University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Capable GV: Capabilities for Session Types in GV

Magdalena J. Latifa

March 30, 2022

Abstract

Session types are a type formalism that describes concurrency through a sequence of actions where each action has a type and a direction of message exchange. They provide useful communication guarantees such as communication safety, privacy and session fidelity. Session types are commonly used to describe communication in process calculi such as the π-calculus, however, this paradigm lacks practicality when it comes to implementation. Hence, session types have been introduced to the functional paradigm and GV is one of the most important works of this type. GV benefits from higher-order functions, separation of run-time configuration and an overall more natural fit for implementations, while also enjoying the guarantees of session types. Since its original introduction by Wadler, GV has benefited from various extensions that improve its expressivity and guarantees. None of these extensions, however, have allowed for channel sharing, which excludes an entire set of programs from GV programs. This project aims to solve this problem by exploring the notion of capabilities in the context of GV with an end goal of creating an extension to GV with channel sharing. Capabilities split channels into a linear capability and an unrestricted endpoint, thus allowing channel sharing. Consequently, the extension should enjoy greater expressivity and allow cyclic processes at the cost of losing deadlock freedom.

Education Use Consent

Contents

1 Introduction

1.1 Motivation

Modern software development heavily leverages distributed and concurrent systems. Large scale concurrent and distributed systems are now the backbone of software from network protocols to cloud data centres. Structuring such systems, however, is complex and introduces bugs that are difficult to find and fix. Thus, formal systems are critical to guarantee reliability and predictability as they can help in preventing whole sets of errors that could occur during the run-time of a program.

Thus, we can look to session types, a formalism for concurrent communication introduced by Honda (1993), to formalise, and subsequently verify, this type of software. Session types provide certain communication guarantees, such as communication safety, privacy and session fidelity, and allow writing correct-by-construction programs. However, session-typed languages often lack the needed flexibility to be usable for implementations. This has been especially the case as they are most commonly introduced in the context of process calculi, in particular π-calculus, which are troublesome both to program in and to reason about at run-time.

In order to address this problem, session types have been introduced into the functional paradigm to take advantage of its practicality. First introduced by Wadler (2012) and inspired by the work of Gay and Vasconcelos (2010), Good Variation (GV) is a functional calculus with session types. However, the current body of work on GV and its extensions do not allow for channel sharing, hence lacking the expressivity to allow programs that require sharing. Hence, exploring how channel sharing could be introduced into a GV-based language and what the implications of that are would allow us to expand the body of work on practical session-typed languages while also expanding the set of programs allowed.

1.2 Aims

As defining and proving a GV extension is outside the scope of this dissertation and the work on the system is on-going, we provide aims for both this dissertation and the final work. For this dissertation, we want to focus on exploring the concepts of channel sharing in the context of a GV-based language which can be further used to complete the system and thus we will not be presenting a complete system. Specifically, the dissertation includes finished work on the language elements explored as well as discussion and reasoning on the future work that could be undertaken when finalising the remaining elements of the system.

Thus, the specific aims of this dissertation are to explore channel sharing in GV by providing the design and considerations for introducing capabilities and our findings. To do so, we first discuss how an extension to GV is developed and what changes ought to be made to the system. Secondly, we provide the static elements of the system which explore in depth a way of introducing channel sharing via capabilities in a system closely based on the previous works on GV. Further, we develop case studies and provide their typing derivations to showcase the expressive power of channel sharing in GV. We provide a discussion on the expected properties of the system and the reasoning behind our claims. Finally, we discuss the remaining elements of the system as future work and discuss possible directions.

The overall goal for the final work will be to develop a full GV extension which allows channel sharing. This includes utilising the findings from this dissertation, in particular how capabilities can be incorporated into the static elements of the system, and defining the run-time elements and operational semantics. Then, the properties of the system can be reasoned about and proved. Thus, the final aim of this research is to develop a more expressive GV calculus that we expect will preserve subject reduction but not progress due to the reintroduction of deadlocks.

1.3 Outline

In this dissertation, we present the following chapters:

- Chapter 2 provides the context for this project. We include background and related works on the metatheory of formal type systems and the key areas of this research i.e. session types, GV and channel sharing.
- Chapter 3 provides the problem statement, research questions and methodology that we have identified and followed.
- Chapter 4 provides the contribution in the form of definitions and typing rules of Capable GV (CGV). The static elements of CGV that have been defined to showcase the use of capabilities for channel sharing.
- Chapter 5 contains the evaluation of CGV by implementing different kinds of programs using CGV's syntax and performing typing derivations in order to reason about the expressive power of CGV.
- Chapter 6 concludes and discusses further direction for the work, including a discussion on remaining elements of the extension.

2 Background and Related Work

2.1 Metatheory

This section focuses on the basic concepts of type theory that are necessary to represent a formal system such as a GV extension. Works by Cardelli (1996) and Winskel (1993) have been used as the main point of reference for the concepts' definitions but we have also considered the common usage of the concepts mentioned in type theory and programming languages literature (Vasconcelos 2012; Dardha and Gay 2018; Voinea et al. 2019; Fowler et al. 2019).

2.1.1 Types and Type Systems

Types define the set of possible values that a variable can be of. For example, a trivial example of a boolean type allows only two values: true and false. Specifying what type a variable is of provides significant advantages to the language, including being able to reason about meaningful operations and error prevention. For example, we can define that an equality check operation is valid exclusively on variables of the same type, which allows us to reason about what makes terms equal. It also allows us to prevent a situation in which we attempt to compare terms of different types, for which defining such operation might not be possible, thus causing an error.

Type systems are a component that acts as a map between variables and their type or, in a broader sense, a map of an expression to its type. The term's type can either by provided directly, for example by explicit declaration, or implicitly based on the language's rules. Type systems are key to deciding whether a program is *well typed* where well-typability indicates what behaviour we can expect from the program. This varies for each system, with some restrictive languages allowing exclusively deadlock-free and terminating programs and some lenient languages allowing any program in which types match the operations used.

It is worth noting that reasoning about allowed operations and potential errors is possible only in *typed languages*. This means that the language needs to encode the notion of types and be able to track these. While typed languages fall on the scale from fully explicitly typed to mostly implicitly typed, as long as the system can decidably infer the types used in the program, it is able to reason about whether the program is *well-typed* or not. This is in contrast to *untyped languages* which do not provide any such benefits as they do not encode any notion of types.

2.1.2 Typing Judgements, Rules and Derivations

In order to prove properties of type systems, we need to formalise them via certain constructs and provide mechanisms to verify that programs belong to the language domain. Typing judgements, typing rules and typing derivations are the core components of reasoning about this.

Typing judgements are a formal way of reasoning about what is allowed under a typing system, usually by relating different entities into a formal assertion. Typically this takes on a form $\Gamma \vdash x : T$ which can be read as *under typing context* Γ*, term* x *is of type* T where Γ usually denotes an abstract type system that maps terms to their types.

Typing rules build upon typing judgements and allow specifying exact rules for a given type system and provide the basis of the language's type system. The language's set of typing rules is specified as a set of inference rules. Each inference rules can have zero or more premises and must have a conclusion. Rules with zero premises are known as axioms and provide the basis for the system that can operate under no assumptions. Specifying typing rules allows defining what programs are well typed given a specified typing context. For example, many languages contain a typing rule for variables which states that a variable x is of type T whenever typing context Γ contains this mapping. A more complex example of a typing rule could describe a function that specifies the return type given some arguments of certain types.

$$\textsc{T-Var}\ \frac{x : T \textit{ is in } \Gamma}{\Gamma \vdash x : T}$$

Typing derivations are a way of utilising typing rules and building upon them in order to obtain a typing judgement for a given term and thus obtaining a typing judgement for a program or a part thereof. Specifically, we construct typing judgements exclusively from the typing rules of the system and some provided context e.g. predefined judgements or other assumptions. This way we can construct a tree-structured derivation that allows us to go from given assumptions to the term typing exclusively via the typing rules of the system. The structure requires each judgement to directly follow from the previous. This means that each leaf of the derivation tree is an assumption or an axiom and the root is the result of the judgement. If we can obtain a typing derivation for a given term, we can conclude that we obtained a *valid judgement* and the term is well typed. The conditions of the term being well typed rely on the assumptions of the derivation and in the case of all leaves being axioms, the term is always well typed in the given type system.

For example, assume our type system consists of the following typing rules.

$$\textsc{T-True}\ \frac{}{\varnothing \vdash \mathit{true} : \mathit{Bool}} \qquad\qquad \textsc{T-Not}\ \frac{\Gamma \vdash x : \mathit{Bool}}{\Gamma \vdash \mathit{not}\ x : \mathit{Bool}}$$

Then we can construct a typing derivation of the term *not true* as follows.

$$\dfrac{\dfrac{}{\varnothing \vdash \mathit{true} : \mathit{Bool}}\ (\text{by T-True})}{\varnothing \vdash \mathit{not}\ \mathit{true} : \mathit{Bool}}\ (\text{by T-Not})$$

Which states that $\varnothing \vdash \mathit{not}\ \mathit{true} : \mathit{Bool}$ is a valid judgement in our system.

2.1.3 Formal Semantics

The type systems described in previous sections define the *static* elements of the system which means that verifying if a program is well typed can be checked statically, e.g. at compile time, via a typechecking algorithm. While these elements are crucial in reasoning about errors that can be prevented by the type system, they are not sufficient to describe the meaning of the program e.g. what to and how it will evaluate when it is being executed. Hence, we need other constructs in order to describe the dynamic elements of the language.

Formal semantics are the construct that defines the meaning, and thus the behaviour, of the system and its programs. Semantics follow different styles and are conventionally split into operational, denotational and axiomatic. Denotational semantics rely on the mathematical foundation of reasoning about objects. Thus, when using this style of semantics, we compose a mapping between the programs of the system and certain mathematical structures. Thus, by reasoning about the mathematical structures we can then reason about the programs. Axiomatic semantics give the programs meaning more directly via proof rules. This means that the program logic is intertwined with the proof of correctness and the program state is dictated by the rules and assertions.

Operational semantics, on the other hand, allow reasoning about the meaning of the programs more abstractly. They formally specify how the program evaluates at run-time and are thus suitable to express evaluation on an abstract machine. In particular, they allow defining rules of the step-by-step evaluation of programs e.g. by reducing expressions until a value is reached. This provides a formal way of reasoning about the run-time behaviour that closely resembles implementation and thus operational semantics are a common method of defining meaning for more practical systems. It is worth noting that some systems may require additional (run-time) typing rules in order to define operational semantics e.g. when its run-time constructs differ from those provided statically.

There are different approaches to defining operational semantics and they can be generally split into small-step and big-step, depending on the technique. Reduction semantics, as introduced by Plotkin (1975), belong to the small-step semantics category. They allow specifying the program in terms of *reduction rules*, which describe the steps we can take to reduce the program. Structural operational semantics, as first introduced by Plotkin (1981), also belong to the small-step semantics category. They allow us to logically separate a program into smaller components, thus allowing us to reason about parts of the run-time instead of the whole program run-time. On the other hand, natural semantics, as introduced by Kahn (1987), belong to the big-step semantics category and are an alternative method of describing operational semantics. Unlike structural operational semantics, which focus on the steps of execution, natural semantics focus on the overall result of execution. While defining natural semantics could be considered easier in comparison to structural semantics, this approach is not suitable for certain programs e.g. those that utilise concurrency.

2.1.4 Properties

As mentioned previously, using a type system allows us to prove the system's behaviour and thus reason about the properties of programs that are typeable in it. An interesting property that we can prove for some systems is type soundness, which states that well-typed programs are well-behaved. If we prove that the soundness theorem for the system holds, we can say that the type system is sound. In the context of languages defined with operational semantics, type soundness can be split into two properties: *subject reduction* and *progress* (Wright and Felleisen 1994). Subject reduction ensures that as expressions are being reduced, they do not lose their property of being well-typed. Progress states that a well-typed program will never reach a state in which no further evaluation is possible i.e. it has to reach a final state or be reducible further.

2.2 Session Types

2.2.1 Key Concepts

Session types are a concurrent communication formalism that was introduced by Honda (1993). They key idea of session types is to allow descriptions of concurrent communication from the standpoint of a given participant by specifying what sequence of actions is allowed and what types of data are exchanged. This is possible via the constructs of receive ($?$), send ($!$), select ($\oplus$), branch ($\&$) and terminate (end). These constructs are sufficient to model different communication protocols, from a simple sequential exchange of information to non-deterministic choices.

Session types are a useful formalism as they provide guarantees of communication safety, privacy and session fidelity. Session fidelity is guaranteed by the fact that a program specified by a session type has to follow the protocol sequentially. This ensures that all communication occurs as expected since the protocol must be followed. Privacy is ensured by every communication taking place in a *session*. This means that whenever two participants want to exchange messages, they get two opposite endpoints of a channel to communicate on and no other participants can interfere with this communication. Communication safety is guaranteed via the concept of *duality*. Session types enforce that communicating participants have to have dual types i.e. when one participants expects to receive a message of type T then the other participants must send a message of type T. Hence, if communication is described by session types, every participant can perform only those actions that can be properly acted upon by the other participant.

2.2.2 Binary and Multiparty Session Types

First we consider binary session types, that is, session types that describe communication between precisely two participants. We can best see how binary protocols can be specified using an example.

Consider the following communication protocol as observed by client **C** who is a user of a smart home controlled by server **S**. When **C** communicates with **S**, they send a $String$ representing a room. Subsequently, they have an option to *select* whether they want to change the *temperature* or see if anything is moving in the room via *checkMotion*. Depending on which option was chosen, the communication proceeds as sending an $Integer$ depicting the desired temperature or respectively receiving a $Boolean$ representing if the room is motionless. Lastly, the communication terminates with end.

$$\begin{aligned}\mathbf{C} \ = \ & !String \oplus \{\ temperature : !Integer.end\ , \\ & \qquad checkMotion : ?Boolean.end\ \}\end{aligned}$$

Naturally, this communication could also be described from the standpoint of **S**. However, as stated previously, in order to ensure that **S**' communication protocol is compatible with **C**'s, we must ensure that the session type is *dual* i.e. $\overline{C} = S$. Hence, **S** communicates as follows:

$$\begin{aligned}\mathbf{S} \ = \ & ?String \,\&\, \{\ temperature : ?Integer.end\ , \\ & \qquad checkMotion : !Boolean.end\ \}\end{aligned}$$

So far, we have discussed *binary* session types, but these are often not sufficient to describe more complex protocols occurring between more than two participants. In order to address this limitation, we can use *multiparty* session types, as introduced by Honda et al. (2016). This formalism allows defining the interaction between multiple *roles* (participants) via a *global type*. The global type can then be projected into *local types*, which describe the communication from the standpoint of each participant.

2.2.3 Typing and Programming Paradigms

It is worth noting that session types often restrict the system, in particular by requiring a linear typing discipline. By linear typing discipline, we mean that only one reference to a channel can be kept at any given time. This restriction emerges from the fact that without linearity, multiple participants could have access to a channel that requires a specific sequence of actions. Thus, we cannot guarantee that the communication occurs as required, e.g. two participants might try to perform the same action simultaneously. An attempt to address this limitation has been made by Vasconcelos (2012) with the introduction of *linear* and *unrestricted* types. This approach splits session typed channels into those that change state and thus require access to be linear and into those that are unrestricted and always allow exactly the same set of actions.

Historically, session types have been mainly introduced into the setting of process calculi, particularly the π-calculus (Dezani-Ciancaglini and de'Liguoro 2009; Hüttel et al. 2016). Still, work has also been done on introducing them into other paradigms. Notably, session types have been introduced into object-oriented (Dezani-Ciancaglini et al. 2005; 2006) and functional programming (Vasconcelos 2009; Gay and Vasconcelos 2010). This approach may be seen as favourable since functional and object-oriented paradigms are a more natural approach for implementations.

2.3 GV

2.3.1 Properties

Functional calculi extended with concurrency constructs have a few advantages over process calculi. In particular, *higher-order functions* and a *high level of abstractions* are better supported for functional languages and are usually not present in process calculi. This fact makes concurrent functional calculi more extendable as we can rely on functional fragments to derive extensions for the communication elements of the language. Another benefit of communication occurring in a functional setting is a separation of the static program and the run-time configuration, which in turn allows precise reasoning about *side effects*. Additionally, as mentioned in the previous section, the functional paradigm may be a more practical approach to describing communication given the abundance of functional programming languages and applications.

One of the most important works on session types in the context of functional algebras is Good Variation (GV). GV is a functional calculus with binary session types, originally introduced by Wadler (2012) but inspired by the work of Gay and Vasconcelos (2010) on LAST. Unlike the previous work on functional calculi with session types, including LAST, GV enjoys a strong correspondence to *classical linear logic*. Wadler achieves this correspondence by closely modelling GV on CP, a session-typed process calculus with strong correspondence to classical linear logic. Subsequently, a translation from GV to CP is presented, which formalises GV's own correspondence to linear logic. This fundamental feature of GV guarantees certain desirable characteristics, namely *deadlock-freedom*, *determinism* and *termination*.

However, as originally presented by Wadler (2012), GV was not as expressive as CP since the translation was only *unidirectional*. This means that not every program typeable in CP was typeable in GV, yet the opposite always held true. Thus, further work has been done on improving the expressivity of GV to make it as expressive as CP with notable works by Lindley and Morris (2014) on Harmonious GV and on GV (Lindley and Morris 2015). Both of these works present translations from GV to CP and from CP to GV. The existence of a *bidirectional* translation means that the expressivity of the functional calculus became the same as that of the process calculus.

It is worth noting that while Harmonious GV and GV presented by Lindley and Morris preserve the desirable traits emerging from correspondence to classical linear logic, i.e. deadlock-freedom, determinism and termination, they allow only *tree-structured processes*. This restriction excludes certain classes of processes, i.e. *good cycles*, meaning that some deadlock-free terminating programs are not typeable in GV. Kokke and Dardha (2021a) address this problem in their work on Priority GV (PGV). By utilising an alternative approach to linear logic in the style of Kobayashi (2006) and Padovani (2014), PGV allows cyclic processes and restores deadlock-freedom with *priorities*. While priorities weaken the correspondence to classical linear logic, PGV enjoys greater expressivity than Lindley and Morris' GV (Lindley and Morris 2014; 2015). A translation from PCP, a priority-based version of CP introduced by Dardha and Gay (2018), is also provided.

2.3.2 Extensions and Implementations

Other GV extensions have been developed either to address problems of previous calculi or to extend them with additional features. In particular, PGV and Hypersequent GV (HGV), HGV being as introduced by Fowler et al. (2021), both fix the problem of configuration equivalence not being type preserving in previous GVs. PGV accomplishes this by using priorities, while

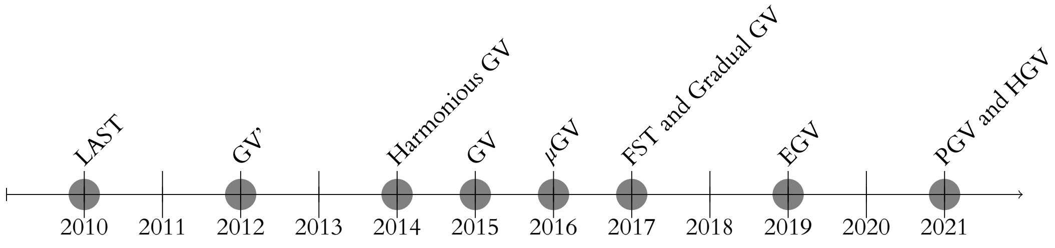

Figure 2.1: *A timeline of selected works related to GV.*

HGV utilises *hyperenvironments* instead and allows only tree-structured processes. The rest of the extensions of the selected GV works focus on introducing additional features into GV. Namely, Lindley and Morris (2016) present μGV with *structural recursion* and FST (Lindley and Morris 2017) with practical features such as *row typing* and *polymorphism*. Igarashi et al. (2017) present Gradual GV with *gradual types*, and Fowler et al. (2019) introduce asynchronous calculus with *exceptions* in work on Exceptional GV (EGV).

It is worth noting that GV-based systems have already been implemented in widely used programming languages, in particular in Haskell[1] (Kokke and Dardha 2021b) and Rust[2]. In fact, GV is the foundation of the only currently available implementations of session types in Rust. The first implementation by Jespersen et al. (2015) is based on LAST (Gay and Vasconcelos 2010). Unfortunately, due to LAST's typing discipline being linear, the implementation has encountered a problem with Rust's affine type system. Thus, Laumann et al.'s implementation guarantees correctness only under the assumption that channels are never dropped. This assumption, however, cannot be guaranteed due to various external factors that affect the program's execution, e.g. connection failures in networked applications; hence session fidelity is violated. The second implementation is Rusty Variation by Kokke (2019) which is a more complete Rust implementation of a session-typed functional calculus. Kokke chose EGV (Fowler et al. 2019) as the basis for the implemented language, which has the advantage of being able to express exceptions. This approach is more adequate for an affine typing discipline since cancellation is embedded into the core of the calculus. Thus, dropped sessions are accounted for and thus handled correctly, which does not violate session fidelity and is, in turn, more realistic.

Figure 2.1 shows a timeline of mentioned works that are related to GV.

2.3.3 Example Program

Since GV has been closely modelled on constructs from CP, writing and understanding GV programs requires getting accustomed to the syntax and constructs. In this section we aim to showcase and explain constructs typical to GV on a simple example. In order to present a simple GV program, we will be using a simplified version of PGV. This is similar to the language Kokke and Dardha (2021a) use to showcase examples with cyclic structures and deadlocks. While this language does not fully reflect any specific GV, it is useful to get familiar with the constructs and the style without the overhead of construct and design choices specific to particular extension features.

The example program consists of a main thread, defined as M, which interacts with another thread, defined as P, in parallel by sending one message. Then, both threads finish the execution by closing the channels they used. In order to understand the syntax of GV, we will now discuss each of the constructs used in the example in more depth.

[1]https://www.haskell.org/
[2]https://rust-lang.org/

```
M =  let (x, y) = new in
     spawn P;                          P =  let ((), y) = recv y in
     let x = send ((), x) in                close y
     close x
```

Figure 2.2: *A simple example of a program in GV syntax describing the communication between participants M and P.*

Firstly, let us consider the constructs that are relevant to threads and channel creation. Creation of channels is straightforward and made possible with the new keyword. This means that new simply returns a pair of channel endpoints that can be used for communication. As for the threads in GV, all parallel threads need to be spawned by other threads, starting from the main thread. This means that the main thread M has to spawn P using the spawn construct, otherwise there would be no parallel thread to interact with. It is worth noting that these two constructs are the core of only PGV and other GV extensions use a $fork$ construct instead. We will not focus on explaining the differences in too much detail since the spawn and new constructs are more applicable in this dissertation. It is worth nothing, however, that both $fork$ or a combination of spawn and new deal with spawning threads and providing communication channels. Thus, spawn P spawns a thread in parallel to the main thread and executed process defined by P on that thread.

In order to make use of spawn, new, and other constructs of the language, we need a way of assigning the values and executing computations. This is achieved using a standard functional concept of let bindings. In particular, let $x = M$ in N can be read as *evaluate the term M and assign the resulting value to x, then evaluate N with the bound x.* This can also be extended to a pair binding which simply lets us deconstruct product types into separate values. Thus, in the case of our example program, let $(x, y) =$ new in ... means that we evaluate new which reduces to two channel endpoints. Then, the channel endpoints are bound to variables x and y and the rest of the program ... is executed with these variables. GV also commonly uses the $M; N$ construct which can be read as *evaluate M and then proceed to evaluating N.* In a linear typing discipline, performing this type of operation is possible only if M is of the unit type. In current GVs, $M; N$ is essentially syntactic sugar for let $() = M$ in N, but as we will see in Chapter 4, this operation can be constructed differently depending on the constructs available in the system. Thus, in our example program spawn $P; \ldots$ allows us to spawn a thread that will execute P and continue as the rest of the program ... which sends a message and terminates.

Lastly, we can take a closer look at the constructs that allow the actual communication between spawned threads i.e. sending a message, receiving a message and closing the channel.

In order to send a message, we need to provide the channel endpoint we want to communicate on and the message itself. Thus, the send constructs takes in an argument that is a pair of the message and the channel endpoint. In our example, we can see that with send$((), x)$ which specifies that we are sending a unit $()$ over the channel endpoint x. Since GV follows linear typing discipline, the channel endpoint is no longer available as soon as it is provided to the send construct. Thus, the construct returns the endpoint after sending on it. This way, we can keep using it even in a linear setting simply by reassigning it using let bindings. In the case of our example, let $x =$ send $((), x)$ in close x specifies that we are sending a unit on x, then rebinding the endpoint to x and executing the rest of the program, in this case close x.

Conversely, receiving a message requires only the channel endpoint in order to communicate, but provides both the message received and the endpoint itself. Again, the endpoint needs to be returned by the construct due to the linear typing discipline. Thus, in our example we can see that recv y performs the operation of receiving a message on the endpoint y. This, in turn, can be used in the let binding in order to bind the returned values to variables. In particular, $\text{let}\ ((), y) = \text{recv}\ y\ \text{in close}\ y$ states that we receive a unit on the channel, rebind the endpoint y and proceed as the rest of the process, in this case $\text{close}\ y$. It is worth noting that we have used the unit $()$ instead of a variable for the sake of simplicity of the example but the message could have also been assigned to an actual variable.

Finally, GV contains constructs that allow terminating a channel endpoint on which communication no longer takes place. This has to be done in order to reduce the channel endpoints to a unit type which is caused, again, by the linear typing discipline which prohibits the channel endpoints from being dropped when they are no longer in use. In fact, all GVs contain two constructs that allow this operation, close and $wait$. They both close channel endpoints but the differ due to the fact that close closes the endpoint after it has been used for sending and $wait$ closes the endpoint after it has been used for receiving. The need for the two separate constructs comes from session types having split end types which in turn comes from the nature of the $fork$ construct. For the sake of our language's simplicity, and to make it more relevant to the system presented in Chapter 4, we are treating the end session type to be self dual and thus we only utilise the close construct. Therefore, in the case of our example, both $\text{close}\ x$ and $\text{close}\ y$ allow us to close the channel endpoints after the message has been sent or received and the channels are no longer in use.

Additionally, GV terms and their syntax that have not been included in the simple example include standard functional terms i.e. lambda abstractions $\lambda x.M$, applications $M\ N$, injections $\text{inl}\ M$ and $\text{inr}\ M$, case sums $\text{case}\ L\ \{\text{inl}\ x \mapsto M;\ \text{inr}\ y \mapsto N\}$. We will not explain these in details as they all have standard behaviour and are not used extensively for the sake of this dissertation.

2.4 Channel Sharing

Channel sharing is the notion of different participants being able to access the same channel endpoint. It is possible to share channels, for example, in an unrestricted typing system simply by distributing the endpoints between different processes. In the case of session-typed communication, however, sharing the endpoints would violate linearity and thus we would lose the guarantees that session types provide.

In this section, we present how allowing channel sharing adds expressivity by providing the ability to reason about the "identity" of endpoints. We also introduce *capabilities*, a technique of introducing channel sharing, as well as alternative approaches.

2.4.1 Added Expressivity

In order to showcase how the expressivity of a system can be expanded by the addition of channel sharing, we present an example of a transaction interaction, a communication flow inspired by the example presented by Voinea et al. (2019). We will be referring to this example as PQB, named after participants involved in the communication. Again, the implementations will be presented using a simplified version of PGV, as it was done in Section 2.3.3. We are also extending the syntax with log and pay constructs that correspond to arbitrary operations that participants may want to perform. While these construct do not change the expressivity of the language, we include them for the sake of clarity of the example and the exact typing rules of those constructs are provided in Section 5.1.

Assume a situation in which an online store, P, is attempting to finalise a transaction with a customer, Q. In order to finalise the transaction, a payment service, B, needs to obtain the required details so that it can process the payment. Specifically, the price of the products sold by the online store and the payment details of the customer need to be provided. After the details are provided

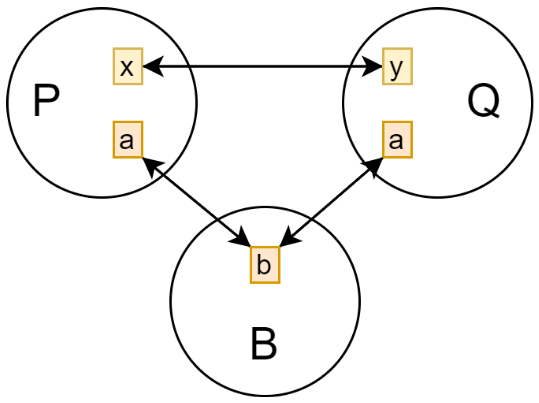

Figure 2.3: The communication between participants in the PQB example. P and Q both use channel a to communicate with B and use channels x and y to communicate with each other.

and the transaction is completed, the payment service provides the outcome of the transaction which is needed by the online store so that they can dispatch the products.

The value of the transaction is held by the store P while the payment details are naturally held by the customer Q. We can assume that the store does not want to trust the customer with providing the transaction value to the payment service B because the customer might have malicious intentions and lower the amount so that they can pay less. The customer, on the other, does not want share their payment details with the store for security reasons and they are only willing to share these with the payment service. Hence, both P and Q want to communicate with B directly i.e. on the opposite endpoint of the channel that B is using. Additionally, P and Q can interact with each other on their own pair of channel endpoints. The diagram of this communication interaction can be seen in Figure 2.3.

Version with compliant participants Initially we can assume all participants are compliant and do not act maliciously in a system without channel sharing. Under these assumptions, an implementation of the PQB scenario in a simplified version of GV is defined in Figure 2.4. In order to ensure session type work as intended, channels are linear i.e. sending and receiving returns a copy of the channel endpoint and each participant is spawned in individual process so that they can interact concurrently.

In this example, all participants interact with each other as intended and without any malicious intent. This means that online store P communicates with B using its channel endpoint a. Then, P passes that endpoint to the customer Q on their separate channel. Then, Q uses the same channel to interact with B and send the payment details. Then, Q returns that endpoint to P and finally P receives the transaction outcome from B and logs it.

Version with malicious participants Now we can assume a system without channel sharing where not all participants are compliant. The implementation of this version of the scenario is defined in Figure 2.5. In this case, store P wants to gain access to payment details of customer Q even though Q does not wish to share these. Thus, P creates a new set of channel endpoints and it will disguise the new channel fa as the actual channel a which Q wishes to use. Specifically, P sends channel fa to Q while holding the opposite endpoint fb. Then, Q communicates on that channel, thinking it is communication with the payment service, and sends their account details. This means that P receives the account details on fb and can now perform a malicious activity. In this case, the details are logged. Then, P continues communicating with B on the real channel a and sends account details itself because Q no longer uses that channel and B still expects the rest of the communication. It is worth noting that while none of this violates linearity of session types, Q ended up communicating with a participant it did not aim to communicate with. Thus,

```
M = let (x, y) = new in          P = let a = send (10, a) in
    let (a, b) = new in              let x = send (a, x) in
    spawn P;                         let (a, x) = recv x in
    spawn Q;                         let (r, a) = recv a in
    spawn B                          log r;
                                     close a; close x
B = let (p, b) = recv b in
    let (n, b) = recv b in       Q = let (a, y) = recv y in
    let r = pay (p, n) in            let a = send (123, a) in
    let b = send (r, b) in           let y = send (a, y) in
    close b                          close y
```

Figure 2.4: *PQB example with all participants interacting properly.*

```
                                 P = let a = send (10, a) in
                                     let (fa, fb) = new in
                                     let x = send (fa, x) in
M = let (x, y) = new in              let (acc, fb) = a in
    let (a, b) = new in              log acc;
    spawn P;                         let (fa, x) = recv x in
    spawn Q;                         close fa; close fb;
    spawn B                          let a = send (acc, a) in
                                     let (r, a) = recv a in
                                     log r;
                                     close a; close x

B = let (p, b) = recv b in       Q = let (a, y) = recv y in
    let (n, b) = recv b in           let a = send (123, a) in
    let r = pay (p, n) in            let y = send (a, y) in
    let b = send (r, b) in           close y
    close b
```

Figure 2.5: *PQB example with a malicious participant P.*

```
M = let (x, y) = new in          P = send (10, a);
    let (a, b) = new in              send ((), x);
    spawn P;                         let () = recv x in
    spawn Q;                         let r = recv a in
    spawn B                          log r;
                                     close a; close x
B = let p = recv b in
    let n = recv b in            Q = let () = recv y in
    let r = pay (p, n) in            send (123, a);
    send (r, b);                     send ((), y);
    close b                          close y
```

Figure 2.6: *PQB example in a system with channel sharing.*

we can see that we could not enforce direct communication between Q and B.

Version with channel sharing Now we can attempt to define this example in a system which allows channel sharing i.e. all participants are aware of and have access to the channels created by the main thread M. The definition of each participant can be found in Figure 2.6. Sharing channels between all participants means that the channel endpoints can no longer be linear so we do not need to keep reassigning them in let bindings. Instead, we treat channels in an unrestricted manner. Hence, both P and Q have direct access to the channel a so the endpoint no longer needs to be passed between them. This way both P and Q are guaranteed to be using the expected channel endpoint and swapping channel endpoints maliciously would not be possible. In this example, avoiding a race condition between P and Q attempting to communicate with B is possible by P and Q synchronising on their own channels x and y. It is crucial to note, however, that session types can be violated in such a system without additional linear constraints. For example, without the synchronising message exchange between P and Q, a race condition would occur with both P and Q attempting to send a message to B.

2.4.2 Capabilities and Alternative Approaches

As we have discussed in the previous section, channel mobility alone is insufficient to describe certain classes of problems. We gave an example of customer Q being unaware that they are communicating with P rather than B which showcases the problem of being unable to reason about the identity of channels that participants wish to use. We also gave an example of a system with channel sharing that allows reasoning about channel identities and ensuring that the participants use the channel they intend to use. Sharing channels in an unrestricted way, however, introduces problems due to the violation of linearity which in turn does not allow the communication to be properly session typed. In this section we discuss techniques of introducing channel sharing while preserving linearity.

Capabilities have been introduced in the context of session types by Voinea et al. (2019). The technique, however, has been originally introduced in works by Crary et al. (1999) and Walker and Morrisett (2000) in the context of region-based memory management. In fact, the technique has been used in the development of the Cyclone[3] programming language (Grossman et al. 2002). Cyclone subsequently became an inspiration for Rust's memory management (Rust 2022) which is renowned for its safety due to the ownership constructs.

They key idea of capabilities for session types is to split every channel into two entities: a channel endpoint itself and a capability of using it. This way, we can make one entity, the channel endpoint, unrestricted and the other entity, the capability, linear. By making the capability linear we are at the same time enforcing the linearity on the whole channel because it cannot be used without the capability. This technique provides a way of reasoning about the identity of the channels that participants are communicating on which in turn means we can impose certain guarantees.

Manifest sharing as presented by Balzer and Pfenning (2017) is a different approach to channel sharing. Rather than splitting channels into a linear entity and an unrestricted entity, the manifest sharing technique relies on acquiring and releasing locks as the session type unravels during the communication. As expected from sharing, added expressivity has introduced deadlocks into the system due to cyclic dependencies of resource management and has been subsequently restricted in the work by Balzer et al. (2019).

The subtle difference between these two approaches, maintaining linearity of use via locks or via a linear entity, leads to interesting results. Namely, manifest sharing allows session types to freely change from linear to unrestricted while capabilities enforce linearity throughout the entire lifetime of the session.

[3]http://cyclone.thelanguage.org/

3 Analysis and Methodology

In this chapter we discuss the process of creating a GV extension which has the expressive power of channel sharing while preserving the guarantees of session types and the convenience of functional programming. In particular, we describe the steps and considerations taken when designing the extension such as introducing new types and constructs and handling unrestricted types. Additionally, we provide reasoning behind the choices made for the extension presented in Chapter 4.

3.1 Problem Analysis

Currently, there is no functional calculus with session types and the guarantees that GV provides, such as determinism and termination, that also allows channel sharing. This means that problems which require channel sharing are not expressible in any of the current GV extensions i.e. they are not valid GV programs. To address this we propose creating an extension of GV which incorporates channel sharing via the introduction of capabilities. Such a system will allow a wider selection of programs to be typeable compared to the base GV while preserving the benefits of the functional calculus, including the practicality of it.

Hence, we define the following as the requirements for a channel sharing GV extension:

- Channel sharing is to be implemented via capabilities.
- The core of the calculus is to be based on current GV systems.
- Most of the GV characteristics are to be preserved where possible.

In particular, implementing channel sharing via capabilities is a requirement due to the nature of the project as we wish to explore the suitability of capabilities for this problem. Hence, every channel endpoint has to be shared in an unrestricted manner while the capability of using it must stay linear. This approach allows preserving linearity of session types, which is a necessity in order to not violate session fidelity, communication safety and privacy.

As previously discussed, we focus on GV because it has session types primitives as well as characteristics such as deadlock-freedom, determinism and termination. Additionally, the functional nature of GV makes it more suitable for implementations compared to e.g. process based approaches. Hence, the extension needs to incorporate core features of GV, which include session types as primitives of the language as well as communication and functional constructs.

Finally, we need to try to preserve GV characteristics. However, we anticipate that in some cases this may not be possible due to the nature of the extension. It is worth noting that while proving the characteristics of the system presented in Chapter 4 is not in the scope of this project, we can still reason about some of the expected outcomes. In particular, we expect to lose deadlock-freedom in order to allow for full expressivity of channel sharing, which we discuss in Section 5.3.

Having considered the overall goals for the extension, specifically for this dissertation, we aim to answer the following research questions:

- Can we introduce capabilities into a GV-based calculus?

- Can we express a set of sharing problems in the extension that we can not express in base GV?
- Can we create the extension in a way that prevents violating the linearity of session types?

Hence, the focus of this work is to investigate the process of constructing a GV extension that allows channel sharing, the changes that need to be made in order to introduce capabilities and to provide a provide directions for further work necessary to finalise the system and prove its properties.

3.2 Extension Design

Based on the requirements discussed, we now present the methodology for creating a channel sharing GV extension. We provide the general steps to be taken to create a GV extension as well as considerations specific to capability-based channel sharing. We also reason about the choices taken when designing the system presented in Chapter 4 and present the challenges that channel sharing introduces to GV.

GV extensions are split into those corresponding to a classical linear logic via a translation (Kokke and Dardha 2021a; Fowler et al. 2021) and those that do not enjoy such tight correspondence (Igarashi et al. 2017; Fowler et al. 2019). Thus, the first necessary step is to recognise if the extension should focus on strictly corresponding to classical linear logic or not. Since channel sharing is useful in cyclic processes, as discussed in Section 2.4, as a consequence we expect deadlocks will be present as well. Thus, we believe it a reasonable choice to forego logical correspondence for this extension and consider restoring it in the future, as discussed in Section 6.1. The next necessary step in creating an extension to GV is identifying core constructs that are present in GV's terms and types. We have identified that all GVs have the following elements in common.

Firstly, communication constructs are required for session typed exchange of messages. This includes encoding session types in the linear type system as well as encoding activities such as sending and receiving, and closing a channel endpoint. Selection and branching are not required due to the functional nature of GV - sum types are fully capable of encoding the constructs of selection and branch. Additionally, all systems need a construct for creating channels and spawning threads. Notably, most GV extensions allow only tree-structured process in order to guarantee deadlock-freedom and due to this fact, the process of creating a channel and spawning threads is combined in one construct - a *fork*. However, PGV presented by Kokke and Dardha (2021a) separates channel creation and thread spawning into separate constructs, *new* and *spawn*, in order to allow cyclic processes, and the *fork* construct can be encoded with the use of the other two constructs. It is worth noting that the *fork* construct requires the *wait* construct to close one of the channel endpoints while the other endpoint in the spawned thread gets closed automatically. The *new* and *spawn* constructs, however, require both the *wait* and *close* constructs that handle closing both endpoints of the channel rather than just one of them.

Secondly, all GVs have terms and types typical to functional languages. These include the previously mentioned sum types as well as injections, the product type for pairs, the unit type, abstractions of linear maps, applications, and let bindings.

Thus, a GV extension that allows channel sharing should share most of these constructs. In particular, the terms and types typical to functional languages do not need significant changes as capabilities should not modify the core of them. The communication constructs, however, require many changes in order to introduce capabilities. Specifically, capabilities require channels to be split into channel endpoints and their capabilities. Thus, the channels will no longer need to be directly of a certain session type. Instead, their endpoint's type will be tracked by a capability and the capability type will contain the session type of the endpoint. This needs to be reflected in the type system by the introduction of a tracked typed analogous to the one introduced by Voinea et al. (2019).

Additionally, channel sharing requires the channel endpoints to be unrestricted. This imposes a challenge in a linear system such as GV since it requires all types to be linear. In order to mitigate this, unrestricted types can be introduced into an otherwise linear typing discipline by the context split operator (Vasconcelos 2012). Alternatively, creating channel aliases could be made possible with explicit copy and discard operations. While both of these approaches are valid, explicit aliases create an additional overhead for the programmers because they have to manage each alias and remember to discard it when it is no longer used. Context split operators, on the other hand, introduce slight complexity into the system and provide less control over the use of each alias by the programmer. Given our focus on practicality and usability, we have decided to opt for the context split approach in the system presented in Chapter 4.

After changes are made to the types, terms and typing discipline, the implementation of the mechanism governing capabilities needs to be considered. Specifically, the system needs a way to track which process owns which capability as well as update them when communication occurs. This is crucial in ensuring linearity of capabilities and thus linearity of session types. We introduce a novel solution to this problem with the use of a flow-sensitive type-and-effect system, similarly to the solution of parametrised monads as presented by Atkey (2009). The key point of this approach is ensuring that each computation has a pre-evaluation and a post-evaluation capability sets associated with it. In this way, we can keep track of required capabilities as well as capabilities produced when evaluating e.g. when sending a message reduces the session type of the channel endpoint used. This approach is particularly suitable for capabilities since they are distributed between processes and stay with them until they are explicitly passed to another process. An alternative approach would be incorporating capabilities into the type system alone, as originally done by Walker and Morrisett (2000). This solution, however, imposes additional overhead to the programmers as it would require the capability to be explicitly used whenever the channel endpoint is used. Thus, for the sake of simplicity of use, in Chapter 4 we have proceeded with the type-and-effect system at the cost of added complexity of the system.

4 Implementation: Capable GV

In this chapter we present our approach to creating Capable GV (CGV) - a GV extension that introduces channel sharing via capabilities. In order to demonstrate how capabilities can be used to share channels while preserving linearity of communication, we present CGV's type system i.e. term syntax, types and static typing rules which allow us to showcase what kind of programs should be, and are, typeable in CGV.

For the sake of clarity and consistency, terms are typeset in red, capabilities are typeset in purple and types are typeset in blue throughout the entire work.

4.1 Terms and Types

In order to define syntax and types for CGV, we have investigated the constructs used in current GVs and the reasoning behind them. We have predominantly focused on work by Lindley and Morris (2015) since it can be considered a base version of GV and on work by Kokke and Dardha (2021a) since it is the only GV extension that does not utilise the $fork$ construct. We have also analysed which constructs are required to introduce capability-based sharing by investigating the design choices of the work done by Voinea et al. (2019). We present the results of our work in this section by introducing the syntax used to write CGV programs as well as the base types of CGV.

Terms While CGV inherits most of its term syntax directly from PGV, we have stratified the terms into values V, W and computations L, M, N. This decision has been made due to CGV implementing a flow-sensitive type-and-effect system which requires such a separation. Additionally, two new terms have been added in order to be able to share channels by passing capabilities.

Thus, CGV's values consist of a standard unit value $()$, a value x, an abstraction $\lambda x.M$, a pair of values (V, W) as well as left and right injections $\mathrm{inl}\ V$ and $\mathrm{inr}\ V$. These are all standard terms for functional calculi.

CGV's calculations are slightly more complex and consist of an application $V\ W$, value return $\mathrm{return}\ V$, case sum $\mathrm{case}\ L\ \{\mathrm{inl}\ x \mapsto M;\ \mathrm{inr}\ y \mapsto N\}$ and three different types of let bindings. We require all three because they perform different functionalities. The term $\mathrm{let}\ x = M\ \mathrm{in}\ N$ evaluates computation M, assigns it to x and evaluates to N. The term $\mathrm{let}\ (x, y) = V\ \mathrm{in}\ M$ allows deconstructing a pair of values V into two separate values x and y and subsequently evaluating M. The term $\mathrm{let}\ () = V\ \mathrm{in}\ M$ evaluates a unit value V while reducing it and subsequently evaluates to M. This let unit construct is necessary in order to discard terms of the unit type in a linear typing system.

Additionally, CGV utilises standard communication primitives that were slightly modified from the ones used by PGV. The term new creates two opposite channel endpoints. Given that V is a pair of values, the term $\mathrm{send}\ V$, sends the first element of the pair on the channel which is specified by the second element of the pair. The term $\mathrm{recv}\ V$ receives a value on channel endpoint specified by V. The term $\mathrm{close}\ V$ closes a channel endpoint V provided that its session type is end and we have the capability of using it. The term $\mathsf{spawn}\ M$ spawns the process specified by M in parallel i.e. it creates a thread on which M is executed.

The terms $\mathsf{act}\ V$ and $\mathsf{inact}\ V$ are the only terms unique to CGV. The term $\mathsf{inact}\ V$ inactivates the channel endpoint specified by V which means that it *disables* the capability of using that endpoint and returns its inactivated form. The term $\mathsf{act}\ V$ acts in an opposite manner so it takes the disabled capability of the inactivated endpoint and activates it by obtaining the capability to use that endpoint. Specifically, when evaluated, $\mathsf{inact}\ V$ removes the capability of using V from the pre-evaluation capability set and returns it while $\mathsf{act}\ V$ puts the capability stored in V into the post-evaluation capability set.

It is worth noting that the standard syntactic sugar $M; N$ which stands for *evaluate M and reduce it to unit, then proceed as N* can be denoted as $\mathsf{let}\ x = M\ \mathsf{in}\ \mathsf{let}\ () = x\ \mathsf{in}\ N$.

The syntax of CGV is thus defined by the following grammar:

$$\begin{array}{lcl}
V, W & ::= & () \mid x \mid \lambda x.M \mid (V, W) \mid \mathsf{inl}\ V \mid \mathsf{inr}\ V \\[1em]
L, M, N & ::= & V\ W \mid \mathsf{return}\ V \mid \mathsf{let}\ x = M\ \mathsf{in}\ N \\
& \mid & \mathsf{let}\ (x, y) = V\ \mathsf{in}\ M \mid \mathsf{let}\ () = V\ \mathsf{in}\ M \\
& \mid & \mathsf{case}\ L\ \{\mathsf{inl}\ x \mapsto M;\ \mathsf{inr}\ y \mapsto N\} \\
& \mid & \mathsf{new} \mid \mathsf{send}\ V \mid \mathsf{recv}\ V \mid \mathsf{close}\ V \\
& \mid & \mathsf{inact}\ V \mid \mathsf{act}\ V \mid \mathsf{spawn}\ M
\end{array}$$

Session types. CGV contains only basic communication constructs in order to be able to express session types i.e. input, output and self-dual end. Thus session types of CGV are defined as follows:

$$S \quad ::= \quad !T.S \mid ?T.S \mid \mathsf{end}$$

It is worth noting that this approach resembles the approach of other GVs - in GV systems, branch and select constructs are not necessary for session types. This is caused by the fact that the sum type present in functional calculi can be used to implement the same functionality. Hence, this approach differs from the process-calculus based applications of session types which typically require input, output, branch, select and end as seen in Voinea et al. (2019).

CGV, however, abandons the convention of using split ends that is present in other GV systems. Instead, we used self-dual end since it can encode the very same functionality while reducing the complexity of the system. This decision was additionally motivated by the fact that CGV is moving away from the classical linear logic correspondence and will not rely on the $fork$ construct to spawn processes in parallel due to the nature of the extension. Thus, preserving the convention may be seen as unnecessary while self-dual end is a convenient simplification.

Duality. The duality relation of session types is defined in a standard way i.e. input is dual to output and end is self-dual. As a consequence, applying duality twice is a neutral operation i.e. a dual of a dual of a session type is just that session type.

$$\overline{!T.S} = ?T.\overline{S} \qquad \overline{?T.S} = !T.\overline{S} \qquad \overline{\mathsf{end}} = \mathsf{end} \qquad \overline{\overline{S}} = S$$

Types. While session types are required in CGV to preserve the properties that they bring, CGV terms cannot be directly of that type as this would risk violating linearity of session types when channels become unrestricted. Instead, the types in CGV are a combination of standard functional calculus types and capability-based types adapted from Voinea et al. (2019).

In particular, introducing capabilities into CGV is made possible by splitting channels into a channel endpoint $tr(\rho)$ and the capability of using it $\rho(S)$. These two types can be read as *channel*

$$\frac{}{\varnothing = \varnothing \circ \varnothing} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad T = tr(\rho)}{\Gamma, x : T = (\Gamma_1, x : T) \circ (\Gamma_2, x : T)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : T = (\Gamma_1, x : T) \circ \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : T = \Gamma_1 \circ (\Gamma_2, x : T)}$$

Figure 4.1: *Context split definition.*

endpoint of type $tr(\rho)$ is tracked by capability ρ and *capability of type $\rho(S)$ manages an endpoint that follows session type S*. This way, the channel endpoint no longer stores information about its session type. Instead, the endpoint references its capability and the capability itself holds the information about the session type. As the channel endpoint no longer is tied to a session type, making it unrestricted is safe and does not violate the linearity that is necessary for session types.

Similarly to session types, capabilities are not a type that can be directly used by CGV terms. Instead, we introduce an inactive capability type $[\rho(S)]$. This distinction allows a visual separation of an inactive channel that cannot be used for communication and an active channel which requires a capability present in a capability set. The specifics of capability sets and CGV's type and effect system are discussed below. While denoting inactive capabilities with square brackets is not necessary, we believe it is a useful visual indication of the way capabilities work. In particular, active capabilities exist in capability sets that make up for the CGV's effect system while inactive capabilities exist at a type level and can subsequently be sent over channels.

Other types include standard functional types i.e. the product of two types $T \times U$, the sum of two types $T + U$ and the unit type 1. Notably, the linear function type $T\ (C_1) \multimap (C_2)\ U$ is unique to CGV. It specifies pre- and post-evaluation capability sets of the function body as well as the mapping of an argument to an outcome. As we will see in Section 4.2, these capability sets are crucial when evaluating function application.

$$\begin{array}{rcl} T, U & ::= & T \times U \mid T + U \mid tr(\rho) \mid [\rho(S)] \\ & \mid & 1 \mid T\ (C) \multimap (C')\ U \end{array}$$

Environments. There are three entities necessary for making typing judgements on CGV terms: *value context* Γ, *capability context* Δ and *capability set* C. All of these are constructed recursively by building upon an empty construct $\varnothing$.

$$\begin{array}{lrcl} \text{Value contexts} & \Gamma & ::= & \varnothing \mid \Gamma, x : T \\ \text{Capability contexts} & \Delta & ::= & \varnothing \mid \Delta, \rho \\ \text{Capability sets} & C & ::= & \varnothing \mid C \otimes \rho(S) \end{array}$$

Value contexts are standard environments for which variables are associated with types. For example, $\Gamma = \varnothing, x : T$ states that context Γ contains an assumption that term x is of type T in the scope. Extending context Γ with $\Gamma, x : T$ is possible only if there are no occurrences of x in Γ. Additionally to the standard linear context combination Γ_1, Γ_2 which denotes the combination of context with disjoint domains, we introduce the context split operator $\circ$. This means that the typing context can be split into $\Gamma_1 \circ \Gamma_2$ according to the rules specified in Figure 4.1 which allows channel endpoints to be unrestricted while everything else remains linear. It is worth noting that although the context split rules are similar to those of Vasconcelos (2012), the unrestricted type of channel endpoints can also be treated linearly if needed. This change allows us to treat the remaining system fully linearly.

Capability contexts are sets of labels which are used to keep track of capabilities in scope. For example, $\Delta = \varnothing, \rho$ states that context Δ contains an assumption that capability ρ is in the scope. Extending context Δ with Δ, ρ is possible only if there are no occurrences of ρ in Δ. The standard linear context combination Δ_1, Δ_2 is defined and denotes the combination of contexts with disjoint domains.

Capability sets are sets of capabilities present in the given scope *i.e.*, on the thread. For example, $C = \varnothing \otimes \rho(S)$ states that capability $\rho(S)$ is in the capability set C. This means that the channel endpoint of type $tr(\rho)$ follows the protocol of session type S and is controlled by the capability in C. Capability sets can either be pre-evaluation to denote which capabilities need to be present prior to computation evaluation or post-evaluation to denote what capabilities the computation produces. Individual capabilities or capability sets can be combined into one capability sets using the $\otimes$ operator. This operator is consistent with other works on capabilities in the literature but it functions analogously to the combination operators for the value and capability contexts since capability set domains will always be disjoint sets.

Typing judgements Typing judgements for values are of the form $\Gamma; \Delta \vdash V : T$ which states that *under typing environment* Γ *and capability environment* Δ*, term* V *is of type* T.

Typing judgements for computations are of the form $\Gamma; \Delta; C \vdash M : T \triangleright C'$ which states that *under typing environment* Γ *and capability environment* Δ *and with capability set* C*, term* M *is of type* T *and produces capability set* C'. For the sake of fitting longer typing judgements, a three-line version is typeset as $\begin{pmatrix} \Gamma; \Delta; \\ C \triangleright C' \\ \vdash M : T \end{pmatrix}$ which puts environments on the first line, capability sets on the second and the term typing on the last line. Since the effect system of CGV is done via capabilities, they are only necessary when evaluating computations. Values on their own should not need nor modify capabilities and are thus types without any effects in place. The special case includes functions which have type $T\ (C) \multimap (C')\ U$ which, as mentioned before, denotes the pre- and post-evaluation capability sets for the function body.

4.2 Typing Rules

In order to reason about the static elements of CGV, we present static typing rules in Figure 4.2. Most of the typing rules are standard and as expected for a GV extension, especially in the case of typing rules for values.

4.2.1 Rules for Values

T-Var states that a value x is of type T given that the value context is $x : T$ and capability context is Δ. It is worth noting here that the value context requirement is necessary in order to ensure a linear typing system while capability context can be arbitrary since it does not affect the majority of terms in CGV. T-Unit states that unit is well typed under empty value context and, similarly to T-Var, an arbitrary capability context. Typing rules for injections T-Inl and T-Inr state that they are well typed under the same contexts as values that are being injected. T-Pair states that a pair of values is well types under combined value and capability contexts where combination of value contexts is specified by the context split operator. T-Lam is the most interesting rule for values as it “stores away” pre- and post-evaluation capability sets in the the of the function. It also removes the bound variable from the value context in a standard manner.

Values.

$$\textsc{T-Var}\quad \frac{}{x : T; \Delta \vdash x : T} \qquad \textsc{T-Unit}\quad \frac{}{\varnothing; \Delta \vdash () : 1} \qquad \textsc{T-Inl}\quad \frac{\Gamma; \Delta \vdash V : T}{\Gamma; \Delta \vdash \mathsf{inl}\ V : T + U} \qquad \textsc{T-Inr}\quad \frac{\Gamma; \Delta \vdash V : U}{\Gamma; \Delta \vdash \mathsf{inr}\ V : T + U}$$

$$\textsc{T-Lam}\quad \frac{\Gamma, x : T; \Delta; C \vdash M : U \triangleright C'}{\Gamma; \Delta \vdash \lambda x.M : T\ (C) \multimap (C')\ U} \qquad \textsc{T-PairVal}\quad \frac{\Gamma_1; \Delta_1 \vdash V : T \qquad \Gamma_2; \Delta_2 \vdash W : U}{\Gamma_1 \circ \Gamma_2; \Delta_1, \Delta_2 \vdash (V, W) : T \times U}$$

Computations.

$$\textsc{T-App}\quad \frac{\Gamma_1; \Delta_1 \vdash V : T\ (C) \multimap (C')\ U \qquad \Gamma_2; \Delta_2 \vdash W : T}{\Gamma_1 \circ \Gamma_2; \Delta_1, \Delta_2; C \vdash V\ W : U \triangleright C'} \qquad \textsc{T-Return}\quad \frac{\Gamma; \Delta \vdash V : T}{\Gamma; \Delta; C \vdash \mathsf{return}\ V : T \triangleright C}$$

$$\textsc{T-LetUnit}\quad \frac{\Gamma_1; \Delta_1 \vdash V : 1 \qquad \Gamma_2; \Delta_2; C \vdash M : T \triangleright C'}{\Gamma_1 \circ \Gamma_2; \Delta_1, \Delta_2; C \vdash \mathsf{let}\ () = V\ \mathsf{in}\ M : T \triangleright C'}$$

$$\textsc{T-LetPair}\quad \frac{\Gamma_1; \Delta_1 \vdash V : T \times T' \qquad \Gamma_2, x : T, y : T'; \Delta_2; C \vdash M : U \triangleright C'}{\Gamma_1 \circ \Gamma_2; \Delta_1, \Delta_2; C \vdash \mathsf{let}\ (x, y) = V\ \mathsf{in}\ M : U \triangleright C'}$$

$$\textsc{T-LetBind}\quad \frac{\Gamma_1; \Delta_1; C \vdash M : T \triangleright C' \qquad \Gamma_2, x : T; \Delta_2; C' \vdash N : U \triangleright C''}{\Gamma_1 \circ \Gamma_2; \Delta_1, \Delta_2; C \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N : U \triangleright C''} \qquad \textsc{T-Close}\quad \frac{\Gamma; \Delta \vdash V : tr(\rho)}{\Gamma; \Delta; C \otimes \rho(\mathsf{end}) \vdash \mathsf{close}\ V : 1 \triangleright C}$$

$$\textsc{T-CaseSum}\quad \frac{\Gamma_1; \Delta_1 \vdash L : T + T' \qquad \Gamma_2, x : T; \Delta_2; C \vdash M : U \triangleright C' \qquad \Gamma_2, y : T'; \Delta_2; C \vdash N : U \triangleright C'}{\Gamma_1 \circ \Gamma_2; \Delta_1, \Delta_2; C \vdash \mathsf{case}\ L\ \{\mathsf{inl}\ x \mapsto M;\ \mathsf{inr}\ y \mapsto N\} : U \triangleright C'}$$

$$\textsc{T-New}\quad \frac{}{\varnothing; \Delta, \rho^+, \rho^-; C \vdash \mathsf{new} : tr(\rho^+) \times tr(\rho^-) \triangleright C \otimes \rho^+(S) \otimes \rho^-(\overline{S})} \qquad \textsc{T-Spawn}\quad \frac{\Gamma; \Delta; C_s \vdash M : 1 \triangleright \varnothing}{\Gamma; \Delta; C \otimes C_s \vdash \mathsf{spawn}\ M : 1 \triangleright C}$$

$$\textsc{T-Send}\quad \frac{\Gamma; \Delta \vdash V : T \times tr(\rho)}{\Gamma; \Delta; C \otimes \rho(!T.S) \vdash \mathsf{send}\ V : 1 \triangleright C \otimes \rho(S)} \qquad \textsc{T-Recv}\quad \frac{\Gamma; \Delta \vdash V : tr(\rho)}{\Gamma; \Delta; C \otimes \rho(?T.S) \vdash \mathsf{recv}\ V : T \triangleright C \otimes \rho(S)}$$

$$\textsc{T-Inact}\quad \frac{\Gamma; \Delta \vdash V : tr(\rho)}{\Gamma; \Delta; C \otimes \rho(S) \vdash \mathsf{inact}\ V : [\rho(S)] \triangleright C} \qquad \textsc{T-Act}\quad \frac{\Gamma; \Delta \vdash V : [\rho(S)]}{\Gamma; \Delta; C \vdash \mathsf{act}\ V : 1 \triangleright C \otimes \rho(S)}$$

Figure 4.2: *Static Typing Rules for CGV.*

4.2.2 Rules for Standard Computations

As for the typing rules for computations, we have to pay close attention to capability sets as they are crucial in CGV's flow-sensitive type-and-effect system. T-App states that capability sets can be extracted from the functon type and used when evaluating an application. T-Return states that returning a value does not modify present capabilities i.e. pre- and post-evaluation capability sets are the same. T-LetUnit states that this type of let binding reduces the value V and then evaluates M so it requires and produces the same capabilities as M. T-LetPair is analogous to T-LetUnit when it comes to the treatment of capabilities. T-LetBind is the most interesting type of the let binding rule as it shows how the capability sets can be chained in order to execute computations sequentially. It is worth noting that M requires capability set C and produces set C' while N needs to require capability set C' and produces set C''. Hence, we can see how the capabilities are chained from C to C' to C'' which results in the let binding requiring capability set C and producing capability set C'' when all parts are evaluated. T-CaseSum states that the case sum requires and provides the same capabilities as the terms M and N without any modifications.

4.2.3 Rules for Communication-Centric Computations

Finally, we can observe the core functionality of capabilities by taking a look at the communication-centric typing rules. T-Send and T-Recv describe how the communication protocols, as defined by session types, unravel when communication takes place. In particular, we can see that in order to send a message of type T on a channel which is tracked by capability ρ, we must first have the capability of using it i.e. it must be present in the pre-evaluaiton capability set. Since we want to send a message on that channel, the capability must hold the session type $!T.S$. Intuitively, after the communication takes place the session type should be just S which we can see being reflected in the post-evaluation capability set. This also shows that a capability stays with the process until it is explicitly sent or given up. Dually, T-Recv states than in order to receive on a channel we need to have the capability to use that channel and the session type needs to allow receiving. Capability sets again reflect the way session types progress as the communication takes place. It is worth noting that, unlike in other GVs, the send and receive primitives no longer return channel endpoint copies. Instead, all channel endpoints are unrestricted so send V can return a unit type and recv V can simply return the received value. This is a simplification when writing CGV programs since we no longer need to keep reassigning channel endpoints. T-Close states how we can terminate a channel endpoint i.e. it is possible to do so only if we have the capability to use that channel and the protocol of the channel has reached an end.

A less explicit use of capabilities can be seen in T-Spawn. This rule states that if a term to be spawned in parallel M requires capability set C_s, the process that is spawning M has to have these capabilities in the pre-evaluation capability set and they will be passed to the spawned process i.e. they will not be present in the post-evaluation capability set of the spawning process. Other than removing a subset of capabilities for the sake of a spawned process, the capability sets should not change. The creation of channels, and thus the capabilities of using them, is possible with the new construct. T-New states that new returns the opposite channel endpoints while also providing the capability of using them. These capabilities can then be used and distributed to other threads. It is crucial to note here that the capability context has to contain the capabilities that are being created. This ensures that no capability is generated twice as that would violate linearity of each channel endpoint. This guarantee is made possible as every single term combines capability contexts which means that every CGV programs has to be well typed under an arbitrary capability context. While the specifics do not matter significantly, it is an important check in ensuring that each capability ρ is in fact unique and thus each type $tr(\rho)$ is unique as well.

Outside of creating capabilities and passing them to new threads, each process should be able to pass an owned capability to a different process, thus allowing channel sharing. That inactive capability should be then received and reactivated in order to use the channel again. This exchange is described via rules T-Inac and T-Act which state how capabilities can be removed from re-evaluation capability sets or re-added to post-evaluation capability sets.

4.2.4 General Considerations

In this section we discuss additional considerations that are important to note about CGV's static typing system i.e. context combination, conditions required for a CGV program to be well-typed and a typing rule simplification.

Firstly, for all typing rules, the value contexts are combined analogously to how they would be combined in any other linear functional calculus except with a different operator. A capability context functions similarly but with an operator that does not allow unrestricted names. As stated before, this allows ensuring that any capability is created at most once.

Hence, a CGV program M is well typed if we can type derive it to the state of $\varnothing; \Delta; \varnothing \vdash M : T \triangleright \varnothing$ where Δ and T are completely arbitrary.

Secondly, in addition to the typing rules stated in Figure 4.2, we can define an additional typing rule for $M; N$ that will simplify typing derivations. It can be defined as follows:

$$\textsc{T-LetShort}\quad \frac{\Gamma_1; \Delta_1; C \vdash M : 1 \triangleright C' \qquad \Gamma_2; \Delta_2; C' \vdash N : T \triangleright C''}{\Gamma_1 \circ \Gamma_2; \Delta_1, \Delta_2; C \vdash M; N : T \triangleright C''}$$

Since $M; N$ is syntactic sugar for $\mathsf{let}\ x = M\ \mathsf{in}\ \mathsf{let}\ () = x\ \mathsf{in}\ N$, we can derive a typing rule for it directly from T-LetUnit and T-LetBind. While this rule does not add any merit to the system, it follows directly from the other two rules, so it's just a "syntactic sugar" rule.

5 Evaluation

In this chapter, we evaluate CGV, as presented in Chapter 4, by creating case studies that show how channel sharing and linearity enforcement work in the language. We verify CGV by performing typing derivations of these programs and verifying that the outcomes conform to our expectations i.e. that the channel sharing example is well-typed and the example that violates linearity is not well-typed. Lastly, we provide a discussion of the expected properties of CGV in terms of progress and subject reduction.

5.1 Case Study: Transaction Example

In order to showcase the functionality of CGV, we present an example that models a simple transaction interaction. In particular, PQB, as previously described in Section 2.4, is suitable to demonstrate sharing. To restate, assume a situation in which an online store, P, and a customer, Q want to finalise a transaction. In order to finalise this transaction, the price of the products and the account details of the customer need to be passed to a payment service, B. Due to the nature of the transaction, the store needs to send the price to the service, as it does not trust the customer with not lowering the price, and it needs to receive a confirmation from the service that the transaction has gone through. The customer, on the other hand, does not trust the store and wants to share the account details directly with the service.

Hence, the customer and the store will share the channel endpoint responsible for communicating with the payment service. In particular, the service requires the payment amount to be sent first and the account details to be sent second. Then, it responds with the outcome of the transaction which is needed for the store to process the order.

For the sake of this example, the payment amount, the bank details as well as the transaction outcome will all be integers. Thus, we will be introducing integer primitives into CGV as well as the following two constructs: log for logging data and pay for performing the payment transaction. While these are not necessary for the system to showcase sharing, it makes the example less artificial and more relevant to the real-world based case study described above. Thus, the following typing rules are added for the constructs that the processes use:

$$\frac{\Gamma; \Delta \vdash V : \mathit{Int} \times \mathit{Int}}{\Gamma; \Delta; C \vdash \mathrm{pay}\ V : \mathit{Int} \triangleright C}\ \textsc{T-Pay} \qquad \frac{\Gamma; \Delta \vdash V : \mathit{Int}}{\Gamma; \Delta; C \vdash \mathrm{log}\ V : 1 \triangleright C}\ \textsc{T-Log}$$

$$\frac{x \text{ is an integer}}{\varnothing; \Delta; C \vdash x : \mathit{Int} \triangleright C}\ \textsc{T-Int}$$

The example in Figure 5.1 shows the implementation of PQB in CGV.

By performing a typing derivation on the program, as shown in sections 5.1.1, 5.1.2, 5.1.3 and 5.1.4, we can see that PQB is indeed well typed in CGV. While this example does not prove that every program that requires sharing is typeable in CGV, it showcases an expressivity advantage of CGV over systems which do not allow unrestricted channels.

```
M =  let t = new in              P =  send (10, a);
     let (x, y) = t in                let t = inact a in
     let t = new in                   send (t, x);
     let (a, b) = t in                let t = recv x in
     spawn P;                         act t;
     spawn Q;                         let r = recv a in
     spawn B                          log r;
                                      close a; close x
B =  let p = recv b in           Q =  let t = recv y in
     let n = recv b in                act t;
     let r = pay (p, n) in            send (123, a);
     send (r, b);                     let t = inact a in
     close b                          send (t, y);
                                      close y
```

Figure 5.1: *Example of channel sharing in the transaction example.*

5.1.1 Typing Derivation for P

Using the typing rules presented before and the definition of process P as provided in Figure 5.1, we present the typing derivation for this part of the example CGV program. In order to follow the derivation gradually and be able to present it visually, the process has been split into several parts and thus typing derivations build upon each other. Using this approach, we can show that:

$$
\mathrm{D}_1\quad
\dfrac{
\dfrac{\dfrac{}{r : \mathit{Int}; \varnothing \vdash r : \mathit{Int}}}{r : \mathit{Int}; \varnothing; \rho_1^+(\text{end}) \otimes \rho_2^+(\text{end}) \vdash \text{log } r : 1 \triangleright \rho_1^+(\text{end}) \otimes \rho_2^+(\text{end})}
\;\vdots
\qquad
\dfrac{
\dfrac{\dfrac{}{a : tr(\rho_2^+); \varnothing \vdash a : tr(\rho_2^+)}}{a : tr(\rho_2^+); \varnothing; \rho_1^+(\text{end}) \otimes \rho_2^+(\text{end}) \vdash \text{close } a : 1 \triangleright \rho_1^+(\text{end})}
\quad
\dfrac{\dfrac{}{x : tr(\rho_1^+); \varnothing \vdash x : tr(\rho_1^+)}}{x : tr(\rho_1^+); \varnothing; \rho_1^+(\text{end}) \vdash \text{close } x : 1 \triangleright \varnothing}\;\vdots
}{a : tr(\rho_2^+), x : tr(\rho_1^+); \varnothing; \rho_1^+(\text{end}) \otimes \rho_2^+(\text{end}) \vdash \text{close } a; \text{close } x : 1 \triangleright \varnothing}
}{r : \mathit{Int}, a : tr(\rho_2^+), x : tr(\rho_1^+); \varnothing; \rho_1^+(\text{end}) \otimes \rho_2^+(\text{end}) \vdash \begin{pmatrix} \text{log } r; \\ \text{close } a; \text{close } x \end{pmatrix} : 1 \triangleright \varnothing}
$$

Rules TVar, TClose, TLog and TLetShort have been used in order to type derive a part of the process. It is worth noting how TClose removes the channel endpoints' capabilities e.g. in $x : tr(\rho_1^+); \varnothing; \rho_1^+(\text{end}) \vdash \text{close } x : 1 \triangleright \varnothing$ or how TLetShort chains the capability sets meaning that the post-evaluation capability set of log r has to be the pre-evaluation capability set of close a; close x. Then, using the derivation D_1 from above we derive:

$$
D_2\ \dfrac{\begin{matrix}\dfrac{\dfrac{}{t : [\rho_2^+(?\mathit{Int}.\mathrm{end})]; \varnothing \vdash t : [\rho_2^+(?\mathit{Int}.\mathrm{end})]}}{t : [\rho_2^+(?\mathit{Int}.\mathrm{end})]; \varnothing; \rho_1^+(\mathrm{end}) \vdash \mathrm{act}\ t : 1 \triangleright \rho_1^+(\mathrm{end}) \otimes \rho_2^+(?\mathit{Int}.\mathrm{end})} \\ \vdots \end{matrix} \quad \dfrac{\dfrac{\dfrac{}{a : tr(\rho_2^+); \varnothing \vdash a : tr(\rho_2^+)}}{a : tr(\rho_2^+); \varnothing; \rho_1^+(\mathrm{end}) \otimes \rho_2^+(?\mathit{Int}.\mathrm{end}) \vdash \mathrm{recv}\ a : \mathit{Int} \triangleright \rho_1^+(\mathrm{end}) \otimes \rho_2^+(\mathrm{end})} \quad D_1}{a : tr(\rho_2^+), x : tr(\rho_1^+); \varnothing; \rho_1^+(\mathrm{end}) \otimes \rho_2^+(?\mathit{Int}.\mathrm{end}) \vdash \begin{pmatrix} \mathrm{let}\ r = \mathrm{recv}\ a\ \mathrm{in} \\ \mathrm{log}\ r; \\ \mathrm{close}\ a; \mathrm{close}\ x \end{pmatrix} : 1 \triangleright \varnothing}}{t : [\rho_2^+(?\mathit{Int}.\mathrm{end})], a : tr(\rho_2^+), x : tr(\rho_1^+); \varnothing; \rho_1^+(\mathrm{end}) \vdash \begin{pmatrix} \mathrm{act}\ t; \\ \mathrm{let}\ r = \mathrm{recv}\ a\ \mathrm{in} \\ \mathrm{log}\ r; \\ \mathrm{close}\ a; \mathrm{close}\ x \end{pmatrix} : 1 \triangleright \varnothing}
$$

In this case, rules TVar, TLetBind, TLetShort, TRecv and TAct have been used to type derive a further part of the process. It is worth noting how TAct activates the channel endpoint a placing the capability of using it in the post-evaluation capability set with $t : [\rho_2^+(?\mathit{Int}.\mathrm{end})]; \varnothing; \rho_1^+(\mathrm{end}) \vdash \mathrm{act}\ t : 1 \triangleright \rho_1^+(\mathrm{end}) \otimes \rho_2^+(?\mathit{Int}.\mathrm{end})$. Thus, it is possible to use a again e.g. in $\mathrm{recv}\ a$. It is also worth noting how TRecv changes the session type that the capability holds. Then, using the derivation D_2 from above we derive:

$$
D_3\ \dfrac{\dfrac{\dfrac{}{x : tr(\rho_1^+); \varnothing \vdash x : tr(\rho_1^+)}}{x : tr(\rho_1^+); \varnothing; \rho_1^+(?[\rho_2^+(?\mathit{Int}.\mathrm{end})].\mathrm{end}) \vdash \mathrm{recv}\ x : [\rho_2^+(?\mathit{Int}.\mathrm{end})] \triangleright \rho_1^+(\mathrm{end})} \quad D_2}{a : tr(\rho_2^+), x : tr(\rho_1^+); \varnothing; \rho_1^+(?[\rho_2^+(?\mathit{Int}.\mathrm{end})].\mathrm{end}) \vdash \begin{pmatrix} \mathrm{let}\ t = \mathrm{recv}\ x\ \mathrm{in} \\ \mathrm{act}\ t; \\ \mathrm{let}\ r = \mathrm{recv}\ a\ \mathrm{in} \\ \mathrm{log}\ r; \\ \mathrm{close}\ a; \mathrm{close}\ x \end{pmatrix} : 1 \triangleright \varnothing}
$$

In this case, rules TVar, TRecv and TLetBind have again been used to type derive a further part of the process. Then, using the derivation D_3 from above we derive:

$$
\mathrm{D}_4\;
\dfrac{
\dfrac{
\dfrac{}{t : [\rho_2^+(!Int.?Int.\mathrm{end})]; \varnothing \vdash t : [\rho_2^+(!Int.?Int.\mathrm{end})]}
\qquad
\dfrac{}{x : tr(\rho_1^+); \varnothing \vdash x : tr(\rho_1^+)}
}{
x : tr(\rho_1^+), t : [\rho_2^+(!Int.?Int.\mathrm{end})]; \varnothing \vdash (t, x) : [\rho_2^+(!Int.?Int.\mathrm{end})] \times tr(\rho_1^+)
}
}{
\left(\begin{array}{c}
x : tr(\rho_1^+), t : [\rho_2^+(!Int.?Int.\mathrm{end})]; \varnothing; \\
\rho_1^+(![\rho_2^+(!Int.?Int.\mathrm{end})].?[\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \triangleright \rho_1^+(?[\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \\
\vdash \mathrm{send}\ (t, x) : 1
\end{array}\right)
}
\qquad \mathrm{D}_3
\over
\left(\begin{array}{c}
a : tr(\rho_2^+), x : tr(\rho_1^+), t : [\rho_2^+(!Int.?Int.\mathrm{end})]; \varnothing; \\
\rho_1^+(![\rho_2^+(!Int.?Int.\mathrm{end})].?[\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \triangleright \varnothing \\
\vdash \left(\begin{array}{l}
\mathrm{send}\ (t, x); \\
\mathrm{let}\ t = \mathrm{recv}\ x\ \mathrm{in} \\
\mathrm{act}\ t; \\
\mathrm{let}\ r = \mathrm{recv}\ a\ \mathrm{in} \\
\mathrm{log}\ r; \\
\mathrm{close}\ a; \mathrm{close}\ x
\end{array}\right) : 1
\end{array}\right)
$$

In this case, rules TVar, TPair, TSend and TLetShort have been used to type derive a further part of the process. It is again worth noting how TSend changes the session type that the capability holds. Then, using the derivation D$_4$ from above we derive:

$$
\mathrm{D}_5\;
\dfrac{
\begin{array}{c}
\dfrac{
\dfrac{}{a : tr(\rho_2^+); \varnothing \vdash a : tr(\rho_2^+)}
}{
\left(\begin{array}{c}
a : tr(\rho_2^+); \varnothing; \\
\left(\begin{array}{l}
\rho_1^+(![\rho_2^+(!Int.?Int.\mathrm{end})].?[\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \\
\otimes \rho_2^+(!Int.?Int.\mathrm{end})
\end{array}\right) \triangleright \rho_1^+(![\rho_2^+(!Int.?Int.\mathrm{end})].?[\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \\
\vdash \mathrm{inact}\ a : [\rho_2^+(!Int.?Int.\mathrm{end})]
\end{array}\right)
} \\
\vdots
\end{array}
\qquad \mathrm{D}_4
}{
\left(\begin{array}{c}
a : tr(\rho_2^+), x : tr(\rho_1^+); \varnothing; \\
\rho_1^+(![\rho_2^+(!Int.?Int.\mathrm{end})].?[\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \otimes \rho_2^+(!Int.?Int.\mathrm{end}) \triangleright \varnothing \\
\vdash \left(\begin{array}{l}
\mathrm{let}\ t = \mathrm{inact}\ a\ \mathrm{in} \\
\mathrm{send}\ (t, x); \\
\mathrm{let}\ t = \mathrm{recv}\ x\ \mathrm{in} \\
\mathrm{act}\ t; \\
\mathrm{let}\ r = \mathrm{recv}\ a\ \mathrm{in} \\
\mathrm{log}\ r; \\
\mathrm{close}\ a; \mathrm{close}\ x
\end{array}\right) : 1
\end{array}\right)
}
$$

In this case, rules TVar, TLetShort and TInact have been used to type derive a further part of the process. It is worth noting how TInact does the opposite of TAct and removes the capability of using the channel endpoint a from the capability set and stores it in a type it returns. Finally, using the derivation D$_5$ from above we derive the full term:

$$
\dfrac{
\dfrac{
\dfrac{}{\varnothing;\varnothing \vdash 10 : Int}
\qquad
\dfrac{}{a : tr(\rho_2^+);\varnothing \vdash a : tr(\rho_2^+)}
}{a : tr(\rho_2^+);\varnothing \vdash (10, a) : Int \times tr(\rho_2^+)}
}{
\left(\begin{array}{c}
a : tr(\rho_2^+);\varnothing; \\
\left(\begin{array}{l}
\rho_1^+(![\rho_2^+(!Int.?Int.\mathrm{end})].?[\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \\
\otimes\ \rho_2^+(!Int.!Int.?Int.\mathrm{end})
\end{array}\right)
\triangleright
\left(\begin{array}{l}
\rho_1^+(![\rho_2^+(!Int.?Int.\mathrm{end})].?[\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \\
\otimes\ \rho_2^+(!Int.?Int.\mathrm{end})
\end{array}\right) \\
\vdash \mathrm{send}\ (10, a) : 1
\end{array}\right)
}
$$

$$
\dfrac{\vdots}{
\left(\begin{array}{c}
a : tr(\rho_2^+), x : tr(\rho_1^+);\varnothing; \\
\rho_1^+(![\rho_2^+(!Int.?Int.\mathrm{end})].?[\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \otimes\ \rho_2^+(!Int.!Int.?Int.\mathrm{end}) \triangleright \varnothing \\
\vdash \left(\begin{array}{l}
\mathrm{send}\ (10, a); \\
\mathrm{let}\ t = \mathrm{inact}\ a\ \mathrm{in} \\
\mathrm{send}\ (t, x); \\
\mathrm{let}\ t = \mathrm{recv}\ x\ \mathrm{in} \\
\mathrm{act}\ t; \\
\mathrm{let}\ r = \mathrm{recv}\ a\ \mathrm{in} \\
\mathrm{log}\ r; \\
\mathrm{close}\ a; \mathrm{close}\ x
\end{array}\right) : 1
\end{array}\right)
}\ \mathrm{D}_5
$$

Rules TVar, TInt, TPair, TLetShort and TSend have been used in order to the remaining part of the process. Therefore, the typing judgement for P that can be used when typing the whole example is

$$
\left(\begin{array}{c}
a : tr(\rho_2^+), x : tr(\rho_1^+);\varnothing; \\
\rho_1^+(![\rho_2^+(!Int.?Int.\mathrm{end})].?[\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \otimes\ \rho_2^+(!Int.!Int.?Int.\mathrm{end}) \triangleright \varnothing \\
\vdash P : 1
\end{array}\right)
$$

5.1.2 Typing Derivation for Q

Similarly for process Q, using the typing rules presented before and the definition of Q provided in Figure 5.1, we present the typing derivation for this part of the example CGV program. Again, in order to follow the derivation gradually and be able to present it visually, the process has been split into several parts and thus typing derivations build upon each other. Using this approach, we can show that:

$$\mathrm{D}_1\;\dfrac{\begin{array}{c}\dfrac{\dfrac{\dfrac{}{t : [\rho_2^+(?Int.\text{end})]; \varnothing \vdash t : [\rho_2^+(?Int.\text{end})]} \qquad \dfrac{}{y : tr(\rho_1^-); \varnothing \vdash y : tr(\rho_1^-)}}{t : [\rho_2^+(?Int.\text{end})], y : tr(\rho_1^-); \varnothing \vdash (t, y) : [\rho_2^+(?Int.\text{end})] \times tr(\rho_1^-)}}{t : [\rho_2^+(?Int.\text{end})], y : tr(\rho_1^-); \varnothing; \rho_1^-(![\rho_2^+(?Int.\text{end})].\text{end}) \vdash \text{send } (t, y) : 1 \triangleright \rho_1^-(\text{end})} \\ \vdots \end{array} \qquad \dfrac{\dfrac{}{y : tr(\rho_1^-); \varnothing \vdash y : tr(\rho_1^-)}}{y : tr(\rho_1^-); \varnothing; \rho_1^-(\text{end}) \vdash \text{close } y : 1 \triangleright \varnothing}}{t : [\rho_2^+(?Int.\text{end})], y : tr(\rho_1^-); \varnothing; \rho_1^-(![\rho_2^+(?Int.\text{end})].\text{end}) \vdash \begin{pmatrix} \text{send } (t, y); \\ \text{close } y \end{pmatrix} : 1 \triangleright \varnothing}$$

Rules TVar, TPair, , TClose and TSend have been used in order to type derive a part of the process. It is worth noting that while the rules are used as previously, we are operating on endpoint y which is opposite to the endpoint x that was used in the process P. Then, using the derivation D$_1$ from above we derive:

$$\mathrm{D}_2\;\dfrac{\dfrac{\dfrac{}{a : tr(\rho_2^+); \varnothing \vdash a : tr(\rho_2^+)}}{\begin{pmatrix} a : tr(\rho_2^+); \varnothing; \\ \rho_1^-(![\rho_2^+(?Int.\text{end})].\text{end}) \otimes \rho_2^+(?Int.\text{end}) \triangleright \rho_1^-(![\rho_2^+(?Int.\text{end})].\text{end}) \\ \vdash \text{inact } a : [\rho_2^+(?Int.\text{end})] \end{pmatrix}} \qquad \mathrm{D}_1}{\begin{pmatrix} a : tr(\rho_2^+), y : tr(\rho_1^-); \varnothing; \\ \rho_1^-(![\rho_2^+(?Int.\text{end})].\text{end}) \otimes \rho_2^+(?Int.\text{end}) \triangleright \varnothing \\ \vdash \begin{pmatrix} \text{let } t = \text{inact } a \text{ in} \\ \text{send } (t, y); \\ \text{close } y \end{pmatrix} : 1 \end{pmatrix}}$$

In this case, rules TVar, TLetBind and TInact have been used to type derive a further part of the process. It is worth noting again how TInact modifies the capability sets by inactivating the channel endpoint. Then, using the derivation D$_2$ from above we derive:

$$\mathrm{D}_3\;\dfrac{\dfrac{\dfrac{\dfrac{}{\varnothing; \varnothing \vdash 123 : Int} \qquad \dfrac{}{a : tr(\rho_2^+); \varnothing \vdash a : tr(\rho_2^+)}}{a : tr(\rho_2^+); \varnothing \vdash (123, a) : Int \times tr(\rho_2^+)}}{\begin{pmatrix} a : tr(\rho_2^+); \varnothing; \\ \rho_1^-(![\rho_2^+(?Int.\text{end})].\text{end}) \otimes \rho_2^+(!Int.?Int.\text{end}) \triangleright \rho_1^-(![\rho_2^+(?Int.\text{end})].\text{end}) \otimes \rho_2^+(?Int.\text{end}) \\ \vdash \text{send}(123, a) : 1 \end{pmatrix}} \qquad \mathrm{D}_2}{\begin{pmatrix} a : tr(\rho_2^+), y : tr(\rho_1^-); \varnothing; \\ \rho_1^-(![\rho_2^+(?Int.\text{end})].\text{end}) \otimes \rho_2^+(!Int.?Int.\text{end}) \triangleright \varnothing \\ \vdash \begin{pmatrix} \text{send } (123, a); \\ \text{let } t = \text{inact } a \text{ in} \\ \text{send } (t, y); \\ \text{close } y \end{pmatrix} : 1 \end{pmatrix}}$$

In this case, rules TVar, TPair, TInt, TLetShort and TSend have been used in order to type derive a further part of the process. Then, using the derivation D₃ from above we derive:

$$
\mathrm{D}_4 \;\; \dfrac{\dfrac{\overline{t : [\rho_2^+(!Int.?Int.\mathrm{end})]; \varnothing \vdash t : [\rho_2^+(!Int.?Int.\mathrm{end})]}}{\left(\begin{array}{c} t : [\rho_2^+(!Int.?Int.\mathrm{end})]; \varnothing; \\ \rho_1^-(![\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \triangleright \rho_1^-(![\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \otimes \rho_2^+(!Int.?Int.\mathrm{end}) \\ \vdash \mathrm{act}\ t : 1 \end{array}\right)} \qquad \mathrm{D}_3}{\left(\begin{array}{c} a : tr(\rho_2^+), y : tr(\rho_1^-), t : [\rho_2^+(!Int.?Int.\mathrm{end})]; \varnothing; \\ \rho_1^-(![\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \triangleright \varnothing \\ \vdash \left(\begin{array}{l} \mathrm{act}\ t; \\ \mathrm{send}\ (123, a); \\ \mathrm{let}\ t = \mathrm{inact}\ a\ \mathrm{in} \\ \mathrm{send}\ (t, y); \\ \mathrm{close}\ y \end{array}\right) : 1 \end{array}\right)}
$$

Rules TVar, TLetBind and TAct have again been used to type derive a further part of the process. As this point it is worth noting the duality of activating and inactivating because whenever process P inactivated/activated a channel, process Q had to activate/inactivate it respectively. Finally, using the derivation D₄ from above we derive the full term:

$$
\dfrac{\dfrac{\overline{y : tr(\rho_1^-); \varnothing \vdash y : tr(\rho_1^-)}}{\left(\begin{array}{c} y : tr(\rho_1^-); \varnothing; \\ \rho_1^-(?[\rho_2^+(!Int.?Int.\mathrm{end})].![\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \triangleright \rho_1^-(![\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \\ \vdash \mathrm{recv}\ y : [\rho_2^+(!Int.?Int.\mathrm{end})] \end{array}\right)} \qquad \mathrm{D}_4}{\left(\begin{array}{c} a : tr(\rho_2^+), y : tr(\rho_1^-); \varnothing; \\ \rho_1^-(?[\rho_2^+(!Int.?Int.\mathrm{end})].![\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \triangleright \varnothing \\ \vdash \left(\begin{array}{l} \mathrm{let}\ t = \mathrm{recv}\ y\ \mathrm{in} \\ \mathrm{act}\ t; \\ \mathrm{send}\ (123, a); \\ \mathrm{let}\ t = \mathrm{inact}\ a\ \mathrm{in} \\ \mathrm{send}\ (t, y); \\ \mathrm{close}\ y \end{array}\right) : 1 \end{array}\right)}
$$

Rules TVar, TLetBind and TRecv have again been used to type derive the remaining part of the process. Therefore, typing judgement for Q that can be used in order to type the whole example is

$a : tr(\rho_2^+), y : tr(\rho_1^-); \varnothing; \rho_1^-(?[\rho_2^+(!Int.?Int.\mathrm{end})].![\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \vdash Q : 1 \triangleright \varnothing$

5.1.3 Typing Derivation for B

Similarly for B, using the typing rules presented before and the definition of process B provided in Figure 5.1, we present the typing derivation for this part of the example CGV program. Although this process is simpler than P or Q, we still split the process into a couple of parts for the sake of readability and thus typing derivations build upon each other. Using this approach, we can show that:

$$
\mathrm{D}_1\;
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{p : Int; \varnothing \vdash p : Int}
\qquad
\dfrac{}{n : Int; \varnothing \vdash n : Int}
}{p : Int, n : Int; \varnothing \vdash (p, n) : Int \times Int}
}{p : Int, n : Int; \varnothing; \rho_2^-(!Int.\mathrm{end}) \vdash \mathrm{pay}\ (p, n) : 1 \triangleright \rho_2^-(!Int.\mathrm{end})}
\quad
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{r : Int; \varnothing \vdash r : Int}
\qquad
\dfrac{}{b : tr(\rho_2^-); \varnothing \vdash b : tr(\rho_2^-)}
}{r : Int, b : tr(\rho_2^-); \varnothing \vdash (r, b) : Int \times tr(\rho_2^-)}
}{r : Int, b : tr(\rho_2^-); \varnothing; \rho_2^-(!Int.\mathrm{end}) \vdash \mathrm{send}\ (r, b) : 1 \triangleright \rho_2^-(\mathrm{end})}
\quad
\dfrac{
\dfrac{}{b : tr(\rho_2^-); \varnothing \vdash b : tr(\rho_2^-)}
}{b : tr(\rho_2^-); \varnothing; \rho_2^-(\mathrm{end}) \vdash \mathrm{close}\ b : 1 \triangleright \varnothing}
}{r : Int, b : tr(\rho_2^-); \varnothing; \rho_2^-(!Int.\mathrm{end}) \vdash \left(\begin{array}{l}\mathrm{send}\ (r, b);\\ \mathrm{close}\ b\end{array}\right) : 1 \triangleright \varnothing}
}{p : Int, n : Int, b : tr(\rho_2^-); \varnothing; \rho_2^-(!Int.\mathrm{end}) \vdash \left(\begin{array}{l}\mathrm{let}\ r = \mathrm{pay}\ (p, n)\ \mathrm{in}\\ \mathrm{send}\ (r, b);\\ \mathrm{close}\ b\end{array}\right) : 1 \triangleright \varnothing}
$$

Standard rules TVar, TPair, TPay, TClose, TSend, TLetShort and TLetBind have been used in order to type derive the first part of the process. Then, using the derivation D_1 as above, we can derive the whole of B as follows:

$$
\dfrac{
\dfrac{
\dfrac{}{b : tr(\rho_2^-); \varnothing \vdash b : tr(\rho_2^-)}
}{b : tr(\rho_2^-); \varnothing; \rho_2^-(?Int.?Int.!Int.\mathrm{end}) \vdash \mathrm{recv}\ b : Int \triangleright \rho_2^-(?Int.!Int.\mathrm{end})}
\quad
\dfrac{
\dfrac{
\dfrac{}{b : tr(\rho_2^-); \varnothing \vdash b : tr(\rho_2^-)}
}{b : tr(\rho_2^-); \varnothing; \rho_2^-(?Int.!Int.\mathrm{end}) \vdash \mathrm{recv}\ b : Int \triangleright \rho_2^-(!Int.\mathrm{end})}
\qquad \mathrm{D}_1
}{p : Int, b : tr(\rho_2^-); \varnothing; \rho_2^-(?Int.!Int.\mathrm{end}) \vdash \left(\begin{array}{l}\mathrm{let}\ n = \mathrm{recv}\ b\ \mathrm{in}\\ \mathrm{let}\ r = \mathrm{pay}\ (p, n)\ \mathrm{in}\\ \mathrm{send}\ (r, b);\\ \mathrm{close}\ b\end{array}\right) : 1 \triangleright \varnothing}
}{b : tr(\rho_2^-); \varnothing; \rho_2^-(?Int.?Int.!Int.\mathrm{end}) \vdash \left(\begin{array}{l}\mathrm{let}\ p = \mathrm{recv}\ b\ \mathrm{in}\\ \mathrm{let}\ n = \mathrm{recv}\ b\ \mathrm{in}\\ \mathrm{let}\ r = \mathrm{pay}\ (p, n)\ \mathrm{in}\\ \mathrm{send}\ (r, b);\\ \mathrm{close}\ b\end{array}\right) : 1 \triangleright \varnothing}
$$

Rules TVar, TRecv and TLetBind have been used in order to type derive the remaining part of the process. It is worth noting that process B uses exclusively channel endpoint b. Although b is the opposite endpoint of a, which is used by both P and Q, process B does not have to worry about who communicates from the other endpoint and can follow the protocol without any special considerations about the other endpoint being shared. Therefore, typing judgement for B that can be used in order to type the whole example is $b : tr(\rho_2^-); \varnothing; \rho_2^-(?Int.?Int.!Int.\mathrm{end}) \vdash B : 1 \triangleright \varnothing$.

5.1.4 Typing Derivation for M

Finally, using the typing judgements resulting from typing derivations for processes P, Q and B we can type derive the type of the whole program M thus verifying it is well typed in CGV. Again, in order to follow the derivation gradually and be able to present it visually, the process has been split into several parts and thus typing derivations build upon each other as well as use the derivations from previous sections. Using this approach, we can show that:

$$
\dfrac{
\dfrac{\dfrac{}{a : tr(\rho_2^+), y : tr(\rho_1^-); \varnothing; \rho_1^-(?[\rho_2^+(!Int.?Int.\mathrm{end})].![\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \vdash Q : 1 \triangleright \varnothing}}
{\left(\begin{array}{c}
a : tr(\rho_2^+), y : tr(\rho_1^-); \varnothing; \\
\left(\begin{array}{l} \rho_2^-(?Int.?Int.!Int.\mathrm{end}) \\ \otimes\ \rho_1^-(?[\rho_2^+(!Int.?Int.\mathrm{end})].![\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \end{array}\right) \triangleright \rho_2^-(?Int.?Int.!Int.\mathrm{end}) \\
\vdash \mathrm{spawn}\ Q : 1
\end{array}\right)}
\\ \vdots \qquad
\dfrac{\dfrac{}{b : tr(\rho_2^-); \varnothing; \rho_2^-(?Int.?Int.!Int.\mathrm{end}) \vdash B : 1 \triangleright \varnothing}}{b : tr(\rho_2^-); \varnothing; \rho_2^-(?Int.?Int.!Int.\mathrm{end}) \vdash \mathrm{spawn}\ B : 1 \triangleright \varnothing}
}{
\left(\begin{array}{c}
a : tr(\rho_2^+), b : tr(\rho_2^-), y : tr(\rho_1^-); \varnothing; \\
\left(\begin{array}{l} \rho_2^-(?Int.?Int.!Int.\mathrm{end}) \\ \otimes\ \rho_1^-(?[\rho_2^+(!Int.?Int.\mathrm{end})].![\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \end{array}\right) \triangleright \varnothing \\
\vdash \left(\begin{array}{l} \mathrm{spawn}\ Q; \\ \mathrm{spawn}\ B \end{array}\right) : 1
\end{array}\right)
} \mathrm{D}_1
$$

Rules TSpawn and TLetShort have been used in order to type derive a part of the process. It is worth noting how TSpawn treats capability sets i.e. it separates capabilities that a thread requires and removes them from own capability set e.g. in $b : tr(\rho_2^-); \varnothing; \rho_2^-(?Int.?Int.!Int.\mathrm{end}) \vdash \mathrm{spawn}\ B : 1 \triangleright \varnothing$ the capability $\rho_2^-(?Int.?Int.!Int.\mathrm{end})$ has been removed from the pre-evaluation capability set because it has been given to the process B which requires it to be well typed. Then, using the derivation D_1 from above we derive:

$$\dfrac{D_2 \qquad \dfrac{\dfrac{}{\left(\begin{array}{c} a : tr(\rho_2^+), x : tr(\rho_1^+); \varnothing; \\ \rho_1^+(![\rho_2^+(!Int.?Int.\mathrm{end})].?[\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \otimes \ \rho_2^+(!Int.!Int.?Int.\mathrm{end}) \triangleright \varnothing \\ \vdash P : 1 \end{array}\right)}}{\left(\begin{array}{c} a : tr(\rho_2^+), x : tr(\rho_1^+); \varnothing; \\ \left(\begin{array}{l} \rho_1^+(![\rho_2^+(!Int.?Int.\mathrm{end})].?[\rho_2^+(?Int.\mathrm{end})].\mathrm{end})\otimes \\ \rho_2^+(!Int.!Int.?Int.\mathrm{end})\otimes \\ \rho_2^-(?Int.?Int.!Int.\mathrm{end})\otimes \\ \rho_1^-(?[\rho_2^+(!Int.?Int.\mathrm{end})].![\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \end{array}\right) \triangleright \left(\begin{array}{l} \rho_2^-(?Int.?Int.!Int.\mathrm{end})\otimes \\ \rho_1^-(?[\rho_2^+(!Int.?Int.\mathrm{end})].![\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \end{array}\right) \\ \vdash \mathrm{spawn}\ P : 1 \end{array}\right)} \qquad D_1}{\left(\begin{array}{c} a : tr(\rho_2^+), b : tr(\rho_2^-), x : tr(\rho_1^+), y : tr(\rho_1^-); \varnothing; \\ \left(\begin{array}{l} \rho_1^+(![\rho_2^+(!Int.?Int.\mathrm{end})].?[\rho_2^+(?Int.\mathrm{end})].\mathrm{end})\otimes \\ \rho_2^+(!Int.!Int.?Int.\mathrm{end})\otimes \\ \rho_2^-(?Int.?Int.!Int.\mathrm{end})\otimes \\ \rho_1^-(?[\rho_2^+(!Int.?Int.\mathrm{end})].![\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \end{array}\right) \triangleright \varnothing \\ \vdash \left(\begin{array}{l} \mathrm{spawn}\ P; \\ \mathrm{spawn}\ Q; \\ \mathrm{spawn}\ B \end{array}\right) : 1 \end{array}\right)}$$

Again, rules TSpawn and TLetShort have been used to type derive a further part of the process. Then, using TNew alone we derive:

$$D_3 \ \dfrac{}{\left(\begin{array}{c} \varnothing; \rho_2^+, \rho_2^-; \\ \left(\begin{array}{l} \rho_1^+(![\rho_2^+(!Int.?Int.\mathrm{end})].?[\rho_2^+(?Int.\mathrm{end})].\mathrm{end})\otimes \\ \rho_1^-(?[\rho_2^+(!Int.?Int.\mathrm{end})].![\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \end{array}\right) \triangleright \left(\begin{array}{l} \rho_1^+(![\rho_2^+(!Int.?Int.\mathrm{end})].?[\rho_2^+(?Int.\mathrm{end})].\mathrm{end})\otimes \\ \rho_2^+(!Int.!Int.?Int.\mathrm{end})\otimes \\ \rho_2^-(?Int.?Int.!Int.\mathrm{end})\otimes \\ \rho_1^-(?[\rho_2^+(!Int.?Int.\mathrm{end})].![\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \end{array}\right) \\ \vdash \mathrm{new} : tr(\rho_2^+) \times tr(\rho_2^-) \end{array}\right)}$$

It is worth noting how for the first time in this typing derivation we are using a non-empty capability context i.e. $\Delta = \rho_2^+, \rho_2^-$. It is necessary to ensure that one capability can govern only one channel endpoint as we will see below. Then using the derivations D_2 and D_3 from above we derive:

$$
\dfrac{
D_3 \quad
\dfrac{
\dfrac{}{t : tr(\rho_2^+) \times tr(\rho_2^-); \varnothing \vdash t : tr(\rho_2^+) \times tr(\rho_2^-)} \quad D_2
}{
\left(\begin{array}{c}
t : tr(\rho_2^+) \times tr(\rho_2^-), x : tr(\rho_1^+), y : tr(\rho_1^-); \varnothing; \\
\left(\begin{array}{l}
\rho_1^+(![\rho_2^+(!Int.?Int.\mathrm{end})].?[\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \otimes \\
\rho_2^+(!Int.!Int.?Int.\mathrm{end}) \otimes \\
\rho_2^-(?Int.?Int.!Int.\mathrm{end}) \otimes \\
\rho_1^-(?[\rho_2^+(!Int.?Int.\mathrm{end})].![\rho_2^+(?Int.\mathrm{end})].\mathrm{end})
\end{array}\right) \triangleright \varnothing \\
\vdash \left(\begin{array}{l}
\mathrm{let}\ (a, b) = t\ \mathrm{in} \\
\mathrm{spawn}\ P; \\
\mathrm{spawn}\ Q; \\
\mathrm{spawn}\ B
\end{array}\right) : 1
\end{array}\right)
}
}{
\left(\begin{array}{c}
x : tr(\rho_1^+), y : tr(\rho_1^-); \rho_2^+, \rho_2^-; \\
\left(\begin{array}{l}
\rho_1^+(![\rho_2^+(!Int.?Int.\mathrm{end})].?[\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \otimes \\
\rho_1^-(?[\rho_2^+(!Int.?Int.\mathrm{end})].![\rho_2^+(?Int.\mathrm{end})].\mathrm{end})
\end{array}\right) \triangleright \varnothing \\
\vdash \left(\begin{array}{l}
\mathrm{let}\ t = \mathrm{new}\ \mathrm{in} \\
\mathrm{let}\ (a, b) = t\ \mathrm{in} \\
\mathrm{spawn}\ P; \\
\mathrm{spawn}\ Q; \\
\mathrm{spawn}\ B
\end{array}\right) : 1
\end{array}\right)
}\ D_4
$$

In this case, rules TVar, TLetBind and TLetPair have been used to type derive a further part of the process. It is worth noting how the capability contexts have been combined i.e. the current capability context is no longer empty since TNew requires a pair of capabilities to be present in it. Then we can use TNew again to derive:

$$
D_5 \ \dfrac{}{
\left(\begin{array}{c}
\varnothing; \rho_1^+, \rho_1^-; \\
\varnothing \triangleright \left(\begin{array}{l}
\rho_1^+(![\rho_2^+(!Int.?Int.\mathrm{end})].?[\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \otimes \\
\rho_1^-(?[\rho_2^+(!Int.?Int.\mathrm{end})].![\rho_2^+(?Int.\mathrm{end})].\mathrm{end})
\end{array}\right) \\
\vdash \mathrm{new} : tr(\rho_1^+) \times tr(\rho_1^-)
\end{array}\right)
}
$$

It is again worth noting how the capability context is non-empty but it contains capabilities different than previously i.e. $\Delta = \rho_1^+, \rho_1^-$ rather than ρ_2^+, ρ_2^-. Finally, using the derivations D_4 and D_5 from above we derive:

$$\dfrac{D_5 \qquad \dfrac{\dfrac{}{t : tr(\rho_1^+) \times tr(\rho_1^-); \varnothing \vdash t : tr(\rho_1^+) \times tr(\rho_1^-)} \qquad D_4}{\left(\begin{array}{c} t : tr(\rho_1^+) \times tr(\rho_1^-); \rho_2^+, \rho_2^-; \\ \left(\begin{array}{l} \rho_1^+(![\rho_2^+(!Int.?Int.\mathrm{end})].?[\rho_2^+(?Int.\mathrm{end})].\mathrm{end})\otimes \\ \rho_1^-(?[\rho_2^+(!Int.?Int.\mathrm{end})].![\rho_2^+(?Int.\mathrm{end})].\mathrm{end}) \end{array}\right) \triangleright \varnothing \\ \vdash \left(\begin{array}{l} \mathrm{let}\ (x, y) = t\ \mathrm{in} \\ \mathrm{let}\ t = \mathrm{new}\ \mathrm{in} \\ \mathrm{let}\ (a, b) = t\ \mathrm{in} \\ \mathrm{spawn}\ P; \\ \mathrm{spawn}\ Q; \\ \mathrm{spawn}\ B \end{array}\right) : 1 \end{array}\right)}}{\varnothing; \rho_1^+, \rho_1^-, \rho_2^+, \rho_2^-; \varnothing \vdash \left(\begin{array}{l} \mathrm{let}\ t = \mathrm{new}\ \mathrm{in} \\ \mathrm{let}\ (x, y) = t\ \mathrm{in} \\ \mathrm{let}\ t = \mathrm{new}\ \mathrm{in} \\ \mathrm{let}\ (a, b) = t\ \mathrm{in} \\ \mathrm{spawn}\ P; \\ \mathrm{spawn}\ Q; \\ \mathrm{spawn}\ B \end{array}\right) : 1 \triangleright \varnothing}$$

Rules TVar, TLetBind and TLetPair have been used in order to type derive the remaining part of the process. It is crucial to note how the capability contexts have been combined and resulted in $\Delta = \rho_1^+, \rho_1^-, \rho_2^+, \rho_2^-$. This context combination would be impossible if the new produced channels that are controlled by the same capabilities multiple times since it only allows unique names and they cannot be overriden as specified by the combination operator.

Therefore, we can see that the example is well typed in CGV since we can type derive it and reach the form $\varnothing; \rho_1^+, \rho_1^-, \rho_2^+, \rho_2^-; \varnothing \vdash M : 1 \triangleright \varnothing$ which is a valid CGV program i.e. it is well typed under empty typing context, empty capability sets and with an arbitrary capability context Δ.

5.2 Case Study: Linearity Violation

In order to showcase the enforcement of linearity of communication channels present in CGV, we will present an example that violates that linearity which makes it not well typed in CGV.

The example program can be seen in Figure 5.2. For the sake of simplicity, the program operates on one set of endpoints only and the session type of the channel is end. In this example, the program creates a new channel and bind the endpoints to x and y. Then it spawns two threads where both of these are trying to close the channel x. The main thread tries to close the channel y after spawning the other threads.

The part that violates linearity in this example is that both spawned threads want to close x which would violate communication safety.

$$
\begin{aligned}
M = \;& \text{let } t = \text{new in} \\
& \text{let } (x, y) = t \text{ in} \\
& \text{spawn (close } x); \\
& \text{spawn (close } x); \\
& \text{close } y
\end{aligned}
$$

Figure 5.2: *Example of a program that violates linearity of communication.*

We can attempt a typing derivation for this example as follows:

$$
\dfrac{\dfrac{\dfrac{}{x : tr(\rho_1^+); \varnothing \vdash x : tr(\rho_1^+)}}{x : tr(\rho_1^+); \varnothing; \rho_1^+(\text{end}) \vdash \text{close } x : 1 \triangleright \varnothing}}{x : tr(\rho_1^+); \varnothing; \rho_1^+(\text{end}) \otimes \rho_1^-(\text{end}) \vdash \text{spawn (close } x) : 1 \triangleright \rho_1^-(\text{end})}
$$

$$
\mathrm{D}_1 \;\; \dfrac{\vdots \qquad\qquad \dfrac{\dfrac{}{y : tr(\rho_1^-); \varnothing \vdash y : tr(\rho_1^-)}}{y : tr(\rho_1^-); \varnothing; \rho_1^-(\text{end}) \vdash \text{close } y : 1 \triangleright \varnothing}}{x : tr(\rho_1^+), y : tr(\rho_1^-); \varnothing; \rho_1^+(\text{end}) \otimes \rho_1^-(\text{end}) \vdash \begin{pmatrix} \text{spawn (close } x); \\ \text{close } y \end{pmatrix} : 1 \triangleright \varnothing}
$$

By using rules TVar, TClose, TSpawn and TLetShort as expected. Then we can continue trying with

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{x : tr(\rho_1^+); \varnothing \vdash x : tr(\rho_1^+)}}{x : tr(\rho_1^+); \varnothing; \rho_1^+(\text{end}) \vdash \text{close } x : 1 \triangleright \varnothing}}{x : tr(\rho_1^+); \varnothing; \rho_1^+(\text{end}) \otimes C \vdash \text{spawn (close } x) : 1 \triangleright C} \qquad \mathrm{D}_1}{\textit{Term we want to type: } \begin{pmatrix} \text{spawn (close } x); \\ \text{spawn (close } x); \\ \text{close } y \end{pmatrix}}
$$

but we are unable to proceed given the static typing rules of CGV, in particular because of TLet-Bind chaining capability sets. This is caused by the fact that executing the first spawn (close x) produces capability set C. This set cannot contain the capability of the x channel endpoint, i.e. ρ_1^+, because it has been passed to the spawned thread. In particular, the capability $\rho_1^+(\text{end})$ is required by the thread as dictated by TClose. Thus, it follows that it can no longer exist in the post-evaluation capability set after spawning the thread. However, in order to be able to type spawn (close x); spawn (close x); close y we need $C = \rho_1^+(\text{end}) \otimes \rho_1^-(\text{end})$ as dictated by TShort. This leads to contradiction meaning that this example is not well typed in CGV.

While this example does not prove that every program that violates linearity is ill-typed in CGV, it showcases that the system does have ways of ensuring that session type integrity is not violated while keeping the channels unrestricted.

```
M =  let t = new in
     let (x, y) = t in
     let t = new in          P =  recv y;                  Q =  recv a;
     let (a, b) = t in            send ((), b);                 send ((), x);
     spawn P;                     close y; close b              close x; close a
     spawn Q
```

Figure 5.3: *Example of a CGV program with a deadlock.*

5.3 Expected Properties

As mentioned in 2.1.4, the type soundness of type systems can be split into two properties: subject reduction and progress. Proving these properties of CGV is not possible without operational semantics because we cannot reason about how the program evaluates and reduces. Subject reduction is particularly difficult to reason about based on only static elements but we can anticipate that CGV in its current form will not preserve progress. This is caused by the fact that capability-based sharing is useful only if we allow cyclic structures in programs - without cycles, we would not be able to freely pass around capabilities to use unrestricted channel endpoints. In the case of GV, all versions other than PGV allowed exclusively tree-structured programs because spawning processes in parallel was allowed via the $fork$ construct. This feature has guaranteed progress for those GVs because the tree-structure ruled out the possibility of deadlocks. On the other hand, PGV has allowed cyclic processes while preserving deadlock-freedom via priorities. Thus, PGV allows only those cycles that do not lead to deadlocks while making deadlockable programs not typeable. This means that without an extra mechanism for prohibiting deadlocks, cyclic structures can lead to a lack of progress.

Due to the fact that CGV does not utilise any additional mechanisms to prevent deadlocks from occurring and allows cyclic processes, we expect progress to be lost. This can be intuitively seen in the example illustrated in Figure 5.3 which is adapted from Kokke and Dardha (2021a). It is a simple scenario where two processes are spawned, P and Q. They use two sets of channel endpoints to communicate with each other. The communication on these channels is interleaved in a way that causes a deadlock in the case of synchronous communication. In particular, process P is supposed to receive on y and then send on b but process Q cannot send on x before receiving on a thus leading to a deadlock because both processes will infinitely wait until they receive a message. This example shows that CGV should not preserve progress on its own and additional means are necessary to restore deadlock-freedom.

6 Conclusion

We conclude by summarising the work that was done in this dissertation including the contributions made in the form of CGV, discussions of language design choices behind CGV and a case-study-based evaluation. Additionally, we provide a discussion of the future work that will be conducted to finalise the extension and potential directions for further research.

In our work, we have identified the requirements for a GV extension that allows channel sharing via capabilities. To understand the process of creating such an extension, we have surveyed the literature and extracted the core concepts of current GV extensions and capabilities and discussed them in Section 3.2. Specifically, we have explained our reasoning for moving away from correspondence to classical linear logic and the choice of constructs that should be included. A consequence of moving away from logical correspondence is losing deadlock-freedom which we consider necessary in order to take full advantage of the added expressivity of channel sharing. We have also discussed interesting challenges that we have identified when adding capabilities to GV, particularly the design choices directly related to capabilities, i.e. the way of handling unrestricted types and type-and-effect system, and their trade-offs.

Further, we have presented components of CGV which show how capabilities can be incorporated into a GV-based language. Specifically, we have defined the required terms and types, type-and-effect environments, combination operators and judgements for the system. We have also provided typing rules that allow reasoning about how CGV programs are constructed and motivated them by explaining the expected behaviour.

Lastly, we have provided two case studies that showcase the expressivity of the system. We have shown that CGV allows some programs that require channel sharing using the example of PQB, that has been discussed in depth in Section 2.4. We have also shown that CGV prohibits some programs that violate linearity of communication and thus preserves the guarantees of session types. We have noted that proving the expressivity formally would require the remaining parts of the system, i.e. operational semantics, run-time configurations and run-time typing rules, to be defined. Overall, we have found that CGV behaves as expected when it comes to the treatment of capabilities and prohibiting violations of linearity.

6.1 Further Work

Our proposed further work can be split into two parts: finalising CGV, which includes defining the remaining elements of the system and proving its properties, and expanding upon CGV into a few potential research directions.

6.1.1 Finalising CGV

Given the static elements of CGV as provided in Chapter 4, the remaining parts that are left to be defined are operational semantics and structural congruence.

In order to define operational semantics for CGV, we will need to specify the configuration of processes and how they evaluate analogously to other GVs. Specifically, run-time typing rules for main thread, child thread, parallel composition and name restriction need to be defined. This will allow CGV to specify how processes communicate with each other in parallel and to ensure that there is only one thread that can evaluate to a value. In the case of CGV, extra attention needs to be paid when defining name restriction due to the unrestricted nature of channels. Thus, we will need to consider a definition that is able to capture all occurrences of the channel that is being restricted.

After operational semantics are defined in terms of term and configuration reductions, we will be able to define what structural congruence holds for CGV. We expect to be able to define the same structural congruence rules as PGV (Kokke and Dardha 2021a) due to the separation of $fork$ construct into $spawn$ and new constructs. This is due to the fact that other GVs do not hold structural congruence, or require additional techniques to be able to preserve it, because of the nature of the $fork$ construct.

Lastly, properties of the system can be proven using standard proof techniques. As mentioned in the Section 5.3, we expect CGV to preserve subject reduction, given a correct definition of operational semantics, but not progress.

6.1.2 Further Research Opportunities

A few exciting research directions can be suggested, starting from a finalised CGV system. Firstly, given our expectation of deadlocks being introduced in CGV, we plan to investigate potential ways of restoring deadlock-freedom. One of the possible ways of achieving this would be utilising priorities in the style of Kokke and Dardha (2021a) by tying priorities to the capabilities rather than to channel endpoints. While the order of communication described by priorities should work analogously on capabilities, this approach could impose an interesting challenge in combination with type-and-effect systems because so far priorities have been reasoned about only in standard type systems.

Secondly, we could look into combining techniques into CGV with the focus of making it more practical. In particular, combining the ideas of gradual typing Igarashi et al. (2017) and exceptions Fowler et al. (2019) would provide significant practical advantages while preserving added expressivity of channel sharing. These extensions, however, may introduce additional challenge depending on the way deadlock-freedom gets restored. In particular, combining exceptions with priorities or gradual typing with priorities has not been studied so it could impose an interesting theoretical challenge.

Lastly, we plan to implement CGV for the sake of providing a language that has useful practical components as well as strong guarantees. As mentioned by Kokke (2019), affine typing is particularly important given that very few programming languages support linear typing as well as due to the fact that linear typing is not practical in many circumstances. Thus, finalising CGV is crucial in allowing added expressivity of channel sharing, restoring deadlock-freedom will be needed in order to preserve valuable properties of GV and practical features are key to making the CGV implementation usable.

6 Bibliography

R. Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19, 2009. ISSN 09567968. doi: 10.1017/S095679680900728X.

S. Balzer and F. Pfenning. Manifest sharing with session types. *Proceedings of the ACM on Programming Languages*, 1, 2017. ISSN 2475-1421. doi: 10.1145/3110281.

S. Balzer, B. Toninho, and F. Pfenning. Manifest deadlock-freedom for shared session types. In L. Caires, editor, *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 611–639. Springer, 2019. doi: 10.1007/978-3-030-17184-1_22. URL `https://doi.org/10.1007/978-3-030-17184-1_22`.

L. Cardelli. Type systems. *ACM Computing Surveys (CSUR)*, 28:263–264, 3 1996. ISSN 0360-0300. doi: 10.1145/234313.234418. URL `https://dl.acm.org/doi/abs/10.1145/234313.234418`.

K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, 1999. ISSN 07308566. doi: 10.1145/292540.292564.

O. Dardha and S. J. Gay. A new linear logic for deadlock-free session-typed processes. In C. Baier and U. D. Lago, editors, *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10803 of *Lecture Notes in Computer Science*, pages 91–109. Springer, 2018. doi: 10.1007/978-3-319-89366-2_5. URL `https://doi.org/10.1007/978-3-319-89366-2_5`.

M. Dezani-Ciancaglini and U. de'Liguoro. Sessions and session types: An overview. In C. Laneve and J. Su, editors, *Web Services and Formal Methods, 6th International Workshop, WS-FM 2009, Bologna, Italy, September 4-5, 2009, Revised Selected Papers*, volume 6194 of *Lecture Notes in Computer Science*, pages 1–28. Springer, 2009. doi: 10.1007/978-3-642-14458-5_1. URL `https://doi.org/10.1007/978-3-642-14458-5_1`.

M. Dezani-Ciancaglini, N. Yoshida, A. J. Ahern, and S. Drossopoulou. A distributed object-oriented language with session types. In R. D. Nicola and D. Sangiorgi, editors, *Trustworthy Global Computing, International Symposium, TGC 2005, Edinburgh, UK, April 7-9, 2005, Revised Selected Papers*, volume 3705 of *Lecture Notes in Computer Science*, pages 299–318. Springer, 2005. doi: 10.1007/11580850_16. URL `https://doi.org/10.1007/11580850_16`.

M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. In D. Thomas, editor, *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*, volume 4067 of *Lecture Notes in Computer Science*, pages 328–352. Springer, 2006. doi: 10.1007/11785477_20. URL `https://doi.org/10.1007/11785477_20`.

S. Fowler, S. Lindley, J. G. Morris, and S. Decova. Exceptional asynchronous session types: session types without tiers. *Proceedings of the ACM on Programming Languages*, 3, 2019. ISSN 2475-1421. doi: 10.1145/3290341.

S. Fowler, W. Kokke, O. Dardha, S. Lindley, and J. G. Morris. Separating sessions smoothly. In S. Haddad and D. Varacca, editors, *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference*, volume 203 of *LIPIcs*, pages 36:1–36:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi: 10.4230/LIPIcs.CONCUR.2021.36. URL `https://doi.org/10.4230/LIPIcs.CONCUR.2021.36`.

S. J. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010. doi: 10.1017/S0956796809990268.

D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. *ACM SIGPLAN Notices*, 37, 2002. ISSN 0362-1340. doi: 10.1145/543552.512563.

K. Honda. Types for dyadic interaction. In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science, pages 509–523, Berlin, Heidelberg, 1993. Springer. ISBN 978-3-540-47968-0. doi: 10.1007/3-540-57208-2_35.

K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *Journal of the ACM*, 63, 2016. ISSN 1557735X. doi: 10.1145/2827695.

H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P. M. Deniélou, D. Mostrous, L. Padovani, A. Nióravara, E. Tuosto, H. T. Vieira, and G. Zavattaro. Foundations of session types and behavioural contracts. *ACM Computing Surveys*, 49, 2016. ISSN 15577341. doi: 10.1145/2873052.

A. Igarashi, P. Thiemann, V. T. Vasconcelos, and P. Wadler. Gradual session types. *Proceedings of the ACM on Programming Languages*, 1, 2017. ISSN 2475-1421. doi: 10.1145/3110282.

T. B. L. Jespersen, P. Munksgaard, and K. F. Larsen. Session types for rust. In P. Bahr and S. Erdweg, editors, *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming, WGP@ICFP 2015, Vancouver, BC, Canada, August 30, 2015*, pages 13–22. ACM, 2015. doi: 10.1145/2808098.2808100. URL `https://doi.org/10.1145/2808098.2808100`.

G. Kahn. Natural semantics. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 247 LNCS:22–39, 1987. ISSN 16113349. doi: 10.1007/BFB0039592.

N. Kobayashi. A new type system for deadlock-free processes. In *Proceedings of the 17th International Conference on Concurrency Theory*, CONCUR'06, page 233–247, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3540373764. doi: 10.1007/11817949_16.

W. Kokke. Rusty variation: Deadlock-free sessions with failure in rust. In M. Bartoletti, L. Henrio, A. Mavridou, and A. Scalas, editors, *Proceedings 12th Interaction and Concurrency Experience, ICE 2019, Copenhagen, Denmark, 20-21 June 2019*, volume 304 of *EPTCS*, pages 48–60, 2019. doi: 10.4204/EPTCS.304.4. URL `https://doi.org/10.4204/EPTCS.304.4`.

W. Kokke and O. Dardha. Prioritise the best variation. In K. Peters and T. A. C. Willemse, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 100–119, Cham, 2021a. Springer International Publishing. ISBN 978-3-030-78089-0.

W. Kokke and O. Dardha. Deadlock-free session types in linear Haskell. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell*, Haskell 2021, pages 1–13. Association for Computing Machinery, Aug. 2021b. ISBN 9781450386159. doi: 10.1145/3471874.3472979. URL `https://doi.org/10.1145/3471874.3472979`.

S. Lindley and J. Morris. A semantics for propositions as sessions. In J. Vitek, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 560–584. Springer Berlin Heidelberg, Apr. 2015. ISBN 978-3-662-46668-1. doi: 10.1007/978-3-662-46669-8_23. 24th European Symposium on Programming and Systems, ESOP 2015 ; Conference date: 11-04-2015 Through 18-04-2015.

S. Lindley and J. Morris. *Lightweight functional session types.* River Publishers, Feb. 2017.

S. Lindley and J. G. Morris. Sessions as propositions. *Electronic Proceedings in Theoretical Computer Science, EPTCS*, 155:9–16, 6 2014. doi: 10.4204/EPTCS.155.2.

S. Lindley and J. G. Morris. Talking bananas: Structural recursion for session types. *ACM SIGPLAN Notices*, 51, 2016. ISSN 15232867. doi: 10.1145/2951913.2951921.

L. Padovani. Deadlock and lock freedom in the linear π-calculus. In T. A. Henzinger and D. Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 72:1–72:10. ACM, 2014. doi: 10.1145/2603088.2603116.

G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1: 125–159, 12 1975. ISSN 0304-3975. doi: 10.1016/0304-3975(75)90017-1.

G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981. URL `http://citeseer.ist.psu.edu/plotkin81structural.html`.

Rust. Influences - the rust reference, 2022. URL `https://doc.rust-lang.org/reference/influences.html`.

V. T. Vasconcelos. Session types for linear multithreaded functional programming. In A. Porto and F. J. López-Fraguas, editors, *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal*, pages 1–6. ACM, 2009. doi: 10.1145/1599410.1599411. URL `https://doi.org/10.1145/1599410.1599411`.

V. T. Vasconcelos. Fundamentals of session types. *Information and Computation*, 217, 2012. ISSN 08905401. doi: 10.1016/j.ic.2012.05.002.

A. L. Voinea, O. Dardha, and S. J. Gay. Resource sharing via capability-based multiparty session types. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11918 LNCS:437–455, 2019. doi: 10.1007/978-3-030-34968-4_24.

P. Wadler. Propositions as sessions. *SIGPLAN Not.*, 47(9):273–286, sep 2012. ISSN 0362-1340. doi: 10.1145/2398856.2364568. URL `https://doi.org/10.1145/2398856.2364568`.

D. Walker and J. G. Morrisett. Alias types for recursive data structures. In R. Harper, editor, *Types in Compilation, Third International Workshop, TIC 2000, Montreal, Canada, September 21, 2000, Revised Selected Papers*, volume 2071 of *Lecture Notes in Computer Science*, pages 177–206. Springer, 2000. doi: 10.1007/3-540-45332-6_7. URL `https://doi.org/10.1007/3-540-45332-6_7`.

G. Winskel. *The formal semantics of programming languages - an introduction.* Foundation of computing series. MIT Press, 1993. ISBN 978-0-262-23169-5.

A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115, 1994. ISSN 10902651. doi: 10.1006/inco.1994.1093.