtt{q}$.

Definition 7.7 (Properties of Multiparty Session Processes)**.** We say that process P is:

- **safe**, written safe(P), if for all P' such that $P \to^* P'$, $P' \not\to \mathtt{err}$;

- **deadlock-free**, written $\mathrm{df}(P)$, if for all P' such that $P \to^* P' \not\to$, $P' \equiv \mathbf{0}$;
- **fully terminating**, written $\mathrm{term}(P)$, if there is $j \in \mathbb{N}^0$ such that for all $n \geq j$, $P = P_0 \to P_1 \to \cdots \to P_n$ implies $P_n \equiv \mathbf{0}$;
- **never-terminating**, written $\mathrm{nterm}(P)$, if for all P' such that $P \to^* P'$, $P' \to$;
- **live**, written $\mathrm{live}(P)$, if for all P' such that $P \to^* P'$, having either $P' \xrightarrow{s:\mathtt{p}\lhd\mathtt{q}:\mathtt{m}\langle d\rangle}$ or $P' \xrightarrow{s:\mathtt{q}\&\mathtt{p}:\mathtt{m}'\langle d\rangle}$ implies $P' \to^* P'' \xrightarrow{s:\mathtt{p},\mathtt{q}:\mathtt{m}}$ for some P''. ⌟

Example 7.8. The unsafe version P^{unsafe} in Examples 7.4 and 7.6 is not safe, not deadlock-free, not terminating and not never-terminating: one of its branches ends in an error showing in the process (and no further reduction is possible), as seen in Example 7.6. It is not live either, because when the error occurs, $P_{\mathtt{s}}$ attempts to communicate with the client again, but is never met with any possible reduction.

The safe version P^{safe} in the same examples satisfies safety, deadlock-freedom and liveness. It is however not fully terminating, since the cancel branch can be taken arbitrarily many times; and neither is it never-terminating, since the login branch terminates in three steps. ⌟

8 Multiparty Session Types for the Multiparty Session π-calculus

This section introduces a local version of the multiparty session type system (i.e., without global types) with a non-deterministic selection, extending the types presented in the original work [SY19a]. Because we develop a counterexample guided framework, we do not aim for soundness as an inherent property of the typing rules, unlike some variants of MPST. Yet, we need to show (1) our new error definitions are *sound* with respect to the type system; and (2) for which subset of the processes, our error definitions are *complete*. These properties enable us to link a counterexample found by the tool to the matching error in the process. This chapter also details an error in the session fidelity theorem in Scalas and Yoshida's work [SY19a], and provides a fix, as it is a crucial theorem to our approach.

8.1 Syntax of the Types

The syntax of multiparty session types is defined below.

Definition 8.1 (Types)**.** The syntax of multiparty session types is:

$$S, T ::= \mathtt{p}\&_{i\in I}\mathtt{m}_i\langle S_i\rangle.S_i' \mid \mathtt{p}\oplus_{i\in I}\mathtt{m}_i\langle S_i\rangle.S_i' \mid \mathbf{end} \mid \mu\mathbf{t}.S \mid \mathbf{t}$$

with $I \neq \emptyset$, $\mathtt{m}_i$ pairwise distinct, and S_i is a closed type: we require recursion variables to be bound by a recursion operator. We often omit **end**. ⌟

A term in the types corresponds to the behaviour of a specific channel as an endpoint (since channels are directional in our framework), containing a model of the actions of this channel only; when a π-calculus term could make use of several channels in succession. This allows to have a model for the interactions of each channel with the others, as they are the key participants in the system.

The syntax above describes in order external choice (branching) and internal choice (selection), inactive types represented as **end**, recursive types, and finally type recursion variables. For choices, we first have the target role $\mathtt{p}$, then the set of possible choices described as labels $\mathtt{m}_i$, behaviour of the communicated data (if applicable) as type S_i, and the behaviour of the continuation of our endpoint if this choice was made as type S_i'. Note the fact that we look at the behaviour of communicated data only, instead of the data itself: this makes it possible to check the system communicates and works properly, while reducing the strain on

model checking by not imposing the verification of big graphs of inter-dependent channels used as both communicating endpoints and communicated data.

Example 8.2 (Types). The types corresponding to $s[\mathtt{s}]$, $s[\mathtt{c}]$ (unsafe version), $s[\mathtt{a}]$ and $s[\mathtt{c}]$ (safe version) from Example 7.4 are:

$$S_{\mathtt{s}} = \mu \mathbf{t}.\mathtt{c}\oplus\left\{\begin{array}{l}\mathtt{login}.\mathtt{a}\&\mathtt{auth}\langle \mathrm{Bool}\rangle,\\ \mathtt{cancel}.\mathbf{t}\end{array}\right\} \qquad S_{\mathtt{c}}^{\mathrm{err}} = \mu \mathbf{t}.\mathtt{s}\&\left\{\begin{array}{l}\mathtt{login}.\mathtt{a}\oplus\mathtt{passwd}\langle \mathrm{Str}\rangle,\\ \mathtt{cancel}.\mathtt{a}\&\mathtt{cancel}.\mathbf{t}\end{array}\right\}$$

$$S_{\mathtt{a}} = \mu \mathbf{t}.\mathtt{c}\&\left\{\begin{array}{l}\mathtt{passwd}\langle \mathrm{Str}\rangle.\mathtt{s}\oplus\mathtt{auth}\langle \mathrm{Bool}\rangle,\\ \mathtt{cancel}.\mathbf{t}\end{array}\right\} \quad S_{\mathtt{c}}^{\mathrm{safe}} = \mu \mathbf{t}.\mathtt{s}\&\left\{\begin{array}{l}\mathtt{login}.\mathtt{a}\oplus\mathtt{passwd}\langle \mathrm{Str}\rangle,\\ \mathtt{cancel}.\mathtt{a}\oplus\mathtt{cancel}.\mathbf{t}\end{array}\right\}$$

⌟

8.2 LTS of the Types and Type Safety

Types need to be used together in the form of contexts, in order to assign them to processes in the typing discipline.

Definition 8.3 (Typing Contexts). Θ denotes a partial mapping from process variables to n-tuples of types, and Γ denotes a partial mapping from channels to types, defined as:

$$\Theta \ ::= \ \emptyset \mid \Theta, X{:}S_1,\ldots,S_n \qquad\qquad \Gamma \ ::= \ \emptyset \mid \Gamma, x{:}S \mid \Gamma, s[\mathtt{p}]{:}S$$

The *composition* Γ_1,Γ_2 is defined if and only if $\mathrm{dom}(\Gamma_1)\cap\mathrm{dom}(\Gamma_2)=\emptyset$, where $\mathrm{dom}(\Gamma)$ denotes the set of channels described in Γ. We write $s\notin\Gamma$ if and only if $\forall \mathtt{p} : s[\mathtt{p}]\notin\mathrm{dom}(\Gamma)$ (i.e., session s does not occur in Γ). We write $\mathrm{dom}(\Gamma)=\{s\}$ if and only if $\forall c\in\mathrm{dom}(\Gamma)$ there is $\mathtt{p}$ such that $c=s[\mathtt{p}]$ (i.e., Γ only contains session s). Finally, we write $\Gamma\leqslant\Gamma'$ if and only if $\mathrm{dom}(\Gamma)=\mathrm{dom}(\Gamma')$ and $\forall c\in\mathrm{dom}(\Gamma):\Gamma(c)\leqslant\Gamma'(c)$, where $\leqslant$ is the standard subtyping relation defined below. ⌟

Definition 8.4 (Subtyping). The **subtyping relation** $\leqslant$ for session types is *co*inductively defined by:

$$\frac{\forall i\in I \quad S_i\leqslant T_i \quad S_i'\leqslant T_i'}{\mathtt{p}\&_{i\in I\cup J}\mathtt{m}_i\langle S_i\rangle.S_i' \leqslant \mathtt{p}\&_{i\in I}\mathtt{m}_i\langle T_i\rangle.T_i'}\ [\textsc{Sub}\text{-}\&] \qquad \frac{I\subseteq J \quad \forall i\in I \quad T_i\leqslant S_i \quad S_i'\leqslant T_i'}{\mathtt{p}\oplus_{i\in I}\mathtt{m}_i\langle S_i\rangle.S_i' \leqslant \mathtt{p}\oplus_{i\in I\cup J}\mathtt{m}_i\langle T_i\rangle.T_i'}\ [\textsc{Sub}\text{-}\oplus]$$

$$\frac{}{\mathbf{end}\leqslant\mathbf{end}}\ [\textsc{Sub}\text{-}\mathbf{end}] \qquad \frac{S\{\mu \mathbf{t}.S/\mathbf{t}\}\leqslant T}{\mu \mathbf{t}.S\leqslant T}\ [\textsc{Sub}\text{-}\mu\mathrm{L}] \qquad \frac{S\leqslant T\{\mu \mathbf{t}.T/\mathbf{t}\}}{S\leqslant\mu \mathbf{t}.T}\ [\textsc{Sub}\text{-}\mu\mathrm{R}]$$

⌟

In Definition 8.4, rules [Sub-&] and [Sub-$\oplus$] define **subtyping on branch/select types**: [Sub-&] is contravariant in both carried types and number of branches, while [Sub-$\oplus$] is covariant in both. This formalises the intuition of smaller types having more external choices and less internal ones. The rest of the rules are usual in the literature: [Sub-**end**] gives that **end** is the only subtype of itself, and the recursion rules relate types up-to their unfolding as coinductive subtyping prescribes.

Example 8.5 (Subtyping)**.** We can use a part of the types in Example 8.2 to showcase how subtyping works, by showing that a supertype of S_c^{err} can contain fewer capabilities in the branching action:

$$\dfrac{\dfrac{\mathtt{a}\oplus\mathtt{passwd}\langle\mathrm{Str}\rangle \leqslant \mathtt{a}\oplus\mathtt{passwd}\langle\mathrm{Str}\rangle}{\mathtt{s}\&\left\{\begin{array}{l}\mathtt{login}.\mathtt{a}\oplus\mathtt{passwd}\langle\mathrm{Str}\rangle\,,\\ \mathtt{cancel}.\mathtt{a}\&\mathtt{cancel}.S_c^{\text{err}}\end{array}\right\} \leqslant \mathtt{s}\&\mathtt{login}.\mathtt{a}\oplus\mathtt{passwd}\langle\mathrm{Str}\rangle}\,[\text{SUB-}\&]}{S_c^{\text{err}} = \mu\mathbf{t}.\mathtt{s}\&\left\{\begin{array}{l}\mathtt{login}.\mathtt{a}\oplus\mathtt{passwd}\langle\mathrm{Str}\rangle\,,\\ \mathtt{cancel}.\mathtt{a}\&\mathtt{cancel}.\mathbf{t}\end{array}\right\} \leqslant \mathtt{s}\&\mathtt{login}.\mathtt{a}\oplus\mathtt{passwd}\langle\mathrm{Str}\rangle}\,[\text{SUB-}\mu\text{L}]$$

⌟

We introduce the LTS of the types in order to have context reduction, as well as to enable us to define desirable properties for the types similarly to the processes.

Definition 8.6 (Typing Context Reduction)**.** Reduction labels, respectively corresponding to (1) branching; (2) selection; (3) communication, are defined as: $\alpha ::= s{:}\mathtt{p}\&\mathtt{q}{:}\mathtt{m}(S) \mid s{:}\mathtt{p}\oplus\mathtt{q}{:}\mathtt{m}(S) \mid s{:}\mathtt{p},\mathtt{q}{:}\mathtt{m}$.

The *typing context transition* $\xrightarrow{\alpha}$ is inductively defined by the rules:

$$[\Gamma\text{-}\oplus]\ \frac{k \in I}{s[\mathtt{p}]{:}\mathtt{q}\oplus_{i\in I}\mathtt{m}_i\langle S_i\rangle.S_i' \xrightarrow{s:\mathtt{p}\oplus\mathtt{q}:\mathtt{m}_k(S_k)} s[\mathtt{p}]{:}S_k'} \qquad [\Gamma\text{-}\mu]\ \frac{\Gamma, c{:}S\{\mu\mathbf{t}.S/\mathbf{t}\} \xrightarrow{\alpha} \Gamma'}{\Gamma, c{:}\mu\mathbf{t}.S \xrightarrow{\alpha} \Gamma'}$$

$$[\Gamma\text{-}\&]\ \frac{k \in I}{s[\mathtt{p}]{:}\mathtt{q}\&_{i\in I}\mathtt{m}_i\langle S_i\rangle.S_i' \xrightarrow{s:\mathtt{p}\&\mathtt{q}:\mathtt{m}_k(S_k)} s[\mathtt{p}]{:}S_k'} \qquad [\Gamma\text{-CONG}]\ \frac{\Gamma \xrightarrow{\alpha} \Gamma'}{\Gamma, c{:}S \xrightarrow{\alpha} \Gamma', c{:}S}$$

$$[\Gamma\text{-COMM}]\ \frac{\Gamma_1 \xrightarrow{s:\mathtt{p}\oplus\mathtt{q}:\mathtt{m}(S)} \Gamma_1' \quad \Gamma_2 \xrightarrow{s:\mathtt{q}\&\mathtt{p}:\mathtt{m}(T)} \Gamma_2' \quad T \leqslant S}{\Gamma_1, \Gamma_2 \xrightarrow{s:\mathtt{p},\mathtt{q}:\mathtt{m}} \Gamma_1', \Gamma_2'}$$

We write $\Gamma \xrightarrow{\alpha}$ if and only if $\Gamma \xrightarrow{\alpha} \Gamma'$ for some Γ' and $\Gamma \not\xrightarrow{\alpha}$ for its negation (i.e., when there is no Γ' such that $\Gamma \xrightarrow{\alpha} \Gamma'$). The *reduction* $\Gamma \to \Gamma'$ is defined if and only if $\Gamma \xrightarrow{s:\mathtt{p},\mathtt{q}:\mathtt{m}} \Gamma'$ for some $s, \mathtt{p}, \mathtt{q}, \mathtt{m}$. We write $\Gamma \to$ if and only if $\Gamma \to \Gamma'$ for some Γ', and $\Gamma \not\to$ for its negation (i.e., when there is no Γ' such that $\Gamma \to \Gamma'$). We define $\to^*$ as the reflexive and transitive closure of $\to$. ⌟

The rules are standard [SY19a] and describe how typing contexts evolve after each (partial) communicating action, with $\leqslant$ the standard subtyping relation described in Definition 8.4. In particular, selection and branching are compatible for communication only when the payload types are compatible, which is shown using subtyping; a reducing recursive type that was unfolded to show the reduction can be folded and preserve the reduction; and we can concatenate both the redex and the reduct types with the same typing context and preserve the reduction.

Definition 8.7 (Typing Context Safety)**.** φ is a *safety property* of typing contexts if and only if:

[S-⊕&] $\varphi\Big(\Gamma,\, s[\mathtt{p}]{:}\mathtt{q}{\oplus_{i\in I}}\mathtt{m}_i\langle S_i\rangle.S_i',\, s[\mathtt{q}]{:}\mathtt{p}\&_{j\in J}\mathtt{m}_j\langle T_j\rangle.T_j'\Big) \;\Rightarrow\; I\subseteq J$, and $\forall i\in I : T_i \leqslant S_i$;

[S-⊕] $\varphi\big(\Gamma,\, s[\mathtt{p}]{:}\mathtt{q}{\oplus_{i\in I}}\mathtt{m}_i\langle S_i\rangle.S_i'\big) \;\Rightarrow\; \forall \mathtt{m}, T : \neg\Gamma \xrightarrow{\not\;\; s:\mathtt{q}\oplus\mathtt{p}:\mathtt{m}(T)}$;

[S-&] $\varphi\big(\Gamma,\, s[\mathtt{p}]{:}\mathtt{q}\&_{i\in I}\mathtt{m}_i\langle S_i\rangle.S_i'\big) \;\Rightarrow\; \forall \mathtt{m}, T : \neg\Gamma \xrightarrow{\not\;\; s:\mathtt{q}\&\mathtt{p}:\mathtt{m}(T)}$;

[S-μ] $\varphi(\Gamma,\, s[\mathtt{p}]{:}\mu\mathbf{t}.S) \;\Rightarrow\; \varphi(\Gamma,\, s[\mathtt{p}]{:}S\{{}^{\mu\mathbf{t}.S}/_{\mathbf{t}}\})$;

[S-$\to$] $\varphi(\Gamma)$ and $\Gamma \to \Gamma' \;\Rightarrow\; \varphi(\Gamma')$.

We say Γ *is safe*, written safe(Γ), if $\varphi(\Gamma)$ for some safety property φ. We define safe as the largest safety property, i.e., the union of all safety properties. ⌟

There are two new safety items w.r.t. Scalas and Yoshida's definition [SY19a], [S-⊕] and [S-&]. Both are formed on the same principle: if the typing context has a send (resp. receive) action available on $s[\mathtt{p}]$ towards $\mathtt{q}$, then it must not have a send (resp. receive) action immediately available on $s[\mathtt{q}]$ towards $\mathtt{p}$. Those rules allow to detect the wider range of errors introduced back in Definition 7.5.

Example 8.8. Form the following typing context from Example 8.2: $\Gamma = s[\mathtt{s}]{:}S_\mathtt{s}, s[\mathtt{c}]{:}S_\mathtt{c}^{\mathrm{err}}, s[\mathtt{a}]{:}S_\mathtt{a}$. This typing context is not safe, because one of its reductions after unfolding does not respect [S-&]. Thus, it then does not respect [S-$\to$] and [S-μ]. By contradiction, suppose safe(Γ). Then, by [S-μ] we have safe(Γ'), where Γ' is:

$$s[\mathtt{s}]{:}\mathtt{c}\oplus\left\{\begin{array}{l}\mathtt{login.a}\&\mathtt{auth}\langle\mathrm{Bool}\rangle.\mathbf{end},\\ \mathtt{cancel}.S_\mathtt{s}\end{array}\right\}, s[\mathtt{c}]{:}\mathtt{s}\&\left\{\begin{array}{l}\mathtt{login.a}\oplus\mathtt{passwd}\langle\mathrm{Str}\rangle.\mathbf{end},\\ \mathtt{cancel.a}\&\mathtt{cancel}.S_\mathtt{c}^{\mathrm{err}}\end{array}\right\},$$

$$s[\mathtt{a}]{:}\mathtt{c}\&\left\{\begin{array}{l}\mathtt{passwd}\langle\mathrm{Str}\rangle.\mathtt{s}\oplus\mathtt{auth}\langle\mathrm{Bool}\rangle.\mathbf{end},\\ \mathtt{cancel}.S_\mathtt{a}\end{array}\right\}$$

In turn, by [S-$\to$] we deduce safe(Γ''), where:

$$\Gamma'' = s[\mathtt{s}]{:}S_\mathtt{s}, s[\mathtt{c}]{:}\mathtt{a}\&\mathtt{cancel}.S_\mathtt{c}^{\mathrm{err}}, s[\mathtt{a}]{:}\mathtt{c}\&\left\{\begin{array}{l}\mathtt{passwd}\langle\mathrm{Str}\rangle.\mathtt{s}\oplus\mathtt{auth}\langle\mathrm{Bool}\rangle.\mathbf{end},\\ \mathtt{cancel}.S_\mathtt{a}\end{array}\right\}$$

But Γ'' does not validate [S-&]. Thus, Γ'' is not safe, so Γ is not safe either. ⌟

8.3 Typing Rules

We now justify our error definitions satisfy type soundness. The type system uses two kinds of typing contexts: Θ to assign an n-tuple of types to each process variable X (one type per argument), and Γ to map variables and channels with roles to session types. Then the judgement is defined as:

$$\Theta \vdash P \blacktriangleright \Gamma \qquad \text{(with } \Theta \text{ omitted when empty)} \tag{8.1}$$

meaning: "given the process types in Θ, P uses its variables and channels *linearly* according to Γ." In this context, using the channels and variables linearly according to Γ means that the process follows for each

$$\frac{\Theta(X) = S_1, \ldots, S_n}{\Theta \vdash X{:}S_1, \ldots, S_n} \text{ [T-X]} \qquad \frac{S' \leqslant S}{c{:}S \vdash c{:}S'} \text{ [T-SUB]} \qquad \frac{\forall i \in 1..n \quad c_i{:}S_i \vdash c_i{:}\mathbf{end}}{\mathrm{end}(c_1{:}S_1, \ldots, c_n{:}S_n)} \text{ [T-end]} \qquad \frac{\mathrm{end}(\Gamma)}{\Theta \vdash \mathbf{0} \blacktriangleright \Gamma} \text{ [T-0]}$$

$$\frac{\Theta \vdash X{:}S_1, \ldots, S_n \quad \mathrm{end}(\Gamma_0) \quad \forall i \in 1..n \quad \Gamma_i \vdash c_i{:}S_i}{\Theta \vdash X\langle c_1, \ldots, c_n\rangle \blacktriangleright \Gamma_0, \Gamma_1, \ldots, \Gamma_n} \text{ [T-CALL]} \qquad \frac{\Theta \vdash P_1 \blacktriangleright \Gamma_1 \qquad \Theta \vdash P_2 \blacktriangleright \Gamma_2}{\Theta \vdash P_1 \mid P_2 \blacktriangleright \Gamma_1, \Gamma_2} \text{ [T-|]}$$

$$\frac{\Theta, X{:}S_1, \ldots, S_n \vdash P \blacktriangleright x_1{:}S_1, \ldots, x_n{:}S_n \qquad \Theta, X{:}S_1, \ldots, S_n \vdash Q \blacktriangleright \Gamma}{\Theta \vdash \mathbf{def}\ X(x_1{:}S_1, \ldots, x_n{:}S_n) = P\ \mathbf{in}\ Q \blacktriangleright \Gamma} \text{ [T-def]}$$

$$\frac{\Gamma_1 \vdash c{:}\mathtt{q}\&_{i\in I}\mathtt{m}_i\langle S_i\rangle.S_i' \quad \forall i \in I \quad \Theta \vdash P_i \blacktriangleright \Gamma, y_i{:}S_i, c{:}S_i'}{\Theta \vdash c[\mathtt{q}]\textstyle\sum_{i\in I}\mathtt{m}_i(y_i{:}S_i).P_i \blacktriangleright \Gamma, \Gamma_1} \text{ [T-\&]}$$

$$\frac{\Gamma_1 \vdash c{:}\mathtt{q}\oplus_{i\in I}\mathtt{m}_i\langle S_i\rangle.S_i' \quad \forall i \in I \quad \Gamma_i \vdash d_i{:}S_i \quad \Theta \vdash P_i \blacktriangleright \Gamma, c{:}S_i'}{\Theta \vdash c[\mathtt{q}]\oplus_{i\in I}\mathtt{m}_i\langle d_i\rangle.P_i \blacktriangleright \Gamma, \Gamma_1, \bigcup_{i\in I}\Gamma_i} \text{ [T-}\oplus\text{]}$$

$$\frac{\Gamma' = \{s[\mathtt{p}]{:}S_\mathtt{p}\}_{\mathtt{p}\in I} \quad \varphi(\Gamma') \quad s \notin \Gamma \quad \Theta \vdash P \blacktriangleright \Gamma, \Gamma'}{\Theta \vdash (\nu s{:}\Gamma')\ P \blacktriangleright \Gamma} \text{ [T-}\nu\text{]} \quad \varphi \text{ a safety property}$$

Figure 8.1: Typing rules for the multiparty session π-calculus

of them the actions supported by Γ for that object in a linear pattern: it executes each action for each type only once, after which the action is consumed from the type and unavailable to the process. The typing rules are shown in Figure 8.1, with highlight on the non-deterministic selection.

Typing a process call is done through having the process variable associated in Θ to as many types as it has arguments, and those types should be the ones expected for said arguments in order, matching with the types of each argument present in the typing context Γ. A parallel composition is typed by simply concatenating the typing environments of both sides, which means that we can only type parallel processes that use disjoint sets of subject channel endpoints ($\mathrm{sc}(P) \cap \mathrm{sc}(Q) = \emptyset$). When typing a process definition, since the process variable can appear in both the defined process (allowing recursion) and the target process (right-hand side of the "in"), we type each of them with the process variable in the typing environment, but the definition itself being a binder for the process variable used, it does not appear in the typing of the whole definition; and neither do the variables in the typing context, while they appear in that of the defined process (right-and side of the equal, left-hand side of the "in"): they are instead annotated to each variable in the process variable part of the definition (left-hand side of the equal).

When typing a branch or select construct, each branch continuation is typed separately in the premises, and a typing context is build around the behaviour of the channel in each branch. For branching, the types of the bound variables are absent from the typing context of the conclusion, since they are bound, but is annotated to the process, like the types of the variables for the arguments of a process variable in a definition. For selection, the types of the payloads are present in the typing context of the conclusion, as

they can be used by recipient processes, thus being made available through the combination of this rule and the parallel composition rule. Finally, typing restriction is done by isolating all the types of channels with roles on session s, thus we decide to verify at this moment that the typing context formed by all those endpoints is safe (Definition 8.7). This environment is then annotated to the restricted session name in the process, and absent from the typing environment in the conclusion of the rule.

The four rules at the top describe meta-typing rules, the first one is how we check that a typing environment types a process variable properly, the second rule is how we define for a typing context to type a channel, and the last two give what we call an inactive environment: the end() operation defines an environment in which all channels are inactive (typed by **end**), with the empty environment being trivially inactive. An inactive process can only be typed by an inactive environment — including the empty environment.

Lemma 8.9. The following rule for the shortcut is deduced from the typing rules:

$$[\text{T-rec}]\ \frac{\Theta, X{:}S_1, \ldots, S_n \vdash P \blacktriangleright x_1{:}S_1, \ldots, x_n{:}S_n \quad \text{end}(\Gamma_0) \quad \forall i \in 1..n \quad \Gamma_i \vdash c_i{:}S_i}{\Theta \vdash \mu X(x_1{:}S_1, \ldots, x_n{:}S_n).P\langle c_1, \ldots, c_n\rangle \blacktriangleright \Gamma_0, \Gamma_1, \ldots, \Gamma_n}$$

⌟

Proof. By combining [T-CALL] and [T-**def**], and annotating the argument variables in the shortcut in the same way they are in the process variable definition in the conclusion of [T-**def**]. □

Example 8.10. The authenticator of our running example is typed as follows. Let $\Gamma_1 = s[\mathtt{a}]{:}S_\mathtt{a}$, some Γ_0 such that end(Γ_0), and $S'_\mathtt{a} = \mathtt{c}\&\left\{\begin{array}{l}\mathtt{passwd}\langle\mathrm{Str}\rangle.\mathtt{s}\oplus\mathtt{auth}\langle\mathrm{Bool}\rangle\,,\\ \mathtt{cancel}.S_\mathtt{a}\end{array}\right\}$. Using [SUB-$\mu$L], we have that $S'_\mathtt{a} \leqslant S_\mathtt{a}$, where we remind $S_\mathtt{a} = \mu\mathbf{t}.\mathtt{c}\&\left\{\begin{array}{l}\mathtt{passwd}\langle\mathrm{Str}\rangle.\mathtt{s}\oplus\mathtt{auth}\langle\mathrm{Bool}\rangle\,,\\ \mathtt{cancel}.\mathbf{t}\end{array}\right\}$. Then:

$$\dfrac{\dfrac{x{:}\mathtt{s}\oplus\mathtt{auth}\langle\mathrm{Bool}\rangle \vdash x{:}\mathtt{s}\oplus\mathtt{auth}\langle\mathrm{Bool}\rangle \quad X{:}S_\mathtt{a} \vdash \mathbf{0} \blacktriangleright x{:}\mathbf{end}}{X{:}S_\mathtt{a} \vdash x[\mathtt{s}]\oplus\mathtt{auth}\langle tt\rangle \blacktriangleright y{:}\mathrm{Str}, x{:}\mathtt{s}\oplus\mathtt{auth}\langle\mathrm{Bool}\rangle}\,[\text{T-}\oplus] \quad \dfrac{\dfrac{S'_\mathtt{a} \leqslant S_\mathtt{a}}{x{:}S_\mathtt{a} \vdash x{:}S'_\mathtt{a}}\,[\text{T-Sub}] \quad \dfrac{X{:}S_\mathtt{a} \vdash X{:}S_\mathtt{a} \quad x{:}S_\mathtt{a} \vdash x{:}S_\mathtt{a}}{X{:}S_\mathtt{a} \vdash X\langle x\rangle \blacktriangleright x{:}S_\mathtt{a}}\,[\text{T-Call}]}{}\,[\text{T-}\&]}{X{:}S_\mathtt{a} \vdash x[\mathtt{c}]\sum\left\{\begin{array}{l}\mathtt{passwd}(y).x[\mathtt{s}]\oplus\mathtt{auth}\langle tt\rangle\,,\\ \mathtt{cancel}.X\langle x\rangle\end{array}\right\} \blacktriangleright x{:}S_\mathtt{a} \qquad \text{end}(\Gamma_0) \qquad \Gamma_1 \vdash s[\mathtt{a}]{:}S_\mathtt{a}}\,[\text{T-rec}]$$

$$\emptyset \vdash P_\mathtt{a} \blacktriangleright \Gamma_0, \Gamma_1$$

⌟

1. Γ is **safe**, written safe(Γ), if and only if it satisfies a safety property as described in Definition 8.7
2. Γ is **deadlock-free**, written df(Γ), if and only if: $\Gamma \to^* \Gamma' \not\to$ implies end(Γ')
3. Γ is **terminating**, written term(Γ), if and only if: Γ is deadlock-free, and there is $j \in \mathbb{N}^0$ such that for all $n \geq j$, $\Gamma = \Gamma_0 \to \Gamma_1 \to \cdots \to \Gamma_n$ implies end(Γ_n)
4. Γ is **never-terminating**, written nterm(Γ), if and only if: $\Gamma \to^* \Gamma'$ implies $\Gamma' \to$
5. Γ is **live**, written live(Γ), if and only if: $\varphi(\Gamma)$, for some φ such that

   [L-&] whenever $\varphi(\Gamma', s[\mathtt{q}]:S)$ with $S = \mathtt{p}\&_{i\in I}\mathtt{m}_i\langle S_i\rangle.S_i'$, $\exists i \in I$: $\exists \Gamma''$: $\Gamma', s[\mathtt{q}]:S \to^* \Gamma'', s[\mathtt{q}]:S_i'$

   [L-$\oplus$] whenever $\varphi(\Gamma', s[\mathtt{p}]:S)$ with $S = \mathtt{q}\oplus_{i\in I}\mathtt{m}_i\langle S_i\rangle.S_i'$, $\forall i \in I$: $\exists \Gamma''$: $\Gamma', s[\mathtt{p}]:S \to^* \Gamma'', s[\mathtt{p}]:S_i'$

   [S-μ] $\varphi(\Gamma,\ s[\mathtt{p}]:\mu\mathbf{t}.S) \ \Rightarrow\ \varphi(\Gamma,\ s[\mathtt{p}]:S\{\mu\mathbf{t}.S/\mathbf{t}\})$;

   [S-$\to$] $\varphi(\Gamma)$ and $\Gamma \to \Gamma' \ \Rightarrow\ \varphi(\Gamma')$.

Figure 8.2: Properties of typing contexts

8.4 Properties of Typed Multiparty Session Processes

With the type system above, we can expect to express our safety and liveness properties in the types world, where they are easier to model check thanks to the erasing of data dependency in the reductions.

In Figure 8.2, we define a list of desirable properties of the typing contexts. *Safety* is as defined earlier; *deadlock-freedom* is achieved when a typing context stops reducing only when it has no more available actions; a typing context is *terminating* when it is deadlock-free and the number of its reductions in any path is bounded; *never-termination* is on the contrary when there is always a reduction possible, whatever the previously chosen actions; and finally a typing context is *live* if **(1)** whenever a branching action is available, one of its branches can eventually be reduced, **(2)** whenever a selection action is available, all its branches can eventually be reduced, both of those at all times.

8.5 Relating Properties Between the π-calculus and the Types

Figure 8.3 shows the relation between properties on the processes and properties on the type system. We need four main theorems in order to relate all of our desired properties accordingly and make sure the model checking of types proves that our properties are respected by the corresponding π-calculus process.

First, like Scalas and Yoshida [SY19a], we have Subject Reduction, since we extend their framework:

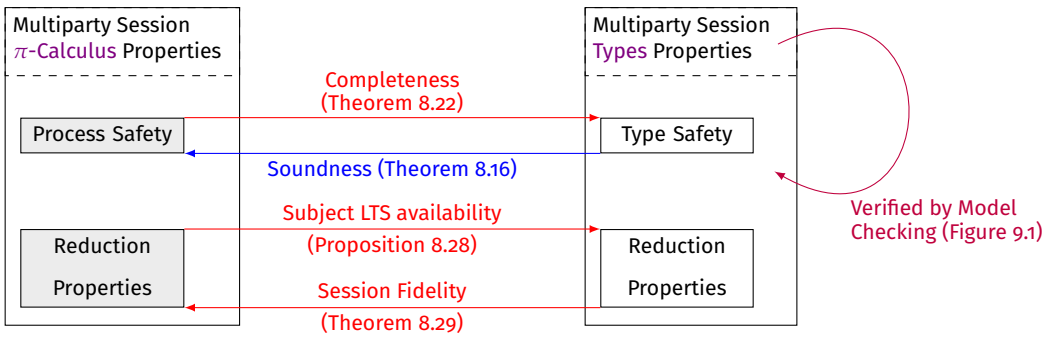


Figure 8.3: Relation of the π-calculus properties, types properties and model checking

Proposition 8.11 (Subject Reduction)**.** Suppose $\vdash P \blacktriangleright \Gamma$, safe$(\Gamma)$, and $P \to P'$, then there exists Γ' safe such that: $\Gamma \to^* \Gamma'$ and $\vdash P' \blacktriangleright \Gamma'$. ⌟

We will need a few basic lemmata for some cases, so we prove them first:

Lemma 8.12 (Substitution)**.** *Assume* $\Theta \vdash P \blacktriangleright \Gamma, x{:}S$ *and* $\Gamma' \vdash s[\mathtt{p}]{:}S$*, with* Γ, Γ' *defined. Then,* $\Theta \vdash P\{^{s[\mathtt{p}]}/_x\} \blacktriangleright \Gamma, \Gamma'$. ⌟

Proof. A standard induction on the typing judgement, where the case is made on the last typing rule applied. Special care is taken for rule [T-|], where only one of the two processes can use the substituted variable, and thus only this one applies the substitution non-trivially. □

Lemma 8.13 (Typing Inversion)**.** *Assume* $\Theta \vdash P \blacktriangleright \Gamma$*. Then:*

1. $P = \mathbf{0}$ *implies* end(Γ);
2. $P = \mathbf{def}\ X(x_1{:}S_1, \ldots, x_n{:}S_n) = Q\ \mathbf{in}\ P'$ *implies:*
   (a) $\Theta, X{:}S_1, \ldots, S_n \vdash Q \blacktriangleright x_1{:}S_1, \ldots, x_n{:}S_n$ *and*
   (b) $\Theta, X{:}S_1, \ldots, S_n \vdash P' \blacktriangleright \Gamma$;
3. $P = X\langle c_1, \ldots, c_n \rangle$ *implies:*
   (a) $\Theta \vdash X{:}S_1, \ldots, S_n$;
   (b) $\Gamma = \Gamma_0, \Gamma_1, \ldots, \Gamma_n$;
   (c) end(Γ_0);
   (d) $\forall i \in 1..n : \Gamma_i \vdash c_i{:}S_i$;
4. $P = (\nu s{:}\Gamma')\ P'$ *implies:*

   (a) $s \notin \Gamma$;

   (b) $\Gamma' = \{s[\mathtt{p}]{:}S_\mathtt{p}\}_{\mathtt{p}\in I}$;

   (c) $\varphi(\Gamma')$ *with* φ *a safety property and*

   (d) $\Theta \vdash P' \blacktriangleright \Gamma, \Gamma'$;

5. $P = P_1 \mid P_2$ *implies:*

   (a) $\Gamma = \Gamma_1, \Gamma_2$ *such that*

   (b) $\Theta \vdash P_1 \blacktriangleright \Gamma_1$ *and*

   (c) $\Theta \vdash P_2 \blacktriangleright \Gamma_2$;

6. $P = c[\mathtt{q}]\oplus_{i\in I}\mathtt{m}_i\langle d_i\rangle.P_i$ *implies:*

   (a) $\Gamma = \Gamma_0, \Gamma_1, \Gamma_2$ *such that*

   (b) $\Gamma_1 \vdash c{:}\mathtt{q}\oplus_{i\in I}\mathtt{m}_i\langle S_i\rangle.S_i'$ *and*

   (c) $\Gamma_2 = \bigcup_{i\in I}\Gamma_i$ *and* $\forall i \in I : \Gamma_i \vdash d_i{:}S_i$ *and*

   (d) $\forall i \in I : \Theta \vdash P_i \blacktriangleright \Gamma_0, c{:}S_i'$;

7. $P = c[\mathtt{q}]\sum_{i\in I}\mathtt{m}_i(y_i{:}S_i).P_i$ *implies:*

   (a) $\Gamma = \Gamma_0, \Gamma_1$ *such that*

   (b) $\Gamma_1 \vdash c{:}\mathtt{q}\&_{i\in I}\mathtt{m}_i\langle S_i\rangle.S_i'$ *and*

   (c) $\forall i \in I : \Theta \vdash P_i \blacktriangleright \Gamma_0, y_i{:}S_i, c{:}S_i'$;

⌟

Proof. This follows straightforwardly from the rules in Figure 8.1, as they are syntax-driven. For each P in cases 1–7, the typing judgement can be obtained by exactly one rule, specifically:

- case 1 by rule [T-**0**];
- case 2 by rule [T-**def**];
- case 3 by rule [T-CALL];
- case 4 by rule [T-ν];
- case 5 by rule [T-$|$];
- case 6 by rule [T-$\oplus$];
- case 7 by rule [T-&]. □

Lemma 8.14 (Subject Congruence). *Assume* $\Theta \vdash P \blacktriangleright \Gamma$ *and* $P \equiv P'$. *Then,* $\Theta \vdash P' \blacktriangleright \Gamma$. ⌟

Proof. By observing the cases where $P \equiv P'$ holds and using the Inversion Lemma (Lemma 8.13), it follows from the typing rules:

- commutativity and associativity of parallel composition are straightforward, splitting Γ between the parallel components using Lemma 8.13, then rearranging them with the order of the terms or of the compositions switched around, which ends up to Γ since composing typing environments is not order dependent;
- for $P \equiv P \mid \mathbf{0}$, we can type $\mathbf{0}$ with $\emptyset$, since trivially $\mathrm{end}(\emptyset)$, and use [T-|];
- when $P = (\nu s{:}\Gamma')\,\mathbf{0}$, using Lemma 8.13 twice gives us that $\mathrm{end}(\Gamma, \Gamma')$, thus $\mathrm{end}(\Gamma)$, allowing us to apply [T-**0**];
- simplifying a definition whose target process is $\mathbf{0}$ works the same as the previous item;
- when $P = (\nu s{:}\Gamma')\,(\nu s'{:}\Gamma'')\,P'$, by applying Lemma 8.13 twice, then [T-ν] twice in the reverse order, we get the result, using the fact that by construction $s \notin \Gamma''$, and $s' \notin \Gamma$ as $\Gamma \subset \Gamma, \Gamma'$;
- when $P = (\nu s{:}\Gamma')\,(Q \mid R)$ with $s \notin \mathrm{fc}(Q)$, using Lemma 8.13 twice with the fact that $s \notin \mathrm{fc}(Q)$ shows that $\Theta \vdash Q \blacktriangleright \Gamma_1$ and $\Theta \vdash R \blacktriangleright \Gamma_2, \Gamma'$, allowing to then apply [T-ν] to R, then [T-|] to Q and $(\nu s{:}\Gamma')\,R$;
- commuting a restriction and a definition works the same way as the previous one, but with [T-**def**] instead of [T-|];
- commuting two definitions works similarly as well, using Lemma 8.13 twice and using the fact that the defined process variables to not intersect and are not in Θ either by construction to rebuild the typing with switched definition order. □

Lemma 8.15 (Subsumption). *If* $\Theta \vdash P \blacktriangleright \Gamma$ *and* $\Gamma \leqslant \Gamma'$, *then* $\Theta \vdash P \blacktriangleright \Gamma'$. ⌟

Proof. By combining rule [T-SUB] with the other rules in Figure 8.1. □

We can now prove Proposition 8.11:

Proof. For a reduction, we have either $P \xrightarrow{\tau} P'$ or $P \xrightarrow{s:\mathtt{p},\mathtt{q}:\mathtt{m}} P'$. We will work by induction on the structure of P, since those reductions can be the result of their own rules or inside a context.

First, note that P' cannot contain an error, because this would contradict the safety of Γ, since an error can only occur as the result of rules [II-ERR], [OO-ERR] or [MISM-ERR], and those directly contradict the first three items in Definition 8.7.

Base cases:
- Rules [COM] and [R-COM] both use the first and last items in Definition 8.7, as well as the typing reduction rule [Γ-COMM];
- Rule [TAU] uses one of the previous cases and the restriction typing rule, such that $\Gamma' = \Gamma$;
- Rule [DEF-1] is a substitution, and thus uses the [T-**def**] typing rule combined with Lemma 8.12.

Inductive cases:
- For [RES], we have that $P0(\nu s)\, P_0 \to (\nu s)\, P_0' = P'$ and by induction on P_0, we have $\Gamma_0' = \Gamma', \Gamma_s$ such that $\vdash P_0' \blacktriangleright \Gamma_0'$ and Γ_0' is safe. Then by restriction over s using [T-ν], we have $\vdash P' \blacktriangleright \Gamma'$, and Γ' is safe, as it is contained in Γ_0' which is safe;
- [DEF-2] works similarly but with the [T-**def**] typing rule instead, and thus the type of the subprocess we use the induction hypothesis on is the final type;
- [PAR] works by induction on the reduced subprocess — the left one in the rule description. The composition of safe typing contexts is safe, thus since the left side has a safe type by induction hypothesis and the right side has a safe type as it is part of the original typing context for P, using the [T-|] typing rule gives a safe typing context;
- Finally, rule [≡] is a case of Lemma 8.14 on both sides. □

With that, we have that type safety is sound.

Theorem 8.16 (Type Safety Soundness)**.** Suppose $\vdash P \blacktriangleright \Gamma$ and safe(Γ). Then for all P' such that $P \to^* P'$, P' has no error. ⌟

Proof. Suppose $\vdash P \blacktriangleright \Gamma$ and safe(Γ). Let us prove by induction on n that for all s, P' such that $(\nu s)\, P = P_0 \to \ldots \to P_n = P'$, P' has no error subterm.

Base case: if $n = 0$ then $P' = (\nu s)\, P$ which has no error, since it is typeable. Furthermore, its type is safe.

Induction: Suppose for $n \geq 0$ that for all s, P_n such that $(\nu s)\, P = P_0 \to \ldots \to P_n$, P_n has no error and has a safe type. Let s', P' such that $(\nu s')\, P = P_0' \to \ldots P_n' \to P_{n+1}' = P'$. By induction hypothesis, P_n' has no error subterm since it has a safe type. Then by Proposition 8.11, P' also has a (safe) type, thus has no error subterm.

This proves, by induction, that for all $n \geq 0$, for all s, P' such that $(\nu s)\, P = P_0 \to \ldots P_n = P'$, P' has no error. Thus, for all s, P' such that $(\nu s)\, P \to^* P'$, P' has no error. □

Remark 8.17. Type safety is sound, but is not complete in the general case. Below is a counterexample that behaves safely in every regard, but cannot be typed with a safe type w.r.t the typing system. ⌟

Example 8.18 (Type unsafe, yet reduction safe process)**.** Process $P \mid Q$, with

$$P = s[\mathtt{p}][\mathtt{q}] \oplus \mathtt{resp}\langle s'[\mathtt{q}]\rangle.s[\mathtt{p}][\mathtt{q}] \oplus \mathtt{m}_1\langle s'[\mathtt{r}]\rangle.s'[\mathtt{p}][\mathtt{q}] \sum \mathtt{m}_1(x).s'[\mathtt{p}][\mathtt{q}] \sum \mathtt{m}_2(y).s[\mathtt{p}][\mathtt{q}] \oplus \mathtt{m}_1\langle y\rangle.\mathbf{0},$$

$$Q = s[\mathtt{q}][\mathtt{p}] \sum \mathtt{resp}(x).s[\mathtt{q}][\mathtt{p}] \sum \left\{ \begin{array}{l} \mathtt{m}_1(y).x[\mathtt{p}] \oplus \mathtt{m}_1\langle y\rangle.(\nu s'')\, x[\mathtt{p}] \oplus \mathtt{m}_2\langle s''[\mathtt{r}]\rangle.s[\mathtt{q}][\mathtt{p}] \sum \mathtt{m}_1(z).\mathbf{0} \\ \mathtt{m}_2(y).x[\mathtt{p}] \oplus \mathtt{m}_2\langle y\rangle.(\nu s'')\, x[\mathtt{p}] \oplus \mathtt{m}_1\langle s''[\mathtt{r}]\rangle.s[\mathtt{q}][\mathtt{p}] \sum \mathtt{m}_2(z).\mathbf{0} \end{array} \right\}$$

is typed by the following environment:

$$s[\mathtt{p}]{:}\mathtt{q} \oplus \mathtt{resp}\langle S'_{\mathtt{q}}\rangle.\mathtt{q} \oplus \mathtt{m}_1\langle \mathbf{end}\rangle.\mathtt{q} \oplus \mathtt{m}_1\langle \mathbf{end}\rangle.\mathbf{end}, s[\mathtt{q}]{:}\mathtt{p} \& \mathtt{resp}\langle S'_{\mathtt{q}}\rangle.\mathtt{p} \& \left\{ \begin{array}{l} \mathtt{m}_1\langle \mathbf{end}\rangle.\mathtt{p} \& \mathtt{m}_1\langle \mathbf{end}\rangle.\mathbf{end} \\ \mathtt{m}_2\langle \mathbf{end}\rangle.\mathtt{p} \& \mathtt{m}_2\langle \mathbf{end}\rangle.\mathbf{end} \end{array} \right\},$$

$$s'[\mathtt{p}]{:}\mathtt{q} \& \mathtt{m}_1\langle \mathbf{end}\rangle.\mathtt{q} \& \mathtt{m}_2\langle \mathbf{end}\rangle.\mathbf{end}, s'[\mathtt{q}]{:}S'_{\mathtt{q}}, s'[\mathtt{r}]{:}\mathbf{end} \text{ with } S'_{\mathtt{q}} = \mathtt{p} \oplus \left\{ \begin{array}{l} \mathtt{m}_1\langle \mathbf{end}\rangle.\mathtt{p} \oplus \mathtt{m}_2\langle \mathbf{end}\rangle.\mathbf{end} \\ \mathtt{m}_2\langle \mathbf{end}\rangle.\mathtt{p} \oplus \mathtt{m}_1\langle \mathbf{end}\rangle.\mathbf{end} \end{array} \right\}$$

This typing environment is the most restrictive, yet it is not safe: $s'[\mathtt{p}]$ can only receive on label $\mathtt{m}_1$, but $s'[\mathtt{q}]$ can send on either $\mathtt{m}_1$ or $\mathtt{m}_2$. ⌟

Since type-safety is *not* complete in all generality, we achieve completeness for a subset of the typed processes, defined below:

Definition 8.19 (Unique-Role Process)**.** Assume $\emptyset \vdash P \blacktriangleright \Gamma$. We say that P:

1. **has guarded definitions** if and only if in each subterm of the form
   def $X(x_1{:}S_1, \ldots, x_n{:}S_n) = Q$ **in** P', for all $i \in 1..n$, $S_i \not\leqslant \mathbf{end}$ implies that a call $Y\langle \ldots, x_i, \ldots\rangle$ can only occur in Q as subterm of $x_i[\mathtt{q}] \sum_{j \in J} \mathtt{m}_j(y_j).P_j$ or $x_i[\mathtt{q}] \oplus_{j \in J} \mathtt{m}_j\langle c_j\rangle.P_j$ (i.e., after using x_i for input/output);
2. **only plays role** $\mathtt{p}$ **in** s**, by** Γ, if and only if *(i)* P has guarded definitions; *(ii)* $\mathrm{fv}(P) = \emptyset$; *(iii)* $\Gamma = \Gamma_0, s[\mathtt{p}]{:}S$ with $S \not\leqslant \mathbf{end}$ and $\mathrm{end}(\Gamma_0)$;
   (iv) in all subterms $c[\mathtt{q}] \sum_{i \in I} \mathtt{m}_i(x_i{:}S_i).P_i$ of P, for all $i \in I$, $x_i \notin \mathrm{sc}(P_i)$ (or equivalently, $S_i \leqslant \mathbf{end}$);
   (v) in all subterms $(\nu s'{:}\Gamma')\, P'$ of P, we have $\mathrm{end}(\Gamma')$.

We say "P **only plays role** $\mathtt{p}$ **in** s" if and only if $\exists \Gamma : \emptyset \vdash P \blacktriangleright \Gamma$, and item 2 holds. ⌟

This subset of processes is composed of processes that, at any point in their execution, use a unique role in a unique session for all future actions. The newly added Item *(iv)* in comparison to the original work [SY19a] will be explained in detail later.

Remark 8.20. Each subprocess of our running example is a unique-role process, as they all use one unique channel from beginning to end. ⌟

Definition 8.21 (Partial Subtyping)**.** The **partial subtyping relation** $\lesssim$ is *co*inductively defined by:

$$\frac{\forall i \in I \quad S_i' \lesssim T_i'}{\mathtt{p}\&_{i\in I\cup J}\mathtt{m}_i\langle S_i\rangle.S_i' \lesssim \mathtt{p}\&_{i\in I}\mathtt{m}_i\langle T_i\rangle.T_i'}\ [\text{PSUB-}\&] \qquad \frac{\forall i \in I \quad S_i' \lesssim T_i'}{\mathtt{p}\oplus_{i\in I}\mathtt{m}_i\langle S_i\rangle.S_i' \lesssim \mathtt{p}\oplus_{i\in I\cup J}\mathtt{m}_i\langle T_i\rangle.T_i'}\ [\text{PSUB-}\oplus]$$

$$\frac{}{\mathbf{end} \lesssim \mathbf{end}}\ [\text{SUB-}\mathbf{end}] \qquad \frac{S\{\mu\mathbf{t}.S/\mathbf{t}\} \lesssim T}{\mu\mathbf{t}.S \lesssim T}\ [\text{SUB-}\mu\text{L}] \qquad \frac{S \lesssim T\{\mu\mathbf{t}.T/\mathbf{t}\}}{S \lesssim \mu\mathbf{t}.T}\ [\text{SUB-}\mu\text{R}]$$

⌟

We now state type safety completeness for unique-role processes: if a process behaves well and is typeable, then it accepts a safe type.

Theorem 8.22 (Type Safety Completeness for Unique-Role Processes)**.** Suppose $\vdash P \blacktriangleright \Gamma$, with:

1. $P \equiv \Big|_{\mathtt{p}\in I} P_\mathtt{p}$, and each $P_\mathtt{p}$ only plays $\mathtt{p}$ in s;
2. $\Gamma = \Gamma_0, \Gamma_1$ with $\Gamma_1 = \{s[\mathtt{p}]{:}S_\mathtt{p}\}_{\mathtt{p}\in I}$;
3. $\forall P'$ such that $P \to^* P'$, P' has no error.

Then there exists $\Gamma' = \Gamma_0', \Gamma_1'$ such that $\vdash P \blacktriangleright \Gamma'$, safe$(\Gamma')$ and $\Gamma_1' \lesssim \Gamma_1$. ⌟

Proof. Suppose $\vdash P \blacktriangleright \Gamma$, such that:

1. $P \equiv \Big|_{\mathtt{p}\in I} P_\mathtt{p}$, and each $P_\mathtt{p}$ either is $\mathbf{0}$ (up-to $\equiv$), or only plays $\mathtt{p}$ in s;
2. $\Gamma = \Gamma_0, \Gamma_1$ with $\Gamma_1 = \{s[\mathtt{p}]{:}S_\mathtt{p}\}_{\mathtt{p}\in I}$;
3. $\forall P'$ such that $P \to^* P'$, P' has no error.

To prove : there exists $\Gamma' = \Gamma_0', \Gamma_1'$ such that $\vdash P \blacktriangleright \Gamma'$, safe$(\Gamma')$ and $\Gamma_1' \lesssim \Gamma_1$. To prove this, we will propose a way to build the typing context Γ'. First we abstract the payloads from the typing context by producing adapted abstract typing rules to replace the selection and parallel composition rules. The abstract typing environment obtained is then unified to replace as many abstract payload types as possible with the type the corresponding receivers expect.

We then ensure the set of labels for the typing context of senders and receivers match the set of labels of the corresponding part of the process, which are the minimal sets of labels for the typing rules to work with the order described by (partial) subtyping. Following the process structure with modified typing rules ensures the unicity of our construction, with the replacement of unidentified abstract variables with **end** types.

Step 1: abstracting payloads. By induction on P, Γ, we construct the typing tree of $\vdash P \blacktriangleright \Gamma$ and modify it by introducing unification variables α as continuation for the types of sent data, effectively constructing a new type environment Γ_α where α is a type variable for the unification:

$$\frac{\Gamma_1 \vdash c{:}\mathtt{q}\oplus\mathtt{m}\langle S\rangle.S' \quad \Gamma_2 \vdash d{:}S \quad \Theta \vdash P \blacktriangleright \Gamma, c{:}S'}{\Theta \vdash c[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.P \blacktriangleright \Gamma, \Gamma_1, \Gamma_2}\ [\text{T-}\oplus]$$

Goes to

$$\frac{\Gamma_1 \vdash c{:}\mathtt{q}\oplus\mathtt{m}\langle \alpha\rangle.S' \quad \Gamma_2 \vdash d{:}\alpha \quad \Theta \vdash P \blacktriangleright \Gamma, c{:}S' \quad \alpha \notin \Gamma}{\Theta \vdash c[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.P \blacktriangleright \Gamma, \Gamma_1, \Gamma_2}\ [\text{T-}\oplus\text{-}\alpha]$$

We also add a condition on parallel composition: the unification variables from both sides must be disjoint (free up-to-renaming of those variables on one side of the tree):

$$\frac{\Theta \vdash P_1 \blacktriangleright \Gamma_1 \qquad \Theta \vdash P_2 \blacktriangleright \Gamma_2 \qquad \alpha(\Gamma_1) \cap \alpha(\Gamma_2) = \emptyset}{\Theta \vdash P_1 \mid P_2 \blacktriangleright \Gamma_1, \Gamma_2}\ [\text{T-}|\text{-}\alpha]$$

We then use the type Γ_α, and unify the unification variables with the corresponding receiving types. Linearity of sessions ensure that there will be unicity for each unification variable of their receiving end, if a receiving end exists. The remaining unification variables after unification can safely be replaced by **end**, as it means the corresponding sender is non-existent. This process guarantees the partial subtyping of the active part of the types between original typing context and the safety-enabled one we compute anew.

Step 2: Minimal extraction for actions. Since there is no merging required in the current case, because of the definition of unique role processes following session fidelity, it can be done at the same time as the construction of Γ_α. All that is needed in this case is to replace the typing clause for the main action in send and receive statements with the type that matches the action exactly, that is the minimal type for which the process will typecheck. For example, with the above rule for sending, we get:

$$\frac{\Gamma_1 = c{:}\mathtt{q}\oplus\mathtt{m}\langle \alpha\rangle.S' \vdash c{:}\mathtt{q}\oplus\mathtt{m}\langle \alpha\rangle.S' \quad \Gamma_2 \vdash d{:}\alpha \quad \Theta \vdash P \blacktriangleright \Gamma, c{:}S' \quad \alpha \notin \Gamma}{\Theta \vdash c[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.P \blacktriangleright \Gamma, \Gamma_1, \Gamma_2}\ [\text{T-}\oplus\text{-}\alpha]$$

This minimal extraction process uses both subtyping and the typing rules to ensure that the types show the same capabilities as the processes, in particular receiving types will not have fewer capabilities than their associated processes and sending types will not have more, because of the uniqueness of endpoint in each parallel subprocess.

Unification is a standard procedure in algorithmics, and here the unicity of the unification variables is guaranteed by the typing rules. There may remain non-unified variables, in the case of sends without a matching receive, but those can be freely replaced with **end** if we wanted the type to be closed, or left as is.

Remark 8.23. Note this process could allow for implementing proper minimal type extraction from a safe process, since we need only use the typing rules modified for unification to obtain a type that corresponds to the process capabilities. This is however out of scope in this work, we shall now finish the proof. ⌟

Summary and conclusions. Safety comes for free, since the types exactly match the process in capabilities thanks to minimal extraction and unification: labels of send actions can always be received if there is a communication on that endpoint (minimal extraction), and types of sent data will match exactly the required type of the receiving end, giving us subtyping of delegated endpoints for free (unification). The clause for receiving data in Definition 8.19, Item 2.*(iv)* makes it so the types do not use merging, which was the only possible cause of unsafe behaviours in the types when the process behaves safely. □

Example 8.24. Take Examples 8.2 and 7.4, using the safe version $P \equiv P_{\mathtt{s}} \mid P_{\mathtt{c}}^{\mathsf{safe}} \mid P_{\mathtt{a}}$. Then $P_{\mathtt{s}}$ (resp. $P_{\mathtt{c}}^{\mathsf{safe}}$, $P_{\mathtt{a}}$) only plays role $\mathtt{s}$ (resp. $\mathtt{c}$, $\mathtt{a}$) in s, and P never has any error during execution. Thus, extracted types from this process are safe. In this case, they correspond to the types given in the example: $\Gamma = s[\mathtt{s}]{:}S_{\mathtt{s}}, s[\mathtt{c}]{:}S_{\mathtt{c}}^{\mathsf{safe}}, s[\mathtt{a}]{:}S_{\mathtt{a}}$. ⌟

In fact, due to the restrictive nature of the unique-role subset of processes, we can construct a type from a process that follows the premises of Theorem 8.22 by simply using the typing rules and ensuring the typing of all payloads to be **end**. It makes a basic form of type inference for the subset of processes on which we can ensure proper verification from automatically generated types.

We also prove *subject LTS reduction*, first for a simple step where if a process that reduces following an action has a safe type, and that type is minimal, then the type admits the same action for reduction.

We begin with the one-step version of subject LTS reduction. We start by saying that a safe type Γ is minimal for $\leqslant$ to type P if $\vdash P \blacktriangleright \Gamma$ and the types for selection and branching are minimal for $\leqslant$. That is the case when the sets of labels presented in the typing context for branching and selection match exactly those of the processes at the same position, and this applies inductively to the continuations and to the payloads. This characterisation of minimality comes from the definitions of subtyping and of the typing rules Definition 8.4 and Figure 8.1 and the safety enforcing the labels of the selections to be contained in the labels of their matching branchings.

Proposition 8.25 (Subject LTS Step)**.** Suppose $\vdash P \blacktriangleright \Gamma$ where safe(Γ), Γ is minimal for $\leqslant$ to type P, and $P \xrightarrow{\ell} P'$. Then if $\ell \neq \tau$, then $\Gamma \xrightarrow{[\![\ell]\!]_\Gamma}$; else $\vdash P' \blacktriangleright \Gamma$ where $[\![\ell]\!]_\Gamma$ denotes a mapping of LTS labels from processes to types defined as:

- $[\![s : \mathtt{p}\&\mathtt{q} : \mathtt{m}_k\langle d_k\rangle]\!]_\Gamma = s : \mathtt{p}\&\mathtt{q} : \mathtt{m}_k(S_k)$ if $\Gamma \vdash s[\mathtt{p}]{:}\mathtt{q}\&_{i\in I}\mathtt{m}_i\langle S_i\rangle.S_i'$ and $k \in I$,
- $[\![s : \mathtt{p}\triangleleft\mathtt{q} : \mathtt{m}_k\langle d_k\rangle]\!]_\Gamma = s : \mathtt{p}\oplus\mathtt{q} : \mathtt{m}_k(S_k)$ if $\Gamma \vdash s[\mathtt{p}]{:}\mathtt{q}\oplus_{i\in I}\mathtt{m}_i\langle S_i\rangle.S_i'$ and $k \in I$,

- $[\![s : \mathtt{p} \triangleleft \mathtt{q} : \mathtt{m}_k(d_k)]\!]_\Gamma = s : \mathtt{p} \oplus \mathtt{q} : \mathtt{m}_k(S_k)$ if $\Gamma \vdash s[\mathtt{p}]{:}\mathtt{q} \oplus_{i \in I} \mathtt{m}_i\langle S_i \rangle.S'_i$ and $k \in I$,
- $[\![s : \mathtt{p},\mathtt{q} : \mathtt{m}]\!]_\Gamma = s : \mathtt{p},\mathtt{q} : \mathtt{m}$.

⌟

Proof. We prove this by induction on the last rule of the typing tree. We treat the base cases first:

- $P = s[\mathtt{p}][\mathtt{q}]\sum_{i \in I} \mathtt{m}_i(x_i).P_i \xrightarrow{s:\mathtt{p}\&\mathtt{q}:\mathtt{m}_k\langle d_k \rangle} P_k\{d_k/x_k\} = P'$. We type P from [T-&]:

$$\frac{\Gamma_2 \vdash s[\mathtt{p}]{:}\mathtt{q} \&_{i \in I} \mathtt{m}_i\langle S_i \rangle.S'_i \quad \forall i \in I \quad \Theta \vdash P_i \blacktriangleright \Gamma_1, x_i{:}S_i, s[\mathtt{p}]{:}S'_i}{\Theta \vdash s[\mathtt{p}][\mathtt{q}]\sum_{i \in I} \mathtt{m}_i(x_i{:}S_i).P_i \blacktriangleright \Gamma_1, \Gamma_2} \text{ [T-\&]}$$

  This is where minimality is important, especially Γ_2 should be minimal so it contains every label in P, i.e. $\Gamma_2 = s[\mathtt{p}]{:}\mathtt{q} \&_{i \in I} \mathtt{m}_i\langle S_i \rangle.S'_i$.

  We can reduce Γ_2 with [Γ-&] on the same label as P:

$$s[\mathtt{p}]{:}\mathtt{q} \&_{i \in I} \mathtt{m}_i\langle S_i \rangle.S'_i \xrightarrow{s:\mathtt{p}\&\mathtt{q}:\mathtt{m}_k(S_k)} s[\mathtt{p}]{:}S'_k = \Gamma'_2$$

  Hence, the result, by [Γ-Cong].

- $P = s[\mathtt{p}][\mathtt{q}] \oplus_{i \in I} \mathtt{m}_i\langle d_i \rangle.P'_i \xrightarrow{s:\mathtt{p}\triangleleft\mathtt{q}:\mathtt{m}_k\langle d_k \rangle} P'_k$. We type P from [T-$\oplus$]:

$$\frac{\Gamma_2 \vdash s[\mathtt{p}]{:}\mathtt{q} \oplus_{i \in I} \mathtt{m}_i\langle S_i \rangle.S'_i \quad \forall i \in I \quad \Gamma_i \vdash d_i{:}S_i \quad \Theta \vdash P'_i \blacktriangleright \Gamma_1, s[\mathtt{p}]{:}S'_i}{\Theta \vdash s[\mathtt{p}][\mathtt{q}] \oplus_{i \in I} \mathtt{m}_i\langle d_i \rangle.P'_i \blacktriangleright \Gamma_1, \Gamma_2, \bigcup_{i \in I} \Gamma_i} \text{ [T-}\oplus\text{]}$$

  While minimality is not needed for this case, as we are sure to contain more labels in Γ_2 than in P by subtyping, for our use case it does not harm to suppose $\Gamma_2 = s[\mathtt{p}]{:}\mathtt{q} \oplus_{i \in I} \mathtt{m}_i\langle S_i \rangle.S'_i$.

  We can reduce Γ_2 with [Γ-$\oplus$]:

$$s[\mathtt{p}]{:}\mathtt{q} \oplus_{i \in I} \mathtt{m}_i\langle S_i \rangle.S'_i \xrightarrow{s:\mathtt{p}\oplus\mathtt{q}:\mathtt{m}_k(S_k)} s[\mathtt{p}]{:}S'_k = \Gamma'_2$$

  Hence, using [Γ-Cong], the result.

- $P = (\nu s')\, P_1 \xrightarrow{s:\mathtt{p}\triangleleft\mathtt{q}:\mathtt{m}(s'[\mathtt{r}])} P'$ with $P_1 \xrightarrow{s:\mathtt{p}\triangleleft\mathtt{q}:\mathtt{m}\langle s'[\mathtt{r}] \rangle} P'$ and $s' \neq s$. We type P from [T-ν]:

$$\frac{\Gamma_1 = \{s'[\mathtt{p}]{:}S_\mathtt{p}\}_{\mathtt{p} \in I} \quad \varphi(\Gamma_1) \quad s' \notin \Gamma \quad \Theta \vdash P_1 \blacktriangleright \Gamma, \Gamma_1}{\Theta \vdash (\nu s'{:}\Gamma_1)\, P_1 \blacktriangleright \Gamma} \text{ [T-}\nu\text{]}$$

  By induction on $\Theta \vdash P_1 \blacktriangleright \Gamma, \Gamma_1$, and since $s \notin \mathrm{dom}(\Gamma_1)$, we get $\Gamma \xrightarrow{s:\mathtt{p}\oplus\mathtt{q}:\mathtt{m}(S_\mathtt{r})} \Gamma'$, allowing to conclude.

- $P = P_1 \mid P_2 \xrightarrow{s:\mathtt{p},\mathtt{q}:\mathtt{m}_k} P'_1 \mid P'_2 = P'$ with $P_1 \xrightarrow{s:\mathtt{p}\triangleleft\mathtt{q}:\mathtt{m}_k\langle d_k \rangle} P'_1$, $P_2 \xrightarrow{s:\mathtt{q}\&\mathtt{p}:\mathtt{m}_k\langle d_k \rangle} P'_2$. We type P from [T-|]:

$$\frac{\Theta \vdash P_1 \blacktriangleright \Gamma_1 \qquad \Theta \vdash P_2 \blacktriangleright \Gamma_2}{\Theta \vdash P_1 \mid P_2 \blacktriangleright \Gamma_1, \Gamma_2} \text{ [T-|]}$$

  By induction on $\Theta \vdash P_1 \blacktriangleright \Gamma_1$ and $\Theta \vdash P_2 \blacktriangleright \Gamma_2$, we get

- $\Gamma_1 = \Gamma_{11}, s[\mathtt{p}]{:}\mathtt{q}\oplus_{i\in I}\mathtt{m}_i\langle S_i\rangle.S'_i \xrightarrow{s:\mathtt{p}\oplus\mathtt{q}:\mathtt{m}_k(S_k)} \Gamma_{11}, s[\mathtt{p}]{:}S'_k$;
- $\Gamma_2 = \Gamma_{21}, s[\mathtt{q}]{:}\mathtt{p}\&_{i\in I}\mathtt{m}_i\langle T_i\rangle.T'_i \xrightarrow{s:\mathtt{q}\&\mathtt{p}:\mathtt{m}_k(T_k)} \Gamma_{21}, s[\mathtt{q}]{:}T'_k$.

From $\Gamma = \Gamma_1, \Gamma_2$ safe, we get $I \subseteq J$ and $\forall i \in I, T_i \leqslant S_i$, in particular $T_k \leqslant S_k$. With [Γ-Comm] we get $\Gamma_1, \Gamma_2 \xrightarrow{s:\mathtt{p},\mathtt{q}:\mathtt{m}_k}$, allowing to conclude.

- $P = P_1 \mid P_2 \xrightarrow{s:\mathtt{p},\mathtt{q}:\mathtt{m}_k} (\nu s')\,(P'_1 \mid P'_2) = P'$ with $P_1 \xrightarrow{s:\mathtt{p}\triangleleft\mathtt{q}:\mathtt{m}_k(s'[\mathtt{r}])} P'_1$ and $P_2 \xrightarrow{s:\mathtt{q}\&\mathtt{p}:\mathtt{m}_k\langle s'[\mathtt{r}]\rangle} P'_2$. We type P from [T-|]:

$$\frac{\Theta \vdash P_1 \blacktriangleright \Gamma_1 \qquad \Theta \vdash P_2 \blacktriangleright \Gamma_2}{\Theta \vdash P_1 \mid P_2 \blacktriangleright \Gamma_1, \Gamma_2}\ [\text{T-}|]$$

By induction on $\Theta \vdash P_1 \blacktriangleright \Gamma_1$ and $\Theta \vdash P_2 \blacktriangleright \Gamma_2$, we get

- $\Gamma_1 = \Gamma_{11}, s[\mathtt{p}]{:}\mathtt{q}\oplus_{i\in I}\mathtt{m}_i\langle S_i\rangle.S'_i \xrightarrow{s:\mathtt{p}\oplus\mathtt{q}:\mathtt{m}_k(S_k)} \Gamma_{11}, s[\mathtt{p}]{:}S'_k$;
- $\Gamma_2 = \Gamma_{21}, s[\mathtt{q}]{:}\mathtt{p}\&_{i\in I}\mathtt{m}_i\langle T_i\rangle.T'_i \xrightarrow{s:\mathtt{q}\&\mathtt{p}:\mathtt{m}_k(T_k)} \Gamma_{21}, s[\mathtt{q}]{:}T'_k$.

From $\Gamma = \Gamma_1, \Gamma_2$ safe, we get $I \subseteq J$ and $\forall i \in I, T_i \leqslant S_i$, in particular $T_k \leqslant S_k$. With [Γ-Comm] we get $\Gamma_1, \Gamma_2 \xrightarrow{s:\mathtt{p},\mathtt{q}:\mathtt{m}_k}$, allowing to conclude.

- $P = (\nu s)\,P_1 \xrightarrow{\tau} (\nu s)\,P'_1 = P'$ with $P_1 \xrightarrow{s:\mathtt{p},\mathtt{q}:\mathtt{m}} P'_1$. We type P from [T-ν]:

$$\frac{\Gamma_1 = \{s[\mathtt{p}]{:}S_\mathtt{p}\}_{\mathtt{p}\in I} \quad \varphi(\Gamma_1) \quad s \notin \Gamma \quad \Theta \vdash P_1 \blacktriangleright \Gamma, \Gamma_1}{\Theta \vdash (\nu s{:}\Gamma_1)\,P_1 \blacktriangleright \Gamma}\ [\text{T-}\nu]$$

By induction on $\Theta \vdash P_1 \blacktriangleright \Gamma, \Gamma_1$ and with $s \notin \Gamma$, we get $\Gamma_1 \xrightarrow{s:\mathtt{p},\mathtt{q}:\mathtt{m}} \Gamma'_1$ and $\Theta \vdash P'_1 \blacktriangleright \Gamma, \Gamma'_1$. We then conclude with [T-ν] again:

$$\frac{\Gamma'_1 = \{s[\mathtt{p}]{:}S_\mathtt{p}\}_{\mathtt{p}\in I} \quad \varphi(\Gamma'_1) \quad s \notin \Gamma \quad \Theta \vdash P'_1 \blacktriangleright \Gamma, \Gamma'_1}{\Theta \vdash (\nu s{:}\Gamma'_1)\,P'_1 \blacktriangleright \Gamma}\ [\text{T-}\nu]$$

- $P = \mathbf{def}\ X(x_1, \ldots, x_n) = P_1\ \mathbf{in}\ (X\langle s_1[\mathtt{p}_1], \ldots, s_n[\mathtt{p}_n]\rangle \mid Q) \xrightarrow{\tau} P'$ with $P' = \mathbf{def}\ X(x_1, \ldots, x_n) = P_1\ \mathbf{in}\ (P_1\{s_1[\mathtt{p}_1]/x_1\} \cdots \{s_n[\mathtt{p}_n]/x_n\} \mid Q)$. The typing goes like this with $\Gamma = \Gamma_1, \Gamma_2$:

$$\frac{\begin{array}{c}\Theta, X{:}S_1, \ldots, S_n \vdash P_1 \blacktriangleright x_1{:}S_1, \ldots, x_n{:}S_n \\ \Theta, X{:}S_1, \ldots, S_n \vdash X\langle s_1[\mathtt{p}_1], \ldots, s_n[\mathtt{p}_n]\rangle \mid Q \blacktriangleright \Gamma_1, \Gamma_2\end{array}}{\Theta \vdash \mathbf{def}\ X(x_1{:}S_1, \ldots, x_n{:}S_n) = P_1\ \mathbf{in}\ (X\langle s_1[\mathtt{p}_1], \ldots, s_n[\mathtt{p}_n]\rangle \mid Q) \blacktriangleright \Gamma_1, \Gamma_2}\ [\textbf{T-def}]$$

Then:

$$\frac{\Theta, X{:}S_1, \ldots, S_n \vdash X\langle s_1[\mathtt{p}_1], \ldots, s_n[\mathtt{p}_n]\rangle \blacktriangleright \Gamma_1 \qquad \Theta, X{:}S_1, \ldots, S_n \vdash Q \blacktriangleright \Gamma_2}{\Theta, X{:}S_1, \ldots, S_n \vdash X\langle s_1[\mathtt{p}_1], \ldots, s_n[\mathtt{p}_n]\rangle \mid Q \blacktriangleright \Gamma_1, \Gamma_2}\ [\text{T-}|]$$

And to finish the interesting part of the tree:

$$\frac{\Theta, X{:}S_1, \ldots, S_n \vdash X{:}S_1, \ldots, S_n \qquad \Gamma_1 = \Gamma_{10}, \Gamma_{11}, \ldots, \Gamma_{1n} \qquad \mathrm{end}(\Gamma_{10}) \qquad \forall i \in 1..n \quad \Gamma_{1i} \vdash s_i[\mathtt{p}_i]{:}S_i}{\Theta, X{:}S_1, \ldots, S_n \vdash X\langle s_1[\mathtt{p}_1], \ldots, s_n[\mathtt{p}_n]\rangle \blacktriangleright \Gamma_1} \text{ [T-CALL]}$$

Then, we extract the left hypothesis from [T-**def**] and use Lemma 8.12 to get

$$\Theta, X{:}S_1, \ldots, S_n \vdash P_1\{s_1[\mathtt{p}_1]/x_1\} \cdots \{s_n[\mathtt{p}_n]/x_n\} \blacktriangleright \Gamma_{11}, \ldots, \Gamma_{1n}$$

From there, using [T-|] along with Lemma 8.14, we get:

$$\Theta, X{:}S_1, \ldots, S_n \vdash P_1\{s_1[\mathtt{p}_1]/x_1\} \cdots \{s_n[\mathtt{p}_n]/x_n\} \blacktriangleright \Gamma_1$$

Using that and its subtree to replace the subtree expanding from [T-CALL] on the left side of rule [T-|], we get:

$$\Theta \vdash \mathbf{def}\ X(x_1{:}S_1, \ldots, x_n{:}S_n) = P_1\ \mathbf{in}\ (P_1\{s_1[\mathtt{p}_1]/x_1\} \cdots \{s_n[\mathtt{p}_n]/x_n\} \mid Q) \blacktriangleright \Gamma$$

allowing us to conclude.

- $P = (\nu s)\, P_1 \xrightarrow{\ell} (\nu s)\, P_1' = P'$ with $P_1 \xrightarrow{\ell} P_1'$ and $s \notin \mathrm{fn}(\ell)$. We type P from [T-ν]:

$$\frac{\Gamma_1 = \{s[\mathtt{p}]{:}S_\mathtt{p}\}_{\mathtt{p}\in I} \quad \varphi(\Gamma_1) \quad s \notin \Gamma \quad \Theta \vdash P_1 \blacktriangleright \Gamma, \Gamma_1}{\Theta \vdash (\nu s{:}\Gamma_1)\, P_1 \blacktriangleright \Gamma} \text{ [T-}\nu\text{]}$$

  By induction on $\Theta \vdash P_1 \blacktriangleright \Gamma, \Gamma_1$ and with $\mathrm{fn}(\ell) \cap \mathrm{dom}(\Gamma_1) = \emptyset$, we get (1) if $\ell \neq \tau$, $\Gamma \xrightarrow{[\![\ell]\!]_\Gamma}$ and conclude, (2) else $\Theta \vdash P_1' \blacktriangleright \Gamma, \Gamma_1$ and conclude with [T-ν]:

$$\frac{\Gamma_1 = \{s[\mathtt{p}]{:}S_\mathtt{p}\}_{\mathtt{p}\in I} \quad \varphi(\Gamma_1) \quad s \notin \Gamma \quad \Theta \vdash P_1' \blacktriangleright \Gamma, \Gamma_1}{\Theta \vdash (\nu s{:}\Gamma_1)\, P_1' \blacktriangleright \Gamma} \text{ [T-}\nu\text{]}$$

- $P = \mathbf{def}\ X(x_1, \ldots, x_n) = Q\ \mathbf{in}\ P_1 \xrightarrow{\ell} \mathbf{def}\ X(x_1, \ldots, x_n) = Q\ \mathbf{in}\ P_1' = P'$, with $P_1 \xrightarrow{\ell} P_1'$ and $\mathrm{fn}(\ell) \cap \mathrm{bn}(X(x_1, \ldots, x_n) = Q) = \emptyset$. The typing follows:

$$\frac{\Theta, X{:}S_1, \ldots, S_n \vdash Q \blacktriangleright x_1{:}S_1, \ldots, x_n{:}S_n \qquad \Theta, X{:}S_1, \ldots, S_n \vdash P_1 \blacktriangleright \Gamma}{\Theta \vdash \mathbf{def}\ X(x_1{:}S_1, \ldots, x_n{:}S_n) = Q\ \mathbf{in}\ P_1 \blacktriangleright \Gamma} \text{ [T-}\mathbf{def}\text{]}$$

  Then, by induction on the right hypothesis from [T-**def**], we conclude in the same way as before, injecting P_1' in the typing rule for the $\ell = \tau$ case.

- $P = P_1 \mid P_2 \xrightarrow{\ell} P_1' \mid P_2 = P'$, with $P_1 \xrightarrow{\ell} P_1'$ and $\mathrm{bn}(\ell) \cap \mathrm{fn}(P_2) = \emptyset$. The typing goes with $\Gamma = \Gamma_1, \Gamma_2$:

$$\frac{\Theta \vdash P_1 \blacktriangleright \Gamma_1 \qquad \Theta \vdash P_2 \blacktriangleright \Gamma_2}{\Theta \vdash P_1 \mid P_2 \blacktriangleright \Gamma_1, \Gamma_2} \text{ [T-|]}$$

  Then by induction hypothesis on $\Theta \vdash P_1 \blacktriangleright \Gamma_1$ we work the same way as the last two cases again, using [Γ-CONG] if $\ell \neq \tau$.

- The other parallel case is symmetric with induction on $\Theta \vdash P_2 \blacktriangleright \Gamma_2$. □

To generalise that, we wish to have a conservation of type minimality through reduction. While this is not possible in the general case due to the necessity of extending types when a subprocess has several roles to play, it is conveniently the case in our narrow subset of unique-role processes.

Example 8.26. To show that in general type minimality is not carried over by reduction, we look at the following example:

$$P = s[\mathtt{p}][\mathtt{q}] \oplus \left\{ \begin{array}{l} \mathtt{m}_1\langle d_1\rangle.s[\mathtt{r}][\mathtt{q}] \oplus \mathtt{m}_1\langle d_3\rangle.\mathbf{0} \\ \mathtt{m}_2\langle d_2\rangle.s[\mathtt{r}][\mathtt{q}] \oplus \mathtt{m}_2\langle d_4\rangle.\mathbf{0} \end{array} \right\}$$

$$Q = s[\mathtt{q}][\mathtt{p}] \sum \left\{ \begin{array}{l} \mathtt{m}_1(x).s[\mathtt{q}][\mathtt{r}] \sum \{\mathtt{m}_1(y).\mathbf{0},\ \mathtt{m}_2(y).\mathbf{0}\} \\ \mathtt{m}_2(x).s[\mathtt{q}][\mathtt{r}] \sum \{\mathtt{m}_1(y).\mathbf{0},\ \mathtt{m}_2(y).\mathbf{0}\} \end{array} \right\}$$

Its minimal type would be:

$$s[\mathtt{p}]{:}\mathtt{q}\oplus \left\{ \begin{array}{l} \mathtt{m}_1\langle \mathbf{end}\rangle.\mathbf{end} \\ \mathtt{m}_2\langle \mathbf{end}\rangle.\mathbf{end} \end{array} \right\},\ s[\mathtt{r}]{:}\mathtt{q}\oplus \left\{ \begin{array}{l} \mathtt{m}_1\langle \mathbf{end}\rangle.\mathbf{end} \\ \mathtt{m}_2\langle \mathbf{end}\rangle.\mathbf{end} \end{array} \right\},\ s[\mathtt{q}]{:}\mathtt{p}\& \left\{ \begin{array}{l} \mathtt{m}_1\langle \mathbf{end}\rangle.\mathtt{r}\&\{\mathtt{m}_1\langle \mathbf{end}\rangle.\mathbf{end},\ \mathtt{m}_2\langle \mathbf{end}\rangle.\mathbf{end}\} \\ \mathtt{m}_2\langle \mathbf{end}\rangle.\mathtt{r}\&\{\mathtt{m}_1\langle \mathbf{end}\rangle.\mathbf{end},\ \mathtt{m}_2\langle \mathbf{end}\rangle.\mathbf{end}\} \end{array} \right\}$$

This is because we need to merge the types of both branches for $s[\mathtt{r}]$ when building the selection on $\mathtt{p}$ to be able to apply [T-⊕]. However, we can wee that after one reduction step, the type of $s[\mathtt{r}]$ will not be minimal to type the continuation that was chosen in P in either case, as the typing contains two branches but both continuations only have one. ⌟

Proposition 8.27 (Type Minimality Conservation)**.** Assume $\emptyset \vdash P \blacktriangleright \Gamma$, where Γ is safe and minimal for $\leqslant$ to type P, $P \equiv \left|_{\mathtt{p}\in I} P_{\mathtt{p}}\right.$, each $P_{\mathtt{p}}$ either is $\mathbf{0}$ (up-to $\equiv$), or only plays $\mathtt{p}$ in s, and $P \to P'$. Then there exists Γ' safe such that $\Gamma \to^* \Gamma'$, $\vdash P' \blacktriangleright \Gamma'$ and Γ' is minimal for $\leqslant$ to type P'. ⌟

Proof. By Theorem 8.16, we deduce that P' has no error because Γ is safe. This means that the reduction is necessarily due to a communication, whether explicit or captured ba a restriction; or the result of a definition expansion. The definition expansion case holds because the minimality holds for P if, and only if, it holds for both sides of the definition, and Lemma 8.12 replaces exactly one variable with one channel with the same type.

Let us treat the case of a communication. By Definition 8.19, we further deduce that this communication has to be explicit, as any restricted session has to be inactive due to typing restrictions. It also gives us that both sides of the communication happen in two distinct subprocesses, suppose $P_{\mathtt{p}}$ and $P_{\mathtt{q}}$. This means that their respective typing environments $\Gamma_{\mathtt{p}}$ and $\Gamma_{\mathtt{q}}$ are minimal as well — otherwise their composition would not be minimal.

Minimality holds for rules [T-⊕] and [T-&] if the respective structure is typed with the exact same labels, no more or less, and each continuation has minimal typing, since each term can only play one role in s. Thus, after reduction the typing for each continuation is minimal, the existence of the reduced type for each subprocess being guaranteed by Proposition 8.25. Any context around each of them keeps minimality by construction of their respective typing rules where we substitute the original term for the chosen continuation. Then, parallel composition for minimally typed processes is minimally typed, as for when we took the composition apart earlier. □

Proposition 8.28 (Subject LTS Reduction)**.** Assume $\emptyset \vdash P \blacktriangleright \Gamma$, where Γ is safe and minimal for $\leqslant$ to type P, $P \equiv \big|_{\mathtt{p} \in I} P_{\mathtt{p}}$, each $P_{\mathtt{p}}$ either is $\mathbf{0}$ (up-to $\equiv$), or only plays $\mathtt{p}$ in s, and $P \to^* P' \xrightarrow{\ell}$. Then for some Γ' such that $\vdash P' \blacktriangleright \Gamma'$, if $\ell \neq \tau$, then $\Gamma \to^* \Gamma' \xrightarrow{[\![\ell]\!]_{\Gamma'}}$; else $\Gamma \to^* \Gamma'$. ⌟

Proof. By Proposition 8.27 applied to each reduction step between P and P', we get the existence of Γ' safe such that $\Gamma \to^* \Gamma'$, $\vdash P' \blacktriangleright \Gamma'$ and Γ' is minimal for $\leqslant$ to type P'. Then, by Proposition 8.25, since Γ' is safe and minimal for $\leqslant$ to type P', we get the last step on ℓ. □

Note that the reduction is either an explicit communication or a τ-action, so the typing context for the reduced process is safe by [S-→] or equality, respectively.

In this subset of unique-role processes, we now revisit the *session fidelity theorem*, which brings reduction properties from types to processes. Specifically, if a P is composed as above and further has a safe type Γ, then if Γ reduces: **(1)** P also reduces following the same path, and its reduction is typed by the reduction of Γ; and **(2)** the reduced process is also a composition of unique-role subprocesses.

Theorem 8.29 (Session Fidelity)**.** Assume $\emptyset \vdash P \blacktriangleright \Gamma$, where Γ is safe, $P \equiv \big|_{\mathtt{p} \in I} P_{\mathtt{p}}$, and each $P_{\mathtt{p}}$ either is $\mathbf{0}$ (up-to $\equiv$), or only plays $\mathtt{p}$ in s. Then, $\Gamma \to$ implies $\exists \Gamma', P'$ such that $\Gamma \to \Gamma'$, $P \to^* P'$ and $\emptyset \vdash P' \blacktriangleright \Gamma'$, where $P' \equiv \big|_{\mathtt{p} \in I} P'_{\mathtt{p}}$ and each $P'_{\mathtt{p}}$ either is $\mathbf{0}$ (up-to $\equiv$), or only plays $\mathtt{p}$ in s. ⌟

Note the new and highlighted Item *(iv)* in Definition 8.19, not present in the previous work [SY19a]. Indeed, the original definition did not take into account variables bound by reception; but their type does not appear in Γ, and that can break the session fidelity theorem [SY19a, Theorem 5.4], which is the same as ours. We show a counterexample that breaks session fidelity with the original definition from Scalas and Yoshida [SY19a], without Item *(iv)*:

Example 8.30. Let $P = P_{\mathtt{p}} \mid P_{\mathtt{q}}$, with:

$$\begin{aligned} P_{\mathtt{p}} &= s[\mathtt{p}][\mathtt{q}] \oplus \mathtt{m}\langle s[\mathtt{r}] \rangle . s[\mathtt{p}][\mathtt{q}] \sum \mathtt{m}(x) . x[\mathtt{q}] \oplus \mathtt{m}\langle s[\mathtt{p}] \rangle . \mathbf{0} \\ P_{\mathtt{q}} &= s[\mathtt{q}][\mathtt{p}] \sum \mathtt{m}(x) . s[\mathtt{q}][\mathtt{p}] \oplus \mathtt{m}\langle x \rangle . s[\mathtt{q}][\mathtt{r}] \sum \mathtt{m}(x) . s[\mathtt{q}][\mathtt{p}] \sum \mathtt{m}(y) . x[\mathtt{q}] \oplus \mathtt{m}\langle \mathbf{0} \rangle . \mathbf{0} \end{aligned}$$

We have the following typing judgements:

$$\vdash P_{\mathtt{p}} \blacktriangleright s[\mathtt{r}]{:}\mathbf{end}, s[\mathtt{p}]{:}\mathtt{q}\oplus\mathtt{m}\langle\mathbf{end}\rangle.\mathtt{q}\&\mathtt{m}\langle\mathtt{q}\oplus\mathtt{m}\langle\mathbf{end}\rangle.\mathbf{end}\rangle.\mathbf{end}$$

$$\vdash P_{\mathtt{q}} \blacktriangleright s[\mathtt{q}]{:}\mathtt{p}\&\mathtt{m}\langle\mathbf{end}\rangle.\mathtt{p}\oplus\mathtt{m}\langle\mathbf{end}\rangle.\mathtt{r}\&\mathtt{m}\langle\mathtt{q}\oplus\mathtt{m}\langle\mathbf{end}\rangle.\mathbf{end}\rangle.\mathtt{p}\&\mathtt{m}\langle\mathbf{end}\rangle.\mathbf{end}$$

Hence $P_{\mathtt{p}}$ only plays role $\mathtt{p}$ in s and $P_{\mathtt{q}}$ only plays role $\mathtt{q}$ in s.

Let $S_{\mathtt{r}} = \mathtt{q}\oplus\mathtt{m}\langle\mathtt{q}\oplus\mathtt{m}\langle\mathbf{end}\rangle.\mathbf{end}\rangle.\mathbf{end}$. Then the whole process P also accepts the following typing context, which is *valid*, *safe*, as well as *deadlock-free*:

$$s[\mathtt{r}]{:}S_{\mathtt{r}} \qquad s[\mathtt{p}]{:}\mathtt{q}\oplus\mathtt{m}\langle S_{\mathtt{r}}\rangle.\mathtt{q}\&\mathtt{m}\langle S_{\mathtt{r}}\rangle.\mathtt{q}\oplus\mathtt{m}\langle\mathbf{end}\rangle.\mathbf{end}$$

$$s[\mathtt{q}]{:}\mathtt{p}\&\mathtt{m}\langle S_{\mathtt{r}}\rangle.\mathtt{p}\oplus\mathtt{m}\langle S_{\mathtt{r}}\rangle.\mathtt{r}\&\mathtt{m}\langle\mathtt{q}\oplus\mathtt{m}\langle\mathbf{end}\rangle.\mathbf{end}\rangle.\mathtt{p}\&\mathtt{m}\langle\mathbf{end}\rangle.\mathbf{end}$$

However, after 3 reduction steps for the process, we get $P \to^3 P' = \mathbf{0} \mid s[\mathtt{q}][\mathtt{p}]\sum\mathtt{m}(y).s[\mathtt{p}][\mathtt{q}]\oplus\mathtt{m}\langle\mathbf{0}\rangle.\mathbf{0}$ with the right side playing both roles $\mathtt{q}$ and $\mathtt{p}$ in s, breaking the conclusion of the theorem.

Furthermore, this process is *deadlocked*, indicating that there was an issue: the goal of the session fidelity theorem is to show correspondence between reduction-related properties being true on the types and their counterparts being true on the processes. The deadlock-freedom property not passing this test gave us good reason to look that it also broke the theorem itself.

In our revised framework, $P_{\mathtt{q}}$ is no longer unique-role, fixing this issue. ⌟

We now show the proof for Theorem 8.29:

Proof. Suppose $\Gamma\to$, then:
We know $P \equiv \left|_{\mathtt{p}\in I} P_{\mathtt{p}}\right.$ and $\emptyset\vdash P \blacktriangleright \Gamma$. Thus, by Lemma 8.14: $\emptyset\vdash \left|_{\mathtt{p}\in I} P_{\mathtt{p}}\right. \blacktriangleright \Gamma$.
Let $\Gamma = \Gamma_0, s[\mathtt{p}]{:}\mathtt{q}\oplus_{i\in I}\mathtt{m}_i\langle S_i\rangle.S_i', s[\mathtt{q}]{:}\mathtt{p}\&_{j\in J}\mathtt{m}_j\langle S_j''\rangle.S_j'''$
then $I \subseteq J$ and $\forall i \in I, S_i'' \leqslant S_i$ by safe(Γ).

There are four cases: **(1)** neither $P_{\mathtt{p}}$ and $P_{\mathtt{q}}$ is a definition, **(2)** (resp. **(3)**) $P_{\mathtt{p}}$ (resp. $P_{\mathtt{q}}$) is a definition, and **(4)** both are definitions.

We treat the first case: let $P_{\mathtt{p}} = s[\mathtt{p}][\mathtt{q}]\oplus_{k\in K}\mathtt{m}_k\langle s'[\mathtt{r}_k]\rangle.P_{\mathtt{p}k}'$,
and $P_{\mathtt{q}} = s[\mathtt{q}][\mathtt{p}]\sum_{l\in L}\mathtt{m}_l(x_l).P_{\mathtt{q}l}'$.
We get:

- for $P_{\mathtt{p}}$:
  - by hypothesis, $P_{\mathtt{p}}$ only plays role $\mathtt{p}$ in s by
    $\Gamma_{\mathtt{p}} = \Gamma_{\mathtt{p}0}, s[\mathtt{p}]{:}\mathtt{q}\oplus_{k\in K}\mathtt{m}_k\langle T_{\mathtt{p}k}\rangle.T_{\mathtt{p}k}', \{s'[\mathtt{r}_k]{:}T_{\mathtt{p}k}\}_{k\in K}$
  - by typing rules, $K \subseteq I$;
  - by hypothesis, end($\Gamma_{\mathtt{p}0}$) and $\forall k \in K, T_{\mathtt{p}k} \leqslant \mathbf{end}$

- let $k \in K$, `if` $P'_{\mathtt{p}k} \not\equiv \mathbf{0}$, implying $T'_{\mathtt{p}k} \not\leqslant \mathbf{end}$:
  - Item *(iii)* is validated by the previous item and the hypothesis;
  - $\forall k \in K, \mathrm{fv}\left(P'_{\mathtt{p}k}\right) = \emptyset$ by construction (validating item *(ii)*);
  - Items *(i)*, *(iv)* and *(v)* are validated by subterms by construction;

  `then` $P'_{\mathtt{p}k}$ only plays role $\mathtt{p}$ in s by $\Gamma_{\mathtt{p}0}, s[\mathtt{p}]{:}T'_{\mathtt{p}k}$;

  `else` $P'_{\mathtt{p}k} \equiv \mathbf{0}$ `then` trivially true;

- for $P_{\mathtt{q}}$:
  - by hypothesis, $P_{\mathtt{q}}$ only plays role $\mathtt{q}$ in s by $\Gamma_{\mathtt{q}} = \Gamma_{\mathtt{q}0}, s[\mathtt{q}]{:}\mathtt{p}\&_{l\in L}\mathtt{m}_l\langle T_{\mathtt{q}l}\rangle.T'_{\mathtt{q}l}$;
  - by typing rules, $J \subseteq L$, and $\forall j \in J, T_{\mathtt{q}j} \leqslant S''_j$ and $T'_{\mathtt{q}j} \leqslant S'''_j$; so we only need to check the communication for $k \in K$ (as $K \subseteq I \subseteq J \subseteq L$);
  - by hypothesis, $\mathrm{end}(\Gamma_{\mathtt{q}0})$;
  - by item *(iv)* for $P_{\mathtt{q}}$, $\forall l \in L$, either $T_{\mathtt{q}l} \leqslant \mathbf{end}$ or $T'_{\mathtt{q}l} \leqslant \mathbf{end}$; since we check only on happening communications, let $k \in K$ and suppose x_k and $s[\mathtt{q}]$ are not both left out of $P'_{\mathtt{q}k}$:
    - `if` $s[\mathtt{q}] \notin \mathrm{sc}\left(P'_{\mathtt{q}k}\right)$, then $T'_{\mathtt{q}k} \leqslant \mathbf{end}$ and we get $P'_{\mathtt{q}k}\{^{s'[\mathtt{r}_k]}/_{x_k}\}$ only plays role $\mathtt{r}_k$ in s' by $\Gamma_{\mathtt{q}0}, s[\mathtt{q}]{:}\mathbf{end}, s'[\mathtt{r}_k]{:}T_{\mathtt{p}k}$:
      - $T_{\mathtt{q}k} \leqslant S_k$ by transitivity;
      - Item *(iii)* is validated by the previous item and the hypothesis;
      - $\mathrm{fv}\left(P'_{\mathtt{q}k}\{^{s'[\mathtt{q}_k]}/_{x_k}\}\right) = \emptyset$ by construction (validating item *(ii)*);
      - Items *(i)*, *(iv)* and *(v)* are validated by subterms by construction;
    - `else` $x_k \notin \mathrm{sc}\left(P'_{\mathtt{q}k}\right)$, then $T_{\mathtt{q}k} \leqslant \mathbf{end}$ and we get $P'_{\mathtt{q}k}\{^{s'[\mathtt{r}_k]}/_{x_k}\}$ only plays role $\mathtt{q}$ in s by $\Gamma_{\mathtt{q}0}, s[\mathtt{q}]{:}T'_{\mathtt{q}k}, s'[\mathtt{r}_k]{:}\mathbf{end}$:
      - $\mathbf{end} \leqslant S_k$ by transitivity, so subtyping rules give $S_k = \mathbf{end}$;
      - Item *(iii)* is validated by the previous item and the hypothesis;
      - $\mathrm{fv}\left(P'_{\mathtt{q}k}\{^{s'[\mathtt{q}_k]}/_{x_k}\}\right) = \emptyset$ by construction (validating item *(ii)*);
      - Items *(i)*, *(iv)* and *(v)* are validated by subterms by construction;

    `else` if both are left out, then $P'_{\mathtt{q}k} \equiv \mathbf{0}$ and the conclusion is trivially true.

The other parallel processes are not changed, and thus still validate the conclusion by hypothesis. Then Γ' can be selected as the reduction of Γ for one of the labels or $P_{\mathtt{p}}$ and P' is selected as the reduction of P for the same label; typing comes for free from the previous points and the typing rules, and the composition of P' is what we just proved.

The other cases are similar: the main difference is the potential τ-transitions to unroll the definitions via [DEF-1] if needed, but those do not change the typing environment (Proposition 8.28), and when the action is immediately available without unrolling the proof is the same as case **(1)**. □

9 Model Checking the Session Types

This time, we use a fragment of the μ-calculus with data to analyse our communication protocols. Section 9.1 presents this variant of the μ-calculus. Note it is functionally mostly equivalent to that of Section 4.1, where we replace the big conjunctions over sets with universal quantification, and the disjunctions over sets with existential quantification. Where the data is implicit in the actions, and thus ranged over by conjunction and disjunction in Section 4.1, it is now explicit and ranged over by first order quantification, but the achieved result is the same.

Then, Section 9.2 presents the proof-of-concept tool from this part of our work, explaining how the different components work together and what they do, and how it integrates with our goal for counterexample extraction to guide the verification process. The tool is available on GitHub [Gab22].

9.1 Model Checking and Counterexample Extraction

Let α range over labels in Definition 8.6: α can have the form $s{:}\mathtt{p}\&\mathtt{q}{:}\mathtt{m}(S)$ for input, $s{:}\mathtt{p}\oplus\mathtt{q}{:}\mathtt{m}(S)$ for output, or $s{:}\mathtt{p},\mathtt{q}{:}\mathtt{m}$ for communication. Then, *μ-calculus formulae* are defined as follows, where d ("data") ranges over sessions, roles, message labels, and session types:

$$\phi ::= \top \mid \bot \mid [\alpha]\,\phi \mid \langle\alpha\rangle\,\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \Rightarrow \phi_2 \mid \mu Z.\phi \mid \nu Z.\phi \mid Z \mid \forall d.\phi \mid \exists d.\phi$$

Formulae contain the same constructs as in the modal μ-calculus from Chapter 4 — true and false constants, box and diamond modalities, conjunction, disjunction, implication, fixed point operators and recursion variables. On top of those base constructs, the μ-calculus used in this section contains existential and universal quantification of the first order on data [SY19a]. We remind the meaning of some operators, as their interpretation differs very slightly from Chapter 4. The *box modality*, $[\alpha]\,\phi$, is valid when, for every state reachable by following an action in α from the current state, ϕ is true. This set of states can be empty: if no action in α is available, $[\alpha]\,\phi$ is vacuously true. The *diamond modality*, $\langle\alpha\rangle\,\phi$, is true when at least one action in α is available from the current state and ϕ is true in the resulting state. If no action in α is available, $\langle\alpha\rangle\,\phi$ is vacuously false.

The *least fixed point* $\mu Z.\phi$ and *greatest fixed point* $\nu Z.\phi$ are still the standard recursive constructs as defined in temporal systems works [Bra92]: on trivial recursion, the least fixed point defaults to false and the greatest fixed point defaults to true. In general, the least fixed point can be expanded on the left side of an implication: if $\phi[Z := \psi] \Rightarrow \psi$ then $\mu Z.\phi \Rightarrow \psi$. Conversely, the greatest fixpoint can be expanded

1. Γ is **safe**, written $\mathrm{safe}(\Gamma)$, iff:
$$\Gamma \models \nu Z. \left(\begin{array}{l} \forall s, \mathtt{p}, \mathtt{q}, \mathtt{m}, \mathtt{m}', S, S'.\ \left(\langle s{:}\mathtt{p} \oplus \mathtt{q}{:}\mathtt{m}(S)\rangle\, \top \wedge \langle s{:}\mathtt{q} \,\&\, \mathtt{p}{:}\mathtt{m}'(S')\rangle\, \top \Rightarrow \langle s{:}\mathtt{p},\mathtt{q}{:}\mathtt{m}\rangle\, \top\right) \\ \wedge\ \left(\langle s{:}\mathtt{p} \oplus \mathtt{q}{:}\mathtt{m}(S)\rangle\, \top \Rightarrow [s{:}\mathtt{q} \oplus \mathtt{p}{:}\mathtt{m}'(S')]\, \bot\right) \\ \wedge\ \left(\langle s{:}\mathtt{p} \,\&\, \mathtt{q}{:}\mathtt{m}(S)\rangle\, \top \Rightarrow [s{:}\mathtt{q} \,\&\, \mathtt{p}{:}\mathtt{m}'(S')]\, \bot\right) \ \wedge\ [s{:}\mathtt{p},\mathtt{q}{:}\mathtt{m}]\, Z \end{array} \right)$$
2. Γ is **deadlock-free**, written $\mathrm{df}(\Gamma)$, iff:
$$\Gamma \models \nu Z. \left(\begin{array}{l} \left((\forall s, \mathtt{p}, \mathtt{q}, \mathtt{m}.[s{:}\mathtt{p},\mathtt{q}{:}\mathtt{m}]\, \bot) \ \Rightarrow\ \forall s, \mathtt{p}, \mathtt{q}, \mathtt{m}, S.[s{:}\mathtt{p} \,\&\, \mathtt{q}{:}\mathtt{m}(S)]\, \bot \wedge [s{:}\mathtt{p} \oplus \mathtt{q}{:}\mathtt{m}(S)]\, \bot \right) \\ \wedge\ \forall s, \mathtt{p}, \mathtt{q}, \mathtt{m}.[s{:}\mathtt{p},\mathtt{q}{:}\mathtt{m}]\, Z \end{array} \right)$$
3. Γ is **terminating**, written $\mathrm{term}(\Gamma)$, iff:
$$\Gamma \models \mu Z. \left(\begin{array}{l} \left((\forall s, \mathtt{p}, \mathtt{q}, \mathtt{m}.[s{:}\mathtt{p},\mathtt{q}{:}\mathtt{m}]\, \bot) \ \Rightarrow\ \forall s, \mathtt{p}, \mathtt{q}, \mathtt{m}, S.[s{:}\mathtt{p} \,\&\, \mathtt{q}{:}\mathtt{m}(S)]\, \bot \wedge [s{:}\mathtt{p} \oplus \mathtt{q}{:}\mathtt{m}(S)]\, \bot \right) \\ \wedge\ \forall s, \mathtt{p}, \mathtt{q}, \mathtt{m}.[s{:}\mathtt{p},\mathtt{q}{:}\mathtt{m}]\, Z \end{array} \right)$$
4. Γ is **never-terminating**, written $\mathrm{nterm}(\Gamma)$, iff:
$$\Gamma \models \nu Z. (\exists s, \mathtt{p}, \mathtt{q}, \mathtt{m}.\langle s{:}\mathtt{p},\mathtt{q}{:}\mathtt{m}\rangle\, \top \ \wedge\ \forall s, \mathtt{p}, \mathtt{q}, \mathtt{m}.[s{:}\mathtt{p},\mathtt{q}{:}\mathtt{m}]\, Z)$$
5. Γ is **live**, written $\mathrm{live}(\Gamma)$, iff:
$$\Gamma \models \nu Z. \left(\begin{array}{l} \forall s, \mathtt{p}, \mathtt{q}.\ \left((\exists \mathtt{m}, S.\langle s{:}\mathtt{q} \,\&\, \mathtt{p}{:}\mathtt{m}(S)\rangle\, \top) \ \Rightarrow\ \mu Z'.\exists \mathtt{m}.\langle s{:}\mathtt{p},\mathtt{q}{:}\mathtt{m}\rangle\, \top \ \vee\ \exists \mathtt{p}', \mathtt{q}', \mathtt{m}'.\langle s{:}\mathtt{p}',\mathtt{q}'{:}\mathtt{m}'\rangle\, Z' \right) \\ \wedge\ \forall \mathtt{m}.\left((\exists S.\langle s{:}\mathtt{p} \oplus \mathtt{q}{:}\mathtt{m}(S)\rangle\, \top) \ \Rightarrow\ \mu Z'.\langle s{:}\mathtt{p},\mathtt{q}{:}\mathtt{m}\rangle\, \top \ \vee\ \exists \mathtt{p}', \mathtt{q}', \mathtt{m}'.\langle s{:}\mathtt{p}',\mathtt{q}'{:}\mathtt{m}'\rangle\, Z' \right) \\ \wedge\ \forall \mathtt{m}.[s{:}\mathtt{p},\mathtt{q}{:}\mathtt{m}]\, Z \end{array} \right)$$

Figure 9.1: Properties of typing contexts as μ-calculus formulae

on the right side.

Since our system does not sport immediate capabilities with barbs this time around, we do not use a pointed LTS to run the μ-calculus properties on, and instead look directly at the typing context. When a typing context Γ satisfies a formula ϕ, we write $\Gamma \models \phi$.

We show in Figure 9.1 the properties of our types in the μ-calculus defined above. Those μ-calculus formulae can be exploited by model checking tools using a μ-calculus variant for expressing their properties, e.g. `mCRL2`. It is important that those formulae abstract over silent and internal actions: the μ-calculus model and properties are designed to be applied on the multiparty session types, which make silent actions (such as definition expansion) invisible, and internal actions either invisible — when the related session is restricted and was independently verified — or explicit — when the related session is in the process of being verified. As such, no τ action exists in either the types reduction or the `mCRL2` model. The properties expressed in Figure 9.1 are respectively read as follows:

- **safety (1)** checks that, if an output on $\mathtt{m}$ and an input on $\mathtt{m}'$ are enabled between two roles $\mathtt{p}$ and $\mathtt{q}$,

then they can communicate via $\mathtt{m}$ (i.e., by Definition 8.6, the output message $\mathtt{m}$ must be supported by the recipient); and that if an output (resp. input) is enabled between roles $\mathtt{p}$ and $\mathtt{q}$ on a session, then no output (resp. input) is enabled in the reverse direction between $\mathtt{q}$ and $\mathtt{p}$. This must hold for any context reachable via communication transitions, which is enforced by the greatest fixed point $\nu Z.\ldots$ and the recursion clause $\ldots \wedge [s{:}\mathtt{p},\mathtt{q}{:}\mathtt{m}]\, Z$;

- **deadlock-freedom (2)** checks whether communication is possible; if not (clause "$\forall\ldots.[s{:}\mathtt{p},\mathtt{q}{:}\mathtt{m}]\,\bot$", holds only when no roles can interact), then ($\Rightarrow$) there must be no input nor output transitions enabled — i.e., all typing context entries must be **end**. This must hold for any context reachable via communications, which is enforced by $\nu Z.\ldots$ and $\ldots \wedge \forall\ldots.[s{:}\mathtt{p},\mathtt{q}{:}\mathtt{m}]\, Z$;

- **termination (3)** is similar to deadlock-freedom, but uses a *least* fixed point $\mu Z.\ldots$: hence, the clause $\ldots \wedge \forall\ldots.[s{:}\mathtt{p},\mathtt{q}{:}\mathtt{m}]\, Z$ can only iterate for a *finite* number of times, at which point no communications, nor inputs, nor outputs must be enabled — i.e., all context entries are **end**;

- **never-termination (4)** checks that in any context reachable via communication transitions ($\nu Z.\ldots$ and $\ldots \wedge \forall\ldots.[s{:}\mathtt{p},\mathtt{q}{:}\mathtt{m}]\, Z$), some further communication is possible ($\exists\ldots.\langle s{:}\mathtt{p},\mathtt{q}{:}\mathtt{m}\rangle\, \top$);

- **liveness (5)** checks that, if an input or output between two roles $\mathtt{p}$ and $\mathtt{q}$ is enabled, then ($\Rightarrow$) a corresponding communication can be fired, after a finite sequence of communications among any role. The sequence is built with a least fixed point $\mu Z'.\ldots$, that can iterate on the clause $\ldots \vee \exists\ldots.\langle s{:}\mathtt{p}',\mathtt{q}'{:}\mathtt{m}'\rangle\, Z'$ for a finite number of times. The top-level greatest fixed point $\nu Z.\ldots$ repeats the check for all contexts reachable via communication (clause $\ldots \wedge \forall\ldots.[s{:}\mathtt{p},\mathtt{q}{:}\mathtt{m}]\, Z$)

As a corollary of Theorems 8.16, 8.22 and 8.29, we have a precise characterisation for the unique-role process subset:

Theorem 9.1 (Precise Characterisation)**.** Suppose $\vdash P \blacktriangleright \Gamma$ such that $P \equiv \big|_{\mathtt{p}\in I} P_{\mathtt{p}}$, each $P_{\mathtt{p}}$ either is $\mathbf{0}$ (up-to $\equiv$), or only plays $\mathtt{p}$ in s, and Γ is deducted from P using the typing rules with minimal typing for $\leqslant$; then, the following properties are equivalent: **(1)** $\mathrm{safe}(\Gamma)$ (cf. Definition 8.7); **(2)** Γ satisfies the μ-calculus safety property (cf. Figure 9.1 Item 1); and **(3)** $\mathrm{safe}(P)$ (cf. Definition 7.7).

Furthermore, if $\mathrm{safe}(\Gamma)$, then for $\varphi \in \{\mathrm{df}, \mathrm{term}, \mathrm{nterm}, \mathrm{live}\}$, the following properties are equivalent: **(1)** $\varphi(\Gamma)$ (cf. Figure 8.2); **(2)** Γ satisfies the μ-calculus property for φ (cf. Figure 9.1); and **(3)** $\varphi(P)$ (cf. Definition 7.7). ⌟

Given backtracking techniques on the exploration of the reduction tree by the μ-calculus-enabled tools, we can assume the possibility of extracting witnesses to the answer of proposition checking: if one of the properties, say safety, is not satisfied, we can backtrack from the node where the checking algorithm

encountered a `false` ($\bot$), extracting the path taken in the reduction, and reconstruct a counterexample from the nodes on that path.

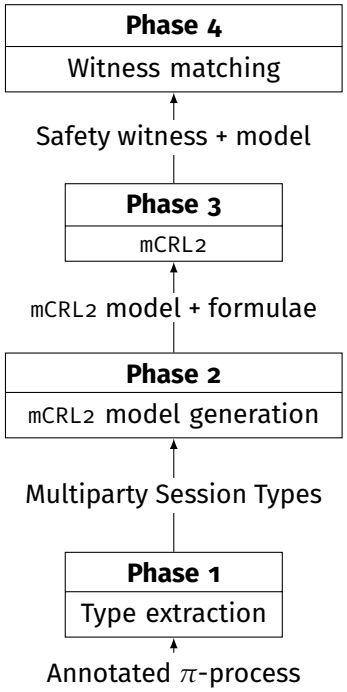
This counterexample is then given in the types (or equivalent representation), and given enough external information, we could match this type to a part of the typed process. Hence, a tool can give indications on liveness or safety issues of processes solely based on the syntax, without the need to compile it with instrumentation. See Section 9.2 for the tool development.

Remark 9.2 (Session Fidelity). Linking our results, we explain how we have discovered an error of the original definition of *unique-role processes* [SY19a]. The counterexample processes given in Example 8.30 satisfies *safety* according to our tool (described in Section 9.2 below), and satisfies the original definition of unique-role processes [SY19a]. Hence, by Theorem 9.1, both tool and theory should verify the process properly. This is however not the case, since the types for this example show as deadlock-free, while the process clearly deadlocks in normal execution. We found it to be due to how channel delegation is handled in the original model, which allows both the receiving endpoint and the endpoint received as data to remain active in the same process. We decided to handle that by adding a clause to the definition of unique-role processes, as given in Definition 8.19 in our work.

Notified of the issue, the authors of the seminal work went another route for their own fix, and instead changed the session fidelity statement, as visible in the revised version of the technical report [SY19b] associated with their work [SY19a]. Their fix resolves the loophole in their proof without having to change it, by adding the necessary hypothesis in the theorem's premises instead. Specifically, they assumed proper handling of delegation in point (2) of Equation (68) in their proof, but that assumption was not reflected anywhere originally, which was the source of the inconsistency. Their fix still allows channel passing in the definition of unique-role processes, but forbids it in the session fidelity theorem, while ours forbids channel passing at the source, hence is more restrictive, but it works for our needs in this framework and keeps the scope of the theorems easier to grasp in general. ⌟

9.2 Implementation and Counterexample Extraction

Our tool is written essentially in `Scala` and is composed of four phases, shown in Figure 9.2. The general behaviour of our tool is as follows: when fed with a multiparty-session π-calculus process, the tool first extracts multiparty session types from it. It then uses the types to generate an `mCRL2`-compatible model, including μ-calculus formulae corresponding to predefined properties such as *type safety*; then uses `mCRL2` to check whether the model (and thus the extracted types) comply with said properties. It extracts witnesses during that verification process; and then returns the results to the user. If only one protocol and one property were analysed, the tool also returns the translated version of the witness to the user, so they



Phase 4 is run if only one protocol and property are verified. Matches the witness with the information kept track of in **Phase 2**.

Phase 3 model checks the mCRL2 model against the formulae, and returns the verification results.

Phase 2 uses the Multiparty Session Types to generate a mCRL2 model and modal μ-calculus formulae, while keeping track of the translation between the names in the MPST model and the mCRL2 model.

Phase 1 extracts Multiparty Session Types from the input annotated π-calculus process from the user.

Figure 9.2: Workflow of the tool

can use it to efficiently determine where the error lies, if any. Both the original tool and our extension are available on GitHub [SY19c, Gab22].

We now present the details of our toolchain.

9.2.1 Phase 1: Type Extraction

```
> oauth2-rec-bad.proc
## p = server
## q = client
## r = authenticator

def P = s[p][q]◁{login.s[p][r]Σauth(x@Bool).0, cancel.P()} in
def Q = s[q][p]Σ{login.s[q][r]◁passwd.0, cancel.s[q][r]◁quit.Q()} in
def R = s[r][q]Σ{passwd.s[r][p]Σauth(true@Bool).0, quit.R()} in
( P() | ( Q() | R() ))
```

```
❯ ./bin/mpstk extract examples/oauth2-rec-bad.proc
s[p]: μ(P)q⊕{login(end).r&{auth(bool).end}, cancel(end).P},
s[r]: μ(R)q&{passwd(end).p&{auth(bool).end}, quit(end).R},
s[q]: μ(Q)p&{login(end).r⊕{passwd(end).end}, cancel(end).r⊕{quit(end).Q}}
```

Figure 9.3: Top: syntax of the processes in the tool; Bottom: extracted types output

The first action of the tool is type extraction. The tool parses the process, and provides projected multiparty session types as an output — or an error if the types cannot be extracted. Figure 9.3 shows on the

top side the *new* syntax of the processes in our tool and on the bottom the extracted types from that same set of processes. The extraction works by reading each action and adding it to the types of its endpoint, and currently relies on user annotation for the types of payloads, as we allow sorts but do not expect any channel passing to comply with the premises of Theorems 8.22 and 8.29. Recursion in definitions is not implemented in this proof-of-concept, but should be achievable with the use of unification algorithms. This is however left for future work, as it would also fall into the scope of a proper type inference tool for our formalism — also left for future work as it would warrant a full publication of its own.

9.2.2 Phase 2: Model Generation

```
%%% Processes representing typing context entries
% E0 => (s[p],μ(P)q⊕{login(end).r&{auth(bool).end}, cancel(end).P})
% E1 => (s[r],μ(R)q&{passwd(end).p⊕{auth(bool).end}, quit(end).R})
% E2 => (s[q],μ(Q)p&{login(end).r⊕{passwd(end).end}, cancel(end).r⊕{quit(end).Q}})

%%% Processes generated from multiparty session types recursion variables
% P0 => (s[p],P)
% P1 => (s[r],R)
% P2 => (s[q],Q)

proc context = (E0) || (E1) || (E2);

proc E0 = (P0);

proc P0 = ((o(s(0), r(1), r(2), m(1), pEnd) . (i(s(0), r(0), r(1), m(0), pBool)))
+ o(s(0), r(1), r(2), m(2), pEnd) . P0);

proc E1 = (P1);

proc P1 = ((i(s(0), r(2), r(0), m(3), pEnd) . (o(s(0), r(0), r(1), m(0), pBool)))
+ i(s(0), r(2), r(0), m(4), pEnd) . P1);

proc E2 = (P2);

proc P2 = ((i(s(0), r(1), r(2), m(1), pEnd) . (o(s(0), r(2), r(0), m(3), pEnd))) +
 i(s(0), r(1), r(2), m(2), pEnd) . (o(s(0), r(2), r(0), m(4), pEnd) . P2));
```

Figure 9.4: The extracted mCRL2 model

Another parser instance translates the types from our user-readable format to mCRL2 format. The result of this translation is shown in Figure 9.4. This mCRL2 format is still readable, but less intuitive than MPST and not intended to be read by the user in general use. The tool then generates the μ-calculus formulae of the properties. In this phase, the tool also creates a table to keep track of the translation between the mCRL2 format names and the MPST sessions, roles, labels and payloads. This table is kept as a comment block above the process description commented at the top of Figure 9.4, in a similar format. This is important to later be able to do the reverse translation automatically in phase 4, from generated witness using mCRL2

names to user-readable MPST counterexample.

9.2.3 Phase 3: Model Checking with `mCRL2`

The tool feeds the model and formulae to `mCRL2`, which model checks the given model against those formulae. Each formula is checked independently, which can take some time for larger examples (this is a known limitation of model checking tools). This is also the step where a witness can first be extracted for the properties. The user is later able to use the returned version of this witness as a counterexample to pinpoint the location of the error, if there was one.

9.2.4 Phase 4: Witness Extraction

```
> ./bin/mpstk verify -p=safety oauth2-rec-bad.ctx
protocol            safe
oauth2-rec-bad.ctx  false

witness:
s[p][r]Σauth(@Bool)
s[r][p]Σauth(@Bool)
```

Figure 9.5: Verification results with witness

The last pass of the tool is the translation of the witness to MPST format. This is run when only one protocol and one property are being verified. It is a simple parsing step, which returns the generic form of the process actions at the end of the witness, including the correct name for the session, roles and labels. While it does not include all previous actions, the amount of information returned at this step is usually enough to pinpoint the location of the error in the main program with a simple text search, should there be one. Figure 9.5 shows such result with a witness for the unsafe OAuth fragment.

```
> ./bin/mpstk verify oauth2-rec.ctx oauth2-rec-bad.ctx
protocol            df     live   live+  live++  nterm  safe   term
oauth2-rec.ctx      true   true   true   true    false  true   false
oauth2-rec-bad.ctx  false  false  false  false   false  false  false
```

Figure 9.6: Full verification results

Figure 9.6 shows the full verification output for the example in Figure 9.3. It presents itself in the form of a table, each line corresponds to one checked protocol, and each column one property, as the tool can check several of both at a time. A legend at the bottom helps keep track of which column represents what property (not shown here as it takes space and the names correspond to this work's nomenclature). Note that `live+` and `live++` are more stringent liveness properties kept from the base tool and work [SY19a,

Table 9.1: Verification results of our recursive OAuth2 example, and other protocols from [SY19a]

Protocol	states	safe	df	live	nterm	term
`OAuth2`	37	✓	✓	✓	×	×
`OAuth2 (unsafe)`	37	×	×	×	×	×
`Rec. two-buyers`	85	✓	✓	✓	×	×
`Rec. map/reduce`	2561	✓	✓	✓	×	×
`MP workers`	442369	✓	✓	✓	×	×

df: deadlock-free, (n)term: (never-)termination,

✓: formula is true, ×: formula is false

SY19c]. Though we do not explain them in this paper, they can be useful for specific use cases and remain in the tool.

Table 9.1 shows the results our tool gives when verifying our recursive versions of OAuth2, both safe and unsafe, along the other protocols given for examples by Scalas and Yoshida [SY19a]: an implementation of a recursive two-buyers protocol; and both a recursive map and reduce protocol and an independent multiparty workers protocol instantiated with $n = 3$ workers each time. Those last three protocols are given to show how our tool gives consistent results with the previous iteration — barring potentially for safety, which is extended to handle more errors in our work, but is still verified true here as the examples are properly constructed. The number of states is given for indication: our tool only explores transitions of proper communications and not partial actions (see Figure 9.1), meaning it does not have to go through the whole state-space. As a result, the time increase for bigger protocols is less pronounced than in full state-space exploration solutions like our previous work [GY20a]. For example on our machine, the biggest example in Table 9.1, *multiparty workers*, took around 30× more time than the OAuth fragment took to verify all properties, for a 12k-ratio in state-space size. Meanwhile, similar size differences in our previous work [GY20a] led to a time increase of more than 1000× (two orders of magnitude in difference), extrapolating from the number of summands that is given in their table to compare the smaller lock-versions of dining philosophers examples they provided to the channel versions of the same protocol.

10 Related Works

There is a vast number of studies on concurrent behavioural type theories [HLV+16], their integrations with programming languages [ABB+16] and tools [SG17]. Here we focus on the most closely related work. Scalas and Yoshida [SY19a] introduce the simplified multiparty session types we use in this work, and applies the `mCRL2` model checker to verify the local types. While our work takes their aproach [SY19a] (rather than the standard top-down approach [HYC16, HY16]), we met several challenges to accomplish this framework: first, they did not provide a proof of completeness for the errors (Theorem 8.22), nor a precise characterisation to link models, types and processes (Theorem 9.1). To achieve this, we needed to find out a precise error set, and prove Subject LTS (Proposition 8.28), defining a new set of LTS for MPST processes. We established both soundness (Theorem 8.16) and completeness in this work, along with a tighter integration between types and processes in our theory. Our tool provides more in comparison as well, with type extraction from processes and counterexample extraction, allowing for precise error reporting.

Moreover, we found and fixed an error in their work's session fidelity section (see Example 8.30). Telling the authors about the error allowed them to provide a fix for this issue: their own fix allows them to not change the proof that was in their technical report [SY19b]. Their mistake was due to an assumption in point (2) after Equation (68) in that report, that was not reflected in the theorem, and adding that restriction to the theorem fixes it. Our fix forbids channel passing just like theirs does, but we decided to include it in the definition of unique-role processes instead of keeping it under an additional restriction in the theorem, making the scope of the related theorems clearer to the reader.

On the model checking side, Cranen, Luttik and Willemse [CLW15] lay out theoretical grounds for counterexample extraction for models using fixed-point logic, matching the μ-calculus we use with `mCRL2`. Abdulla, Atig and Diep [AAD16] use counterexamples to guide verification of programs and let users fix the detected problems, motivating us to implement that for multiparty session types. More works [WW18, BGK+19] detail analysing concurrent systems and extracting counterexamples using model checking with `mCRL2`, supporting our use of `mCRL2` in our tool.

Liu, Zhu, Qin, Chen and Song [LZQ+21] give a different approach: an algorithmic refinement first extracts parts of the communications in the analysed code that may pose communication problems; those are then checked along their intertwined communications to determine if they are causing a bug or not. This approach is neither sound nor complete (resulting in both false positives and negatives), but has the

advantage of being lighter to run, making the analysis more scalable on larger programs. The approach is complementary and can be combined with a model checking approach like ours towards more precise theoretically-founded verification. A complementary approach to detecting channel-related bugs by extensive use of fuzzing techniques is proposed by Liu, Xia, Liang, Song and Hu [LXL+22]. Their approach identifies concurrent messages in the target program, then mutates it to enable processing those messages in any order, before applying fuzzing techniques to generate new processing orders from previous ones. Veileborg, Saioc and Møller [VSM23] go even further than both previous works by using localised abstract interpretation techniques to cut the program in independently verifiable components. While this approach is still neither sound nor complete, it manages to detect more bugs that went unnoticed by the previous works [LZQ+21] while remaining highly scalable, which makes it a promising approach for large scale detection of blocking bugs in channel-based concurrent programs.

Scalas, Yoshida and Benussi [SYB19] develop an approach of type-level verification using model-checking, and apply it in the Dotty language (now Scala 3). While they use dependent behavioural types where we use multiparty session types, the approach uses a μ-calculus for property-verification on the types and the `mCRL2` model checker. Our own previous work and the approaches it extends upon [GY20a, LNTY17, LNTY18] make use of LTS semantics for their process calculi and type systems, though they use CCS-like concurrent behavioural types. Their target is the `Go` language. These previous works [SYB19, GY20a, LNTY17, LNTY18] suggest a promising future work: an application of our framework towards real message-passing programming languages.

Note that none of the mentioned works on behavioural types study a counterexample guided approach, focusing only on reliable detection of errors with no further help for locating or fixing them.

Part IV

Conclusion & Future Works

On behavioural types for Go programs

The Go language provides a unique programming environment where both explicit communication and shared memory concurrency primitives co-exist. Our work introduced MiGo$^+$ as an abstraction layer for Go code, as well as behavioural types to propose a static verification framework for detecting concurrency bugs in Go. Detected bugs include deadlocks and safety for both mutual exclusion locks and channel communication, as well as data races for shared memory primitives.

Shared memory locks and channels cover by themselves a substantial amount of Go's concurrency features. The former is a low-level, standard library provision and the latter is a high-level, built-in language feature. Go mainly features these two basic building blocks, and one can use them to implement most higher levels of concurrency abstraction, for example actors models.

Our framework, built on previous works [LNTY17, LNTY18], extends their channel-only approach to account for more core features of Go, using the latter's own official memory model in the process to account for action ordering in the design of the verified properties.

Future works for this framework include studying the soundness and completeness of the happens-before relation provided by the Go memory model, i.e. studying if the definition of data race derived from it covers all data races that can happen in Go, and whether it does not provide false positives; speeding-up the analysis using more mCRL2 options and the extension to an incremental analysis based on happens-before relations, as taken in other languages [LH18, ZH16]; as well as possibly counter-example extraction for code failing verification, to provide direct access to the detected bugs to developers. There is also the possibility to work on handling dynamic process creation, widening the analysis scope of our current tool and model.

On the multiparty session π-calculus

Our work proposed a counterexample-guided framework for multiparty session type theories, identifying a precise relationship with modal μ-formulae. Based on the new MPST framework devised by Scalas and Yoshida [SY19a], we extended its properties of communication safety; expanded the scope of the processes to account for a form of non-determinism; and spotted — and fixed — an error in their original framework in the form of an oversight of some channel-passing behaviours in their main theorem.

To complement our framework, we took on extending their tool into a tool which can extract types from untyped multiparty session π-processes and identify errors, using the `mCRL2` model checker. Our proof-of-concept tool can extract simple processes without recursion into matching types, then feeds the extracted types to the original pipeline to analyse them. We adapted the verification formulae to our extended model in that pipeline, and adapted it to leverage the witness extraction capabilities present in `mCRL2`. The returned witness is then read by another addition of ours at the end of the pipeline, to return a user-readable position for the detected error, should there be one.

Future work on this framework includes refining the tool to extract types on a bigger fragment of the processes, since it does not yet handle recursion in particular; applying the framework to main-stream programming languages such as `Go` and Scala; and integrating with other tools from adjacent works [SYB19, GY20a, LNTY17, LNTY18], some of which integrate language-specific features or bugs which may benefit from more recent and general techniques. Integrating our extension with the crash-stop failure extension by Barwell, Scalas, Yoshida and Zhou [BSYZ22] is another topic that needs research, to produce a general and versatile session types-based framework covering both kinds of errors at once.

Bibliography

[AAD16] Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Bui Phi Diep. Counter-example guided program verification. In John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM 2016: Formal Methods*, pages 25–42, Cham, 2016. Springer International Publishing.

[ABB+16] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Found. Trends Program. Lang.*, 3(2–3):95–230, July 2016. doi:10.1561/2500000031.

[ABB+17] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral Types in Programming Languages. *Foundations and Trends in Programming Languages*, 3(2-3), 2017. doi:10.1561/2500000031.

[AFF06] Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, March 2006. doi:10.1145/1119479.1119480.

[BGK+19] Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs, and Tim A. C. Willemse. The mcrl2 toolset for analysing concurrent systems. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 21–39, Cham, 2019. Springer International Publishing.

[Bra92] Julian Charles Bradfield. *Verifying temporal properties of systems*. Progress in theoretical computer science. Birkhauser, Boston, 1992.

[BSYZ22] Adam D. Barwell, Alceste Scalas, Nobuko Yoshida, and Fangyi Zhou. Generalised Multiparty Session Types with Crash-Stop Failures. In Bartek Klin, Sławomir Lasota, and Anca Muscholl, editors, *33rd International Conference on Concurrency Theory (CONCUR 2022)*, volume 243 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 35:1–35:25, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2022/17098, doi:10.4230/LIPIcs.CONCUR.2022.35.

[CDY14] Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. On the preciseness of subtyping in session types. In *PPDP*, 2014. doi:10.1145/2643135.2643138.

[CHJ+19] David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. Distributed Programming Using Role Parametric Session Types in Go. In *46th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 1–30. ACM, 2019. doi:10.1145/3290342.

[CLW15] Sjoerd Cranen, Bas Luttik, and Tim A. C. Willemse. Evidence for Fixpoint Logic. In Stephan Kreutzer, editor, *24th EACSL Annual Conference on Computer Science Logic (CSL 2015)*, volume 41 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 78–93, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CSL.2015.78.

[CR22] Milind Chabbi and Murali Krishna Ramanathan. A study of real-world data races in golang. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 474–489, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3519939.3523720.

[CRR02] Sagar Chaki, Sriram K. Rajamani, and Jakob Rehof. Types as models: model checking message-passing programs. In *POPL'02*, pages 45–57, 2002.

[DK15] Alan A.A. Donovan and Brian W. Kernighan. *The Go Programming Language*. Addison-Wesley Professional, 1st edition, 2015.

[DL19] Nicolas Dilley and Julien Lange. An empirical study of messaging passing concurrency in Go projects. In *SANER*. IEEE, 2019.

[FB08] Stephan Falke and Mark Brockschmidt. KITTeL/KoAT on GitHub, 2008. URL: https://github.com/s-falke/kittel-koat.

[Fra18] Steve Francia. Nine years of Go. https://blog.golang.org/9years, 2018.

[Gab20a] Julia Gabet. Godel 2 Benchmarks on GitHub, 2020. Based on original scripts from Julien Lange and Bernardo Toninho. URL: https://github.com/JujuYuki/godel2-benchmark.

[Gab20b] Julia Gabet. Godel 2 source code on GitHub, 2020. Based on original code from Julien Lange and Bernardo Toninho. URL: https://github.com/JujuYuki/godel2.

[Gab20c] Julia Gabet. migoinfer+ source code on GitHub, 2020. Forked from the original work of Nicholas Ng. URL: https://github.com/JujuYuki/gospal.

[Gab22] Julia Gabet. mpstk version 2, 2022. URL: https://github.com/JujuYuki/mpstk.

[Git18] GitHub. The fastest growing languages, 2018. http://octoverse.github.com/.

[Git22] GitHub. The fastest growing languages, 2022. https://octoverse.github.com/2022/top-programming-languages.

[GM14] Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and Analysis of Communicating Systems*. The MIT Press, 2014.

[Go 22a] Go developers. mutex.go, 2009-2022. URL: https://go.dev/src/sync/mutex.go.

[Go 22b] Go developers. rwmutex.go, 2009-2022. URL: https://go.dev/src/sync/rwmutex.go.

[Go 22c] Go developers. The Go Memory Model, 2022. URL: https://go.dev/ref/mem.

[Go13] Go. Data Race Detector. https://golang.org/doc/articles/race_detector.html, 2013.

[GY20a] Julia Gabet and Nobuko Yoshida. Static Race Detection and Mutex Safety and Liveness for Go Programs. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:30, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2020.4.

[GY20b] Julia Gabet and Nobuko Yoshida. Static Race Detection and Mutex Safety and Liveness for Go Programs (Artifact). *Dagstuhl Artifacts Series*, 6(2):12:1–12:3, 2020. URL: https://drops.dagstuhl.de/opus/volltexte/2020/13209, doi:10.4230/DARTS.6.2.12.

[HBK19] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris: Session-type based reasoning in separation logic. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:10.1145/3371074.

[HLV+16] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1), 2016. doi:doi:10.1145/2873052.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hon93] Kohei Honda. Types for dyadic interaction. In *CONCUR*, 1993. doi:10.1007/3-540-57208-2_35.

[HVK98] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, 1998. doi:10.1007/BFb0053567.

[HY16] Raymond Hu and Nobuko Yoshida. Hybrid session verification through endpoint API generation. In *FASE*, 2016.

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, 2008. Full version in [HYC16]. doi:doi:10.1145/1328438.1328472.

[HYC16] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1), 2016. Full version of the eponymous POPL'08 paper http://doi.acm.org/10.1145/1328438.1328472. doi:doi:10.1145/2827695.

[IK01] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. In *POPL*, 2001. doi:doi:10.1145/360204.360215.

[Kla19] Klaus v. Gleissenthall and Rami Gökhan Kıcı and Alexander Bakst and Deian Stefan and Ranjit Jhala. Pretend synchrony: Synchronous verification of asynchronous distributed programs. *Proc. ACM Program. Lang.*, 3(POPL):59:1–59:30, January 2019. URL: http://doi.acm.org/10.1145/3290372, doi:10.1145/3290372.

[Koz83] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983. Special Issue Ninth International Colloquium on Automata, Languages and Programming (ICALP) Aarhus, Summer 1982. URL: https://www.sciencedirect.com/science/article/pii/0304397582901256, doi:https://doi.org/10.1016/0304-3975(82)90125-6.

[KS08] Naoki Kobayashi and Davide Sangiorgi. A hybrid type system for lock-freedom of mobile processes. *TOPLAS*, 32(5):16:1–16:49, May 2008.

[LH18] Bozhen Liu and Jeff Huang. D4: Fast concurrency debugging with parallel differential analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 359–373, New York, NY, USA, 2018. ACM. URL: http://doi.acm.org/10.1145/3192366.3192390, doi:10.1145/3192366.3192390.

[LNTY17] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. Fencing off Go: Liveness and Safety for Channel-based Programming. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 748–761, New York, NY, USA, 2017. ACM. doi:10.1145/3009837.3009847.

[LNTY18] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. A Static Verification Framework for Message Passing in Go using Behavioural Types. In *40th International Conference on Software Engineering*, pages 1137–1148. ACM, 2018. doi:10.1145/3180155.3180157.

[LXL+22] Ziheng Liu, Shihao Xia, Yu Liang, Linhai Song, and Hong Hu. Who goes first? detecting go concurrency bugs via message reordering. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 888–902, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3503222.3507753.

[LY17] Julien Lange and Nobuko Yoshida. On the undecidability of asynchronous session subtyping. In *FOSSACS*, 2017. doi:10.1007/978-3-662-54458-7_26.

[LZQ+21] Ziheng Liu, Shuofei Zhu, Boqin Qin, Hao Chen, and Linhai Song. Automatically detecting and fixing concurrency bugs in go software systems. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 616–629, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3445814.3446756.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92. Springer-Verlag, 1980.

[MNN18] Jan Midtgaard, Flemming Nielson, and Hanne Riis Nielson. Process-local static analysis of synchronous processes. In Andreas Podelski, editor, *Static Analysis*, pages 284–305, Cham, 2018. Springer International Publishing.

[NN94] Hanne Riis Nielson and Flemming Nielson. Higher-order concurrent programs with finite communication topology (extended abstract). In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, page 84–97, New York, NY, USA, 1994. Association for Computing Machinery. doi:10.1145/174675.174538.

[NY16] Nicholas Ng and Nobuko Yoshida. Static deadlock detection for concurrent Go by global session graph synthesis. In *CC*, pages 174–184. ACM, 2016.

[OAu12] OAuth Working Group. RFC 6749: OAuth 2.0 framework, 2012. URL: http://tools.ietf.org/html/rfc6749.

[Pik12] Rob Pike. Go at Google. In *SPLASH*, pages 5–6, New York, NY, USA, 2012. ACM.

[RR01] Sriram K. Rajamani and Jakob Rehof. A behavioral module system for the pi-calculus. In Patrick Cousot, editor, *Static Analysis*, pages 375–394, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[SCH23] Vivek Shahare, Milind Chabbi, and Nikhil Hegde. Protecting locks against unbalanced unlock(). In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '23, page 199–211, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3558481.3591091.

[Scr17] Scribble. project home page. http://www.scribble.org, 2017.

[SG17] António Ravara Simon Gay, editor. *Behavioural Types: from Theory to Tools*. River Publisher, 2017. URL: https://www.riverpublishers.com/research_details.php?book_id=439.

[SI09] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM. URL: http://doi.acm.org/10.1145/1791194.1791203, doi:10.1145/1791194.1791203.

[SPIV12] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. Dynamic race detection with LLVM compiler. In *Proceedings of the Second International Conference on Runtime Verification*, RV'11, pages 110–114, Berlin, Heidelberg, 2012. Springer-Verlag. URL: http://dx.doi.org/10.1007/978-3-642-29860-8_9, doi:10.1007/978-3-642-29860-8_9.

[SST16] Kai Stadmüller, Martin Sulzmann, and Peter Thiemann. Static Trace-Based Deadlock Analysis for Synchronous Mini-Go. In *APLAS*, volume 10017 of *LNCS*, 2016.

[Sta22] StackOverflow. Stackoverflow's 2022 developer survey. https://survey.stackoverflow.co/2022/, 2022.

[Sti01] Colin Stirling. *Modal and Temporal Properties of Processes*. Texts in Computer Science. Springer New York, New York, NY, 1st edition, 2001.

[SY19a] Alceste Scalas and Nobuko Yoshida. Less is More: Multiparty Session Types Revisited. *Proc. ACM Program. Lang.*, 3(POPL):30:1–30:29, January 2019. doi:10.1145/3290343.

[SY19b] Alceste Scalas and Nobuko Yoshida. Less is More: Multiparty Session Types Revisited. Technical report, Imperial College London, 2019. Technical report, updated July 21, 2021, latest version available at: http://mrg.cs.ox.ac.uk/publications/.

[SY19c] Alceste Scalas and Nobuko Yoshida. Less is More: Multiparty Session Types Revisited (Artifact), 2019. Peer-reviewed artifact of [SY19a]. Latest version available at: https://alcestes.github.io/mpstk. doi:10.1145/3291638.

[SYB19] Alceste Scalas, Nobuko Yoshida, and Elias Benussi. Verifying message-passing programs with dependent behavioural types. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 502–516, New York, NY, USA, 2019. Association for Computing Machinery. doi:doi:10.1145/3314221.3322484.

[Syz15] Syzkaller. Randomized testing for Go. https://github.com/google/syzkaller, 2015.

[Tec18] Technische Universiteit Eindhoven. mCRL2. https://www.mcrl2.org/web/user_manual/index.html, 2018.

[The15] The Clang Team. ThreadSanitizer. http://clang.llvm.org/docs/ThreadSanitizer.html, 2015.

[THK94] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE*, 1994. doi:10.1007/3-540-58184-7_118.

[TLSZ19] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. Understanding real-world concurrency bugs in Go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 865–878, New York, NY, USA, 2019. ACM. URL: http://doi.acm.org/10.1145/3297858.3304069, doi:10.1145/3297858.3304069.

[VG13] Dmitry Vyukov and Andrew Gerrand. Introducing the Go Race Detector. https://blog.golang.org/race-detector, 2013.

[VSM23] Oskar Haarklou Veileborg, Georgian-Vlad Saioc, and Anders Møller. Detecting blocking errors in go programs using localized abstract interpretation. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3551349.3561154.

[Vyu15] Dmitry Vyukov. Randomized testing for Go. https://github.com/dvyukov/go-fuzz, 2015.

[WW18] Wieger Wesselink and Tim A.C. Willemse. Evidence extraction from parameterised boolean equation systems. In Christoph Benzmüller and Jens Otten, editors, *Proceedings of the 3rd International Workshop on Automated Reasoning in Quantified Non-Classical Logics (ARQNL 2018)*, CEUR Workshop Proceedings, pages 86–100. CEUR-WS.org, January 2018. 3rd International Workshop on Automated Reasoning in Quantified Non-Classical Logics, (ARQNL 2018), ARQNL2018 ; Conference date: 18-07-2018 Through 18-07-2018. URL: http://ceur-ws.org/Vol-2095/.

[YDBH10] Nobuko Yoshida, Pierre-Malo Deniélou, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. In *FOSSACS*, 2010. doi:doi:10.1007/978-3-642-12032-9_10.

[ZH16] Sheng Zhan and Jeff Huang. Echo: Instantaneous in situ race detection in the IDE. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 775–786, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2950290.2950332, doi:10.1145/2950290.2950332.