View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Illinois Digital Environment for Access to Learning and Scholarship Repository

© 2018 Minas Charalambides

ACTOR PROGRAMMING WITH STATIC GUARANTEES

BY

MINAS CHARALAMBIDES

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Doctoral Committee:

Professor Gul A. Agha, Chair
Associate Professor Madhusudan Parthasarathy
Research Professor Elsa L. Gunter
Assistant Professor António Ravara

Abstract

This thesis discusses two methodologies for applying type discipline to concurrent programming with actors: process types, and session types. A system based on each of the two is developed, and used as the basis for a comprehensive overview of process- and session- type merits and limitations. In particular, we analyze the trade-offs of the two approaches with regard to the expressiveness of the resulting calculi, versus the nature of the static guarantees offered. The first system discussed is based on the notion of a *typestate*, that is, a view of an actor's internal state that can be statically tracked. The typestates used here capture what each actor handle *may* be used for, as well as what it *must* be used for. This is done by associating two kinds of tokens with each actor handle: tokens of the first kind are consumed when the actor receives a message, and thus dictate the types of messages that can be sent through the handle; tokens of the second kind dictate messaging obligations, and the type system ensures that related messages have been sent through the handle by the end of its lifetime. The next system developed here adapts session types to suit actor programming. Session types come from the world of process calculi, and are a means to statically check the messaging taking place over communication channels against a pre-defined protocol. Since actors do not use channels, one needs to consider pairs of actors as participants in multiple, concurrently executed—and thus interleaving—protocols. The result is a system with novel, parameterized type constructs to capture communication patterns that prior work cannot handle, such as the sliding window protocol. Although this system can statically verify the implementation of complicated messaging patterns, it requires deviations from industry-standard programming models—a problem that is true for all session type systems in the literature. This work argues that the typestate-based system, while not enforcing protocol fidelity as the session-inspired one does, is nevertheless more suitable for model actor calculi adopted by practical, already established frameworks such as Erlang and Akka.

To my parents, Antonios and Chrystalla – the Greek Cypriot refugees who started their adult lives armed only with education and no physical possessions, yet provided me with a life of privilege.

Τῆς παιδείας αἱ μεν ῥίζαι πικραί εἰσιν, οἱ δὲ καρποὶ γλυκεῖς.
(Ἀριστοτέλης)

Acknowledgments

This has been a long journey—one that I did not travel alone. I will take this opportunity to thank all those who stood by me, and in their own ways, made this thesis possible.

First and foremost, special mention goes to my parents, Antonios and Chrystalla, for their endless and unconditional support. My parents' constant concern and purpose in life is to make sure that neither myself, nor my sister ever have to experience the hardships that they have lived through. Their unconditional support towards us is to the detriment of their own physical and financial health, and for that, I am infinitely grateful. Nothing can replace that kind of support system. Μάμμα, παπά, ευχαριστώ σας πολλά. Οι κόποι απέδωσαν.

Secondly, I would like to express my gratitude to all the wonderful people I met in the United States, who welcomed me to their country with open arms, and made me feel at home. They have undoubtedly provided me with a friendly environment in which to work, live, and be happy—treating me as an equal, and being true friends.

Special thanks go to my wife, Vasundhara Vigh, for transforming herself from an industrious scientist to a traditional housewife, just to make the last stretch of writing this thesis easier for me. I am also very thankful to my sister Niki, for her endless love. Moreover, thank you to all my friends back in Greece, for caring and being as close to me as the day I got on the plane to come to the States.

I feel the need to thank some people from my undergraduate alma mater; namely Associate Professor Vasilis Vassalos, Professor Ioannis Kontoyiannis, and Professor Andreas Veneris, who believed in me, and—I must assume—wrote some great recommendation letters back in 2008. Special thanks go to my undergraduate thesis advisor Vasilis Vassalos, who first got me involved with research, and played a most significant role in the beginning of my academic journey.

Moreover, I would like to express my gratitude to my soon-to-be boss, Dr. Vincent Reverdy, for his patience in waiting for the completion of this thesis. I am also truly thankful to the people who contributed to the papers that gave rise to this material: Dr. Peter Dinges and Dr. Karl Palmskog, who have been wonderful collaborators.

In addition, I would like to thank the members of my doctoral committee, who pushed me hard enough so that this thesis is at least somehow useful to anyone interested in the topic.

Last, but not least, I would like to thank my academic advisor, Professor Gul Agha, for bearing with me all these years. He has played a most significant role in honing my research and writing skills, and has made sure I am ready to take the next step in academia.

Table of Contents

Chapter 1

Introduction

The field of computing has always strived to achieve high performance in terms of speed. Given the physical constraints that limit the maximum circuit density per unit of silicone area, we have now turned to parallel computer architectures [44], with remarkable results. The achieved increase in performance has rendered large-scale numeric calculations practical [94, 140], accelerating research in fields such as cosmology [54] and biology [77]. It has enabled—in terms of tractability—the training of artificial intelligence agents on very large data sets [85], producing systems that can play Atari games [105] and beat human champions in Go [123]. Moreover, the power consumption of processing units tends to scale cubically with their clock frequency, which makes multi-processing architectures fiscally preferable to fewer, power-hungry processing units [82, 83]. It is thus no coincidence that companies such as Google and Yahoo own large parallel computing infrastructures, and spend many man-hours developing parallel algorithms for data processing [34, 42].

Writing correct parallel programs is more difficult than writing sequential code: in addition to common problems such as dereferencing invalid pointers and memory leaks, programmers have to deal with deadlocks, data races and synchronization issues, to name a few. Adopting the actor model of computation [1] alleviates many of these problems: first, actors are concurrent objects, and hence capture parallelism naturally; second, they are fully encapsulated, i.e., they cannot access each other's state directly—thus, data races cannot occur; third, actors communicate with one another via asynchronous message passing, closely resembling the asynchronous nature of the physical world [65]. Because of these traits, and particularly due to their independent nature, actors encourage code modularity and the separation of concerns. In fact, actor-based languages have been used to implement industry-strength, widely used systems, such as the original back-ends of WhatsApp and Facebook chat [92, 130], both written in Erlang [86].

Despite the model's success, there are still concurrency-related problems that are left to the programmer to address. For example, it is difficult to verify that a collection of actors communicate according to some pre-defined specification. Deadlocks are also possible: situations where each actor requires the receipt of a particular message from another actor to make progress, in a cyclic manner. In general, synchronization issues are inherent to concurrent programming, since one needs to reason about how asynchronous events may be ordered, and whether some orderings can cause problems. Avoiding problematic event sequences, and conversely, enforcing the desired ones, is at the heart of what makes concurrency difficult to get right.

This thesis is concerned with countering some types of synchronization issues at compile-time, i.e., statically. We focus on actors in order to capture programming models where the concurrent entities communicate via message-passing—implemented, for example, by MPI [100], Akka [93], and Erlang [86]. This is in contrast to shared-memory programming, where concurrent entities can access shared variables—captured, for example, by Java threads [110]. Given a program with fully encapsulated concurrent processes that exchange messages, we are interested in the static enforcement of two different, yet related types of properties: first, we want to ensure that communication follows a pre-defined pattern exactly; second, we want to guarantee that certain necessary messages are eventually sent to the processes that require them.

Consider the example of two actors communicating via a sliding window protocol [127]. This consists of actor a sending a sequence of messages to actor b, where each message needs to be acknowledged. The protocol dictates that a can keep sending, but only while the number of unacknowledged messages remains below an agreed upon threshold n. If this threshold is reached, then a needs to pause and wait until the *window* of unacknowledged messages drops below n, i.e., one more acknowledgment is received from b. Given an implementation, we seek to answer the question of whether the code corresponds to this protocol. The question is, in other words, whether it is possible that program execution produces a communication trace which violates the given constraints—and we want to know this at compile-time.

While above we discussed strict protocol adherence, this thesis is also concerned with guaranteeing a second type of property: the eventual delivery of required messages. Consider a scenario where a server creates worker processes on-demand, lets them serve requests received from client processes, and then needs to kill each worker process as they finish their jobs. Assuming that the system terminates processes by delivering a kill message, we want to guarantee that the server does indeed send such a message to each dynamically spawned worker. As before, we are concerned with static enforcement: given an implementation, ensure that it is impossible to produce an execution where an idle worker waits for a kill message indefinitely.

In strongly-typed programming languages such as C++ and Java, the compiler is usually tasked with guaranteeing *plug-compatibility*. Consider, for example, typed variable declarations: the compiler ensures that a memory location corresponding to an integer variable never contains a value intended for use as a floating-point number. Allowing so would not be *safe*, because the bitstring representation of floating-point numbers differs from that of integers.

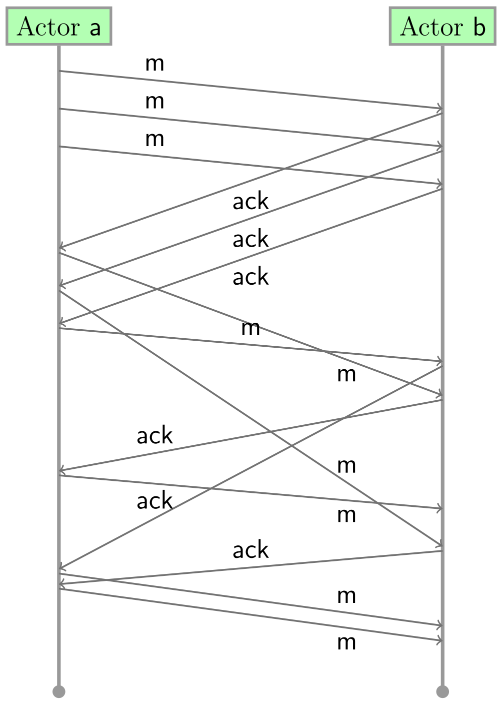
Plug-compatibility can be guaranteed via the use of a suitable *type system*. A type system consists of rules that *typecheck* program statements, i.e., decide whether they can violate a desired property at runtime. For example, to avoid the aforementioned issue of integer vs floating-point bitstring representation, a simple type system would deem an assignment statement to be safe if the left- and right- hand sides of said assignment have the same type. The reason for such checks is that we need to ensure that operations on data work as expected. For example, the hardware does not employ the same algorithm for multiplying integer values, as for multiplying floating-point ones.

It is worth noting that, in reality, we don't need to guarantee that both sides of an assignment statement have the exact same type. Rather, we merely need to ensure that the two sides are *compatible*—meaning that the operations applicable to the variable on the left are also applicable to the data on the right. This is, in essence, the meaning of plug-compatibility: it is safe to replace a piece of data with another that admits at least the same operations. This is known as the Liskov substitutability principle [95, 96].

Plug-compatibility questions arise in the context of concurrent programming as well: given two processes, determine if they can be used interchangeably in a concurrent program. Closely related are issues of *composition*: given a system of concurrently executing processes, determine whether it is safe to compose them with a given external process. In the sequential world, the relevant questions concern data and admissible operations, which can be decided by looking at the types of said data. However, the problem is more difficult when considering processes, because processes evolve—and with them, so do their types. Consider the example of a single-cell buffer, which can receive a set message, making it full, and an unset message, making it empty. The buffer's type thus evolves as program execution progresses. As it turns out, it is still possible to use a type system to decide composition in this context. In fact, in his seminal 1993 paper, Honda [67] proposed a type system to check communication compatibility for pairs of concurrent processes: a type is assigned to each process, describing the communication interactions that it is allowed to engage in. Typechecking then requires that two processes that exchange messages have *dual* types, meaning that each process expects precisely the message sequence that the other sends.

Plug-compatibility is an example of a *safety property*. Intuitively, a safety property can be understood as the absence of a specific undesired situation: the fact that execution will never enter a pre-defined set of bad states. As such, the previously described sliding window example is one where we require a safety property too: that of the messaging pattern not deviating from the protocol. As it turns out, this property can also be enforced with a type system. In fact, Takeuchi et al. [126] do exactly that for concurrent processes that communicate over channels: a type is assigned to each channel, describing the patterns of messages it is allowed to carry. Assuming suitable restrictions on the class of protocols we are interested in, looking at the relevant use sites suffices to decide whether each channel is used in accordance to its type.

Figure 1.1: Two actors communicating via a sliding-window protocol, with a window size of 3. Actor a can only initiate the sending of an m message if the total number of acks it has received is within 2 of the number of ms it has sent.



While our work is based on the above ideas, our focus is on actors, where channels do not exist: actors can address each other by name, and communicate via asynchronous message-passing. This thesis looks at types as communication protocol specifications, and presents a typing methodology that decides adherence to such specifications by looking at the behavior of individual actors. The most important extension over other similar works is the treatment of cases that combine asynchrony with parameterization over the allowed message interleavings. In the example of the sliding window protocol, shown in Figure 1.1, the size of the window determines the allowed interleavings of messages and acknowledgments: no more than n messages can be unacknowledged at any given time, and up to a combined total of n messages and acknowledgments can arrive to their destination in any order. In fact, prior work cannot capture such complex interactions (see page 65) unless the value of n is known at compile-time.

We have thus far discussed safety properties, i.e., the continuous absence of a "bad" situation. Orthogonally, a *liveness* property holds if something "good" happens *eventually*. The most commonly treated liveness property is that of *progress*: the requirement that each process takes the next intended step, eventually. The term "next intended step" leaves room for interpretation, and many flavors of progress have been explored in the literature—the most common being deadlock-freedom. Works in this area differ with regard to their assumptions on the allowed communication patterns. For example, Kobayashi et al. [81] propose a type system that looks at the order of sending and receiving actions on communication channels, and detects cyclic communication dependencies—but only if the channels are designated for use in one of a few, restricted ways. Honda et al. [70] assume a programming model with concurrent processes that can simultaneously participate in multiple communication protocols (a.k.a. *sessions*), and propose a type system that guarantees the absence of cyclic communication dependencies on each session.

This thesis looks at progress from a higher-level perspective: we are interested in allowing the programmer to designate important messages, and letting the type system guarantee, statically, that those messages will be sent to their destinations. For instance, in the previously described scenario of a server spawning workers to process requests in parallel, we want to guarantee that the server eventually does send a kill message to each of the spawned workers. The main contribution of this thesis with regard to guaranteeing progress is that we allow the programmer to define the meaning of progress in a simple, intuitive manner: a special library call is used to add requirements to an actor, and the typing guarantees, statically, that all requirements generated at runtime are eventually satisfied. It is worth noting that this methodology can be easily incorporated in a mainstream actor framework, since type systems can run on top of existing compilers. Moreover, since the requirement generation call is only intended for use by the type checker, it can have an empty implementation, and hence impose no run-time performance penalty.

Thesis Outline

We discuss background material in Chapter 2, where we begin with a formal definition of the actor model of computation, used in the rest of this thesis. In this context, we touch upon the notion of *fairness*, which will be useful in Chapter 4. We then proceed to discuss the topic of *session types*, originally introduced to guarantee safety and liveness properties in channel-based communication. This thesis discusses actor programming, which does not use channels; nonetheless, our proposal for the enforcement of protocols such as the sliding window falls in the category of session type methodologies, since we look at protocols from a global, whole-system perspective. We then proceed to introduce *typestates* and how these relate to *process types*. The latter are designed to guarantee properties of specific concurrent processes, which is in contrast to guaranteeing properties of communication protocols as a whole. Our treatment of cases such as a server being required to send kill messages to spawned workers is based on process types, because we associate a kill requirement with each worker, and there is no explicit mention of a specific protocol satisfying these requirements.

We proceed with an overview of related work in Chapter 3. The chapter discusses work on session types, as it evolved over time to be able to statically verify increasingly complex protocols. We discuss the introduction of type parameters to these methodologies, and the gradual lifting of restrictions on programmability. In this spirit, the chapter discusses approaches based on process types, and how the idea deviates from work on session types: it seems more intuitive to program systems without the explicit mention of protocols—rather, it is enough to specify the properties desired to hold always, i.e., safety, and those that the programmer wants satisfied eventually, i.e., liveness, on a per-process basis.

Chapter 4 proposes process types for the static guaranteeing of eventual message delivery. We motivate our approach with examples, such as the aforementioned case of a server spawning and killing workers dynamically; then, we proceed to introduce the type system that makes such examples work. We show that our methodology can be applied to a pure actor calculus, with limitations only on the class of programs that can be statically verified. After giving formal proofs for our claims, we discuss possible extensions. The chapter concludes by incorporating an important approach on coordination constraints from the literature into our system.

Chapter 5 details our methodology for guaranteeing the adherence of concurrent programs to pre-defined protocols. We first motivate our approach with examples that prior work cannot handle. Then, we present novel type constructs that capture complex cases such as the sliding window protocol, and prove the correctness of our algorithms. After reviewing possible extensions, the chapter concludes with an in-depth discussion of the deviations from the actor model that our system requires. Transferring results from the session type literature into an actor setting reveals limitations that are inherent in the original ideas underpinning session types—they simply transfer over to actors in a different form.

While our typestate-based system imposes limitations only in the shape of programs that can be statically verified, our session-based system deviates from the actor model on the level of the calculus itself. We therefore conclude the thesis in Chapter 6 with a discussion on the practical applicability of the typestate- and session- type approaches, both for our own systems, and with regard to related work. We offer a critique on the way related work handles issues underlying the practical applicability of session types, and conclude that an approach similar to the one in Chapter 4 aligns better with widely used programming frameworks.

Chapter 2

Background

This chapter introduces basic terminology, and presents the core ideas built upon in the rest of the thesis. We begin with an overview of the actor model of computation, and make our assumptions on fairness precise. We then introduce the basic ideas underlying session type systems, and discuss how they can be used to ensure that communication channels are used correctly. We initially present session types for the π-calculus, and then proceed to discuss the differences in application with regard to actors. At that point, we justify our choice of the actor model as opposed to the asynchronous π-calculus. Finally, the chapter introduces typestates, and concludes with a discussion over their connection with concepts from the session type literature.

Notation

We abbreviate (possibly empty) sequences of the form $x_1 \dots x_k$ with $\overline{x}$, sequences of the form $u_1 \dots u_k$ with $\overline{u}$, et cetera. We write $\{x \mapsto \alpha\}$ for the function that maps x to α, and $f[x \mapsto \alpha]$ for the result of altering the function f such that it maps x to α. Assuming $\mathit{dom}(f_1) \cap \mathit{dom}(f_2) = \emptyset$, we write $f_1 \cup f_2$ for the mapping for which $(f_1 \cup f_2)(x) = f_i(x)$ when $x \in \mathit{dom}(f_i)$, $i \in \{1, 2\}$. We denote the capture-avoiding substitution of x for y in P with $P[x/y]$. This is extended to sequences such that $P[\overline{x}/\overline{y}]$ is the result of the capture-avoiding simultaneous substitution of $\overline{x}$ for $\overline{y}$ in P, where $\overline{x}$ and $\overline{y}$ are sequences of the same length.

Our notation conventions include variants of $\longrightarrow$ for reduction relations, and $\equiv$ for congruences. For $k \geq 0$, we write $\longrightarrow^k$ to denote k-step reductions, and $\longrightarrow^*$ for the relation $\bigcup_{k=0}^{\infty} \longrightarrow^k$. We write $X\longrightarrow$ iff there exists X' such that $X \longrightarrow X'$, and $X\not\longrightarrow$ iff there is no X' such that $X \longrightarrow X'$. Additionally, we use $\rightsquigarrow$ for "complete" reductions, i.e., $X \rightsquigarrow X'$ iff $X \longrightarrow^* X'$ and $X' \not\longrightarrow$. We reserve the symbol $\Longrightarrow$ for logical implications.

We use a double-edged arrow for communication actions, such that $x \xrightarrow{m} y$ denotes the sending of message m from x to y. Note that m can be a value or a type, and the meaning will be clear from context. We overload this notation so that constructs of the form $x \xrightarrow{m} y$ can be in message buffers. If a buffer contains $x \xrightarrow{m} y$, then m is waiting to be delivered to y.

2.1 Actors

This section provides an introduction to the actor model of concurrent computation [1, 3, 65]. While this section discusses "pure" actors, subsequent chapters will discuss variants with restrictions. These will be made clear as needed.

Actors combine object-oriented programming with concurrency: rather than using synchronous method calls, actors communicate by sending asynchronous messages to each other. As in object-oriented programming, actors have state, i.e., internal variables. However, directly accessing one-another's state is not allowed; in other words, actors offer full encapsulation: information is only communicated by sending each other messages.

Actors can be in one of two states during program execution: (a) busy, meaning that the actor had previously received a message and is now executing the code associated with it; and (b) idle, meaning that the actor is currently not processing a message, and is capable of receiving the next message sent to it. Execution of actors proceeds concurrently, meaning that at each step, the scheduler can either (i) choose a busy actor and execute the next instruction available to it; or (ii) deliver a message directed to an idle actor, bringing that actor to the busy state. Note that this description implies that message processing is non-preemptive: once an actor starts processing a message, it does so to completion before accepting the next one. Additionally, there is no restriction on the order of message delivery: messages can arrive out of (sending) order, even when having the same sender and recipient.

Figure 2.1: Actors as independent, fully-encapsulated, active, concurrent objects.

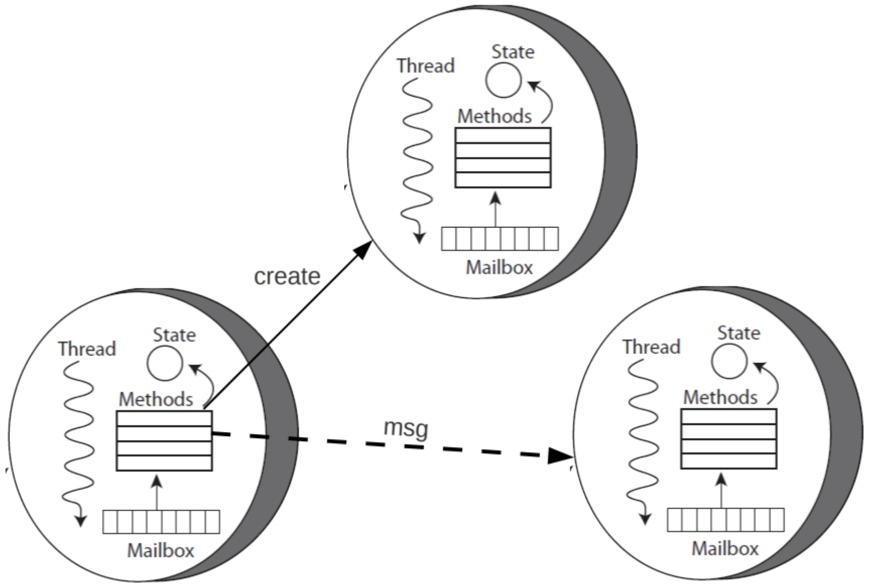


Figure 2.1 shows some co-existing actors in a hypothetical system, assuming a naive implementation in a language with thread support. Each actor has an infinite-length mailbox, where messages sent to the actor are placed. Each actor is further associated with an independent control thread, which removes messages from the mailbox, and passes them to one of the actor's methods for processing, one at a time, in a non-preemptive manner; that is, once an actor starts processing a message, it does so to completion before picking the next one from the mailbox. Note that while Figure 2.1 helps conceptualize the *semantics* of the model, this thesis does not employ the *implementation* implied by the figure.

The actor model has been formalized as an extension of the λ-calculus with primitives for message sending, actor creation, and behavior assumption. In this context, an actor is essentially a closure that is applied to the next available message. This model is detailed in the work of Agha et al. [3], and is at the core of Erlang [86], arguably the most widely adopted actor programming language.

2.1.1 Syntax

We deviate slightly from the actors-as-closures model, and allow behaviors to include a set of message handlers. The intention here is the following: consider an actor α with behavior b, where the definition of b includes a handler h with parameters $x_1 \ldots x_k$ and body $\texttt{S}$. Then, the receipt of a message $h(u_1 \ldots u_k)$ by actor α will invoke the code $\texttt{S}$, replacing the formal parameters $x_1 \ldots x_k$ with the values $u_1 \ldots u_k$, and **self** with α. The reserved name **self** refers to the actor in which it is evaluated. The calculus syntax is given in Figure 2.2: programs $\texttt{P}$ consist of a list of behavior definitions $\overline{\texttt{B}}$ and an initial statement $\texttt{S}$. An actor behavior definition $\texttt{B}$ includes a name b that identifies the behavior, variables $\overline{x}$ that store the assuming actor's state, and a list of message handler definitions $\overline{\texttt{H}}$. In turn, a message handler definition $\texttt{H}$ includes a name h that identifies the handler, a list of message parameters $\overline{x}$, and a statement $\texttt{S}$ to be executed upon invocation of the handler.

Statements generally consist of single operations followed by another statement. For example, $\texttt{x}!h(\overline{e}).\texttt{S}$ sends a message for handler h of the actor $\texttt{x}$, with argument list $\overline{e}$, and then proceeds as $\texttt{S}$. The statement $\nu x{:}b(\overline{e}).\texttt{S}$ creates a new actor (whose name is bound to $\texttt{x}$ in $\texttt{S}$) with behavior b and initial state variables set to the values of the expressions $\overline{e}$. The statement **update**$(\overline{e})$ updates the values of the actor state variables, and the **if** statement has the usual meaning of a conditional. A **ready** statement belongs to the runtime syntax, signifying the end of handler execution.

Figure 2.2: Actor calculus syntax.

$\mathtt{P}$	$::=$	$\overline{\mathtt{B}}\ \mathtt{S}$		
$\mathtt{B}$	$::=$	$\textbf{bdef}\ b(\overline{x}) = \{\overline{\mathtt{H}}\}$	$b \in$ behavior names	
$\mathtt{H}$	$::=$	$\textbf{hdef}\ h(\overline{x}) = \mathtt{S}$	$h \in$ handler names	
$\mathtt{S}$	$::=$	$\mathtt{x}!h(\overline{e}).\mathtt{S}$		
	$\mid$	$\textbf{if}\ e\ \textbf{then}\ \mathtt{S}_1\ \textbf{else}\ \mathtt{S}_2$	$e \in$ expressions (values, function calls, etc.)	
	$\mid$	$\nu x{:}b(\overline{e}).\mathtt{S}$	actor creation	
	$\mid$	$\textbf{update}(\overline{e})$	state update	
	$\mid$	$\textbf{ready}$		[runtime syntax]
$\mathtt{x}$	$::=$	$\textbf{self} \mid x, y, z, \ldots \mid \alpha, \beta, \ldots$	$x, y, z, \ldots \in$ variables	
			$\alpha, \beta, \ldots \in$ actor names	[runtime syntax]
C	$::=$	(Δ, M, A)	configuration	[runtime syntax]
Δ	$::=$	program information		[runtime syntax]
M	$::=$	$\{\alpha_1!h_1(\overline{u}_1) \ldots \alpha_\kappa!h_\kappa(\overline{u}_\kappa)\}$	multiset of pending messages	[runtime syntax]
A	$::=$	$\{\langle \mathtt{S}_1\rangle_{\alpha_1}^{b_1(\overline{w}_1)} \ldots \langle \mathtt{S}_\kappa\rangle_{\alpha_\kappa}^{b_\kappa(\overline{w}_\kappa)}\}$	actor map	[runtime syntax]
			$u, w \in$ values	

2.1.2 Operational Semantics

As is common in the formalization of process calculi [3, 104], program execution is described by a *labeled transition semantics* [106]. The semantics consists of named rules which apply to program configurations that appear during execution: when a rule's preconditions are satisfied, the rule transforms the current configuration into the next one—thus defining execution as a sequence of rule applications from an initial program state. For an actor program $\mathtt{P}$, we write $\mathit{init}(\mathtt{P})$ to denote this initial state, also referred to as the initial configuration.

We assume a reduction relation on expressions, such that the notation $e \rightsquigarrow_\Delta u$ means that expression e reduces to the value u, given static program information Δ. The latter is assumed to contain information extracted from the program, such as the parameters of message handlers. The transition relation for statements is defined in Figure 2.3, where we write $\mathtt{S} \xrightarrow{l}_\Delta \mathtt{S}'$ to say that a statement $\mathtt{S}$ reduces to $\mathtt{S}'$ via l. The label l records the action being taken; for example, $\alpha!h(\overline{e}).\mathtt{S}$ reduces to $\mathtt{S}$, and $l = \alpha!h(\overline{u})$ records the sent message. The values $\overline{u}$ are computed from the expressions $\overline{e}$, i.e., $\overline{e} \rightsquigarrow_\Delta \overline{u}$.

The transition relation $\mathtt{S} \xrightarrow{l}_\Delta \mathtt{S}'$ is referenced in the program-level rules of Figure 2.4, which transform runtime configurations. A runtime configuration C is a tuple (Δ, M, A), where Δ records static program information, M is the multiset of pending (sent, but not

Figure 2.3: Labeled transition semantics for statements. Expressions follow standard semantics, and Δ is static program information.

$$\frac{\overline{e} \rightsquigarrow_\Delta \overline{u} \qquad l = \alpha!h(\overline{u})}{\alpha!h(\overline{e}).\mathsf{S} \xrightarrow{l}_\Delta \mathsf{S}}$$

$$\frac{\alpha \ \ \mathit{fresh} \qquad \overline{e} \rightsquigarrow_\Delta \overline{u} \qquad l = \alpha{:}b(\overline{u})}{\nu x{:}b(\overline{e}).\mathsf{S} \xrightarrow{l}_\Delta \mathsf{S}[\alpha/x]} \qquad \frac{\overline{e} \rightsquigarrow_\Delta \overline{u} \qquad l = \textsf{\textbf{update}}(\overline{u})}{\textsf{\textbf{update}}(\overline{e}) \xrightarrow{l}_\Delta \textsf{\textbf{ready}}}$$

$$\frac{e \rightsquigarrow_\Delta \textsf{\textbf{true}}}{\textsf{\textbf{if}}\ e\ \textsf{\textbf{then}}\ \mathsf{S}_1\ \textsf{\textbf{else}}\ \mathsf{S}_2 \xrightarrow{\textsf{\textbf{if}}}_\Delta \mathsf{S}_1} \qquad \frac{e \rightsquigarrow_\Delta \textsf{\textbf{false}}}{\textsf{\textbf{if}}\ e\ \textsf{\textbf{then}}\ \mathsf{S}_1\ \textsf{\textbf{else}}\ \mathsf{S}_2 \xrightarrow{\textsf{\textbf{if}}}_\Delta \mathsf{S}_2}$$

Figure 2.4: Labeled transition semantics for actor configurations.

$$\textsc{Send}_{\alpha,\beta,h}\ \frac{\mathsf{S} \xrightarrow{\beta!h(\overline{u})}_\Delta \mathsf{S}'}{\Delta,\ M,\ A \cup \{\langle \mathsf{S} \rangle_\alpha^{b(\overline{w})}\} \longrightarrow \Delta,\ M \cup \{\beta!h(\overline{u})\},\ A \cup \{\langle \mathsf{S}' \rangle_\alpha^{b(\overline{w})}\}}$$

$$\textsc{Receive}_{\alpha,\beta,h}\ \frac{\mathsf{S} = \mathit{body}(\Delta, h) \qquad \overline{y} = \mathit{params}(\Delta, h) \qquad \overline{x} = \mathit{params}(\Delta, b)}{\Delta,\ M \cup \{\alpha!h(\overline{u})\},\ A \cup \{\langle \textsf{\textbf{ready}} \rangle_\alpha^{b(\overline{w})}\} \longrightarrow \Delta,\ M,\ A \cup \{\langle \mathsf{S}[\alpha/\textsf{\textbf{self}}][\overline{u}\,\overline{w}/\overline{y}\,\overline{x}] \rangle_\alpha^{b(\overline{w})}\}}$$

$$\textsc{Update}_{\alpha}\ \frac{\mathsf{S} \xrightarrow{\textsf{\textbf{update}}(\overline{u})}_\Delta \mathsf{S}'}{\Delta,\ M,\ A \cup \{\langle \mathsf{S} \rangle_\alpha^{b(\overline{w})}\} \longrightarrow \Delta,\ M,\ A \cup \{\langle \mathsf{S}' \rangle_\alpha^{b(\overline{u})}\}}$$

$$\textsc{New}_{\alpha}\ \frac{\mathsf{S} \xrightarrow{\beta{:}b(\overline{u})}_\Delta \mathsf{S}'}{\Delta,\ M,\ A \cup \{\langle \mathsf{S} \rangle_\alpha^{b_\alpha(\overline{w})}\} \longrightarrow \Delta,\ M,\ A \cup \{\langle \mathsf{S}' \rangle_\alpha^{b_\alpha(\overline{w})},\ \langle \textsf{\textbf{ready}} \rangle_\beta^{b(\overline{u})}\}}$$

$$\textsc{If}_{\alpha}\ \frac{\mathsf{S} \xrightarrow{\textsf{\textbf{if}}}_\Delta \mathsf{S}'}{\Delta,\ M,\ A \cup \{\langle \mathsf{S} \rangle_\alpha^{b(\overline{w})}\} \longrightarrow \Delta,\ M,\ A \cup \{\langle \mathsf{S}' \rangle_\alpha^{b(\overline{w})}\}} \qquad \textsc{Prog}\ \frac{\Delta = \mathit{info}(\overline{\mathsf{B}})}{\overline{\mathsf{B}}\,\mathsf{S} \longrightarrow \Delta, \emptyset, \{\langle \mathsf{S} \rangle_{\mathit{in}}^{\mathit{in}()}\}}$$

received) messages, and A contains running actors. Elements of M have the form $\alpha!h(\overline{u})$, where α is the destination actor, h is the handler to be invoked upon receipt, and $\overline{u}$ are values constituting the message payload. A maps each actor name to a behavior, state, and executing statement. We denote A as a set of elements of the form $\langle \mathtt{S} \rangle_{\alpha}^{b(\overline{w})}$, where α is the actor's name, b corresponds to its behavior, the values $\overline{w}$ constitute its state, and $\mathtt{S}$ is the statement the actor is currently executing.

We write $C \longrightarrow C'$ to say that the runtime configuration C reduces to C' via an application of some rule in Figure 2.4. Execution of a program $\mathtt{P} = \overline{\mathtt{B}}\,\mathtt{S}$ then consists of a sequence of transformations that starts from the program's initial configuration. Such an initial configuration is created via rule Prog in Figure 2.4, and it records information Δ from the program, has an empty message multiset, and includes a single initial actor executing $\mathtt{S}$. This actor has reserved name and behavior *in*, and no state variables. Note that the subscripts in the rule labels are there to identify rule applications uniquely, which will be useful in Section 2.1.3. The current discussion will ignore them for the time being.

Rule Send adds the sent message $h(\overline{u})$ to the multiset of pending messages. Rule Update writes new values $\overline{u}$ to the state variables of α. Rule New creates a new actor $\langle \textbf{ready} \rangle_{\beta}^{b(\overline{u})}$ with unique name β, initialized with the given behavior b and values $\overline{u}$ for state variables. Rule If has the usual effect of deciding a conditional.

Only idle actors can receive messages [2]. Since statements take the form **ready** when completely reduced, Receive describes an idle actor α receiving a message to be processed by handler h. The statement $\mathtt{S}$ to execute is extracted from the program information Δ, and on it, the rule performs substitution of current values for handler and state variables. These values are taken from the message contents $\overline{u}$ and actor state $\overline{w}$. Handler and behavior (i.e., state) parameters are looked up via the auxiliary function *params*.

2.1.3 Executions and Fairness

In a labeled transition semantics such as that presented in the previous section, each application of a rule is associated with a transition label t. For this purpose, the rule labels in Figure 2.4 have subscripts that uniquely identify the specific transition; for example, $\textsc{Send}_{\alpha,\beta,h}$ records the sender α, the receiver β, and the message h. We write $C \xrightarrow{t} C'$ to say that the transition rule t transforms the runtime configuration C to C'. By extension, we write $C \xrightarrow{t_1} C_1 \xrightarrow{t_2} \cdots \xrightarrow{t_k} \cdots$ to denote a possibly infinite sequence of configurations where each adjacent pair follows the transition rules. Such a sequence is called an *execution*.

For a given runtime configuration, multiple rules can be enabled (i.e., their preconditions satisfied). We write *enabled*(C) for the set of labels of those transition rules that are applicable on the runtime configuration C. Assuming an execution from C has reached a configuration C_i, it is possible that $|\mathit{enabled}(C_i)| > 1$. Thus, starting from C, one can draw a tree of possible executions. Upon encountering a branching point, the system makes a non-deterministic choice as to which rule to apply. Such non-determinism at each computation step does not guarantee *fairness* [55], which we require in Section 4.5. We therefore do not consider all sequences of rule applications admissible, and demand *fair* executions, defined in a way reminiscent of the work of Agha et al. [3].

Definition 2.1 (Fair Execution)**.** Let C be a runtime configuration, and assume some execution $s = C_1 \xrightarrow{t_1} C_2 \xrightarrow{t_2} \cdots$ starting from C, i.e., $C = C_1$. We say that s is *fair* when for every $C_i \in s$ it is true that for all $t \in \mathit{enabled}(C_i)$, there exists some k with $k \geq i$ s.t. either $t = t_k$, or $t \notin \mathit{enabled}(C_j)$ for all $j > k$. We write $\mathbb{F}(C)$ to denote the set of fair executions originating from C.

Intuitively, (weakly) fair executions are those where no transition label becomes continuously enabled without the associated rule ever being applied to trigger the transition. In other words, under fairness, an enabled transition either occurs eventually, or becomes permanently disabled after a finite number of steps. Observe that in our system, once a transition is enabled, it remains enabled until taken.

2.2 Session Types

Session types have been proposed as a means to apply type discipline to the channel-based communication of concurrent processes. This section offers an introduction to the basic concepts from the literature; for a comprehensive review of the area, we refer the interested reader to the work of Hüttel et al. [74].

Most works in this domain use a variant of Milner's π-calculus [103], which is a process algebra that allows the definition and concurrent composition of processes that send and receive names. For example, we write $c!a\,;P$ for a process that sends the name a via the channel c, and then proceeds as P. Conversely, receiving a value via c, then proceeding as P', is written $c?x\,;P'$. The concurrent composition of these two is written $(c!a\,;P)\,\|\,(c?x\,;P')$, which reduces to $P\,\|\,(P'[a/x])$. In the latter, x is replaced by a in P', to capture the fact that the name a was received via c.

We use $\nu c.P$ for scope restriction, so that c is only visible in P. This way, for example, $\nu c.(c!a\,;P)\,\|\,\nu c.(c?x\,;P')$ does not perform any communications, because c on the left is effectively not the same channel as c on the right. In fact, the above would reduce to $(c_1!a\,;P)\,\|\,(c_2?x\,;P')$ for some fresh names c_1 and c_2. Note that the result $(c_1!a\,;P)\,\|\,(c_2?x\,;P')$

cannot be reduced any further, i.e., cannot make *progress*. This is a prime example of the class of errors that session type systems are designed to prevent.

Session type calculi associate each channel with a type, so that one can write $(\nu c\!:\!t).P$ to bind the name c to a fresh channel in P, with this new channel having type t. The typechecker is then responsible for verifying that P uses the channel c as per its type, t. For example, consider the program $(\nu c : P_1 \xrightarrow{\text{Int}} P_2).(P_1 \,\|\, P_2)$ with $P_1 = c!1$ and $P_2 = c?x$. It is easy to see that c is indeed used to send an integer value from P_1 to P_2, which is exactly what the type $P_1 \xrightarrow{\text{Int}} P_2$ prescribes.

In general, session type syntax follows that of processes: $P_1 \xrightarrow{m} P_2\,;T$ is a type that describes the action of sending a message of type m from P_1 to P_2, then proceeding as T. Some calculi allow process definitions to be recursive, as in, for example,

$$P_1 \overset{\text{def}}{=} c_1?x\,;c_2!x\,;P_1$$
$$P_2 \overset{\text{def}}{=} c_1!1\,;c_2?x\,;P_2$$

where P_1 receives a value from channel c_1, sends it through c_2, and repeats in the same fashion. P_2 performs the complementary actions of sending the value 1 on c_1, receiving it back on c_2, and repeating indefinitely. In such calculi, session types can take a recursive form as well. Given the concurrent composition of P_1 with P_2, i.e., $P_1 \,\|\, P_2$, the type of c_1 is given by

$$T_1 \overset{\text{def}}{=} (P_2 \xrightarrow{\text{Int}} P_1)\,;T_1$$

that is, the action of P_2 sending an integer value to P_1 over and over again.

2.2.1 Projection

In the previous examples, the communication protocol was given from a global perspective, meaning that the type mentions the actions of all participants. Such a session type is often called a *global* type, since it describes the entire protocol. In order to localize typechecking, i.e., typecheck the code of each process separately, we need to consider the actions that a session type implies for each protocol participant. Given a session type t and a participant p, we write $t \rhd p$ to denote the *projection* of t onto p, i.e., the *local* type associated with p. Such a local type can then be checked against the participant's code in a straightforward manner.

Consider the previous example of $P_1 \,\|\, P_2$ and the global type associated with c_1, that is, $T_1 \overset{\text{def}}{=} (P_2 \xrightarrow{\text{Int}} P_1)\,;T_1$. Then the projection of T_1 onto P_1, i.e., $T_1 \rhd P_1$ gives the local type L_1

with $L_1 \overset{\text{def}}{=} P_2?\text{Int}\,;L_1$. This type describes the interactions over c_1 from the perspective of P_1 exactly: it repeatedly receives an integer value from P_2.

2.2.2 Multi-party Session Types

So far we have treated protocols with two participants, where we simplify the presentation by involving the names P_1 and P_2 in the session type syntax. However, processes in the π-calculus are anonymous; what makes typechecking possible in this context is that, in general, two-party communication can be dealt with by ignoring the participant's name. For example, the local type L_1 above does not need to mention P_2 explicitly, i.e., which process placed the integer on the channel. Since L_1 describes the communication restrictions that the type of c_1 places on P_1, all that matters in this example is to verify that P_1 engages in the continuous receipt of an integer from the channel. This describes the interaction from the point of view of P_1 exactly; moreover, since only two processes are involved, the precise identity of the sender is irrelevant. We can thus write $L_1 \overset{\text{def}}{=} ?\text{Int}\,;L_1$, which contains all the information necessary to typecheck the code of P_1 with respect to its use of channel c_1. In the same manner, the projection of the same channel's type onto P_2 is defined as $L_2 \overset{\text{def}}{=} !\text{Int}\,;L_2$. Note that from these two definitions, one can easily reconstruct T_1, i.e., the session type associated with c_1.

The situation is more complicated when more than two participants are involved, because we can no longer ignore participant names. Since the π-calculus does not name processes, but does name channels, additional syntax is employed in order to enable projection. A session over channels $\overline{c}$ is initialized with

$$\mathsf{request}\ s[1..n](\overline{c})\ \mathsf{in}\ P_1$$

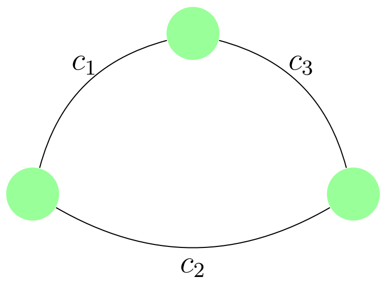
which binds the names $\overline{c}$ in the remaining actions P_1. Here, s is a name that uniquely identifies the initialized session, and enables other processes to join. The number n denotes the total number of participants, who can join with

$$\mathsf{accept}\ s[k](\overline{c})\ \mathsf{in}\ P_k$$

which binds the names $\overline{c}$ in P_k. Note the role of the natural number k, which is used to uniquely identify the joining process. The session initiator is automatically assigned the number 1, and so $1 < k \leq n$.

These syntactic extensions enable the static association of each process with a natural number, which is made explicit at the points where request and accept are used. For example,

Figure 2.5: Three processes connected with channels pairwise.



consider three processes arranged in a circle, as in Figure 2.5. Each pair of processes is connected with a distinct channel, and the program passes the number 0 around the circle once. This program can be written as

$$\begin{aligned}&\mathsf{request}\ s[1..3](c_1, c_2, c_3)\ \mathsf{in}\ c_1!0\,;\, c_3?x\,;\,\mathsf{end}\\ \|\ &\mathsf{accept}\ s[2](c_1, c_2, c_3)\ \mathsf{in}\ c_1?x\,;\, c_2!x\,;\,\mathsf{end}\\ \|\ &\mathsf{accept}\ s[3](c_1, c_2, c_3)\ \mathsf{in}\ c_2?x\,;\, c_3!x\,;\,\mathsf{end}\end{aligned}$$

where the first process sends 0 along c_1 to be received by the second process, which in turn sends it along c_2 to be received by the third process, who sends it back to the first one via c_3. Assuming this is the intended behavior, one can describe it with the following association of types to channels:

$$\begin{aligned}c_1 &: 1 \xrightarrow{\text{Int}} 2\\ c_2 &: 2 \xrightarrow{\text{Int}} 3\\ c_3 &: 3 \xrightarrow{\text{Int}} 1\end{aligned}$$

Now assume we are interested in verifying that the actions on c_2 follow the above specification; that would involve: (i) projecting the type $2 \xrightarrow{\text{Int}} 3$ onto each participant, and (ii) verifying that the code of each participant conforms to the resulting local type. The first step gives the empty projection for participant 1, since participant 1 does not appear in the type. On the other hand, $(2 \xrightarrow{\text{Int}} 3) \triangleright 2 = !\text{Int}$ and $(2 \xrightarrow{\text{Int}} 3) \triangleright 3 = ?\text{Int}$. From the code, we can see that the process that begins with $\mathsf{accept}\ s[2]$ indeed sends an integer over c_2, and similarly, that the process that begins with $\mathsf{accept}\ s[3]$ indeed receives an integer over c_2. These match the projected types !Int and ?Int exactly, and it is easy to verify that this is the case for the other two channels as well—implying that the program adheres to the specified protocol.

2.2.3 Session Types for Actors

In Chapter 5, we are concerned with applying session-type discipline to actor programming. Actors differ from the calculi discussed above: first, there are no channels to type; actors are named entities, and one can write $a!x$ to mean sending the message x to the actor whose name is bound to a. Second, communication in actors is asynchronous, i.e., sent messages can arrive to their destinations later, and the sending actor can continue with the next instruction before the message arrives.

Since we are not typing channels, the purpose of a session type essentially becomes to describe the entire communication protocol among a system of actors. We do away with session initiation and acceptance, and so, all actors in a program are participants to the globally specified protocol. For instance, in an actor setting, communication for the example of page 17 cannot be captured by three separate channel types; rather, the example would involve three actors a, b and c, and their communication pattern can be fully described by the type $a \xrightarrow{\text{Int}} b\,;\, b \xrightarrow{\text{Int}} c\,;\, c \xrightarrow{\text{Int}} a$.

In this setting, we deal with *global* and *local* types; the former describe the protocol that the program must adhere to, and the latter describe the constraints that such a protocol implies on the individual participants. The type $a \xrightarrow{\text{Int}} b\,;\, b \xrightarrow{\text{Int}} c\,;\, c \xrightarrow{\text{Int}} a$, mentioned above, is a global type. Its projection onto a, that is, $(a \xrightarrow{\text{Int}} b\,;\, b \xrightarrow{\text{Int}} c\,;\, c \xrightarrow{\text{Int}} a) \triangleright a$, is the local type $b!\text{Int}\,;\, c?\text{Int}$. Similarly, the local types for actors b and c are given by

$$(a \xrightarrow{\text{Int}} b\,;\, b \xrightarrow{\text{Int}} c\,;\, c \xrightarrow{\text{Int}} a) \triangleright b = a?\text{Int}\,;\, c!\text{Int}$$
$$(a \xrightarrow{\text{Int}} b\,;\, b \xrightarrow{\text{Int}} c\,;\, c \xrightarrow{\text{Int}} a) \triangleright c = b?\text{Int}\,;\, a!\text{Int}$$

As before, the purpose here is to check the derived local types against the the program code for a, b and c. For a discussion on the issues encountered when applying session type discipline to actors, we refer the interested reader to the short paper by Masini and Francalanza [97].

Why not the asynchronous π-calculus?

The asynchronous π-calculus [20] uses the constructs introduced at the start of Section 2.2. The main difference lies in the semantics of messaging: in the asynchronous case, a message send is a non-blocking action, and does not require a matching receive to proceed. For example, the process

$$(\nu c).(c!1\,;\, c!2\,;\, \mathsf{end})$$

can reduce to end with the values 1 and 2 “floating” in the channel, waiting (in this case, forever) for a process to receive them. In contrast, in the synchronous case, the above code would block indefinitely right after creating the channel, since no other process knows c, and thus cannot issue a receive action on it.

One could indeed choose the asynchronous π-calculus to demonstrate the ideas in this thesis. The actor model was chosen because the author feels it offers a more natural programming style, where the decomposition of a system into concurrent entities allows for the direct addressing of those entities, by name. This is in contrast to the π-calculus, where processes communicate via channels. Since actors can be addressed directly, the role of constructs such as request and accept is mute—circumventing the technicalities addressed on page 16.

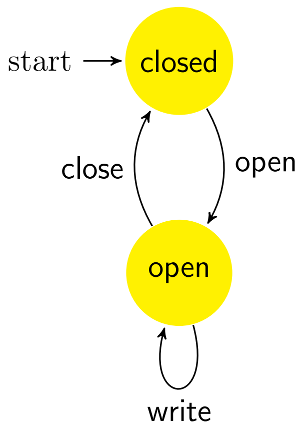
Furthermore, session types for the π-calculus essentially consist of a two-level specification: the conceptual level of the protocol as a whole, and the type of each channel. For instance, in the case of Figure 2.5 one needs to think both in terms of the protocol "each process passes an integer to the next one", and in terms of what that means for the individual channels—resulting in three channel types, which are then projected to the participating processes. In contrast, as discussed in Section 2.2.3, the actor case only requires the specification of one global type: $a \xrightarrow{\mathsf{Int}} b\,; b \xrightarrow{\mathsf{Int}} c\,; c \xrightarrow{\mathsf{Int}} a$. This can then be projected to the participants directly.

2.3 Typestates

Typestates [124] were introduced by Strom and Yemini in 1986, with the goal of enabling the static catching of errors relating to the dynamic evolution of a data object's state. In their words, *whereas the type of a data object determines the set of operations ever permitted on the object, typestate determines the subset of these operations which is permitted in a particular context.* For example, a variable can start as uninitialized, and reading from it can give unpredictable results. After value assignment, the variable enters the initialized typestate, and subsequent code can safely read from it. Thus, by tracking changes in a name's typestate in the program, we can infer the operations that it can be meaningfully involved in.

The idea is applicable to more than just simple data; in fact, it has been used to catch pointer-related errors in C [45], and extended to object-oriented programming [46]. For instance, consider an object that represents a file: the object can be written to when in state open, but not when in state closed. The available state transitions for this object can be described by the automaton in Figure 2.6. Initializing the file object sets its state to closed, and from there, one can call open(), bringing it to state open. In that state, one can repeatedly call write(data); or call close() once, which will bring the file back to state closed. With certain limitations, it is possible to track such changes in object state statically. For

Figure 2.6: A finite state automaton for the typestate transitions of a file object.



example, the following program is erroneous in that it attempts to write to the file without opening it:

```
File f = new File();
f.write("this text");
```

A typestate is a static abstraction over the dynamic state of a data object: it captures the operations allowed on the object in a given context, without knowledge of the exact values of the object's internal variables. In the previous example, it is not necessary to know the actual values and internal references that the File object has—to catch the above error, it suffices to know which *typestate* the file object is in. The diagram in Figure 2.6 thus describes the possible typestate transitions of the object, capturing the operations available in a given static context.

However, deciding whether a program uses an object according to such a transition diagram is not always possible to do statically, even for simple cases. At its core, the problem lies in the use of aliases, as demonstrated by the program below:

```
File f = new File();
f.open("/usr/local/data.txt");
File g = f;
f.close();
g.write("this text");
```

Clearly, this program contains a similar error as before: it attempts to write to the file after closing it. This case is obvious because it is easy to infer that f and g refer to the same (closed) file by looking at the assignment g = f. However, method calls can introduce complications:

```
File f = new File();
f.open("/usr/local/data.txt");
```

```
File g = someMethod(f);
g.write("this text");
```

In order to know the typestate of g at the point of the call to write, we would need to know the typestate of the file returned by someMethod. It could be, for example, that someMethod changes the typestate of f before returning a reference to it. Clearly, predicting the exact effects of method calls is—in the general case—an undecidable problem. However, the issue can be mitigated in practice by using a type-and-effect system such as those proposed by DeLine and Fähndrich [45, 46]. In such systems, methods are annotated with the changes they cause to their arguments' typestates. Nevertheless, static alias tracking, and the related problem of deciding the effects of procedure calls, are both deep topics that go well beyond the scope of this thesis. Specifically for alias tracking, the interested reader is referred to, as a starting point, the classic book on compiler principles by Aho and Ullman [5].

2.3.1 Typestates for Concurrent Programming

In the case of sequential programming, an object's typestate can be used to determine the operations we can apply to it in a given context. For concurrent programs, the problem becomes one of enforcing synchronization constraints: ensure that processes send one-another messages in an acceptable pattern. We can thus associate an evolving typestate with each concurrent process, determining the acceptability of messages in each context. For instance, consider the actor equivalent of the file example discussed above, where the methods are message handlers: the file actor starts in the open state, and can receive write or close messages, with the transitions shown in Figure 2.6 as before. Using ! to denote the action of sending a message, code that uses such an actor can have the form

```
FileActor f = new FileActor();
f ! open("/usr/local/data.txt");
f ! write("this text");
f ! close();
```

The intention of the code might be to open a file, write some data, and then close it; however, it is possible for the messages to arrive in the order write, open, close, or open, close, write, both resulting in a runtime error when the program attempts to write to a closed file. To circumvent cases like this, many authors assume single-source FIFO[1] communication, so that the messages open, write and close in the example above arrive in the same order as they were sent. With this assumption in mind, it is possible to apply the same typechecking mechanism that makes the sequential case work, to the case of actors.

[1]First In First Out

Puntigam [120] furthers this idea by associating each actor in the program with a multiset of tokens, and annotating message handlers with token requirements. This way, he can demand that a message only be received if the actor owns the necessary tokens. In his system, message handlers consume some tokens upon invocation, and produce a (potentially different) bag of tokens when they finish executing. Applying the idea to the previous example, the file actor behavior can be written as below, where the effect of each message on the owned token set is given explicitly:

```
behavior FileActor = {
  handler write(String data) [o⟶o]  { ... }
  handler open()             [c⟶o]  { ... }
  handler close()            [o⟶c]  { ... }
}
```

Receipt of a write message consumes and produces an o token, i.e., it can only be called when o is available, but leaves the actor's token set unchanged. Receiving an open message consumes a c token, and produces an o one; message close does the opposite: it consumes an o and produces a c. In this example, handlers only consume/produce one token at a time, which suffices to impose the desired message ordering in this simple case. However, Puntigam's original system allows the consumption and production of multisets of tokens whose exact form is controlled by type parameters—a system that can capture more complex synchronization patterns.

Process Types. The rules governing the transformation of a given object's typestate can be given in any form that can help the typechecker track the changes statically. Finite state automata, such as the one shown in Figure 2.6, are usually good candidates because they offer a well-understood mechanism that is easy to implement. In the case of concurrent processes, the relevant transition descriptions are usually referred to as *process types*.

Consider the definitions that follow, corresponding to, respectively, a file and a client process:

$$F(c) \overset{\text{def}}{=} c?\mathsf{open}\,;\, c?\mathsf{write}(m)\,;\, c?\mathsf{close}$$
$$P(c,m) \overset{\text{def}}{=} c!\mathsf{open}\,;\, c!\mathsf{write}\langle m\rangle\,;\, c!\mathsf{close}$$

The notation is that of Section 2.2, extended to allow arguments in process definitions. Above, c is the channel that the processes will use to communicate, and m is the text to be written to the file. We can thus write the following program, which is the composition of a file and a client process communicating over a fresh channel c:

$$(\nu c).(F\langle c\rangle \,\|\, P\langle c, \text{“}\mathsf{hello}\text{”}\rangle)$$

The program writes “hello” to the file. With regard to the channel c, the process type of the file is $?\mathsf{open}\,;?\mathsf{write}\,;?\mathsf{close}$. Sending it an open message consumes the first part, and leaves us with $?\mathsf{write}\,;?\mathsf{close}$, at which point, only the write message is acceptable. Sending the file such a write message further changes its type to $?\mathsf{close}$. In other words, the process type of the file evolves as computation progresses—in exactly the same manner as captured by an evolving typestate. Thus, a process type captures both the current typestate, and the rules governing the ways said typestate can change. When the meaning is clear from the context, we will use the terms “typestate” and “process type” interchangeably in the rest of this thesis.

2.4 Typestates vs Session Types

Typestates subsume session types in two different ways. First, a session type describes message patterns over a communication channel; it can thus be viewed as a description of the channel's typestate evolution: as messaging progresses, the channel's (type-) state changes—and so do the messages allowed. For example, consider the following session type, intended for the channel c on page 22:

$$t \overset{\text{def}}{=} P \xrightarrow{\mathsf{open}} F\,;(P \xrightarrow{\mathsf{write}} F)^*\,;P \xrightarrow{\mathsf{close}} F\,;t$$

The starred term denotes repetition of zero or more times. The diagram in Figure 2.7 describes the available transitions, such that the channel starts at typestate A, and the interaction $P \xrightarrow{\mathsf{open}} F$ brings it to typestate B, enabling the write message. From there, the interaction $P \xrightarrow{\mathsf{write}} F$ leaves the typestate unchanged, while the interaction $P \xrightarrow{\mathsf{close}} F$ brings it back to typestate A, allowing the open message again.

The same techniques that enable the static tracking of an actor's typestate, as in Section 2.3.1, can be used to track the typestate transitions of channels. This enables the static typing, for example, of the program on page 22, augmented to associate the type t defined above with the channel c:

$$(\nu c : t).(F\langle c\rangle \,\|\, P\langle c, \text{“}\mathsf{hello}\text{”}\rangle)$$

The second way in which typestates subsume session types concerns the projection of global types onto the individual protocol participants. The result of said projection gives a

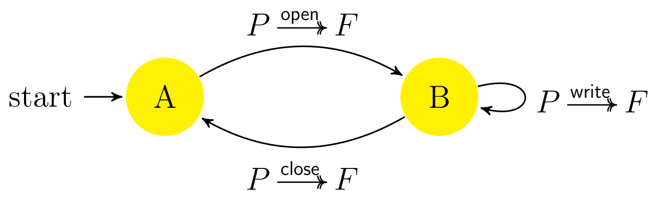

$P \xrightarrow{\text{open}} F$
start
A
B
$P \xrightarrow{\text{write}} F$
$P \xrightarrow{\text{close}} F$

Figure 2.7: A finite state automaton for the typestate transitions of a file channel.

local type for each process, which describes the pattern of actions that the process may take with respect to a given channel. Hence, a local type is, by definition, a process type.

Chapter 3

Related Work

The idea of enforcing type discipline on communicating processes goes at least as far back as the work of Honda [67], and Pierce and Sangiorgi [115, 116]. Both approaches address the problem of guaranteeing—statically—that communicating processes have compatible behavior with regard to messaging: if a process attempts to send some value, the action will be matched with a compatible receive instruction, issued by a second process.

Honda presented his ideas in a calculus reminiscent of CSP [66], and defined the notion of *duality* to capture process compatibility: two process types are *dual*, if processes that implement them communicate such that each process sends precisely the message sequence that the other expects. The basic actions a process P can take are to send and receive a value v, respectively written $!v$ and $?v$. The actions of two processes P_1 and P_2 can be sequenced, as in $P_1\,; P_2$. Choosing between P_1 and P_2 is denoted with $P_1 \oplus P_2$, while branching according to the receipt of a label is written $?l_1\,; P_1 \,\&\, ?l_2\,; P_2$. In a typed universe, each value v is assigned a type t, written $v{:}t$. Thus, processes can be typed; for example, the action of sending a value of type t is written $!t$, and conversely, receiving a value of type t is written $?t$. Types then follow the syntax rules for processes: the type of a process that sequences the actions T_1 and T_2, in this order, is $T_1\,; T_2$. Choosing between two actions with types T_1 and T_2 is written $T_1 \oplus T_2$, and similarly, branching according to the receipt of a label is written $?l_1\,; T_1 \,\&\, ?l_2\,; T_2$. For a simple value-type t, Honda defines type duality such that $!t$ is the dual of $?t$, that is, $\mathit{dual}(!t) = ?t$. Consequently, $\mathit{dual}(T_1\,; T_2) = \mathit{dual}(T_1)\,; \mathit{dual}(T_2)$, and $\mathit{dual}(T_1 \oplus T_2) = \mathit{dual}(T_1) \,\&\, \mathit{dual}(T_2)$. Honda thus assigns a type to each process, with the intention that processes with dual types can be safely composed. In addition, he defines the subset of typeable terms (in the proposed universe) for which deadlock-freedom is guaranteed.

Honda's notion of duality was extended by Takeuchi et al. [126] to apply to channel-based communication: for two processes P_1 and P_2 and a channel c, we can safely compose P_1 with P_2 if the actions they take on c have types T_1 and T_2 with $\mathit{dual}(T_1) = T_2$. One limitation of these early systems was that they did not allow for the disciplined sending of channels as values. The idea that typed channels (and thus the associated sessions) can themselves be sent along other typed channels, in turn to be used by the recipient in a type-safe manner, is known as *session delegation*. In later work, Honda et al.[1] [68], extended the system of

[1]later corrected by Yoshida and Vasconcelos [137]

Takeuchi et al. to capture session delegation and recursive communication patterns. Types that describe such patterns have the form $\mu X.T$, the meaning of which is that we can replace X in T by the whole expression: $\mu X.T \equiv T[\mu X.T/X]$. For example, the pattern of repeatedly sending a value of type t can be written as $\mu X.(!t\,; X)$. In presence of recursion, Honda et al. treat the composition of actions by extending the notion of duality, such that, $dual(\mu X.T) = \mu X.dual(T)$.

Independently from the above line of work, and around the same time as Honda's original 1993 paper, Pierce and Sangiorgi [115] proposed a system to apply type discipline on communication in the π-calculus [103]. The main difference from Honda's early work is an extended notion of compatibility, in terms of a subtyping relation—allowing for the disciplined communication of channels as values. In general, sending a value of type t_1 along some channel has to be met with a receipt action on the same channel; such a receipt action should expect a value of type t_2, with the constraint $t_1 \preccurlyeq t_2$ for some subtyping relation $\preccurlyeq$. Informally, for values $v_1 : t_1$ and $v_2 : t_2$, the meaning of subtyping is that $t_1 \preccurlyeq t_2$ if and only if v_1 can be safely used in place of v_2. This relation can be extended to processes, such that for two processes $P_1 : T_1$ and $P_2 : T_2$, it is $T_1 \preccurlyeq T_2$ if and only if process P_1 can safely replace process P_2 in all contexts—an idea made concrete by defining subtyping for processes in terms of bisimilarity [102]. Finally, it is worth mentioning that Pierce and Sangiorgi's early system dealt with the safety of recursive communication patterns as well, expressible through the bang operator [101].

Multi-Party Sessions. The early works described above treated communication in the two-party case, in the sense that they answered the question of whether two processes P_1 and P_2 can be safely composed into $P_1 \,\|\, P_2$. For instance, in the notation of Takeuchi et al. [126], two processes can initiate a session over a name s as such:

$$(\mathsf{request}\ s(c)\ \mathsf{in}\ c!0\,; \mathsf{end})\ \|\ (\mathsf{accept}\ s(c)\ \mathsf{in}\ c?x\,; \mathsf{end})$$

The above contrived example initiates a session s involving a single channel c, which is used for the first process to send the number zero to the second one. The reader might observe that this is the notation of Section 2.2.2, with only one channel, and two participants. The notation of Section 2.2.2, involving multiple channels and participants, was introduced by Honda et al. [69, 70]. Their work allows the specification of protocols from a global perspective, where a session type describes the interactions taking place over all channels in the session; such a type is called a *global type*. A *projection* algorithm then mechanically derives the behavior specification of each individual process, that is, their *local type*. Typing the individual processes with regard to these local specifications ensures two fundamental properties: (i) *session fidelity*, i.e., any reduction on processes is predicted by the associated session type; and (ii) *progress*, i.e, interactions predicted by a session type are certain to take place in the program. One limitation of the work of Honda et al. is that progress is ensured on a per-session basis—the case of communication in one session blocking actions in another was left untreated.

The assumption on the absence of inter-session hindrance was lifted by Bettini et al. [14], who analyzed the flow of dependencies on the use of channels, taking session delegation into account and ensuring the absence of cyclic dependencies. The system of Honda et al. was further extended by Gay and Hole [57], who proposed a subtyping relation for session types along the lines of Pierce and Sangiorgi [115, 116]. Deniélou and Yoshida introduced multiparty session automata [50], a formalism that allows the capturing of interactions such as the alternating bit protocol, that cannot be expressed in the original multiparty session types of Honda et al. In their work, Deniélou and Yoshida show that in some cases, questions that are not decidable for general-purpose communicating finite state machines are in fact decidable for multiparty session automata.

It is worth mentioning that while the above discussion is concerned with the theoretical foundations of multi-party sessions, those came after the topic was first investigated in practice, with the introduction of the Web Services Choreography Description Language (WSCDL) [135]. The idea behind WSCDL is to enable conformance verification for multiple, concurrently executing web services and clients against protocols described in XML [134]. The latter is a general-purpose language for expressing arbitrary graph structures, and thus, WSCDL expresses protocols in an accordingly compositional fashion: simple actions (e.g., sends and receives) are nodes; a collection of action nodes can be connected to a sequencing node, forcing the relevant actions to be sequenced; a collection of nodes can also be connected to a parallel node, denoting the concurrent composition of the related actions, et cetera. The notion of an end-point *projection* of such specifications onto the individual participants was first studied by Carbone et al. [27], who also investigated the associated correctness requirements—that is, the conditions under which a set of local types behaves according to the global type they were projected from.

Parameterized Session Types. Yoshida et al. [138] and Deniélou et al. [51] extended the syntax of Honda et al. [70] to allow the use of parameters—both in process definitions, and in the language of types. The example that follows uses an extended form of the syntax used in Section 2.2.2, and consists of a process sending the value 5 to another process n times:

$$\begin{array}{rcll} P(n) & = & \lambda x{:}\mathsf{Int}\,.\,\big(\mathsf{request}\ s[1..1](c)\ \mathsf{in}\ \mathsf{foreach}(i < n).(c!x)\big) & \text{process definition} \\ Q(n) & = & \mathsf{accept}\ s[1](c)\ \mathsf{in}\ \mathsf{foreach}(i < n).(c!x) & \text{process definition} \\ & & \Pi n.\big(\,P(n)(5) \parallel Q(n)\,\big) & \text{program} \end{array}$$

The construct Πn binds the parameter n in the subsequent expression, which in this case controls the number of times the message gets sent. The session type for s is thus

$$T = \Pi n\,.\,\mathsf{foreach}(i < n)\,.\,(0 \xrightarrow{\mathsf{Int}} 1)$$

which is a dependent type, i.e., the type parameter n is a name bound in the program.

In the above example, the parameter controls the number of repetitions; in addition to such patterns, the system of Yoshida et al. supports type-safe interactions among a fixed, but not statically known, number of participants. The calculus underlying their approach simplifies the treatment of parametric constructs with the use of Gödel's $\mathtt{R}$ operator [8], and allows communication patterns whose repetitive nature is expressible through operations on the indices. One example would be the use of a single global type to capture a butterfly network (used, for example, in performing fast Fourier transforms), the very structure of which is highly dependent on the number of participants. Static verifiability—without instantiating the type parameters—is maintained by projecting onto parameterized local types, which can be checked against process implementations as long as the operations on the indices allow syntactic comparisons. A simple example would be the following global type, describing a ring of $n + 1$ processes that pass some integer value around once:

$$\Pi n.\Big(\mathsf{foreach}(i < n).\big(i \xrightarrow{\mathsf{Int}} (i+1)\big)\,;\,\big(n \xrightarrow{\mathsf{Int}} 0\big)\Big)$$

The work of Yoshida et al. [138] was later improved by Bejleri [12, 13], who allowed indices to range over infinite sets of natural numbers, and introduced the concept of a *role*. The idea was made precise by Deniélou and Yoshida [49], whose system facilitates modular implementations of protocols such that each role is implemented and typechecked separately—similar to classes in object-oriented programming. This new system did away with explicit numeric parameters, and allowed programs to contain expressions that poll the participants of a given role. Deniélou and Yoshida allow, for example, the global type of a map-reduce [47] interaction among a server and multiple clients to be written as

$$\mu t.(\forall x : \mathsf{client}).\big((\mathsf{server} \xrightarrow{\mathsf{map}} x)\,;\,(x \xrightarrow{\mathsf{reduce}} \mathsf{server})\,;\,t\big).$$

The above type states that the server sends a map message to all clients in the program, each one of which then replies with a reduce message. Note that with the use of primitive recursion (μt), the interaction continues forever. An important improvement of this system over the previous ones is that it allows processes to join and leave roles dynamically, while maintaining the benefits of typechecking their behavior. In the above example, processes that join the client role must conform to the given global type. It is worth noting that the lack of numeric parameters results in global types that are less precise, and the role polling mechanism requires related information to be retained at runtime.

Asynchrony. Gay and Vasconcelos [58] and Gay and Vasconcelos [61] were the first to consider asynchronous session types for a functional programming language. Their work develops a type system that is based on linear logic, which, contrary to previous proposals, allows the re-use of well understood typechecking mechanisms. Kouzapas et al. [84] propose asynchronous session types for an event-based flavor of the π-calculus, and study the semantic relation between their calculus and multi-threaded programs. The proposed extensions are also found in the work of Hu et al. [73], who extend the Java programming language with event-based primitives and asynchronous type-safe sessions.

A fundamentally different approach to concurrency is taken by Carbone and Montesi [24], who propose that the program itself should be written from a global perspective. Static typing ensures the absence of deadlocks, while concurrent composition is regained through a suitable swap relation.

Castagna et al. [31] propose a specification language for global types that is closer to regular expressions; they investigate the requirements for correctness after projection, and guarantee certain liveness properties. Their work was the starting point of the author's original papers [35, 36], which give rise to the material in Chapter 5.

Subtyping for local types in asynchronous multi-party sessions has been studied by Mostrous et al. [107], as an extension to the work of Honda et al. [70]. Their work also presents an algorithm for the synthesis of a global type from a set of end-point (local) types, with the property that the resulting global type is minimal with respect to the proposed subtyping relation.

Object-Oriented Languages. Hu et al. [72] were the first to present a full implementation of a Java flavor with support for session types. Dezani-Ciancaglini et al. propose MOOSE [53], an extension to Java with session types that guarantee progress. The same authors extend [52] MOOSE with bounded polymorphism [29] for communicable values, while support for asynchronous channels was added by Coppo et al. [43]. Balzer and Pfenning [9] capture object-oriented programming in an implementation of a π- derived process calculus. In their work, each process is associated with a channel, which allows it to communicate with other processes. Such a channel serves as a "reference" to the process, allowing the outside world to treat it as an object. Each channel is associated with a linear session type, which allows sending a channel to another process, albeit making it unavailable to the sender immediately afterwards.

Better object encapsulation is offered by the system of Gay et al. [59], who mix typestates [46, 132] and session types, by allowing typed channels to be stored as class fields. A typestate is associated with each object reference, and the relevant transitions are given on the class level. These dictate the allowed sequences of method calls on objects, and by extension, the use of the channels stored as fields. Object aliases are controlled via a linear type system, i.e., one that utilizes destructive reads [136].

Process Types. As the current state of the literature in choreographic types suggests, it is difficult to capture dynamic process creation as a behavior in global types. Subsequently, methodologies that rely solely on local types have been proposed, with the intention that the typing will guarantee safety and liveness properties without caring about adherence to a pre-defined protocol. Local types in this context are referred to as *process types* in the literature.

Bonelli and Compagnoni [17] propose process types akin to the local types found in the work of Honda et al. [68][2], and rely on type preservation theorems on (dual) type compositions for correctness. Reminiscent of this approach is the work of Lange and Tuosto [89], who discuss whether the local types of Honda et al. can be composed into a global type that satisfies certain progress and safety properties, in a CCS-style process calculus. As discussed in Section 2.3, process types can be viewed as communicating finite state automata, i.e., automata where input and output actions label the transitions. Correctness criteria for the composition of such automata is discussed by Brand and Zafiropulo [21], and Lange et al. [90].

Programmer Intent. When composing processes, deadlock-freedom does not necessarily guarantee progress in the intuitive sense. As a result, more general notions have been proposed, such as *lock-freedom.* A lock-free process is one where all I/O operations in sub-processes succeed (eventually, under fair scheduling), even if the process as a whole diverges. The relation between lock-freedom and the ability of a process to reduce has been studied, for example, by Carbone et al. [26].

[2]their work was developed around the same time, but independently from Honda et al.

Lock-freedom has been taken one step further, by allowing the programmer to make the set of I/O operations that are required to succeed explicit. For example, Kobayashi's system [79] allows the use of a variety of channel types, with different progress guarantees for each. In general, his type system associates a time tag with each channel, inferred from its relative usage order. The type system then enforces an ordering relation on these tags, breaking cycles. The use of a partial order to break cyclic dependencies is also found in the work of Padovani [112]. Sumii and Kobayashi [125] take these ideas further, so that channels are annotated with capabilities and obligations. The resulting type system ensures that if a process has the capability of performing an I/O action on a channel, that action will eventually succeed; similarly, it guarantees that if a process has the obligation to perform an action on a channel, the action is eventually taken. This strategy for lock-freedom is improved on in Kobayashi's later work [80], where even more precise information is used: channels are additionally associated with the minimum number of reduction steps needed until capabilities are met, and also the maximum number of steps until obligations are fulfilled. The automatic inference of similar type annotations has also been considered [81].

Puntigam's work on token-tracking [120] for actors allows the programmer to annotate each message handler with two multisets of tokens: one to be consumed when receiving the message, and one to be produced when the handler completes. A message is only accepted if the required tokens are available, a property ensured (statically) by the type system, so that every sent message can—and will—be processed. In other words, static typing prevents the sending of messages for which the recipient is not guaranteed to have the required tokens. In this regard, Puntigam's local types make synchronization constraints explicit, by means of static evaluation of *method guards* [99]. The nomenclature is due to the fact that the latter "guard" methods with boolean expressions, and such methods can only be invoked if the associated conditions are satisfied. Related is the concept of *enabled sets* [99], first appearing in the description of Rosette [4]. The change in the token set upon message receipt effectively changes the set of messages that are currently acceptable, in other words, *enabled.*

An important contribution of Puntigam's work is the handling of conditional statements in the type system. For example, for the statement

$$\mathsf{if}\ x_1 = x_2\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$$

P_1 is typed in an environment where the types of x_1 and x_2 have been combined, i.e., their token sets have been merged.

In later work, Puntigam and Peter [122] further improve this idea, such that the typing handles dynamic type comparisons such as

$$\mathsf{if}\ s_1 \preccurlyeq s_2\ \mathsf{then}(t)\ P_1\ \mathsf{else}\ P_2$$

which tests whether the type s_1 is a subtype of s_2. In typing the above, the type variable t is replaced in P_1 by the difference of s_1 and s_2, to account for consumed tokens. The type system ensures that messages sent in P_1 adhere to the tokens available in t, and respectively, that messages sent in P_2 adhere to the tokens available in s_2.

In the same work, Puntigam and Peter add progress guarantees with the inclusion of *obligatory* tokens that the type system requires be consumed. Puntigam's token-tracking ideas [119] are the starting point for the material in Chapter 4, where we address some of his system's shortcomings. For instance, given an actor reference with associated obligatory tokens, Puntigam demands that one of the following happens by the end of the current scope: (a) the actor reference is sent suitable messages to consume the obligatory tokens; (b) the actor reference is sent to another actor; or (c) it is is passed as the formal argument to a become [3] -style invocation. The first item ensures requirement satisfaction. However, the second item allows actor names to be passed around in circles, while the third allows the current actor to diverge—in both cases without guaranteeing the consumption of obligatory tokens. In contrast, the system we develop in Chapter 4 does not allow such infinite requirement delegation; we additionally provide a simpler formal treatment of the associated progress guarantees.

Realizability of Sessions. The question of whether it is possible to implement a given choreography (i.e., a global type) by a suitable set of participants has been addressed in the work of Basu et al. [11]. Given a choreography in the form of an automaton where transitions are labeled with message send actions, Basu et al. show that the question of whether the choreography is implementable by a set of communicating peers is a decidable problem. The authors define equivalence such that two systems are equivalent if they generate the same sequences of message send actions, ignoring the order of receives. In contrast, the realizability question for the more general case of *message sequence graphs* [64], is undecidable. Decidability and complexity class results for the case of message sequence graphs have been derived by Alur et al. [7].

Castagna et al. [30, 31] discuss structural criteria for global types that contain the more common sequence, choice and concurrent composition constructs, to ensure that projection maintains the specification's trace semantics. Related is the approach of Lanese et al. [88], who address the question of whether a choreography in WSCDL [135] has an equivalent set of BPEL4Chor [48] processes. Modeling both in simple calculi that support messaging, choice, and concurrent composition, Lanese et al. determine the criteria for the existence of a bisimulation relation between the WSCDL specification and the set of BPEL4Chor processes, for both synchronous and asynchronous communication.

Generalization of Session Type Systems. A general methodology for deciding the completeness of a subtyping relation is presented by Chen et al. [40, 41]. Denoting substitutability with $<$ and subtyping with $\prec$, a subtyping relation is complete when for all types t_1, t_2 with $t_1 < t_2$, the relation $t_1 \prec t_2$ is decidable statically. Their methodology is based on (i) defining the relation "not a subtype of" syntactically; (ii) defining a *characteristic process* for each session type; and (iii) showing that the relation $t_1 \not\prec t_2 \Leftrightarrow t_1 < t_2$ is not derivable for the characteristic processes of t_1 and t_2.

Igarashi and Kobayashi [75] present a generic session type system for the π-calculus. Equipped with a generic subtyping relation as well as a generic "OK" predicate, they discuss how other session type systems can be seen as concretizations of their system. Igarashi and Kobayashi's proofs for standard properties such as subject reduction provide a framework where one can create correct session type systems without the need to provide proofs anew. In fact, Gay et al. [60] define an encoding from session types in the monadic π-calculus with polarities [57] to the system of Igarashi and Kobayashi above.

Runtime Monitoring. Consider a global type that includes elements of the form $P \xrightarrow{x:A} Q$, signifying the sending of a message from process P to process Q. Here, x is a name bound to the message payload, and A is a boolean assertion. Depending on the form of A, it might not be possible to enforce $A(x)$ statically; in that case, we can guarantee the assertion through run-time checks, i.e., runtime monitoring. Note that it is much more efficient to apply such checks locally, that is, on the level of each process, as opposed to checks that require data from multiple sites—which would be impractical in large distributed systems.

The conditions under which such localized, end-point runtime checks suffice to enforce globally specified assertions were first studied by Bocchi et al. [16]. In a similar spirit, Neykova et al. [108] extended Scribble [71] with timing constraints, and studied the conditions that enable purely local, end-point runtime monitoring of distributed systems with real-time constraints.

Other Related Work. The relationship between linear logic and session types has been investigated by Caires and Pfenning [23]. Their work interprets the connectives of linear logic as standard actions from the session type domain, i.e., the communication of values and channels, continuations, et cetera; standard properties such as subject reduction are a consequence of the close correspondence between the operational semantics of the synchronous π-calculus and the standard sequent calculus proof system for dual intuitionistic linear logic.

A rich and expressive session type system for a multi-threaded version of the typed λ-calculus has been proposed by Gay et al. [62, 133], which treats the dynamic creation of asynchronous channels seamlessly. Typechecking is made possible by annotating functions with the changes they cause to channel types, as well as incorporating a form of alias tracking into the type checker. Similar ideas have been applied to an object-oriented calculus in the work of Gay et al. [59].

Very interesting from an application perspective is the work of Pucella and Tov [118], who encode and enforce session types in Concurrent Haskell [78] using the language's built-in parametric typing capabilities. This is useful from a practical standpoint, because such a technique does not require programmers to rely on non-standard tools in order to benefit from session types. Similarly, Imai et al. [76] encode and enforce session types in pure OCaml [91]. De'Liguoro and Padovani [141] propose types for mailboxes as first-class values, a concept which finds application in languages such as scala [129]. Their typing builds a call graph, and prevents deadlocks by ensuring the absence of cyclic dependencies among messages, their arguments, and their recipients. The basic ideas of that approach were originally explored by Padovani [111], who treats local types as obligatory specifications, i.e., where all prescribed interactions must take place.

Exception handling has been considered by Carbone et al. [25, 28], allowing the participants of a protocol to escape the normal control flow and coordinate on another. Additional safety properties with regard to the consistency and source of data can be guaranteed by augmenting session types with *correspondence assertions* [18, 19]. Such systems can ensure, for instance, that a "man in the middle" does not change the amount deposited in a bank in an unexpected or malicious way. Nierstrasz [109] and Puntigam [121] model the behavior of active objects as local types that resemble finite state automata, and discuss subtyping in terms of their traces. Probert and Saleh [117] discuss the synthesis of protocols from stated properties and partial specifications, such as some, but not all of the local types involved. For the related problem of verifying properties of protocols expressed as finite state machines, we refer the reader to the work of Yuang [139].

Chapter 4

Typestates for Progress

Liveness properties state that a system will eventually produce an event of interest [6, 87]. For example, a client may request exclusive use of a resource from a server, with the expectation that there will eventually be a reply indicating whether access has been granted or denied. Liveness properties are generally difficult to express and reason about; this is primarily because they are formulated over, and thus require reasoning on, sequences of runtime configurations. In addition, even elementary liveness properties may hinge on assumptions about *fairness*, which disallow the indefinite postponement of basic operations such as the dispatching of messages [1].

Usually, type systems provide a straightforward way to capture *safety* properties of programs, i.e., properties which rule out executions that reach undesirable states. In contrast to liveness, safety can be established by analyzing single runtime transition steps. However, work in session types [126] has shown the feasibility of using a type system to establish certain notions of progress. These works apply type discipline to the use of communication channels in the π-calculus, and have viewed issues of progress under the prism of session fidelity: communication protocols, and hence the types that describe them, are designed so that adhering participants never get stuck. Session types usually constrain cyclic communication dependencies on the level of the protocol itself, so that well-typed processes communicate in a manner that always makes progress [14, 70]. However, as Sumii and Kobayashi [125] remark, there is more to progress than breaking cyclic dependencies: *programmer intent* should be taken into account.

We are interested in guaranteeing that an implementation adheres to the programmer's intent on the delivery of certain messages. For example, the programmer may demand that whenever a client requests resource access from a server, it must eventually receive a reply. This reply does not necessarily need to come from the server process itself, but it does need to specify whether access to the resource has been granted or not. Certain bugs in the implementation can violate this requirement, for example, due to cyclic dependencies, or an omission of a message sending command by the programmer.

This chapter presents a possible solution to the problem in the context of actors [3], which communicate via asynchronous message-passing. Our work is structured around a simple actor calculus that allows the programmer to specify how messaging requirements may be generated at runtime. We regard *progress* to be a persistent property on the (runtime) program state such that a configuration C satisfies progress if every execution trace from C ends in a state where all dynamically generated messaging requirements have been satisfied. We propose a type system to guarantee this property for every runtime configuration resulting from the program. We use the notion of a *typestate* [124] on both the language level, and in the presented meta-theory, tagging actor names with the multiset of message types that the corresponding actor needs to receive. We therefore regard progress as the question of whether an actor eventually receives all the messages included in their typestate. The type system enforces that, for every requirement that appears at runtime, suitable action is taken: either it is fulfilled in the current scope, or it is delegated to another actor. By recursive reasoning, the type system guarantees that such a postponed requirement will be satisfied by the delegate actor. In essence, we show that in all executions of well-typed programs, all requirements generated at runtime will eventually result in the corresponding messages being received.

A preliminary version of this work appeared in 2017, at the Workshop on Actors and Active Objects [37]. The most important shortcoming of that system is that the typing does not terminate for programs with cyclic messaging patterns. In Section 4.5 we show that it is possible to overcome this limitation, by using a memoization technique. Moreover, in order to keep the presentation simple, the material here does not deal with elementary safety properties, such as checking the numbers of handler arguments—those can be established by a separate type system.

Chapter Outline

We first present, in Section 4.1, two example actor programs that demonstrate the usefulness of message requirements with regard to establishing progress. We then define our actor calculus formally in Section 4.2, with its abstract syntax and operational semantics. The type system is defined in Section 4.3, where we give example typings and discuss the system's limitations. The main result is proven in Section 4.4, where we show that executions of well-typed programs eventually satisfy stated requirements. In Section 4.5, we discuss cyclic communication patterns and a possible solution to the issues analyzed on page 46. Lastly, Section 4.6 incorporates Puntigam's ideas on safety properties, as outlined on page 22.

4.1 Motivating Examples

We motivate our approach by discussing two simple programs, given in pseudo-code syntax reminiscent of Scala [129] with the Akka toolkit [93] for actors. Later, in Section 4.2, we will define a minimal calculus to make the underlying ideas precise. The first program, shown in Figure 4.1, embodies a resource sharing scenario among multiple clients via a central server. The second program, shown in Figure 4.2, implements a classic example from the session types literature, where two buyers coordinate to purchase a book from a seller.

4.1.1 Resource Sharing Program

In the resource sharing program of Figure 4.1, there are three kinds of actors: servers, clients, and resources. The clients attempt to acquire exclusive access to resources administered by the server, by repeatedly messaging it until they succeed. The server is responsible for responding to requests, by creating new resource actors and handing out their names, as permitted by system limits. The problem we consider in this example is how to ensure that (a) clients eventually receive some reply after a request, whether positive or negative; (b) resources are properly allocated and de-allocated; and that (c) allocated resources are eventually put to work.

Actors are defined by their *behavior*, i.e., how they respond to messages, and their persistent *state*. For example, server actors have handlers for request and done messages, and a count state variable that represents the number of available resources. The program initially spawns a server actor and two client actors, on lines 40 to 42. The two clients are each sent a start(s) message (lines 43 to 44), informing them of the name of the server s. The clients then send request messages to the server (line 19), to ask for resource access. Upon receiving a request message, the server checks if count is zero or less (line 3), and if so, it sends a later message to the client. If count is positive, the server spawns a new resource actor to which it sends a lock message (line 8) with the client and itself as the payload; then, it decrements count. The resource reacts to the lock message by sending ok to the client (line 33), who replies with a work message (line 22).

The requirements (a), (b), and (c) from above are explicitly embedded in the code via the use of the construct add_req. On line 18, we express that the client actor currently executing this line is required to eventually receive either ok or later. On line 7, add_req expresses the requirement that the actor whose name is stored in rs (a resource actor) must eventually receive a kill message, representing de-allocation. Finally, on line 32, we express that the resource actor executing this line needs to eventually receive a work message.

As it turns out, the stated requirements will be satisfied in all executions of the resource sharing program where message delivery and processing is not indefinitely postponed. With regard to the delivery of either ok or later to the clients, consider what happens when the server actor receives a request message. It is either count ≤ 0, in which case the server sends

Figure 4.1: Example – *Resource Sharing.*

```
Server(count : Int) = {
  request(c : Client) =
    if count ≤ 0 then
      c ! later()
    else
      let rs = new Resource() in
      let rs_ = rs.add_req(kill) in
        rs_ ! lock(c, self);
        update(count - 1)

  done(r : Resource) =
    r ! kill();
    update(count + 1)
}

Client() = {
  start(s : Server) =
    let self_ = self.add_req(ok + later) in
      s ! request( self_ )

  ok(r : Resource, s : Server) =
    r ! work( self, s )

  later() = ...

  done(r : Resource, s : Server) =
    s ! done(r)
}

Resource() = {
  lock(c : Client, s : Server) =
    let self_ = self.add_req(work) in
      c ! ok( self_, s )

  work(c : Client, s : Server) =
    ...
    c ! done(self, s);
}

let s = new Server(1) in
let c1 = new Client() in
let c2 = new Client() in
  c1 ! start(s);
  c2 ! start(s)
```

later to the client actor right away; or count > 0, and the client name is sent to the newly spawned resource actor (line 8), which sends it an ok message on line 33.

Note that if we omit some message sending operation, requirements will be violated; for example, leaving out the statement c ! done(**self**, s) from line 37 would violate the requirement set on line 7, thus resulting in failure to de-allocate the resource. Consequently, satisfaction of requirements captures a form of progress for actor programs, by ruling out that certain actors wait forever for some specific message.

4.1.2 Two Buyer Protocol

The program in Figure 4.2 builds on the two buyer protocol—a classic example found, e.g., in the work of Honda et al. [69]. The general idea is that two buyers need to coordinate to buy a book from a seller. The first buyer sends a quote request to the seller, who replies with a price. When the first buyer receives this quote, it tells the second buyer how much it is willing to contribute; then the second buyer decides if the remaining amount is within their budget, and let the seller know. In this example, the problem we consider is (a) whether the first buyer eventually gets a quote from the seller, and (b) whether the seller eventually receives a response to the quote.

There are three actor behaviors in the program: Seller, Buyer1, and Buyer2. Execution begins with spawning one actor of each behavior, on lines 82 to 85. The protocol starts when the first buyer actor receives a start message with the names of the two other participants. The first buyer then sends a get_quote message to the seller actor with the title of a book (line 64). Requirement (a) is embedded in the use of add_req on line 63. In response to a get_quote message, the seller actor looks up the price of the title and sends it back in a quote message (lines 49 to 51). Notice the assignment with add_req on line 50, which captures requirement (b) by demanding either one of messages yes or no.

The first requirement is easily seen to be fulfilled by the seller actor on line 51, assuming the price_of invocation on line 49 terminates. The second requirement is ultimately fulfilled by the buyer-two actor, which will either reply yes, or no, in response to the ask message from the buyer-one actor. Once again, omitting send operations will result in requirement violations; for example, omitting s ! yes() from line 77 would have that branch of the conditional (line 76) to proceed without a response to the seller. As in the first example, it makes sense to demand that both branches of a conditional satisfy all stated requirements, perhaps via a different messaging path—in this example, via a yes or no (lines 77 and 79, respectively).

As both presented examples hint at, making messaging requirements explicit allows us to reason about the eventual delivery of certain messages—and to do so *statically*. Our approach

Figure 4.2: Example – *Two Buyer Protocol.*

```
Seller() = {
  get_quote(title : String,
            b1 : Buyer1,
            b2 : Buyer2) =
    let price = price_of(title) in
    let self_ = self.add_req(yes + no) in
      b1 ! quote(price, b2, self_)

  yes() =
     ...

  no() =
     ...
}

Buyer1(contr : Int) = {
  start(s : Seller,
        b2 : Buyer2) =
    let self_ = self.add_req(quote) in
      s ! get_quote("1984", self_, b2)

  quote(price : Int,
        b2 : Buyer2,
        s : Seller) =
    b2 ! ask(price, contr, s)
}

Buyer2(contr : Int) = {
  ask(price : Int,
      b1_contr : Int,
      s : Seller) =
    if price - b1_contr ≤ contr then
      s ! yes()
    else
      s ! no()
}

let s = new Seller() in
let b1 = new Buyer1(11) in
let b2 = new Buyer2(5) in
  b1 ! start(s, b2)
```

is to reduce difficult parts of this reasoning to the checking of program conformance to a type system along the lines of process types [67, 119]. If a program passes the check, it is free of the discussed progress issues.

4.2 Actor Calculus

In this section, we present a minimal actor calculus to formalize the above ideas. This calculus is a small extension of the one presented in Section 2.1; its syntax provides an extra construct, **add**($\mathtt{x}, \mathtt{R}$) that captures requirement generation. As before, the language follows standard actor semantics. However, its syntax does not adopt the λ-calculus extension of Agha et al. [3]; instead, to capture the examples above, we allow behaviors to include message handler definitions. As in Section 2.1.1, the intention here is the following: consider an actor α with behavior b, where the definition of b includes a handler h with parameters $x_1 \dots x_k$ and body $\mathtt{S}$. Then, the receipt of a message $h(u_1 \dots u_k)$ by actor α will invoke the code $\mathtt{S}$, replacing the formal parameters $x_1 \dots x_k$ with the values $u_1 \dots u_k$, and **self** with α. The reserved name **self** refers to the actor in which it is evaluated.

In what follows, we abbreviate sequences of the form $x_1 \dots x_k$ with $\overline{x}$, sequences of the form $u_1 \dots u_k$ with $\overline{u}$, et cetera. The calculus syntax is given in Figure 4.3: programs $\mathtt{P}$ consist of a list of behavior definitions $\overline{\mathtt{B}}$ and an initial statement $\mathtt{S}$. An actor behavior definition includes a name b that identifies the behavior, variables $\overline{x}$ that store the assuming actor's state, and a list of message handler definitions $\overline{\mathtt{H}}$. In turn, a message handler definition includes a name h that identifies the handler, a list of message parameters $\overline{x}$, and a statement $\mathtt{S}$ to be executed upon invocation of the handler.

Statements generally consist of single operations followed by another statement. For example, $\mathtt{x}!h(\overline{e}).\mathtt{S}$ sends a message for handler h of the actor $\mathtt{x}$, with argument list $\overline{e}$, and then proceeds as $\mathtt{S}$. The statement $\nu x{:}b(\overline{e}).\mathtt{S}$ creates a new actor (whose name is bound to $\mathtt{x}$ in $\mathtt{S}$) with behavior b and initial state variables set to the values of the expressions $\overline{e}$. The statement **update**($\overline{e}$) updates the values of actor state variables, and the **if** statement has the usual meaning of a conditional. A **ready** statement belongs to the runtime syntax, signifying the end of handler execution.

The call **add**($\mathtt{x}, \mathtt{R}$) adds the requirement $\mathtt{R}$ to the list of requirements already associated with the actor $\mathtt{x}$. Informally, to satisfy a disjunctive requirement $(h_1 + h_2)$ of some actor $\mathtt{x}$, we have to send it a message labeled with either h_1 or h_2. Similarly, to satisfy a conjunctive requirement $h_1 \cdot h_2$, one has to send the actor two messages, h_1 and h_2.

Figure 4.3: Actor calculus syntax.

$\mathtt{P}$	$::=$	$\overline{\mathtt{B}}\ \mathtt{S}$		
$\mathtt{B}$	$::=$	$\textbf{bdef}\ b(\overline{x}) = \{\overline{\mathtt{H}}\}$	$b \in$ behavior names	
$\mathtt{H}$	$::=$	$\textbf{hdef}\ h(\overline{x}) = \mathtt{S}$	$h \in$ handler names	
$\mathtt{S}$	$::=$	$\mathtt{x}!h(\overline{e}).\mathtt{S}$		
	$\mid$	$\textbf{add}(\mathtt{x}, \mathtt{R}).\mathtt{S}$		
	$\mid$	$\textbf{if}\ e\ \textbf{then}\ \mathtt{S}_1\ \textbf{else}\ \mathtt{S}_2$	$e \in$ expressions (values, function calls, etc.)	
	$\mid$	$\nu x{:}b(\overline{e}).\mathtt{S}$	actor creation	
	$\mid$	$\textbf{update}(\overline{e})$	state update	
	$\mid$	$\textbf{ready}$		[runtime syntax]
$\mathtt{x}$	$::=$	$\textbf{self} \mid x, y, z, \ldots \mid \alpha, \beta, \ldots$	$x, y, z, \ldots \in$ variables	
			$\alpha, \beta, \ldots \in$ runtime actor names	
$\mathtt{R}$	$::=$	$(\mathtt{R}_1 + \cdots + \mathtt{R}_k)$	requirement disjunction	
	$\mid$	$(\mathtt{R}_1 \cdot \ldots \cdot \mathtt{R}_k)$	requirement conjunction	[runtime syntax]
	$\mid$	$(\mathtt{R}_1 \div \mathtt{R}_2)$	requirement satisfaction	[runtime syntax]
	$\mid$	h	simple message requirement	
	$\mid$	ϵ	empty requirement	[runtime syntax]
C	$::=$	(Δ, R, M, A)	configuration	[runtime syntax]
Δ	$::=$	program information		[runtime syntax]
R	$::=$	$\{\alpha_1 \mapsto \mathtt{R}_1 \ldots \alpha_\kappa \mapsto \mathtt{R}_\kappa\}$	requirement map	[runtime syntax]
M	$::=$	$\{\alpha_1!h_1(\overline{u}_1) \ldots \alpha_\kappa!h_\kappa(\overline{u}_\kappa)\}$	multiset of pending messages	[runtime syntax]
A	$::=$	$\{\langle \mathtt{S}_1 \rangle_{\alpha_1}^{b_1(\overline{w}_1)} \ldots \langle \mathtt{S}_\kappa \rangle_{\alpha_\kappa}^{b_\kappa(\overline{w}_\kappa)}\}$	actor map	[runtime syntax]
			$u, w \in$ values	

Figure 4.4: Structural congruence on requirements.

$$\begin{array}{cc}
\epsilon + \mathtt{R} \equiv \epsilon & \epsilon \cdot \mathtt{R} \equiv \mathtt{R} \\[1em]
\mathtt{R}_1 + \mathtt{R}_2 \equiv \mathtt{R}_2 + \mathtt{R}_1 & \mathtt{R}_1 \cdot \mathtt{R}_2 \equiv \mathtt{R}_2 \cdot \mathtt{R}_1 \\[1em]
\mathtt{R}_1 \cdot (\mathtt{R}_2 \cdot \mathtt{R}_3) \equiv (\mathtt{R}_1 \cdot \mathtt{R}_2) \cdot \mathtt{R}_3 & \mathtt{R}_1 + (\mathtt{R}_2 + \mathtt{R}_3) \equiv (\mathtt{R}_1 + \mathtt{R}_2) + \mathtt{R}_3 \\[1em]
\mathtt{R} \div (\mathtt{R}_1 + \mathtt{R}_2) \equiv (\mathtt{R} \div \mathtt{R}_1) + (\mathtt{R} \div \mathtt{R}_2) & \mathtt{R} \div (\mathtt{R}_1 \cdot \mathtt{R}_2) \equiv (\mathtt{R} \div \mathtt{R}_1) \div \mathtt{R}_2
\end{array}$$

$$(\mathtt{R}_1 + \mathtt{R}_2) \div \mathtt{R} \equiv (\mathtt{R}_1 \div \mathtt{R}) + (\mathtt{R}_2 \div \mathtt{R})$$

Figure 4.5: Requirement reductions.

$$(\mathtt{R} \cdot h) \div h \longrightarrow \mathtt{R}$$

$$\frac{\mathtt{R}_1 \longrightarrow \mathtt{R}_1'}{\mathtt{R}_1 \cdot \mathtt{R}_2 \longrightarrow \mathtt{R}_1' \cdot \mathtt{R}_2} \qquad \frac{\mathtt{R}_1 \longrightarrow \mathtt{R}_1'}{\mathtt{R}_1 + \mathtt{R}_2 \longrightarrow \mathtt{R}_1' + \mathtt{R}_2}$$

$$\mathtt{R}_1 \cdot (\mathtt{R}_2 + \mathtt{R}_3) \longrightarrow \mathtt{R}_1 \cdot \mathtt{R}_2 + \mathtt{R}_1 \cdot \mathtt{R}_3$$

Figure 4.6: Labeled transition semantics for statements. Expressions e follow standard semantics, and Δ is static program information.

$$\frac{\overline{e} \rightsquigarrow_\Delta \overline{u} \qquad l = \alpha!h(\overline{u})}{\alpha!h(\overline{e}).\mathtt{S} \xrightarrow{l}_\Delta \mathtt{S}} \qquad \textsf{add}(\alpha, \mathtt{R}).\mathtt{S} \xrightarrow{\textsf{add}(\alpha,\mathtt{R})}_\Delta \mathtt{S}$$

$$\frac{\alpha \ \textit{fresh} \qquad \overline{e} \rightsquigarrow_\Delta \overline{u} \qquad l = \alpha{:}b(\overline{u})}{\nu x{:}b(\overline{e}).\mathtt{S} \xrightarrow{l}_\Delta \mathtt{S}[\alpha/x]} \qquad \frac{\overline{e} \rightsquigarrow_\Delta \overline{u} \qquad l = \textsf{update}(\overline{u})}{\textsf{update}(\overline{e}) \xrightarrow{l}_\Delta \textsf{ready}}$$

$$\frac{e \rightsquigarrow_\Delta \textsf{true}}{\textsf{if}\ e\ \textsf{then}\ \mathtt{S}_1\ \textsf{else}\ \mathtt{S}_2 \xrightarrow{\textsf{if}}_\Delta \mathtt{S}_1} \qquad \frac{e \rightsquigarrow_\Delta \textsf{false}}{\textsf{if}\ e\ \textsf{then}\ \mathtt{S}_1\ \textsf{else}\ \mathtt{S}_2 \xrightarrow{\textsf{if}}_\Delta \mathtt{S}_2}$$

4.2.1 Operational Semantics

To formalize the program semantics, we first define an algebra on the extended requirement syntax (including the empty, conjunction, and satisfaction rules of Figure 4.3). The relation $\equiv$ on requirements is the least congruence relation that includes the rules of Figure 4.4. The empty requirement ϵ is the zero element for $+$ (disjunction) and the unit element for $\cdot$ (conjunction). Reductions on requirements are defined in Figure 4.5, and hold up to structural congruence. An empty requirement ϵ is always considered satisfied, while we say that the messages $h_1, \ldots, h_k$ satisfy a non-empty requirement $\mathtt{R}$ iff $(\ldots(\mathtt{R} \div h_1) \div h_2) \div \cdots) \div h_k) \longrightarrow^* \epsilon$.

Furthermore, we assume a reduction relation on expressions, such that the notation $e \rightsquigarrow_\Delta u$ means that expression e reduces to the value u, given static program information Δ. The latter is assumed to contain information extracted from the program, such as the parameters of message handlers. The transition relation for statements is defined in Figure 4.6, where we write $\mathtt{S} \xrightarrow{l}_\Delta \mathtt{S}'$ to say that a statement $\mathtt{S}$ reduces to $\mathtt{S}'$ via l. The label l records the action being taken; for example, $\alpha!h(\overline{e}).\mathtt{S}$ reduces to $\mathtt{S}$, and $l = \alpha!h(\overline{u})$ records the sent message. The values $\overline{u}$ are computed from the expressions $\overline{e}$, i.e., $\overline{e} \rightsquigarrow_\Delta \overline{u}$.

The transition relation $\mathtt{S} \xrightarrow{l}_\Delta \mathtt{S}'$ is referenced in the program-level rules of Figure 4.7, which transform runtime configurations. A runtime configuration C is a tuple (Δ, R, M, A),

Figure 4.7: Labeled transition semantics for actor configurations.

$$\textsc{Send}\ \frac{R(\beta) \div h \rightsquigarrow \mathtt{R}' \quad R' = R[\beta \mapsto \mathtt{R}'] \quad \mathtt{S} \xrightarrow{\beta!h(\overline{u})}_{\Delta} \mathtt{S}'}{\Delta,\, R,\, M,\, A \cup \{\langle \mathtt{S} \rangle_{\alpha}^{b(\overline{w})}\} \longrightarrow \Delta,\, R',\, M \cup \{\beta!h(\overline{u})\},\, A \cup \{\langle \mathtt{S}' \rangle_{\alpha}^{b(\overline{w})}\}}$$

$$\textsc{Receive}\ \frac{\mathtt{S} = \mathit{body}(\Delta, h) \quad \overline{y} = \mathit{params}(\Delta, h) \quad \overline{x} = \mathit{params}(\Delta, b)}{\Delta,\, R,\, M \cup \{\alpha!h(\overline{u})\},\, A \cup \{\langle \textbf{ready} \rangle_{\alpha}^{b(\overline{w})}\} \longrightarrow \Delta,\, R,\, M,\, A \cup \{\langle \mathtt{S}[\alpha/\textbf{self}][\overline{u}\overline{w}/\overline{y}\overline{x}] \rangle_{\alpha}^{b(\overline{w})}\}}$$

$$\textsc{Update}\ \frac{\mathtt{S} \xrightarrow{\textbf{update}(\overline{u})}_{\Delta} \mathtt{S}'}{\Delta,\, R,\, M,\, A \cup \{\langle \mathtt{S} \rangle_{\alpha}^{b(\overline{w})}\} \longrightarrow \Delta,\, R,\, M,\, A \cup \{\langle \mathtt{S}' \rangle_{\alpha}^{b(\overline{u})}\}}$$

$$\textsc{New}\ \frac{\mathtt{S} \xrightarrow{\beta:b(\overline{u})}_{\Delta} \mathtt{S}'}{\Delta,\, R,\, M,\, A \cup \{\langle \mathtt{S} \rangle_{\alpha}^{b_\alpha(\overline{w})}\} \longrightarrow \Delta,\, R,\, M,\, A \cup \{\langle \mathtt{S}' \rangle_{\alpha}^{b_\alpha(\overline{w})},\, \langle \textbf{ready} \rangle_{\beta}^{b(\overline{u})}\}}$$

$$\textsc{AddReq}\ \frac{R(\beta) \cdot \mathtt{R} \rightsquigarrow \mathtt{R}' \quad \mathtt{S} \xrightarrow{\textbf{add}(\beta, \mathtt{R})}_{\Delta} \mathtt{S}' \quad R' = R[\beta \mapsto \mathtt{R}']}{\Delta,\, R,\, M,\, A \cup \{\langle \mathtt{S} \rangle_{\alpha}^{b(\overline{w})}\} \longrightarrow \Delta,\, R',\, M,\, A \cup \{\langle \mathtt{S}' \rangle_{\alpha}^{b(\overline{w})}\}}$$

$$\textsc{If}\ \frac{\mathtt{S} \xrightarrow{\textbf{if}}_{\Delta} \mathtt{S}'}{\Delta,\, R,\, M,\, A \cup \{\langle \mathtt{S} \rangle_{\alpha}^{b(\overline{w})}\} \longrightarrow \Delta,\, R,\, M,\, A \cup \{\langle \mathtt{S}' \rangle_{\alpha}^{b(\overline{w})}\}} \qquad \textsc{Prog}\ \frac{\Delta = \mathit{info}(\overline{\mathtt{B}})}{\overline{\mathtt{B}}\,\mathtt{S} \longrightarrow \Delta, \emptyset, \emptyset, \{\langle \mathtt{S} \rangle_{\mathit{in}}^{\mathit{in}()}\}}$$

where Δ records static program information; R is a map from actor names to requirements; M is the multiset of pending (sent, but not received) messages; and A maps each actor name to a behavior, state, and executing statement. Elements of M have the form $\alpha!h(\overline{u})$, where α is the destination actor, h is the handler to be invoked upon receipt, and $\overline{u}$ are values constituting the message payload. We denote A as a set of elements of the form $\langle \mathtt{S} \rangle_{\alpha}^{b(\overline{w})}$, where α is the actor's name, b corresponds to its behavior, the values $\overline{w}$ constitute its state, and $\mathtt{S}$ is the statement the actor is currently executing.

We write $C \longrightarrow C'$ to say that the runtime configuration C reduces to C' via an application of some rule in Figure 4.7. By extension, $C_1 \longrightarrow C_2 \longrightarrow \cdots$ denotes a possibly infinite sequence of configurations where each adjacent pair follows the transition rules of Figure 4.7. Execution of a program $\mathtt{P} = \overline{\mathtt{B}}\,\mathtt{S}$ then consists of a sequence of transformations that starts from the program's initial configuration. Such an initial configuration is created via rule Prog in Figure 4.7, and it records information Δ from the program, associates no requirements with any actor, has an empty message set, and includes a single initial actor executing $\mathtt{S}$. This actor has reserved name and behavior *in*, and no state variables. When $R = \emptyset$, we define $R(\mathtt{x}) = \epsilon$ for all $\mathtt{x}$; i.e., by convention, $\emptyset$ maps no requirements to any actor. We assume a

Figure 4.8: Requirement algebra extended to requirement mappings and runtime configurations.

$$(R_1 \cdot R_2)(x) \overset{\text{def}}{=} R_1(x) \cdot R_2(x)$$
$$(R_1 + R_2)(x) \overset{\text{def}}{=} R_1(x) + R_2(x)$$
$$(R_1 \div R_2)(x) \overset{\text{def}}{=} R_1(x) \div R_2(x)$$

$$\frac{\mathtt{R} \longrightarrow \mathtt{R}'}{R \cup \{\mathtt{x} \mapsto \mathtt{R}\} \longrightarrow R \cup \{\mathtt{x} \mapsto \mathtt{R}'\}} \qquad \frac{\mathtt{R} \equiv \mathtt{R}'}{R \cup \{\mathtt{x} \mapsto \mathtt{R}\} \equiv R \cup \{\mathtt{x} \mapsto \mathtt{R}'\}}$$

$$\frac{R \longrightarrow R'}{(\Delta, R, M, A) \longrightarrow (\Delta, R', M, A)} \qquad \frac{R \equiv R'}{(\Delta, R, M, A) \equiv (\Delta, R', M, A)}$$

straightforward extension of the requirement algebra to runtime configurations, as shown in Figure 4.8. As always, reductions hold up to $\equiv$.

Rule AddReq deals with calls of the form **add**$(\beta, \mathtt{R})$ by appending $\mathtt{R}$ to $R(\beta)$, i.e., the requirements already associated with β. Note the use of $\div$ in Send: the rule adds the sent message to the multiset of pending messages, and reduces the requirements related to β, by re-mapping β to $R(\beta) \div h$. This corresponds to the fact that β will eventually receive h.

Only idle actors can receive messages [2]. Since statements take the form **ready** when completely reduced, Receive describes an idle actor α receiving a message to be processed by handler h. The statement $\mathtt{S}$ to execute is extracted from the program information Δ, and on it, the rule performs substitution of current values for handler and state variables. These values are taken from the message contents $\overline{u}$ and actor state $\overline{w}$. Handler and behavior (i.e., state) parameters are looked up via the auxiliary function *params*. Rule Update writes new values $\overline{u}$ to the state variables of α. Rule New creates a new actor $\langle \textbf{ready} \rangle_{\beta}^{b(\overline{u})}$ with unique name β, initialized with the given behavior b and values $\overline{u}$ for state variables. Rule If has the usual effect of deciding a conditional.

4.3 Type System

The typing rules are given in Figure 4.9. As before, Δ records static program information (such as the abstract syntax tree) which is used to retrieve, for example, the body of message handlers. R maps names to pending requirements, and $\mathtt{S}$ is the program statement being typed. Judgments have the form $R \vdash_{\Delta} \mathtt{S}$, read "under program information Δ and requirement map R, the statement $\mathtt{S}$ is well typed".

Rule T-Prog types programs, writing $\vdash \mathtt{P}$ to state that the program $\mathtt{P}$ is well-typed. The rule prescribes that $\mathtt{P} = \overline{\mathtt{B}}\,\mathtt{S}$ is well-typed, when using the information Δ extracted from

Figure 4.9: Static typing rules.

$$\textsc{T-Prog}\ \frac{\Delta = \mathit{info}(\overline{\mathtt{B}}) \qquad \emptyset \vdash_\Delta \mathtt{S}}{\vdash \overline{\mathtt{B}}\,\mathtt{S}}$$

$$\textsc{T-New}\ \frac{R \cup \{x' \mapsto \epsilon\} \vdash_\Delta \mathtt{S}[x'/x] \qquad x'\ \mathit{fresh}}{R \vdash_\Delta \nu x{:}b(\overline{e}).\mathtt{S}} \qquad \textsc{T-Add}\ \frac{\mathtt{R}_1 \cdot \mathtt{R} \rightsquigarrow \mathtt{R}' \qquad R \cup \{\mathtt{x} \mapsto \mathtt{R}'\} \vdash_\Delta \mathtt{S}}{R \cup \{\mathtt{x} \mapsto \mathtt{R}_1\} \vdash_\Delta \textbf{add}(\mathtt{x}, \mathtt{R}).\mathtt{S}}$$

$$\textsc{T-If}\ \frac{R_1 \vdash_\Delta \mathtt{S}_1 \qquad R_2 \vdash_\Delta \mathtt{S}_2 \qquad R \equiv R_1 + R_2}{R \vdash_\Delta \textbf{if}\ e\ \textbf{then}\ \mathtt{S}_1\ \textbf{else}\ \mathtt{S}_2} \qquad \textsc{T-Update}\ \frac{\forall \mathtt{x}.(\mathtt{x} \in \mathit{dom}(R) \implies R(\mathtt{x}) \longrightarrow^* \epsilon)}{R \vdash_\Delta \textbf{update}(\overline{e})}$$

$$\textsc{T-Send}\ \frac{\begin{array}{c}\overline{\mathtt{y}} = \mathit{actors}(\Delta, \overline{e}) \qquad \overline{z} = \mathit{actors}(\Delta, \mathit{params}(\Delta, h)) \\ \mathtt{S}_h = \mathit{body}(\Delta, h) \qquad \{\mathtt{x} \mapsto \mathtt{R}_1\,,\ \overline{\mathtt{y}} \mapsto \overline{\mathtt{R}}_2\} \vdash_\Delta \mathtt{S}_h[\mathtt{x}/\textbf{self}][\overline{\mathtt{y}}/\overline{z}] \\ R \cup \{\mathtt{x} \mapsto (\mathtt{R}_\mathtt{x} \div (h \cdot \mathtt{R}_1)),\ \overline{\mathtt{y}} \mapsto (\overline{\mathtt{R}}_\mathtt{y} \div \overline{\mathtt{R}}_2)\} \vdash_\Delta \mathtt{S}\end{array}}{R \cup \{\mathtt{x} \mapsto \mathtt{R}_\mathtt{x},\ \overline{\mathtt{y}} \mapsto \overline{\mathtt{R}}_\mathtt{y}\} \vdash_\Delta \mathtt{x}!h(\overline{e}).\mathtt{S}}$$

the behavior definitions $\overline{\mathtt{B}}$, the statement $\mathtt{S}$ is well-typed under the empty requirement map. Rule T-New requires that the statement following the creation command be typed with no requirements associated with the new actor. Rule T-Add demands that the statement following the **add**$(\mathtt{x}, \mathtt{R})$ command is well-typed under an environment which includes the new requirements $\mathtt{R}$ for $\mathtt{x}$. Per rule T-If, typing conditionals requires that each of the two branches satisfies the known requirements. Consistent with the fact that statements end in a construct of the form **update**$(\overline{e})$, rule T-Update is the base case of the recursive typing algorithm: it demands that all requirements known in the current scope have been satisfied.

Typing the action of sending a message takes into account that the execution of the related handler may satisfy some requirements known in the current context. Thus, rule T-Send demands that the statement $\mathtt{S}$ following the send command must be type-able under a "reduced" requirement map, from which we have removed the sent message h, and the requirements satisfied by the body of h. These include some requirements $\mathtt{R}_1$ associated with $\mathtt{x}$, as well as some requirements $\mathtt{R}_2$ associated with (some of[1]) the arguments $\overline{e}$.

To clarify the use of these rules, consider the example in Figure 4.10, and the respective typing in Figure 4.11. The program's main statement adds the requirement for a message m to y, but then does not contain a $y!m()$ statement; rather, it sends $h(y)$ to x. When x receives that message, the requirement for m will be satisfied in the body of the handler, i.e., the statement $z!m()$.**update**(), with z bound to y. For this reason, when the typing reaches $x!h(y)$, it requires both the typing of the body of h, and the remaining commands—as in the application of rule T-Send from Figure 4.11.

[1]The careful reader might observe that the rule does not care for **self** being part of the message payload. This poses no additional technical difficulty, and is omitted to simplify the presentation.

Figure 4.10: Example of requirement delegation.

```
bdef $b_1() = \{$
  hdef $h(z) = z!m().\textbf{update}()$     [sends $m$ to $z$ and returns]
$\}$
bdef $b_2() = \{$
  hdef $m() = \textbf{update}()$            [empty update, does nothing]
$\}$

$\nu x{:}b_1()$                             [creates $x$ with behavior $b_1$]
 $.\nu y{:}b_2()$                           [creates $y$ with behavior $b_2$]
  $.\textbf{add}(y, m)$                     [associates requirement for $m$ with $y$]
   $.x!h(y)$                                [sends $h$ to $x$, with $y$ in the payload]
    $.\textbf{update}()$                    [empty update, does nothing]
```

Limitations. Our system does not consider a requirement fulfilled, if the necessary messaging happens via state variables. To clarify this limitation, consider the program on the left-hand side of Figure 4.12. It includes two behavior definitions, b_1 and b_2, with one handler each: h_1 in b_1, and h_2 in b_2. Execution starts with the creation of actor x with behavior b_1, and actor y with behavior b_2. Actor y is created with $b_2(x)$, i.e., storing x in the behavior (state) variable z. The program proceeds to associate the requirement h_1 with x, then sends $h_2()$ to y. When y receives h_2, it will send $h_1()$ to x. However, the presented type system will reject the program, because the typing rules for message sending do not consult with the actor's state. Doing so requires the static tracking of dynamically changing actor state, and is the topic of future research.

The example on the right-hand side of Figure 4.12 shows two actors that exchange messages forever. Actor x sends h_2 to y, which replies with h_1, and so on. Let's see what happens when we attempt to type the body of h_1. First, the system encounters the call **add**(**self**, h_1), which adds a requirement for h_1 to **self**, i.e., x. Because the remaining statement is $z!h_2(\textbf{self}).\textbf{update}()$, the typing will have to proceed via rule T-Send. In accordance to the rule premises (page 46), we need to type the body of h_2. In doing so, we will eventually reach the statement **add**(**self**, h_2), adding a requirement for h_2 to **self**, i.e., y. The remaining statement is $z!h_1(\textbf{self}).\textbf{update}()$, and T-Send demands the typing of the body of h_1. Attempting to type h_1 essentially restarts the process, entering an infinite sequence of rule applications.

Note that on an intuitive level, this program has the discussed progress property: every generated requirement is eventually satisfied—even though a new one takes its place imme-

Figure 4.11: Typing the example of Figure 4.10.

$$
\textsc{T-New}\;\dfrac{
  \textsc{T-New}\;\dfrac{
    \textsc{T-Add}\;\dfrac{
      \textsc{T-Send}\;\dfrac{
        \begin{array}{c}
          \textsc{T-Send}\;\dfrac{
            \begin{array}{c}
              \{x' \mapsto \epsilon,\, y' \mapsto \epsilon\} \vdash_\Delta \ \textsf{\textbf{update}}() \\
              \Downarrow \ \mathit{body}(\Delta, m) = \textsf{\textbf{update}}() \\
              \{x' \mapsto \epsilon,\, y' \mapsto \epsilon\} \vdash_\Delta \ \mathit{body}(\Delta, m)[y'/\textsf{\textbf{self}}]
            \end{array}
            \quad
            \begin{array}{c}
              \{x' \mapsto \epsilon,\, y' \mapsto \epsilon\} \vdash_\Delta \ \textsf{\textbf{update}}() \\
              \Downarrow \ (m \div m) \longrightarrow \epsilon \\
              \{x' \mapsto \epsilon,\, y' \mapsto (m \div m)\} \vdash_\Delta \ \textsf{\textbf{update}}()
            \end{array}
          }{
            \{x' \mapsto \epsilon,\, y' \mapsto m\} \vdash_\Delta \ y'!m() \,.\, \textsf{\textbf{update}}()
          } \\
          \Downarrow \ \mathit{body}(\Delta, h) = z!m() \,.\, \textsf{\textbf{update}}() \\
          \{x' \mapsto \epsilon,\, y' \mapsto m\} \vdash_\Delta \ \mathit{body}(\Delta, h)[y'/z]
        \end{array}
        \quad
        \begin{array}{c}
          \{x' \mapsto \epsilon,\, y' \mapsto \epsilon\} \vdash_\Delta \ \textsf{\textbf{update}}() \\
          \Downarrow \ (m \div m) \longrightarrow \epsilon \\
          \{x' \mapsto \epsilon \div h,\, y' \mapsto (m \div m)\} \vdash_\Delta \ \textsf{\textbf{update}}()
        \end{array}
      }{
        \{x' \mapsto \epsilon,\, y' \mapsto m\} \vdash_\Delta \ x'!h(y') \,.\, \textsf{\textbf{update}}()
      }
    }{
      \{x' \mapsto \epsilon,\, y' \mapsto \epsilon\} \vdash_\Delta \ \textsf{\textbf{add}}(y',\, m) \,.\, x'!h(y') \,.\, \textsf{\textbf{update}}()
    }
  }{
    \{x' \mapsto \epsilon\} \vdash_\Delta \ \nu y{:}b_2 \,.\, \textsf{\textbf{add}}(y,\, m) \,.\, x'!h(y) \,.\, \textsf{\textbf{update}}()
  }
}{
  \emptyset \vdash_\Delta \ \nu x{:}b_1() \,.\, \nu y{:}b_2() \,.\, \textsf{\textbf{add}}(y,\, m) \,.\, x!h(y) \,.\, \textsf{\textbf{update}}()
}
$$

Figure 4.12: Untypeable examples.

$$\begin{tikzcd}
x : b_2() & & y : b_2(x) \arrow[ll, "h_1()"'] \\
& & \\
\mathtt{P} \arrow[uurr, "h_2()"'] & &
\end{tikzcd}$$

```
bdef b1() = {
  hdef h1() = update()
}
bdef b2(z) = {
  hdef h2() =
    z!h1()
     .update(z)
}
νx:b1()
 .νy:b2(x)
  .add(x, h1)
   .y!h2()
    .update()
```

```
bdef b1() = {
  hdef h1(z) =
    add(self, h1)
      . z!h2(self)
        . update()
}

bdef b2() = {
  hdef h2(z) =
    add(self, h2)
      . z!h1(self)
        . update()
}

νx : b1()
  . νy:b2()
    . x!h1(y)
      . update()
```

$$\begin{tikzcd}
x : b_1() \arrow[rr, bend left, "h_2(x)"] & & y : b_2() \arrow[ll, bend left, "h_1(y)"] \\
\mathtt{P} \arrow[u, "h_1(y)"] & &
\end{tikzcd}$$

diately after. This can be made obvious from the program's reduction sequence, shown in Figure 4.13 with minor simplifications. Observe that the configurations marked with $(*)$ are identical, and that the part corresponding to the requirement mapping $\{x \mapsto h_1\}$ is tracked by the type system in a manner that mirrors execution, thus failing to terminate. A possible solution to the problem is discussed in Section 4.5.

4.4 Calculus Meta-Theory

In order to establish our main result, we extend the typing relation to runtime configurations:

Definition 4.1 (Runtime Typing)**.** Let C be a runtime configuration. We say that C is well-typed, written $\vdash C$, iff C satisfies the rules shown in Figure 4.14.

Note rule R-Transition, which is defined with the intention of forcing the runtime typing to unfold program execution—facilitating the proofs of this section. For example, we can show that typing holds up to equivalence and requirement reductions (Figure 4.8), captured by the next lemma.

Lemma 4.1. The following rules hold true:

$$\frac{C_1 \equiv C_2 \quad \vdash C_1}{\vdash C_2} \qquad \frac{R \longrightarrow R' \quad \vdash (\Delta, R, M, A)}{\vdash (\Delta, R', M, A)}$$

Figure 4.13: Simplified reduction sequence for the right-hand side program of Figure 4.12.

$$\begin{array}{cccccc}
 & \Delta, & \emptyset, & \emptyset, & \{\langle \nu x{:}b_1().\nu y{:}b_2.x!h_1(y).\textsf{update}()\rangle_{in}^{in()}\} & \\
\longrightarrow^* & \Delta, & \emptyset, & \emptyset, & \{\langle \textsf{ready}\rangle_{x}^{b_1()},\ \langle \textsf{ready}\rangle_{y}^{b_2()},\ \langle x!h_1(y).\textsf{update}()\rangle_{in}^{in()}\} & \\
\longrightarrow^* & \Delta, & \emptyset, & \{x!h_1(y)\}, & \{\langle \textsf{ready}\rangle_{x}^{b_1()},\ \langle \textsf{ready}\rangle_{y}^{b_2()},\ \langle \textsf{ready}\rangle_{in}^{in()}\} & \\
\longrightarrow & \Delta, & \emptyset, & \emptyset, & \{\langle \textsf{add}(\textsf{self},h_1).y!h_2(\textsf{self}).\textsf{update}()\rangle_{x}^{b_1()},\ \langle \textsf{ready}\rangle_{y}^{b_2()},\ \langle \textsf{ready}\rangle_{in}^{in()}\} & \\
\longrightarrow & \Delta, & \{x\mapsto h_1\}, & \emptyset, & \{\langle y!h_2(\textsf{self}).\textsf{update}()\rangle_{x}^{b_1()},\ \langle \textsf{ready}\rangle_{y}^{b_2()},\ \langle \textsf{ready}\rangle_{in}^{in()}\} & \\
\longrightarrow^* & \Delta, & \{x\mapsto h_1\}, & \{y!h_2(x)\}, & \{\langle \textsf{ready}\rangle_{x}^{b_1()},\ \langle \textsf{ready}\rangle_{y}^{b_2()},\ \langle \textsf{ready}\rangle_{in}^{in()}\} & (*) \\
\longrightarrow & \Delta, & \{x\mapsto h_1\}, & \emptyset, & \{\langle \textsf{ready}\rangle_{x}^{b_1()},\ \langle \textsf{add}(\textsf{self},h_2).x!h_1(\textsf{self}).\textsf{update}()\rangle_{y}^{b_2()},\ \langle \textsf{ready}\rangle_{in}^{in()}\} & \\
\longrightarrow & \Delta, & \{x\mapsto h_1,\ y\mapsto h_2\}, & \emptyset, & \{\langle \textsf{ready}\rangle_{x}^{b_1()},\ \langle x!h_1(\textsf{self}).\textsf{update}()\rangle_{y}^{b_2()},\ \langle \textsf{ready}\rangle_{in}^{in()}\} & \\
\longrightarrow^* & \Delta, & \{y\mapsto h_2\}, & \{x!h_1(y)\}, & \{\langle \textsf{ready}\rangle_{x}^{b_1()},\ \langle \textsf{ready}\rangle_{y}^{b_2()},\ \langle \textsf{ready}\rangle_{in}^{in()}\} & \\
\longrightarrow & \Delta, & \{y\mapsto h_2\}, & \emptyset, & \{\langle \textsf{add}(\textsf{self},h_1).y!h_2(\textsf{self}).\textsf{update}()\rangle_{x}^{b_1()},\ \langle \textsf{ready}\rangle_{y}^{b_2()},\ \langle \textsf{ready}\rangle_{in}^{in()}\} & \\
\longrightarrow & \Delta, & \{x\mapsto h_1,\ y\mapsto h_2\}, & \emptyset, & \{\langle y!h_2(\textsf{self}).\textsf{update}()\rangle_{x}^{b_1()},\ \langle \textsf{ready}\rangle_{y}^{b_2()},\ \langle \textsf{ready}\rangle_{in}^{in()}\} & \\
\longrightarrow^* & \Delta, & \{x\mapsto h_1\}, & \{y!h_2(x)\}, & \{\langle \textsf{ready}\rangle_{x}^{b_1()},\ \langle \textsf{ready}\rangle_{y}^{b_2()},\ \langle \textsf{ready}\rangle_{in}^{in()}\} & (*) \\
 & & & & \vdots &
\end{array}$$

Figure 4.14: Runtime typing rules.

$$\textsc{R-Transition}\ \frac{C \longrightarrow \qquad \forall C'.(C \longrightarrow C' \implies \vdash C')}{\vdash C}
\qquad
\textsc{R-Ready}\ \frac{\begin{array}{c}\forall \mathtt{x}.(\mathtt{x} \in \mathit{dom}(R) \implies R(\mathtt{x}) \longrightarrow^* \epsilon) \\ \forall \mathtt{S},\, b, \overline{w},\, \alpha.(\langle \mathtt{S}\rangle_{\alpha}^{b(\overline{w})} \in A \implies \mathtt{S} = \textsf{ready})\end{array}}{\vdash \Delta,\, R,\, \emptyset,\, A}$$

The proof is by induction on the structure of runtime typing derivations, and is omitted. The main result of this chapter is that during executions of well-typed programs, all requirements generated dynamically are eventually satisfied; that is, runtime configurations satisfy the *progress* property:

Definition 4.2 (Progress)**.** Let $C = (\Delta, R, M, A)$ be a runtime configuration. We say that C satisfies the progress property, written $\mathbb{P}(C)$, iff for all executions $C \longrightarrow C_1 \longrightarrow \cdots \longrightarrow C_k$ that start from C, it is $C_k = (\Delta, R_k, M_k, A_k)$ with $R_k(\mathtt{x}) \longrightarrow^* \epsilon$ for all $\mathtt{x} \in \mathit{dom}(R_k)$.

We remind the reader that the initial configuration of a program $\mathtt{P}$ is denoted with $\mathit{init}(\mathtt{P})$, and that $\longrightarrow^*$ is the transitive reflexive closure of the relation $\longrightarrow$. We can now state the main result:

Theorem 4.1. Let $\mathtt{P}$ be a program. Assuming statements $\mathtt{S}$ terminate, $\vdash \mathtt{P}$ and $\mathit{init}(\mathtt{P}) \longrightarrow^* C$ imply $\mathbb{P}(C)$.

The theorem states that all configurations reachable from the initial configuration of a well-typed program satisfy the progress property, notwithstanding the divergence of expressions (denoted with e in Figure 4.3).

Proof outline. The main idea is to show that

(i) well-typed programs generate well-typed initial configurations, that is, $\vdash \texttt{P}$ implies $\vdash \mathit{init}(\texttt{P})$;

(ii) the reduction relation of Figure 4.7 preserves typing, that is, $\vdash C$ and $C \longrightarrow^* C'$ imply $\vdash C'$; and

(iii) well-typed configurations satisfy the progress property, that is, $\vdash C$ implies $\mathbb{P}(C)$.

In other words, we show that the typing of configurations guarantees progress, and that reduction preserves the progress property. We proceed to prove the above items in sequence.

Recalling that satisfying $\texttt{R}_1 \cdot \texttt{R}_2$ requires the satisfaction of both $\texttt{R}_1$ and $\texttt{R}_2$, we state—without proof—an auxiliary lemma, which can be shown by induction on the structure of runtime typing derivations:

Lemma 4.2. If $\vdash (\Delta,\, R_1,\, M_1,\, A_1)$ and $\vdash (\Delta,\, R_2,\, M_2,\, A_2)$, then
$\vdash (\Delta,\ R_1 \cdot R_2,\ M_1 \cup M_2,\ A_1 \cup A_2)$.

Moreover, the following is an immediate consequence of figures 4.4 and 4.5:

Lemma 4.3. $(\texttt{R}_1 \div \texttt{R}_2) \cdot \texttt{R}_2 \longrightarrow^* \texttt{R}_1$

The lemma that follows captures our intuition that the static typing of programs (per Figure 4.9) implies that the respective runtime configurations are well-typed (per Figure 4.14).

Lemma 4.4. Let $\texttt{S}$ be a statement where **self** has been replaced by a runtime name α, and let A consist solely of actors executing **ready**. Then, for any static program information Δ, requirement map R, behavior instantiation $b(\overline{u})$, and variables $\overline{x}$ with $|\overline{x}| = |\overline{u}|$, we have that $R[\overline{u}/\overline{x}] \vdash_\Delta \texttt{S}[\overline{u}/\overline{x}]$ implies $\vdash \big(\Delta,\, R[\overline{u}/\overline{x}],\, \emptyset,\, A \cup \{\langle \texttt{S}[\overline{u}/\overline{x}] \rangle_\alpha^{b(\overline{u})}\}\big)$.

Proof. We proceed by induction on the syntax of statements.

<u>Base case.</u>

From Figure 4.3, the base case is that of the **update** call. Let Δ, R, A, α, $\overline{x}$ and $b(\overline{u})$ be as per the statement of the lemma. Moreover, fix values $\overline{w}$ with $|\overline{w}| = |\overline{u}|$. We need to show that

$$R[\overline{u}/\overline{x}] \vdash_\Delta \underbrace{\textbf{update}(\overline{w})[\overline{u}/\overline{x}]}_{\textbf{update}(\overline{w})} \quad \text{implies}$$

$$\vdash \quad (\Delta, \quad R[\overline{u}/\overline{x}], \quad \emptyset, \quad A \ \cup\ \{\langle \underbrace{\textbf{update}(\overline{w})[\overline{u}/\overline{x}]}_{\textbf{update}(\overline{w})} \rangle_\alpha^{b(\overline{u})}\}).$$

Notice that the variables $\overline{x}$ do not appear in the values $\overline{w}$, and so the substitution $[\overline{u}/\overline{x}]$ leaves

$\textsf{\textbf{update}}(\overline{w})$ unchanged. Assume $R[\overline{u}/\overline{x}] \vdash_\Delta \textsf{\textbf{update}}(\overline{w})$ per rule T-Update in Figure 4.9, i.e.,

$$\forall \mathtt{y}.(\mathtt{y} \in \mathit{dom}(R[\overline{u}/\overline{x}]) \implies R[\overline{u}/\overline{x}](\mathtt{y}) \longrightarrow^* \epsilon) \tag{4.1}$$

From 4.1 and rule R-Ready in Figure 4.14, we have that

$$\vdash (\Delta,\, R[\overline{u}/\overline{x}],\, \emptyset,\, A \cup \{\langle \textsf{\textbf{ready}} \rangle_\alpha^{b(\overline{w})}\}) \tag{4.2}$$

which, by R-Transition, implies

$$\vdash \big(\Delta,\, R[\overline{u}/\overline{x}],\, \emptyset,\, A \cup \{\langle \textsf{\textbf{update}}(\overline{w}) \rangle_\alpha^{b(\overline{u})}\}\big).$$

<u>Inductive step – message sending.</u>

Let Δ, R, A, α, $\overline{x}$ and $b(\overline{u})$ be as per the statement of the lemma. Moreover, fix a message handler h, values $\overline{w}$, an actor name β, a statement $\mathtt{S}$, a behavior instantiation $b'(\overline{u}')$, and variables $\overline{x}'$ with $|\overline{x}'| = |\overline{u}'|$. Assume that $A = A_1 \cup A_2 \cup \{\langle \textsf{\textbf{ready}} \rangle_\beta^{b'(\overline{u}')}\}$ for some A_1 and A_2 consisting solely of **ready** actors, and that $R = R'[\overline{u}/\overline{x}]$ for some R'. Also, assume that $\mathtt{S} = \mathtt{S}_0[\overline{u}/\overline{x}]$ for some $\mathtt{S}_0$ where **self** has been replaced with α. Further assumptions on $\overline{x}'$ and $\overline{u}'$ will become clear in the next few steps. We need to prove that

$$R \vdash_\Delta \beta!h(\overline{w}).\mathtt{S} \;\text{ implies }\; \vdash \big(\Delta,\, R,\, \emptyset,\, A \cup \{\langle \beta!h(\overline{w}).\mathtt{S} \rangle_\alpha^{b(\overline{u})}\}\big).$$

Assume $R \vdash_\Delta \beta!h(\overline{w}).\mathtt{S}$ was derived via an application of rule T-Send, and thus

$$R \;=\; \underbrace{R_0}_{R_{01}[\overline{u}/\overline{x}]} \;\cup\; \underbrace{\{\beta \mapsto \mathtt{R}_\beta,\; \overline{\gamma} \mapsto \overline{\mathtt{R}}_\gamma\}}_{R_{02}[\overline{u}/\overline{x}]} \tag{4.3}$$

for some mapping R_0, requirements $\mathtt{R}_\beta$ and $\overline{\mathtt{R}}_\gamma$, and $\overline{\gamma} = \mathit{actors}(\Delta, \overline{w})$. From T-Send, it is

$$\underbrace{R_0}_{R_{01}[\overline{u}/\overline{x}]} \;\cup\; \underbrace{\{\beta \mapsto \mathtt{R}_\beta \div (h \cdot \mathtt{R}_1),\; \overline{\gamma} \mapsto \overline{\mathtt{R}}_\gamma \div \overline{\mathtt{R}}_2\}}_{R_{03}[\overline{u}/\overline{x}]} \vdash_\Delta \mathtt{S} \tag{4.4}$$

$$\text{and} \quad \{\beta \mapsto \mathtt{R}_1,\; \overline{\gamma} \mapsto \overline{\mathtt{R}}_2\} \vdash_\Delta \underbrace{\mathit{body}(\Delta, h)[\beta/\textsf{\textbf{self}}][\overline{\gamma}/\overline{z}]}_{\mathtt{S}'_h = \mathtt{S}_h[\overline{u}'/\overline{x}']} \tag{4.5}$$

where $\mathtt{R}_1$, $\overline{\mathtt{R}}_2$, $\overline{\gamma}$ and $\overline{z}$ are as in rule T-Send. From the inductive hypothesis, 4.4 implies

$$\vdash \big(\Delta,\ R_0 \cup \{\beta \mapsto \mathtt{R}_\beta \div (h \cdot \mathtt{R}_1),\ \overline{\gamma} \mapsto \overline{\mathtt{R}}_\gamma \div \overline{\mathtt{R}}_2\},\ \emptyset,\ A_1 \cup \{\langle \mathtt{S} \rangle_\alpha^{b(\overline{u})}\}\big) \tag{4.6}$$

since A_1 consists solely of **ready** actors. Note that the mapping $\{\beta \mapsto \mathtt{R}_1, \overline{\gamma} \mapsto \overline{\mathtt{R}}_2\}$ in 4.5 subsumes the substitution $[\overline{u}'/\overline{x}']$. As a result, we can apply the inductive hypothesis on 4.5 to get

$$\vdash \big(\Delta,\ \{\beta \mapsto \mathtt{R}_1,\ \overline{\gamma} \mapsto \overline{\mathtt{R}}_2\},\ \emptyset,\ A_2 \cup \{\langle \mathtt{S}'_h \rangle_\beta^{b'(\overline{u}')}\}\big) \tag{4.7}$$

because A_2 consists solely of **ready** actors. Combining 4.6 and 4.7 per Lemma 4.2, we get

$$\begin{aligned}\vdash \big(\Delta,\ R_0 \cup \{\beta \mapsto (\mathtt{R}_\beta \div (h \cdot \mathtt{R}_1)) \cdot \mathtt{R}_1,\ \overline{\gamma} \mapsto (\overline{\mathtt{R}}_\gamma \div \overline{\mathtt{R}}_2) \cdot \overline{\mathtt{R}}_2\}\\ \emptyset,\ A_1 \cup A_2 \cup \{\langle \mathtt{S} \rangle_\alpha^{b(\overline{u})},\ \langle \mathtt{S}'_h \rangle_\beta^{b(\overline{u}')}\}\big)\end{aligned} \tag{4.8}$$

We apply Lemma 4.3 and Lemma 4.1 to 4.8 to get

$$\vdash \big(\Delta,\ R_0 \cup \{\beta \mapsto \mathtt{R}_\beta \div h,\ \overline{\gamma} \mapsto \overline{\mathtt{R}}_\gamma\},\ \emptyset,\ A_1 \cup A_2 \cup \{\langle \mathtt{S} \rangle_\alpha^{b(\overline{u})},\ \langle \mathtt{S}'_h \rangle_\beta^{b(\overline{u}')}\}\big) \tag{4.9}$$

We remind the reader that $\overline{u}'$ contains (among others) names in $\overline{w}$, and that $\mathtt{S}'_h$ is the body of handler h with the required substitutions. Thus, by rule R-Transition, 4.9 implies

$$\begin{aligned}\vdash \big(\Delta,\ R_0 \cup \{\beta \mapsto \mathtt{R}_\beta \div h,\ \overline{\gamma} \mapsto \overline{\mathtt{R}}_\gamma\},\\ \{\beta!h(\overline{w})\},\ A_1 \cup A_2 \cup \{\langle \mathtt{S} \rangle_\alpha^{b(\overline{u})},\ \langle \textbf{ready} \rangle_\beta^{b'(\overline{u}')}\}\big).\end{aligned}$$

Since $A = A_1 \cup A_2 \cup \{\langle \textbf{ready} \rangle_\beta^{b'(\overline{u}')}\}$, the above can be written

$$\vdash \big(\Delta,\ R_0 \cup \{\beta \mapsto \mathtt{R}_\beta \div h,\ \overline{\gamma} \mapsto \overline{\mathtt{R}}_\gamma\},\ \{\beta!h(\overline{w})\},\ A \cup \{\langle \mathtt{S} \rangle_\alpha^{b(\overline{u})}\}\big).$$

Applying R-Transition again, the above implies

$$\vdash \big(\Delta,\ R_0 \cup \{\beta \mapsto \mathtt{R}_\beta,\ \overline{\gamma} \mapsto \overline{\mathtt{R}}_\gamma\},\ \emptyset,\ A \cup \{\langle \beta!h(\overline{w}).\mathtt{S} \rangle_\alpha^{b(\overline{u})}\}\big).$$

From 4.3, the above is the same as

$$\vdash \big(\Delta,\ R,\ \emptyset,\ A \cup \{\langle \beta!h(\overline{w}).\mathtt{S} \rangle_\alpha^{b(\overline{u})}\}\big)$$

which completes the proof for message sending. The rest of the cases are simpler, and are thus omitted in the interest of space. □

Corollary 4.1 (Static Typing Implies Runtime Typing.)**.** $\vdash \mathtt{P}$ implies $\vdash \mathit{init}(\mathtt{P})$ for all programs $\mathtt{P}$.

Proof. Let $\mathtt{P} = \overline{\mathtt{B}}\,\mathtt{S}$ be a program, and assume **self** does not appear in $\mathtt{S}$ (**self** does not make sense in the context of the initial actor). We apply Lemma 4.4 to $\emptyset \vdash_{\Delta} \mathtt{S}$ and $\vdash (\Delta, \emptyset, \emptyset, \{\langle \mathtt{S} \rangle_{in}^{in()}\})$ with $\Delta = \mathit{info}(\overline{\mathtt{B}})$. □

We now show that the reduction relation of Figure 4.7 preserves typing:

Lemma 4.5 (Type Preservation)**.** Let C be a runtime configuration. Then $\vdash C$ and $C \longrightarrow^* C'$ imply $\vdash C'$.

Proof. Assume $\vdash C$, which means that one of the rules in Figure 4.14 (page 50) applies. If there exists C' s.t. $C \longrightarrow C'$, then $\vdash C'$ from the definition of typing rule R-Transition. If $C \not\longrightarrow$, i.e., $C \longrightarrow^* C$, the only possibility is that C is a quiescent state, i.e., there are no messages to be delivered, and all actor statements have been reduced to **ready**. Per rule R-Ready, C is well-typed. □

Let C be a well-typed runtime configuration. Then a derivation of $\vdash C$ according to the rules of Figure 4.14 forms a tree with root $\vdash C$, such that every path on this tree is a sequence of applications of R-Transition that ends in a single application of R-Ready. On each such sequence, we focus on the configurations on the rule conclusions, say $C, C_1 \ldots C_k$. We write $\mathit{Paths}(\vdash C)$ for the set of all such sequences of configurations. As it turns out, $\mathit{Paths}(\vdash C)$ includes all possible executions from configuration C:

Lemma 4.6 (Typing Unfolds Execution)**.** Let C_1 be a well-typed configuration, i.e., $\vdash C_1$ and $C_1 \longrightarrow \cdots \longrightarrow C_k$ an execution from C_1. Then $(C_1, \ldots, C_k) \in \mathit{Paths}(\vdash C)$.

Proof. Directly from Lemma 4.5. □

Finally, item (iii) from the proof outline is captured in the statement below:

Lemma 4.7 (Runtime Typing Guarantees Progress)**.** Let C be a configuration. Then $\vdash C$ implies $\mathbb{P}(C)$.

Proof. A derivation of $\vdash C$ follows the rules of Figure 4.14, and hence, such a derivation ends in an application of rule R-Ready. Thus, every sequence in $\mathit{Paths}(\vdash C)$ ends in some configuration C_k for which $\vdash C_k$ is given by rule R-Ready. From the definition of the rule, C_k must be a quiescent state with no requirements. By Lemma 4.6 and the fact that $\vdash C$, every execution from C ends in such a state. □

We are now ready to prove the main result:

Proof (Theorem 4.1). Direct consequence of

$$\begin{array}{ll} \vdash \mathtt{P} \text{ implies } \vdash \mathit{init}(\mathtt{P}) & \text{(Corollary 4.1)} \\ \vdash C \text{ and } C \longrightarrow^{*} C' \text{ implies } \vdash C' & \text{(Lemma 4.5)} \\ \vdash C \text{ implies } \mathbb{P}(C) & \text{(Lemma 4.7).} \end{array}$$

□

Error programs. The progress property $\mathbb{P}(C)$ ensures that a configuration C reduces to a state where all requirements have been satisfied. The property implicitly captures the definition of *error* programs: those that can result in a configuration C for which $\mathbb{P}(C)$ does not hold. More formally,

Definition 4.3 (Error Program)**.** Let `P` be a program in the syntax of page 42. It is *error*(`P`) when `P` $\rightsquigarrow (\Delta, R, M, A)$ and there exists α for which $R(\alpha) \not\longrightarrow{}^{*}\epsilon$.

In other words, an error program is one that can reduce to a configuration with non-empty requirements, and where no reduction rules (page 44) apply. Theorem 4.1 directly implies that well-typed programs are not error:

Corollary 4.2. Let `P` be a program. Then $\vdash$ `P` implies that `P` $\notin$ *error*.

Proof. Direct consequence of Theorem 4.1 and Definition 4.2. □

4.5 On Cyclic Communication Patterns

The typing of Figure 4.9 permits some cases of indirect satisfaction of requirements, i.e., actors delegating obligations to one another—a typical behavior in actor systems. However, the presented rules demand that such delegation be linear, in the sense that symbolic tracing of the code should not revisit the same parts in a circular fashion, to avoid infinite loops. As it turns out, this requirement is an artifact of the design of the typing algorithm, which was chosen to make the presentation easier to follow. This section discusses alternative typing rules that would allow many cases of cyclic communication patterns, in large part solving the issue analyzed on pages 46–49.

The strategy we follow is to track the code starting from each **add** call site, and declare success if that particular requirement is found to be satisfied at some point later. If this symbolic execution reaches the same **add** site without finding an appropriate send command, then the algorithm can safely declare failure.

To better understand the proposed extension, consider the program in Figure 4.15, repeated from page 49. The diagram on the right describes the interaction pattern: an

Figure 4.15: Infinite, cyclic requirement generation example.

```
 1  bdef $b_1() = \{$
 2    hdef $h_1(z) =$
 3      add(self, $h_1$)
 4        . $z!h_2$(self)
 5          . update()
 6  $\}$
 7
 8  bdef $b_2() = \{$
 9    hdef $h_2(z) =$
10      add(self, $h_2$)
11        . $z!h_1$(self)
12          . update()
13  $\}$
14
15  $\nu x : b_1()$
16    . $\nu y{:}b_2()$
17      . $x!h_1(y)$
18        . update()
```

$$\begin{tikzcd}
x : b_1() \arrow[rr, bend left, "h_2(x)"] & & y : b_2() \arrow[ll, bend left, "h_1(y)"] \\
\mathtt{P} \arrow[u, "h_1(y)"] & &
\end{tikzcd}$$

initial message h_1 to actor x causes x and y to exchange messages h_1 and h_2 indefinitely. The complication arises because both actors generate a requirement for themselves before satisfying the requirement they already know for the other actor: on line 3, actor x adds a requirement for h_1 to itself; then, on line 4, it satisfies the requirement h_2 for y, to whom it relies for the satisfaction of its own (just added) requirement for h_1.

In summary, the problem with typing this example with the rules in Figure 4.9 is that they unfold program execution, including the generation of requirements—while demanding that the code in a handler ends with an empty requirement set. Thus, for the example of Figure 4.15, the system of Figure 4.9 (a) does not detect that requirements are indeed satisfied before new ones are generated, and (b) does not terminate.

The first problem can be alleviated if we look at each **add** statement separately: by unfolding execution from the **add** statement onward, we can declare success if we find appropriate send commands before looping back to that **add** statement. If no such command was reached and no more unfolding is possible, i.e., the call graph that begins at the particular **add** statement is a chain, then we can safely assume failure. The second problem, i.e., termination, can be tackled by "remembering" the initial **add** call site and breaking the loop if it is encountered again. We make these ideas more precise with the rules in Figure 4.16.

Figure 4.16: Augmented static typing for progress, solving the cyclicity problem of page 46.

$$\textsc{Augm-Send}\ \frac{\mathtt{S}_h = \mathit{body}(\Delta, h) \qquad k, \{\mathtt{x} \mapsto \mathtt{R}_2\} \vdash_\Delta \mathtt{S}_h[\mathtt{x}/\textsf{\textbf{self}}] \qquad k, \{\mathtt{x} \mapsto (\mathtt{R}_1 \div (h \cdot \mathtt{R}_2))\} \vdash_\Delta \mathtt{S}}{k, \{\mathtt{x} \mapsto \mathtt{R}_1\} \vdash_\Delta \mathtt{x}!h(\overline{e}).\mathtt{S}}$$

$$\textsc{Augm-Delegate}\ \frac{\overline{z} = \mathit{params}(\Delta, h) \qquad \mathtt{x} \neq \mathtt{y} \qquad \mathtt{S}_h = \mathit{body}(\Delta, h) \qquad k, \{\mathtt{x} \mapsto \mathtt{R}_2\} \vdash_\Delta \mathtt{S}_h[\mathtt{y}/\textsf{\textbf{self}}][\overline{e}_1\, \mathtt{x}\, \overline{e}_2 \,/\, \overline{z}] \qquad k, \{\mathtt{x} \mapsto (\mathtt{R}_1 \div \mathtt{R}_2)\} \vdash_\Delta \mathtt{S}}{k, \{\mathtt{x} \mapsto \mathtt{R}_1\} \vdash_\Delta \mathtt{y}!h(\overline{e}_1\mathtt{x}\overline{e}_2).\mathtt{S}}$$

$$\textsc{Augm-If}\ \frac{k, \{\mathtt{x} \mapsto \mathtt{R}_1\} \vdash_\Delta \mathtt{S}_1 \qquad k, \{\mathtt{x} \mapsto \mathtt{R}_2\} \vdash_\Delta \mathtt{S}_2 \qquad \mathtt{R}_1 \equiv \mathtt{R}_1 + \mathtt{R}_2}{k, \{\mathtt{x} \mapsto \mathtt{R}\} \vdash_\Delta \textsf{\textbf{if}}\ e\ \textsf{\textbf{then}}\ \mathtt{S}_1\ \textsf{\textbf{else}}\ \mathtt{S}_2}$$

$$\textsc{Augm-New}\ \frac{k, \{\mathtt{x} \mapsto \mathtt{R}\} \vdash_\Delta \mathtt{S}[y'/y] \qquad y' \text{ fresh}}{k, \{\mathtt{x} \mapsto \mathtt{R}\} \vdash_\Delta \nu y{:}b(\overline{e}).\mathtt{S}}$$

$$\textsc{Augm-Update}\ \frac{\mathtt{R} \longrightarrow^* \epsilon}{k, \{\mathtt{x} \mapsto \mathtt{R}\} \vdash_\Delta \textsf{\textbf{update}}(\overline{e})}$$

$$\textsc{Augm-Add}\ \frac{k_1 \neq k_2 \qquad k_1, \{\mathtt{x} \mapsto \mathtt{R}_1 \cdot \mathtt{R}_2\} \vdash_\Delta \mathtt{S}}{k_1, \{\mathtt{x} \mapsto \mathtt{R}_1\} \vdash_\Delta \textsf{\textbf{add}}(\mathtt{x}, \mathtt{R}_2)^{k_2}.\mathtt{S}}$$

$$\textsc{Augm-Add-End}\ \frac{\mathtt{R} \longrightarrow^* \epsilon}{k, \{\mathtt{x} \mapsto \mathtt{R}\} \vdash_\Delta \textsf{\textbf{add}}(\mathtt{x}, \mathtt{R})^{k}.\mathtt{S}}$$

$$\textsc{Augm-Add-Start}\ \frac{k, \{\mathtt{x} \mapsto \mathtt{R}\} \vdash_\Delta \mathtt{S}}{\vdash_\Delta \textsf{\textbf{add}}(\mathtt{x}, \mathtt{R})^{k}.\mathtt{S}}$$

The rules assume that each **add** statement is associated with a unique token k, as in $\textbf{add}(\mathtt{x}, \mathtt{R})^k$. Rule Augm-Add-Start bootstraps the process, with k carried along the relation $\vdash$, which unfolds execution. The algorithm only keeps track of one actor name `x` and its requirements `R`, and declares success if it loops back to the same k with an empty `R` (rule Augm-Add-End). Rule Augm-Send reduces the tracked requirements by the sent message h, as well as the requirements $\mathtt{R}_2$ satisfied in the handler body $\mathtt{S}_h$. Rule Augm-Delegate reduces the requirements of `x` when it is passed as an argument to the message h. Notice that Augm-Update demands that no requirements remain unsatisfied.

The difference from the system of Figure 4.9 lies in the fact that the rules presented here are intended to apply on each **add** call site independently, via application of Augm-Add-Start. Endless looping is prevented by keeping track of the unique token k, and stopping when it is reached again (rule Augm-Add-End).

The Fairness Requirement. The typing strategy of the current section only guarantees requirement satisfaction for fair executions [55]. As discussed in Section 2.1.3, fair executions are those where enabled transitions are not infinitely postponed. Without this assumption, the program of Figure 4.15 would not be guaranteed to satisfy all generated requirements; for instance, $h_2(x)$ is not guaranteed to arrive at y, which is the only way h_1 gets sent to x after line 3.

The typing rules in Figure 4.16 follow the rationale that after encountering an **add** operation, execution must reach a configuration where a suitable **send** transition is taken, i.e., rule Send on page 44. One way that Augm-Add in Figure 4.16 succeeds, is when the typing goes through rule Augm-Send, which removes the added requirement. In other words, for a statement $\textbf{add}(\mathtt{x}, \mathtt{R}).\mathtt{S}$, rule Augm-Add succeeds if `S` contains a suitable send command. The silent assumption here is that all necessary intermediate transitions are taken, that the **send** command is reached, and the message is indeed sent. Note that such intermediate transitions can be ones where `x` is sent over as the payload to another message, captured by the typing rule Augm-Delegate. In those cases, the typing must go through Augm-Send eventually, i.e., guarantee that a suitable **send** command is reached, even if in another actor's code.

4.6 Puntigam's Tokens

The system discussed so far has been based on the idea that the meaning of progress should derive from programmer intent: we allow the programmer to specify the kinds of messages that are necessary to each actor, and at which program states. This section discusses

Figure 4.17: The syntax of Figure 4.3 augmented to support Puntigam's ideas [120].

$\mathtt{P}$	$::=$	$\overline{\mathtt{B}}\ \mathtt{S}$	
$\mathtt{B}$	$::=$	**bdef** $b(\overline{x}) = \{\overline{\mathtt{H}}\}$	$b \in$ behavior names
$\mathtt{H}$	$::=$	**hdef** $h^{\mathtt{T}}(\overline{x}) = \mathtt{S}$	$h \in$ handler names
$\mathtt{T}$	$::=$	$\overline{t}_1 \multimap \overline{t}_2$	$t, t_1, t_2, \ldots \in$ token names
$\mathtt{S}$	$::=$	$\mathtt{x}!h(\overline{e}).\mathtt{S}$	
	$\mid$	**if** e **then** $\mathtt{S}_1$ **else** $\mathtt{S}_2$	$e \in$ expressions (values, function calls, etc.)
	$\mid$	$\nu x{:}b(\overline{e}).\mathtt{S}$	actor creation
	$\mid$	$x :=$ **take**$(\mathtt{y}, \overline{t}).\mathtt{S}$	token extraction
	$\mid$	**update**$(\overline{e})$	state update
	$\mid$	**ready**	[runtime syntax]
$\mathtt{x}$	$::=$	**self** $\mid x, y, z, \ldots \mid \alpha, \beta, \ldots$	$x, y, z, \ldots \in$ variables
			$\alpha, \beta, \ldots \in$ runtime actor names

the converse, which is that the programmer should be able to specify the types of messages that each actor is *allowed* to receive.

The basic idea, originally appearing in the work of Puntigam [120], associates handler definitions to declarations of the form $\overline{t}_1 \multimap \overline{t}_2$. This is a linear function that consumes the tokens $\overline{t}_1$ and produces $\overline{t}_2$. By attaching this function to a handler h, we say that an actor that receives a message $h(\overline{e})$ will consume $\overline{t}_1$ and produce $\overline{t}_2$. The type system is then responsible for enforcing that messages are only sent if the recipient has the necessary tokens, in this case, $\overline{t}_1$. This is achieved by tracking the tokens known to be associated with each actor on a per-scope basis.

We augment our calculus with the $x :=$ **take**$(\mathtt{y}, \overline{t})$ construct, which introduces a fresh alias x to actor $\mathtt{y}$, such that x is associated with tokens $\overline{t}$, "taken" from $\mathtt{y}$. The typing then has to ensure that $\mathtt{y}$ indeed has $\overline{t}$ tokens available for taking (rule Tok-Take).

Figure 4.17 shows the new actor syntax that includes a function $\mathtt{T}$ associated with each handler h. In order to keep the presentation simple, this syntax does not include the **add** construct, or requirements in general. This way, the type system in Figure 4.18 can enforce the safety property outlined above without being concerned with the guarantees of the previous section. It is trivial to superimpose the two type systems to offer both guarantees.

Rule Tok-Init in Figure 4.18 extracts static information Δ from the program $\overline{\mathtt{B}}\mathtt{S}$. Per the conventions used so far, such information is carried along the typing relation $\vdash$ as a subscript, i.e., $\vdash_\Delta$. Rule Tok-New introduces a new actor x' with no tokens, and requires that the statement following the actor creation command be typeable in the augmented environment. Rule TOK-TAKE subtracts the tokens $\{\bar{t}\}$ from `y`, and requires that the statement `S` that follows the **take** command is typeable in an environment where x' is associated with the taken tokens. Note the requirement $\{\bar{t}\} \subseteq T$, meaning that the statement will only type if `y` has enough tokens to take from.

The most complicated rule is that for sending a message, i.e., `x`$!h(\bar{e})$.`S`. Rule TOK-SEND requires that the tokens associated with actors $\bar{\texttt{y}}$ passed as arguments to h are removed from the environment that types the subsequent statement `S`. Moreover, the statement `S` must be typeable in an environment where the tokens associated with `x` have been updated to reflect the application of $\bar{t}_1 \multimap \bar{t}_2$ on `x`'s token set.

It is worth noting that our approach to progress is complementary to the one taken in this section: ours can be used to guarantee the receipt of certain messages—a liveness property—while Puntigam's can be used to withhold their sending—a safety property. Combining the two systems would allow complicated coordination constraints to be statically enforced.

Cyclic Communication. The endless looping problem from Section 4.5 exists in Puntigam's token system as well: the send rule of Figure 4.18 requires the indiscriminate typing of handler bodies, much like the send rule of Figure 4.9. A possible solution along the ideas of Section 4.5 apply in this case as well: by reaching an already seen mark k with the same token set, we can declare success and stop the typing process. A more elaborate solution would be to declare success if (a) k is seen twice, and (b) the token set encountered the second time round is a superset of the one seen the first time. This would detect a messaging pattern where the token set keeps increasing; thus, since the tokens were enough for the first iteration, they would be enough in every subsequent one.

Figure 4.18: Typing rules implementing Puntigam's ideas [120].

$$\textsc{Tok-Init}\ \frac{\Delta = \mathit{info}(\overline{\mathtt{B}}) \qquad \emptyset \vdash \mathtt{S}}{\vdash \overline{\mathtt{B}}\mathtt{S}}$$

$$\textsc{Tok-New}\ \frac{\mathbb{T} \cup \{x' \mapsto \emptyset\} \vdash_\Delta \mathtt{S}[x'/x] \qquad x' \text{ fresh}}{\mathbb{T} \vdash_\Delta \nu x{:}b(\overline{e}).\mathtt{S}}$$

$$\textsc{Tok-Send}\ \frac{\begin{array}{c} \mathit{tok}(\Delta, h) = \overline{t}_1 \multimap \overline{t}_2 \qquad \mathit{body}(\Delta, h) = \mathtt{S}_h \\ \{\overline{t}_1\} \subseteq T \qquad T' = T \setminus \{\overline{t}_1\} \cup \{\overline{t}_2\} \qquad \mathbb{T} \cup \{\mathtt{x} \mapsto T'\} \vdash_\Delta \mathtt{S} \\ \mathit{actors}(\Delta, \overline{e}) = \overline{\mathtt{y}} \qquad \mathit{actors}(\Delta, \mathit{params}(\Delta, h)) = \overline{z} \\ \mathbb{T} \cup \{\overline{\mathtt{y}} \mapsto T_1\} \vdash_\Delta \mathtt{S}_h[\overline{\mathtt{y}}/\overline{z}][\mathtt{x}/\mathbf{self}] \end{array}}{\mathbb{T} \cup \{\mathtt{x} \mapsto T,\ \overline{\mathtt{y}} \mapsto T_1\} \vdash_\Delta \mathtt{x}!h(\overline{e}).\mathtt{S}}$$

$$\textsc{Tok-Take}\ \frac{\begin{array}{c} T'_{\mathtt{y}} = T_{\mathtt{y}} \setminus \{\overline{t}\} \qquad \{\overline{t}\} \subseteq T_{\mathtt{y}} \\ \mathbb{T} \cup \{x' \mapsto \overline{t},\ \mathtt{y} \mapsto T'_{\mathtt{y}}\} \vdash_\Delta \mathtt{S}[x'/x] \qquad x' \text{ fresh} \end{array}}{\mathbb{T} \cup \{\mathtt{y} \mapsto T_{\mathtt{y}}\} \vdash_\Delta x := \mathbf{take}(\mathtt{y}, \overline{t}).\mathtt{S}}$$

$$\textsc{Tok-Update}\quad \mathbb{T} \vdash_\Delta \mathbf{update}(\overline{e}) \qquad\qquad \textsc{Tok-If}\ \frac{\mathbb{T} \vdash_\Delta \mathtt{S}_1 \qquad \mathbb{T} \vdash_\Delta \mathtt{S}_2}{\mathbb{T} \vdash_\Delta \mathbf{if}\ e\ \mathbf{then}\ \mathtt{S}_1\ \mathbf{else}\ \mathtt{S}_2}$$

Chapter 5

Session Types for Actors

The previous chapter dealt with the question of whether a pre-specified set of messages is eventually delivered to the actors that need them. We were not concerned with interactions not involving this set of messages; as long as the ones we cared about were guaranteed to be delivered to the correct recipients, we treated intermediate interactions as a "means to an end". In other words, all interactions that led to the delivery of required messages to certain actors were acceptable.

In this chapter, we care about all interactions. Given a program and a protocol specification, we want to guarantee that execution of the program only entails interactions prescribed in the protocol. We also want to guarantee the converse: that all interactions predicted by the protocol take place. Typing coordination constraints in actors is challenging with regard to asynchronous communication, because it leads to delays that require the handling of arbitrary shuffles. The difficulty arises when considering *parameterized* protocols; for example, assume two actors that communicate through a sliding window protocol: the actors agree on the length of the window (i.e., the number of messages that may be buffered) and then proceed to exchange messages concurrently. While the calculus in the previous chapter can capture the protocol—from an implementation standpoint—the typing does not guarantee that the implementation adheres to the specification. In this chapter, we reduce the expressiveness of the language, in return for an increase in the precision with which protocols can be statically enforced.

This work is inspired by the literature on session types, such that a program is checked for conformance against a given global type. While drawing ideas from classic papers [30, 51, 69, 126] on session types, our system is adapted to suit an actor-based calculus, and extended to allow the checking of more complicated protocols. For example, prior work on session types is not suitable for typing interactions such as the sliding window protocol, in part because their respective type languages depend on formalisms such as the typed λ-calculus [10] or System T [63], which do not explicitly capture the interleavings inherent in concurrency.

This chapter presents a session-type system that allows for parameterized concurrency: the number of participants, the types of messages sent, and the number of such messages are controlled by type parameters. Overall, this work makes the following contributions: (i) we introduce a novel construct that captures the atomic reordering of interactions, useful for the validation of protocols where, say, clients interact with server resources; (ii) the typing allows the number of branches in choice constructs to be controlled by a type parameter—both the number, and the types of different possible execution paths can be unknown at compile time; and (iii) the system can statically verify the conformance of actors to protocols where causality relations are directly dependent on parameter values. For instance, the sliding window protocol dictates that acknowledgments come after data messages, while the number of outstanding messages is capped to the size of the window. It is important to note that type checking in our system can be performed without instantiation of the type parameters.

Preliminary versions of this work were published in 2012 and 2016 [35, 36]. This chapter makes the ideas more precise, and offers a comprehensive review of the limitations inherent in session types in general, and their mix with actor systems in particular. For reasons discussed in Section 5.7, capturing actor creation in a session type system is unintuitive, and is the primary reason we developed the typestate system of the previous chapter.

Chapter Outline

Section 5.1 motivates our approach with examples. The examples assume a language of global types, the syntax and semantics of which are formalized in Section 5.2. We then discuss end-point types, i.e., local types, in Section 5.3. Along with their syntax and semantics, in the same section, we present an algorithm that mechanically derives such local types from a global specification. In Section 5.4 we formalize the actor calculus in which the discussed examples can be programmed, while an algorithm that deduces end-point types from actor implementations is presented in Section 5.5. That section also contains the proofs of various standard sanity checks, such as subject reduction. Possible extensions are discussed in Section 5.6, while an in-depth discussion of the limitations of this work is given in Section 5.7. The latter contains an analysis of the issues present in session type systems designed for the π-calculus, and how those manifest in the context of actors.

5.1 Motivation

This section motivates our approach with examples that are not captured by other systems in the literature: the sliding window protocol, a case of locking–unlocking, and some cases of limited resource sharing. We first demonstrate how the behavior of such protocols can be described in our system, and then proceed with the reasons other related works cannot capture these behaviors in their type systems.

The Sliding Window Protocol. Assume that an actor a sends messages of type m to an actor b, which acknowledges every received message with an ack message. The protocol determines that at most n messages can be unacknowledged at any given time, so that when the threshold is reached, a ceases sending until it receives at least one more ack message. In this example, the window size n is a parameter, which means that we need to express the fact that n sending–acknowledging events can be in transit at any given instant in time. The global type of the protocol is as follows:

$$\underbrace{\left(\mathsf{a} \xrightarrow{\mathsf{m}} \mathsf{b}\,;\,\mathsf{b} \xrightarrow{\mathsf{ack}} \mathsf{a}\right)^* \parallel \left(\mathsf{a} \xrightarrow{\mathsf{m}} \mathsf{b}\,;\,\mathsf{b} \xrightarrow{\mathsf{ack}} \mathsf{a}\right)^* \parallel \cdots \parallel \left(\mathsf{a} \xrightarrow{\mathsf{m}} \mathsf{b}\,;\,\mathsf{b} \xrightarrow{\mathsf{ack}} \mathsf{a}\right)^*}_{\mathsf{n}\text{ times}}$$

where $\mathsf{a} \xrightarrow{\mathsf{m}} \mathsf{b}$ denotes that a sends a message of type m to b. The operator ; is used for sequencing interactions. Operator $\parallel$ is used for the concurrent composition of its left and right arguments, while the Kleene star has the usual semantics of an unbounded—yet finite—number of repetitions.

The above type can be expressed using the notation of Castagna et al. [30, 31], albeit with a fixed window size n. In our system on the other hand, we can parameterize the type in n and statically verify that participants follow the protocol without knowing its runtime value. Using $\parallel_{\mathsf{i}=1}^{\mathsf{n}}$ to denote the concurrent composition of n processes, we obtain the following type:

$$\mathop{\Big\|}_{\mathsf{i}=1}^{\mathsf{n}} \left(\left(\mathsf{a} \xrightarrow{\mathsf{m}} \mathsf{b}\,;\,\mathsf{b} \xrightarrow{\mathsf{ack}} \mathsf{a}\right)^*\right) \tag{5.1}$$

Locking / Unlocking. Consider a set of n client actors $\mathsf{c}_{1..\mathsf{n}}$, each of which needs to acquire exclusive access to a server s, by sending it a lock message. The server replies with ack, the client uses the server's services (not shown) atomically, and then sends an unlock message, at which point the next client can do the same. Using $\otimes$ to denote an arbitrary, atomic reordering of terms, the following type describes the locking–unlocking protocol for a fixed number of participants:

$$\Big((\mathsf{c}_1 \xrightarrow{\mathsf{lock}} \mathsf{s}\,;\,\mathsf{s} \xrightarrow{\mathsf{ack}} \mathsf{c}_1\,;\,\mathsf{c}_1 \xrightarrow{\mathsf{unlock}} \mathsf{s}) \otimes \cdots \otimes (\mathsf{c}_\mathsf{n} \xrightarrow{\mathsf{lock}} \mathsf{s}\,;\,\mathsf{s} \xrightarrow{\mathsf{ack}} \mathsf{c}_\mathsf{n}\,;\,\mathsf{c}_\mathsf{n} \xrightarrow{\mathsf{unlock}} \mathsf{s})\Big)$$

This formula expresses that any ordering of the $(\mathsf{c}_\mathsf{i} \xrightarrow{\mathsf{lock}} \mathsf{s}\,;\,\mathsf{s} \xrightarrow{\mathsf{ack}} \mathsf{c}_\mathsf{i}\,;\,\mathsf{c}_\mathsf{i} \xrightarrow{\mathsf{unlock}} \mathsf{s})$ sequences is acceptable. To support a dynamic network topology, the number of participants should be a parameter. Following is the locking–unlocking example in our system, where conformance to the protocol is statically verifiable without knowledge of the runtime value of n:

$$\bigotimes_{\mathsf{i}=1}^{\mathsf{n}} (\mathsf{c_i} \xrightarrow{\mathsf{lock}} \mathsf{s}\,;\, \mathsf{s} \xrightarrow{\mathsf{ack}} \mathsf{c_i}\,;\, \mathsf{c_i} \xrightarrow{\mathsf{unlock}} \mathsf{s}) \tag{5.2}$$

Limited Resource Sharing. In this scenario, a server s grants two clients $\mathsf{c_1}$ and $\mathsf{c_2}$ exclusive access to a set of n resources. At any given point, a maximum of n resources can be locked, but the relevant lock–ack–unlock messages from both clients can be interleaved in any way. Following is the global type for this situation:

$$\Big\|_{\mathsf{i}=1}^{\mathsf{n}} \Big(\big(\mathsf{c_1} \xrightarrow{\mathsf{lock_i}} \mathsf{s}\,;\, \mathsf{s} \xrightarrow{\mathsf{ack_i}} \mathsf{c_1}\,;\, \mathsf{c_1} \xrightarrow{\mathsf{unlock_i}} \mathsf{s} \oplus \mathsf{c_2} \xrightarrow{\mathsf{lock_i}} \mathsf{s}\,;\, \mathsf{s} \xrightarrow{\mathsf{ack_i}} \mathsf{c_2}\,;\, \mathsf{c_2} \xrightarrow{\mathsf{unlock_i}} \mathsf{s} \big)^* \Big)$$

The concurrent composition is parameterized in n, the number of resources. Each sequence of lock–ack–unlock messages is also parameterized in i, which ranges from 1 to n and signifies the resource it refers to. This is necessary to ensure realizability of the protocol (see Section 5.6.1), because in the case of multiple outstanding requests, it allows the participants to disambiguate the responses they receive. Each concurrent instance subsumed by the $\|_{\mathsf{i}=1}^{\mathsf{n}}$ operator consists of a loop (Kleene star) which entails a choice, indicated by $\oplus$. Either $\mathsf{c_1}$ gets access to a resource, or $\mathsf{c_2}$, and this happens repeatedly.

These type operators can be combined to express more complicated resource sharing. For instance, consider an extended version of the previous example, where not only the number n of resources, but also the number k of clients is a parameter:

$$\Big\|_{\mathsf{i}=1}^{\mathsf{n}} \Bigg(\Big(\bigoplus_{\mathsf{j}=1}^{\mathsf{k}} (\mathsf{c_j} \xrightarrow{\mathsf{lock_i}} \mathsf{s}\,;\, \mathsf{s} \xrightarrow{\mathsf{ack_i}} \mathsf{c_j}\,;\, \mathsf{c_j} \xrightarrow{\mathsf{unlock_i}} \mathsf{s}) \Big)^* \Bigg) \tag{5.3}$$

The Above Examples in Related Work

The above examples can be *programmed* in the calculi of much of the related work on session types. However, their type systems do not capture the intended interactions with the accuracy described here. For example, the sliding window protocol can be programmed with the initiation of n channels between two processes, where each message has to be acknowledged on the channel it was received. However, verifying that such an implementation conforms to the global type shown here involves an extra step that is not obvious. Our typing makes the allowable interleavings explicit, whereas in systems deriving from Honda et al.'s multiparty session types [70], these interleavings would have to be implicitly derived from the semantics of the language—as opposed to the meaning of the types.

Figure 5.1: The syntax of global types.

G	$::=$	$a \xrightarrow{m} b$	G-Interaction	$\mid$	(G^*)	G-KleeneStar
	$\mid$	$(G_1 \,;\, \ldots \,;\, G_\kappa)$	G-Seq	$\mid$	$\bigodot_{i=1}^{n} G$	G-Seq-N
	$\mid$	$(G_1 \oplus \ldots \oplus G_\kappa)$	G-Choice	$\mid$	$\bigoplus_{i=1}^{n} G$	G-Choice-N
	$\mid$	$(G_1 \parallel \ldots \parallel G_\kappa)$	G-Paral	$\mid$	$\big\Vert_{i=1}^{n} G$	G-Paral-N
	$\mid$	$(G_1 \otimes \ldots \otimes G_\kappa)$	G-Shuffle	$\mid$	$\bigotimes_{i=1}^{n} G$	G-Shuffle-N
	$\mid$	(G)	G-Paren	$\mid$	τ	G-Tau

a, b	$::=$	$\mathsf{a} \mid \mathsf{b} \mid \ldots$	actor names	i, j	$::=$	$\mathsf{i} \mid \mathsf{j} \mid \ldots$	index names
	$\mid$	$\mathsf{a}_i \mid \mathsf{b}_i \mid \ldots$	indexed actor names		$\mid$	$1 \mid 2 \mid \ldots$	index values
m	$::=$	$\mathsf{m} \mid \ldots$	message type names	n	$::=$	$\mathsf{n} \mid \ldots$	parameter names
	$\mid$	$\mathsf{m}_i \mid \ldots$	indexed message type names		$\mid$	$1 \mid 2 \mid \ldots$	parameter values

5.2 Global Types

As demonstrated in the previous section, a global type describes a protocol to which the whole system must adhere. This section formalizes our language of global types, with the syntax shown in Figure 5.1.

The basic building block of a global type is an interaction $a \xrightarrow{m} b$, with the meaning that actor a sends a message of type m to actor b. We use a, b, etc. to denote both unindexed actor names, such as a and b, and their indexed versions, such as, for example, $\mathsf{a_i}$ and $\mathsf{b_i}$. In these expressions, the index i is a free name, to be bound by surrounding uses of parameterized operators. Similarly, m stands for possibly indexed message types, such as m and $\mathsf{m_i}$. Sequencing actions is achieved with the ; operator, as in, for example, $G_1 \,;\, G_2$, which means that the actions in G_1 precede the actions in G_2. A protocol that can take one of two paths, G_1 or G_2, has the form $G_1 \oplus G_2$. Similarly, a protocol that is the concurrent composition of the interactions in G_1 and G_2 is written $G_1 \parallel G_2$, which consists of all possible interleavings of the basic interactions in G_1 and G_2. The operator $\otimes$ is used to denote an arbitrary execution order of its arguments, albeit that being atomic. For instance, $(G_1 \otimes G_2)$

is the same as $\big((G_1\,;G_2)\oplus(G_2\,;G_1)\big)$, that is, either G_1 followed by G_2, or the reverse order, i.e., G_2 followed by G_1.

We define structural congruence on global types to be the least equivalence relation $\equiv$ that includes the rules of Figure 5.2. In addition to some basic properties (e.g., commutativity, distributivity, associativity), the figure contains the rules for expanding parameterized constructs. For a known value u, parameterized constructs are expanded as per the rule

$$\overset{u}{\underset{i=1}{\mathtt{OP}}} G \equiv G[1/i]\;\mathtt{op}\ldots\mathtt{op}\;G[u/i]$$

where $\mathtt{OP}\in\{\circledcirc,\bigoplus,\bigotimes,\|\}$, and $\mathtt{op}$ is the non-parametric version of $\mathtt{OP}$. The non-parametric versions of the operators are those on the left column of Figure 5.1, such that, for example, $\circledcirc_{\mathsf{i}=1}^{2}\big(\mathsf{a_i}\xrightarrow{\mathsf{m}}\mathsf{b_i}\big)\;\equiv\;\mathsf{a_1}\xrightarrow{\mathsf{m}}\mathsf{b_1}\,;\mathsf{a_2}\xrightarrow{\mathsf{m}}\mathsf{b_2}$. Shuffling expands to a choice among all possible permutations of the Gs, according to the rule

$$(G_1\otimes\ldots\otimes G_\kappa)\equiv\big((G_{f_1(1)}\,;\ldots\,;G_{f_1(\kappa)})\oplus\;\cdots\;\oplus(G_{f_{\kappa!}(1)}\,;\ldots\,;G_{f_{\kappa!}(\kappa)})\big)$$

where $f_1\ldots f_{\kappa!}$ are distinct bijective functions of type $\{1..\kappa\}\mapsto\{1..\kappa\}$. Each of these functions corresponds to a distinct permutation of the numbers in $\{1..\kappa\}$, and all of them together generate all possible such permutations. Notice that all operators except $\otimes$ are associative; moreover, all operators except for sequencing are commutative.

We make the meaning of global types precise in Figure 5.3, where the relation $\longrightarrow$ is defined to be the least relation that includes the rules of the figure. The notation $G\xrightarrow{e}G'$ means that G reduces to G', generating event e. Events are single interactions of the form $a\xrightarrow{m}b$, or inactions τ. Events $a\xrightarrow{m}b$ are produced when a type of the same form is reduced (rule GS-Event).

We write $G\longrightarrow G'$ when there exists an event e such that $G\xrightarrow{e}G'$, and $\longrightarrow^*$ for the transitive reflexive closure of $\longrightarrow$. The following statement captures our intuition that structural congruence is maintained along reduction sequences:

Theorem 5.1. Let G_1, G_1' and G_2 be global types. If $G_1\equiv G_2$ and $G_1\longrightarrow^* G_1'$, then there exists G_2' with $G_2\longrightarrow^* G_2'$ and $G_1'\equiv G_2'$.

Proof. By induction on the congruence and transition relations.

Figure 5.2: Structural congruence on global types is the least equivalence relation on global types that includes the rules in this figure.

$$\textsc{Commutativity}\ \frac{\mathsf{op} \in \{\|, \oplus, \otimes\}}{(G_1\,\mathsf{op}\,G_2) \equiv (G_2\,\mathsf{op}\,G_1)}$$

$$\textsc{Associativity}\ \frac{\mathsf{op} \in \{;\,, \|, \oplus\}}{(G_1\,\mathsf{op}\,(G_2\,\mathsf{op}\,G_3)) \equiv ((G_1\,\mathsf{op}\,G_2)\,\mathsf{op}\,G_3)}$$

$$\textsc{Distributivity}\ \left[\begin{array}{lll}
\big(G_1\,;(G_2 \oplus G_3)\big) \equiv \big((G_1\,;G_2) \oplus (G_1\,;G_3)\big) & & \\
\big((G_1 \oplus G_2)\,;G_3\big) \equiv \big((G_1\,;G_3) \oplus (G_2\,;G_3)\big) & & \\
\big(G_1 \,\|\, (G_2 \oplus G_3)\big) \equiv \big((G_1 \,\|\, G_2) \oplus (G_1 \,\|\, G_3)\big) & & \\
\big(G_1\,;(\bigoplus_{i=1}^{\mathsf{n}} G_2)\big) \equiv \bigoplus_{i=1}^{\mathsf{n}}(G_1\,;G_2) & & i \notin \text{free-names}(G_1) \\
\big((\bigoplus_{i=1}^{\mathsf{n}} G_1)\,;G_2\big) \equiv \bigoplus_{i=1}^{\mathsf{n}}(G_1\,;G_2) & & i \notin \text{free-names}(G_2) \\
\big(G_1 \,\|\, (\bigoplus_{i=1}^{\mathsf{n}} G_2)\big) \equiv \bigoplus_{i=1}^{\mathsf{n}}(G_1 \,\|\, G_2) & & i \notin \text{free-names}(G_1)
\end{array}\right.$$

$$\textsc{KleeneStar}\quad G^* \equiv (\tau \oplus (G\,;G^*)) \qquad \textsc{Inaction}\ \left[\begin{array}{l} \tau\,;G \equiv G \equiv G\,;\tau \\ \tau \,\|\, G \equiv G \end{array}\right.$$

$$\textsc{Composition}\ \left[\begin{array}{c}
\dfrac{G_1 \equiv G_1' \quad G_2 \equiv G_2' \quad \mathsf{op} \in \{;\,, \|, \oplus\}}{(G_1\,\mathsf{op}\,G_2) \equiv (G_1'\,\mathsf{op}\,G_2')} \\[2ex]
\dfrac{G \equiv G' \quad \mathtt{OP} \in \{\circledcirc, \|, \oplus, \otimes\}}{\underset{i=1}{\overset{\mathsf{n}}{\mathtt{OP}}} G \equiv \underset{i=1}{\overset{\mathsf{n}}{\mathtt{OP}}} G'} \\[2ex]
\dfrac{G_1 \equiv G_2}{(G_1) \equiv (G_2)}
\end{array}\right.$$

$$\textsc{Expansion}\ \frac{u \in \mathbb{N} \quad \mathtt{OP} \in \{\circledcirc, \oplus, \otimes, \|\} \quad \mathsf{op} = \text{non-parametric}(\mathtt{OP})}{\underset{i=1}{\overset{u}{\mathtt{OP}}} G \equiv G[1/i]\,\mathsf{op} \ldots \mathsf{op}\,G[u/i]}$$

$$\textsc{Shuffle}\ \frac{f_1 \ldots f_{\kappa!} : \text{bijective } \{1..\kappa\} \mapsto \{1..\kappa\} \quad f_1 \ldots f_{\kappa!} \text{ distinct}}{(G_1 \otimes \ldots \otimes G_\kappa) \equiv \big((G_{f_1(1)}\,; \ldots\,; G_{f_1(\kappa)}) \oplus \cdots \oplus (G_{f_{\kappa!}(1)}\,; \ldots\,; G_{f_{\kappa!}(\kappa)})\big)}$$

Figure 5.3: The semantics of global types. The rules GS-KleeneStar, GS-Tau, GS-Expansion and GS-Shuffle generate empty events.

$$\text{GS-EVENT}\;\frac{e = \text{``}a \xrightarrow{m} b\text{''}}{e \xrightarrow{e} \tau} \qquad \text{GS-PAREN}\;\frac{G \xrightarrow{e} G'}{(G) \xrightarrow{e} (G')}$$

$$\text{GS-CHOICE}\;\frac{\psi \in \{1 \ldots \kappa\}}{(G_1 \oplus \ldots \oplus G_\kappa) \xrightarrow{\tau} (G_\psi)} \qquad \text{GS-KLEENESTAR}\left[\begin{array}{l} G^* \xrightarrow{\tau} \tau \\ G^* \xrightarrow{\tau} (G\,;G^*) \end{array}\right.$$

$$\text{GS-PARAL}\;\frac{G_\psi \xrightarrow{e} G'_\psi \qquad \psi \in \{1..\kappa\}}{(G_1 \| \ldots \| G_\psi \| \ldots \| G_\kappa) \xrightarrow{e} (G_1 \| \ldots \| G'_\psi \| \ldots \| G_\kappa)}$$

$$\text{GS-SEQ}\;\frac{G_1 \xrightarrow{e} G'_1}{(G_1\,; G_2\,; \ldots\,; G_\kappa) \xrightarrow{e} (G'_1\,; G_2\,; \ldots\,; G_\kappa)}$$

$$\text{GS-TAU}\;\frac{\psi \in \{1..\kappa\} \qquad G_\psi = \tau \qquad \mathtt{op} \in \{;\,, \|\}}{(G_1\,\mathtt{op} \ldots \mathtt{op}\, G_\psi\,\mathtt{op} \ldots \mathtt{op}\, G_\kappa) \xrightarrow{\tau} (G_1\,\mathtt{op} \ldots \mathtt{op}\, G_{\psi-1}\,\mathtt{op}\, G_{\psi+1}\,\mathtt{op} \ldots \mathtt{op}\, G_\kappa)}$$

$$\text{GS-EXPANSION}\;\frac{u \in \mathbb{N} \qquad \mathtt{OP} \in \{\circledcirc, \bigoplus, \bigotimes, \|\} \qquad \mathtt{op} = \text{non-parametric}(\mathtt{OP})}{\overset{u}{\underset{i=1}{\mathtt{OP}}} G \xrightarrow{\tau} G[1/i]\,\mathtt{op} \ldots \mathtt{op}\, G[u/i]}$$

$$\text{GS-SHUFFLE}\;\frac{f_1 \ldots f_{\kappa!} : \text{bijective } \{1..\kappa\} \mapsto \{1..\kappa\} \qquad f_1 \ldots f_{\kappa!} \text{ distinct}}{(G_1 \otimes \ldots \otimes G_\kappa) \xrightarrow{\tau} \big((G_{f_1(1)}\,; \ldots\,; G_{f_1(\kappa)}) \oplus \;\cdots\; \oplus (G_{f_{\kappa!}(1)}\,; \ldots\,; G_{f_{\kappa!}(\kappa)})\big)}$$

Sequence – Associativity:

$$\begin{tikzcd}
(G_1\,;(G_2\,;G_3)) & \equiv & ((G_1\,;G_2)\,;G_3) & \textsc{Associativity} \\
& G_1 \longrightarrow G_1' & & \\
& \Downarrow & & \\
(G_1\,;(G_2\,;G_3)) \longrightarrow (G_1'\,;(G_2\,;G_3)) & & (G_1\,;G_2) \longrightarrow (G_1'\,;G_2) & \text{GS-}\textsc{Seq} \\
& \equiv & \Downarrow & \\
& & ((G_1\,;G_2)\,;G_3) \longrightarrow ((G_1'\,;G_2)\,;G_3) & \text{GS-}\textsc{Seq}
\end{tikzcd}$$

Concurrent composition – Associativity (1/3):

$$\begin{tikzcd}
(G_1 \| (G_2 \| G_3)) & \equiv & ((G_1 \| G_2) \| G_3) & \textsc{Associativity} \\
& G_1 \longrightarrow G_1' & & \\
& \Downarrow & & \\
(G_1 \| (G_2 \| G_3)) \longrightarrow (G_1' \| (G_2 \| G_3)) & & (G_1 \| G_2) \longrightarrow (G_1' \| G_2) & \text{GS-}\textsc{Paral} \\
& \equiv & \Downarrow & \\
& & ((G_1 \| G_2) \| G_3) \longrightarrow ((G_1' \| G_2) \| G_3) & \text{GS-}\textsc{Paral}
\end{tikzcd}$$

Concurrent composition – Associativity (2/3):

$$\begin{tikzcd}
(G_1 \| (G_2 \| G_3)) & \equiv & ((G_1 \| G_2) \| G_3) & \textsc{Associativity} \\
& G_2 \longrightarrow G_2' & & \\
& \Downarrow & & \\
(G_2 \| G_3) \longrightarrow (G_2' \| G_3) & & (G_1 \| G_2) \longrightarrow (G_1 \| G_2') & \text{GS-}\textsc{Paral} \\
\Downarrow & & \Downarrow & \\
(G_1 \| (G_2 \| G_3)) \longrightarrow (G_1 \| (G_2' \| G_3)) & \equiv & ((G_1 \| G_2) \| G_3) \longrightarrow ((G_1 \| G_2') \| G_3) & \text{GS-}\textsc{Paral}
\end{tikzcd}$$

Concurrent composition – Associativity (3/3):

$$\begin{tikzcd}
(G_1 \| (G_2 \| G_3)) & \equiv & ((G_1 \| G_2) \| G_3) & \textsc{Associativity} \\
 & G_3 \longrightarrow G_3' \arrow[d, Rightarrow] & & \\
(G_2 \| G_3) \longrightarrow (G_2 \| G_3') \arrow[d, Rightarrow] & & ((G_1 \| G_2) \| G_3) \longrightarrow ((G_1 \| G_2) \| G_3') \arrow[dll, Rightarrow] & \text{GS-}\textsc{Paral} \\
(G_1 \| (G_2 \| G_3)) \longrightarrow (G_1 \| (G_2 \| G_3')) & & \text{GS-}\textsc{Paral} &
\end{tikzcd}$$

The case of the choice (operator $\oplus$) follows in the same manner as concurrent composition. The case for the Kleene Star is trivial, from $G^* \equiv (\tau \oplus (G\,;G^*))$ and the reduction rules $G^* \longrightarrow \tau$ and $G^* \longrightarrow (G\,;G^*)$. The congruence rules for operator expansion (including shuffling $\otimes$) are mirrored in the semantics. The rules for the distributive property can be dealt with in the same way as associativity, as above. Operator composition follows inductively from the rest of the cases. $\square$

5.3 Local Types

A local type specifies the abstract behavior of a single protocol participant, and the respective syntax is shown in Figure 5.4. As before, a, b and m are possibly indexed, i.e., a can stand for some actor name a, or an indexed name a_i. Message sending is written $a!m$, meaning the action of sending a message of type m to actor a. Similarly, message receiving is written $a?m$, to mean the action of receiving a message of type m from actor a. The rest of the constructs on the top half of the figure are defined as in the case of global types. The bottom part shows the syntax of runtime configurations, which are tuples of the form (M, Λ). These are described in the next section.

5.3.1 Local Type Semantics

Structural congruence on local types is defined to be the least equivalence relation $\equiv$ that includes the rules of Figure 5.5. As before, parameterized constructs with known values for the parameters are expanded to their non-parameterized versions; for example, for a fixed value u, we have

$$\bigoplus_{i=1}^{u} L \equiv L[1/i] \oplus \ldots \oplus L[u/i].$$

Figure 5.4: The syntax of local types.

$$
\begin{array}{rcll|ll}
L & ::= & a!m & \text{L-S{\small END}} & a?m & \text{L-R{\small ECV}} \\
 & | & (L_1\,;\dots;L_\kappa) & \text{L-S{\small EQ}} & \bigodot\limits_{i=1}^{n} L & \text{L-S{\small EQ}-N} \\
 & | & (L_1 \oplus \dots \oplus L_\kappa) & \text{L-C{\small HOICE}} & \bigoplus\limits_{i=1}^{n} L & \text{L-C{\small HOICE}-N} \\
 & | & (L_1 \,\|\, \dots \,\|\, L_\kappa) & \text{L-P{\small ARAL}} & \Big\|_{i=1}^{n} L & \text{L-P{\small ARAL}-N} \\
 & | & (L_1 \otimes \dots \otimes L_\kappa) & \text{L-S{\small HUFFLE}} & \bigotimes\limits_{i=1}^{n} L & \text{L-S{\small HUFFLE}-N} \\
 & | & (L^*) & \text{L-K{\small LEENE}S{\small TAR}} & (L) & \text{L-P{\small AREN}} \\
 & | & \tau & \text{L-T{\small AU}} & &
\end{array}
$$

$$
\begin{array}{rcll@{\qquad}rcll}
a, b & ::= & \mathsf{a} \mid \mathsf{b} \mid \dots & \text{actor names} & i, j & ::= & \mathsf{i} \mid \mathsf{j} \mid \dots & \text{index names} \\
 & | & \mathsf{a}_i \mid \mathsf{b}_i \mid \dots & \text{indexed actor names} & & | & 1 \mid 2 \mid \dots & \text{index values} \\
m & ::= & \mathsf{m} \mid \dots & \text{message type names} & n & ::= & \mathsf{n} \mid \dots & \text{parameter names} \\
 & | & \mathsf{m}_i \mid \dots & \text{indexed message type names} & & | & 1 \mid 2 \mid \dots & \text{parameter values}
\end{array}
$$

$$
\begin{array}{rcll}
C & ::= & (M, \Lambda) & \text{runtime configuration} \\
\Lambda & ::= & \{\langle L_1 \rangle_{a_1}, \dots, \langle L_\kappa \rangle_{a_\kappa}\} & \text{multiset of running local types} \\
M & ::= & \{a_1 \xrightarrow{m_1} b_1, \dots, a_\kappa \xrightarrow{m_\kappa} b_\kappa\} & \text{multiset of pending messages}
\end{array}
$$

Shuffling is as in the case of global types, and represents all possible sequences of the types involved. The rules to the right of the RUNTIME bracket extend the relation to runtime configurations. Runtime configurations have the form (M, Λ), where M is the multiset of pending (sent but not received) messages, and Λ is the multiset of concurrently executing local types. These take the form $\langle L \rangle_a$, where L is a local type, and a the actor it corresponds to.

Lemma 5.1 (Congruence is compositional on set union)**.** Let L_1 and L_2 be local types, a be an actor name, M a message multiset, and Λ a multiset of running local types, all as in Figure 5.4.
Then, $L_1 \equiv L_2$ iff $\Lambda \cup \{\langle L_1 \rangle_a\} \equiv \Lambda \cup \{\langle L_2 \rangle_a\}$.
Equivalently, $L_1 \equiv L_2$ iff $(M,\, \Lambda \cup \{\langle L_1 \rangle_a\}) \equiv (M,\, \Lambda \cup \{\langle L_2 \rangle_a\})$.

Proof. Directly from the RUNTIME rules in Figure 5.5. □

The reduction relation on local types is defined to be the least relation $\longrightarrow$ that includes the rules in Figure 5.6. As in the case of global types, reductions produce events of the form τ and $a \xrightarrow{m} b$. As always, reductions hold up to structural congruence. Rule LS-SEND adds the sent message to M, i.e., the multiset of undelivered messages. Note that this rule produces

Figure 5.5: Structural congruence on local types is the least equivalence relation on local types that includes the rules in this figure.

$$\textsc{Commutativity}\ \frac{\mathsf{op} \in \{\|, \oplus, \otimes\}}{(L_1\,\mathsf{op}\,L_2) \equiv (L_2\,\mathsf{op}\,L_1)}$$

$$\textsc{Associativity}\ \frac{\mathsf{op} \in \{;\,, \|, \oplus\}}{(L_1\,\mathsf{op}\,(L_2\,\mathsf{op}\,L_3)) \equiv ((L_1\,\mathsf{op}\,L_2)\,\mathsf{op}\,L_3)}$$

$$\textsc{Distributivity}\ \left[\begin{array}{lcll}
\big(L_1\,;(L_2 \oplus L_3)\big) & \equiv & \big((L_1\,;L_2) \oplus (L_1\,;L_3)\big) & \\
\big((L_1 \oplus L_2)\,;L_3\big) & \equiv & \big((L_1\,;L_3) \oplus (L_2\,;L_3)\big) & \\
\big(L_1 \,\|\, (L_2 \oplus L_3)\big) & \equiv & \big((L_1 \,\|\, L_2) \oplus (L_1 \,\|\, L_3)\big) & \\
\big(L_1\,;(\bigoplus_{i=1}^{\mathsf{n}} L_2)\big) & \equiv & \bigoplus_{i=1}^{\mathsf{n}}(L_1\,;L_2) & i \notin \text{free-names}(L_1) \\
\big((\bigoplus_{i=1}^{\mathsf{n}} L_1)\,;L_2\big) & \equiv & \bigoplus_{i=1}^{\mathsf{n}}(L_1\,;L_2) & i \notin \text{free-names}(L_2) \\
\big(L_1 \,\|\, (\bigoplus_{i=1}^{\mathsf{n}} L_2)\big) & \equiv & \bigoplus_{i=1}^{\mathsf{n}}(L_1 \,\|\, L_2) & i \notin \text{free-names}(L_1)
\end{array}\right.$$

$$\textsc{KleeneStar}\quad L^* \equiv (\tau \oplus (L\,;L^*)) \qquad \textsc{Inaction}\ \left[\begin{array}{l} \tau\,;L \equiv L \equiv L\,;\tau \\ \tau \,\|\, L \equiv L \end{array}\right.$$

$$\textsc{Composition}\ \left[\begin{array}{c}
\dfrac{L_1 \equiv L_1' \quad L_2 \equiv L_2' \quad \mathsf{op} \in \{;\,, \|, \oplus\}}{(L_1\,\mathsf{op}\,L_2) \equiv (L_1'\,\mathsf{op}\,L_2')} \\[2ex]
\dfrac{L \equiv L' \quad \mathtt{OP} \in \{\circledcirc, \|, \oplus, \otimes\}}{\underset{i=1}{\overset{\mathsf{n}}{\mathtt{OP}}} L \equiv \underset{i=1}{\overset{\mathsf{n}}{\mathtt{OP}}} L'} \\[2ex]
\dfrac{L_1 \equiv L_2}{(L_1) \equiv (L_2)}
\end{array}\right.$$

$$\textsc{Expansion}\ \frac{u \in \mathbb{N} \quad \mathtt{OP} \in \{\circledcirc, \oplus, \otimes, \|\} \quad \mathsf{op} = \text{non-parametric}(\mathtt{OP})}{\underset{i=1}{\overset{u}{\mathtt{OP}}} L \equiv L[1/i]\,\mathsf{op} \ldots \mathsf{op}\,L[u/i]}$$

$$\textsc{Shuffle}\ \frac{f_1 \ldots f_{\kappa!} : \text{bijective } \{1..\kappa\} \mapsto \{1..\kappa\} \quad f_1 \ldots f_{\kappa!} \text{ distinct}}{(L_1 \otimes \ldots \otimes L_\kappa) \equiv \big((L_{f_1(1)}\,;\ldots;L_{f_1(\kappa)}) \oplus \cdots \oplus (L_{f_{\kappa!}(1)}\,;\ldots;L_{f_{\kappa!}(\kappa)})\big)}$$

$$\textsc{Runtime}\ \left[\begin{array}{cc}
\dfrac{L_1 \equiv L_2}{\langle L_1\rangle_a \equiv \langle L_2\rangle_a} & \dfrac{\langle L_1\rangle_a \equiv \langle L_2\rangle_a}{\big\{\langle L_1\rangle_a\big\} \equiv \big\{\langle L_2\rangle_a\big\}} \\[2ex]
\dfrac{\Lambda_1 \equiv \Lambda_2}{\Lambda \cup \Lambda_1 \equiv \Lambda \cup \Lambda_2} & \dfrac{\Lambda_1 \equiv \Lambda_2}{(M, \Lambda_1) \equiv (M, \Lambda_2)}
\end{array}\right.$$

Figure 5.6: The semantics of local types.

$$\textsc{LS-Send}\quad M,\, \Lambda \cup \{\langle b!m\rangle_a\} \xrightarrow{\tau} M \cup \{a \xrightarrow{m} b\},\, \Lambda \cup \{\langle \tau\rangle_a\}$$

$$\textsc{LS-Seq}\,\frac{M,\, \Lambda \cup \{\langle L_1\rangle_a\} \xrightarrow{e} M',\, \Lambda \cup \{\langle L_1'\rangle_a\}}{M,\, \Lambda \cup \{\langle (L_1\,;L_2\,;\ldots\,;L_\kappa)\rangle_a\} \xrightarrow{e} M',\, \Lambda \cup \{\langle (L_1'\,;L_2\,;\ldots\,;L_\kappa)\rangle_a\}}$$

$$\textsc{LS-Tau}\,\frac{\psi \in \{1..\kappa\} \qquad L_\psi = \tau \qquad \mathtt{op} \in \{;\,,\|\}}{M,\, \Lambda \cup \{\langle (L_1\,\mathtt{op}\ldots\mathtt{op}\,L_\psi\,\mathtt{op}\ldots\mathtt{op}\,L_\kappa)\rangle_a\} \xrightarrow{\tau} M,\, \Lambda \cup \{\langle (L_1\,\mathtt{op}\ldots\mathtt{op}\,L_{\psi-1}\,\mathtt{op}\,L_{\psi+1}\,\mathtt{op}\ldots\mathtt{op}\,L_\kappa)\rangle_a\}}$$

$$\textsc{LS-Paral}\,\frac{\psi \in \{1..\kappa\} \qquad M,\, \Lambda \cup \{\langle L_\psi\rangle_a\} \xrightarrow{e} M',\, \Lambda \cup \{\langle L_\psi'\rangle_a\}}{M,\, \Lambda \cup \{\langle (L_1 \,\|\, \ldots \,\|\, L_\psi \,\|\, \ldots \,\|\, L_k)\rangle_a\} \xrightarrow{e} M',\, \Lambda \cup \{\langle (L_1 \,\|\, \ldots \,\|\, L_\psi' \,\|\, \ldots \,\|\, L_k)\rangle_a\}}$$

$$\textsc{LS-Ext-Choice}\,\frac{\psi \in \{1..\kappa\} \qquad \text{first}(L_1) = a_1!m_1 \ldots \text{first}(L_\kappa) = a_\kappa!m_\kappa}{M,\, \Lambda \cup \{\langle (L_1 \oplus \ldots \oplus L_\kappa)\rangle_b\} \xrightarrow{\tau} M,\, \Lambda \cup \{\langle (L_\psi)\rangle_b\}}$$

$$\textsc{LS-Int-Choice}\,\frac{\psi \in \{1..\kappa\} \qquad \text{“}a_\psi \xrightarrow{m_\psi} b_\psi\text{”} \in M \qquad \text{first}(L_1) = a_1?m_1 \ldots \text{first}(L_\kappa) = a_\kappa?m_\kappa}{M,\, \Lambda \cup \{\langle (L_1 \oplus \ldots \oplus L_\kappa)\rangle_b\} \xrightarrow{\tau} M,\, \Lambda \cup \{\langle (L_\psi)\rangle_b\}}$$

$$\textsc{LS-Recv}\,\frac{e = \text{“}a \xrightarrow{m} b\text{”}}{M \cup \{a \xrightarrow{m} b\},\, \Lambda \cup \{\langle a?m\rangle_b\} \xrightarrow{\tau} M,\, \Lambda \cup \{\langle \tau\rangle_b\}}$$

$$\textsc{LS-Paren}\,\frac{M,\, \Lambda \cup \{\langle L\rangle_a\} \xrightarrow{e} M',\, \Lambda \cup \{\langle L'\rangle_a\}}{M,\, \Lambda \cup \{\langle (L)\rangle_a\} \xrightarrow{e} M',\, \Lambda \cup \{\langle (L')\rangle_a\}}$$

$$\textsc{LS-KleeneStar}\left[\begin{array}{l} M,\, \Lambda \cup \{\langle L^*\rangle_a\} \xrightarrow{\tau} M,\, \Lambda \cup \{\langle (L\,; L^*)\rangle_a\} \\ M,\, \Lambda \cup \{\langle L^*\rangle_a\} \xrightarrow{\tau} M,\, \Lambda \cup \{\langle \tau\rangle_a\} \end{array}\right.$$

$$\textsc{LS-Expansion}\,\frac{u \in \mathbb{N} \qquad \mathtt{OP} \in \{\circledcirc, \oplus, \otimes, \|\} \qquad \mathtt{op} = \text{non-parametric}(\mathtt{OP})}{M,\, \Lambda \cup \{\langle \underset{i=1}{\overset{u}{\mathtt{OP}}} L\rangle_a\} \xrightarrow{\tau} M,\, \Lambda \cup \{\langle (L[1/i]\,\mathtt{op}\ldots\mathtt{op}\,L[u/i])\rangle_a\}}$$

an event τ; the actual message event $a \xrightarrow{m} b$ is produced by rule LS-Recv, i.e, at the point of message delivery. Rule LS-Recv removes the message from M and consumes the $a?m$ action at the receiving actor. Rule LS-Ext-Choice represents choice on the sending end, i.e., the sender chooses which of the given actions to take. For this reason, the rule demands that all provided actions start with a message send. In contrast, LS-Int-Choice demands that all provided actions start with a receive action—capturing the selection of a type L_ψ based on the messages available in M. Rule LS-Expansion replaces parameterized operators with their non-parameterized counterparts, mirroring the respective congruence rule in Figure 5.5. The rest of the reduction rules are standard.

Lemma 5.2 (Reduction is compositional on set union)**.** Let L and L' be local types, M and M' be multisets of pending messages, a be some actor name, and Λ be a multiset of running local types. Then,
$(M, \{\langle L\rangle_a\}) \longrightarrow (M', \{\langle L'\rangle_a\})$ iff $(M, \Lambda \cup \{\langle L\rangle_a\}) \longrightarrow (M', \Lambda \cup \{\langle L'\rangle_a\})$. As a direct consequence,
$(M, \{\langle L\rangle_a\}) \longrightarrow^* (M', \{\langle L'\rangle_a\})$ iff $(M, \Lambda \cup \{\langle L\rangle_a\}) \longrightarrow^* (M', \Lambda \cup \{\langle L'\rangle_a\})$.

Proof. By induction on the size of Λ, and using the definition of the reduction relation in Figure 5.6. □

Lemma 5.3 (Congruence follows semantics—single element)**.** Let L_1, L'_1 and L_2 be local types, M, M' be multisets of pending messages, and a be an actor name. If $L_1 \equiv L_2$ and $(M, \{\langle L_1\rangle_a\}) \longrightarrow^* (M', \{\langle L'_1\rangle_a\})$, then there exists L'_2 such that $(M, \{\langle L_2\rangle_a\}) \longrightarrow^* (M', \{\langle L'_2\rangle_a\})$ with $L'_1 \equiv L'_2$.

Proof. By induction on the structural congruence and reduction relations. Omitting the message sets M and M' for brevity, the proof is similar to that of Theorem 5.1 on page 67. As an example, we show the case for the associativity of sequencing.

Sequence – Associativity:

$$\begin{tikzcd}
(L_1\,;(L_2\,;L_3)) & \equiv & ((L_1\,;L_2)\,;L_3) & \textsc{Associativity} \\
 & L_1 \longrightarrow L_1' & & \\
 & \Downarrow & & \\
(L_1\,;(L_2\,;L_3)) \longrightarrow (L_1'\,;(L_2\,;L_3)) & & (L_1\,;L_2) \longrightarrow (L_1'\,;L_2) & \text{LS-}\textsc{Seq} \\
 & \equiv & \Downarrow & \\
 & & ((L_1\,;L_2)\,;L_3) \longrightarrow ((L_1'\,;L_2)\,;L_3) & \text{LS-}\textsc{Seq}
\end{tikzcd}$$

□

Theorem 5.2 (Congruence follows semantics)**.** Let Λ_1, Λ_1', and Λ_2 be multisets of running local types, and let M and M' be multisets of pending messages. If $\Lambda_1 \equiv \Lambda_2$ and $(M, \Lambda_1) \longrightarrow^* (M', \Lambda_1')$, then there exists Λ_2' such that $(M, \Lambda_2) \longrightarrow^* (M', \Lambda_2')$ and $\Lambda_1' \equiv \Lambda_2'$.

Proof. By induction on the size of the multiset of running local types Λ_1.

Base case: $|\Lambda_1| = 0$.

Since Λ_1 is empty, it is (by definition) $\Lambda_1 \equiv \emptyset$. Additionally, it can only "reduce" in zero steps, i.e., $(M, \Lambda_1) \longrightarrow^0 (M, \emptyset)$. Thus the result holds for $\Lambda_2 = \emptyset$ and $M' = M$.

Inductive step: $|\Lambda_1| > 0$.

Fix some Λ with $|\Lambda| \geq 0$, an actor a, message sets M and M', and local types L_1, L_1' and L_2. Let $\Lambda_1 = \Lambda \cup \{\langle L_1 \rangle_a\}$, $\Lambda_1' = \Lambda \cup \{\langle L_1' \rangle_a\}$ and $\Lambda_2 = \Lambda \cup \{\langle L_2 \rangle_a\}$, such that $(M, \Lambda_1) \longrightarrow^* (M', \Lambda_1')$ and $\Lambda_1 \equiv \Lambda_2$. Equivalently, we have

$$(M, \Lambda \cup \{\langle L_1 \rangle_a\}) \longrightarrow^* (M', \Lambda \cup \{\langle L_1' \rangle_a\}) \tag{5.4}$$

$$\Lambda \cup \{\langle L_1 \rangle_a\} \equiv \Lambda \cup \{\langle L_2 \rangle_a\} \tag{5.5}$$

The above relations mirror the predicates of this theorem.

We need to find Λ_2' such that $(M, \Lambda_2) \longrightarrow^* (M', \Lambda_2')$ and $\Lambda_1' \equiv \Lambda_2'$. From 5.4 and Lemma 5.2, we know that

$$(M, \{\langle L_1 \rangle_a\}) \longrightarrow^* (M', \{\langle L_1' \rangle_a\}) \tag{5.6}$$

Similarly, from 5.5 and Lemma 5.1, we have that

$$L_1 \equiv L_2 \tag{5.7}$$

Applying Lemma 5.3 to 5.6 and 5.7 we know that there exists L'_2 such that

$$\begin{aligned}(M, \{\langle L_2\rangle_a\}) \longrightarrow^* (M', \{\langle L'_2\rangle_a\})\\ \text{with } L'_1 \equiv L'_2\end{aligned}$$

Applying Lemmas 5.1 and 5.2 to the above, we get

$$\begin{aligned}(M, \Lambda \cup \{\langle L_2\rangle_a\}) \longrightarrow^* (M', \Lambda \cup \{\langle L'_2\rangle_a\})\\ \text{with } \Lambda \cup \{\langle L'_1\rangle_a\} \equiv \Lambda \cup \{\langle L'_2\rangle_a\}\end{aligned}$$

which completes the proof. □

5.3.2 Projection

Besides the characterization of actor behavior defined in the previous section, local types also specify the behavior restrictions that a global type implies for each participant. The projection of a global type G onto an actor a, written $G \triangleright a$, is given in Figure 5.7. The $\triangleright$ function generates constructs in the syntax of Figure 5.4, augmented by the symbol $\perp$ which stands for the empty projection. In what follows, we assume a post-processing step that removes $\perp$ from the result of projection.

As an example projection, consider the application of the rules of Figure 5.7 onto the global type for the sliding window protocol, deriving the local types of the participants a and b:

$$\begin{aligned}
&\left(\mathop{\Big\|}_{\mathsf{i}=1}^{\mathsf{n}} \big((\mathsf{a} \xrightarrow{\mathsf{m}} \mathsf{b}\,;\, \mathsf{b} \xrightarrow{\mathsf{ack}} \mathsf{a})^*\big)\right) \triangleright \mathsf{a} \\
&\quad = \mathop{\Big\|}_{\mathsf{i}=1}^{\mathsf{n}} \big(((\mathsf{a} \xrightarrow{\mathsf{m}} \mathsf{b}\,;\, \mathsf{b} \xrightarrow{\mathsf{ack}} \mathsf{a})^*) \triangleright \mathsf{a}\big) && \text{P-PARAM} \\
&\quad = \mathop{\Big\|}_{\mathsf{i}=1}^{\mathsf{n}} \big((\mathsf{a} \xrightarrow{\mathsf{m}} \mathsf{b}\,;\, \mathsf{b} \xrightarrow{\mathsf{ack}} \mathsf{a}) \triangleright \mathsf{a})^*\big) && \text{P-KLEENESTAR} \\
&\quad = \mathop{\Big\|}_{\mathsf{i}=1}^{\mathsf{n}} \Big(\big((\mathsf{a} \xrightarrow{\mathsf{m}} \mathsf{b} \triangleright \mathsf{a})\,;\, (\mathsf{b} \xrightarrow{\mathsf{ack}} \mathsf{a} \triangleright \mathsf{a})\big)^*\Big) && \text{P-OPERATOR} \\
&\quad = \mathop{\Big\|}_{\mathsf{i}=1}^{\mathsf{n}} \big((\mathsf{b!m}\,;\, \mathsf{b?ack})^*\big) && \text{P-INTERACTION}
\end{aligned}$$

Figure 5.7: The projection function. It is $\mathsf{op} \in \{;, \oplus, \otimes, \|\}$ and $\mathtt{OP} \in \{\circledcirc, \bigoplus, \bigotimes, \|\}$.

$$
\begin{array}{rcll}
(a \xrightarrow{m} b) \rhd p & = & \begin{cases} b!m & \text{if } p = a \neq b \\ a?m & \text{if } p = b \neq a \\ a!m \,;\, a?m & \text{if } p = a = b \\ \bot & \text{otherwise} \end{cases} & \text{P-INTERACTION} \\
(G_1 \,\mathsf{op}\, \ldots \,\mathsf{op}\, G_\kappa) \rhd p & = & (G_1 \rhd p) \,\mathsf{op}\, \ldots \,\mathsf{op}\, (G_\kappa \rhd p) & \text{P-OPERATOR} \\
G^* \rhd p & = & (G \rhd p)^* & \text{P-KLEENESTAR} \\
\left(\underset{i=1}{\overset{n}{\mathtt{OP}}} G\right) \rhd p & = & \begin{cases} \underset{i=1}{\overset{n}{\mathtt{OP}}} (G \rhd p) & \text{if } p = \mathsf{a} \text{ (no index)} \\ (G \rhd p) & \text{otherwise} \end{cases} & \text{P-PARAM}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{2}{l}{\left(\underset{\mathsf{i}=1}{\overset{\mathsf{n}}{\|}} \left((\mathsf{a} \xrightarrow{\mathsf{m}} \mathsf{b} \,;\, \mathsf{b} \xrightarrow{\mathsf{ack}} \mathsf{a})^*\right)\right) \rhd \mathsf{b}} & \\
= & \underset{\mathsf{i}=1}{\overset{\mathsf{n}}{\|}} \left(\left((\mathsf{a} \xrightarrow{\mathsf{m}} \mathsf{b} \,;\, \mathsf{b} \xrightarrow{\mathsf{ack}} \mathsf{a})^*\right) \rhd \mathsf{b}\right) & \text{P-PARAM} \\
= & \underset{\mathsf{i}=1}{\overset{\mathsf{n}}{\|}} \left((\mathsf{a} \xrightarrow{\mathsf{m}} \mathsf{b} \,;\, \mathsf{b} \xrightarrow{\mathsf{ack}} \mathsf{a}) \rhd \mathsf{b})^*\right) & \text{P-KLEENESTAR} \\
= & \underset{\mathsf{i}=1}{\overset{\mathsf{n}}{\|}} \left(\left((\mathsf{a} \xrightarrow{\mathsf{m}} \mathsf{b} \rhd \mathsf{b}) \,;\, (\mathsf{b} \xrightarrow{\mathsf{ack}} \mathsf{a} \rhd \mathsf{b})\right)^*\right) & \text{P-OPERATOR} \\
= & \underset{\mathsf{i}=1}{\overset{\mathsf{n}}{\|}} \left((\mathsf{a?m} \,;\, \mathsf{a!ack})^*\right) & \text{P-INTERACTION}
\end{array}
$$

In order for the projection rule P-PARAM of Figure 5.7 to work correctly, we disallow global types where the same name appears with more than one index, when those indices are each bound by a different operator application. An example of an unsupported case would be the following:

$$\underset{\mathsf{i}=1}{\overset{\mathsf{n}_1}{\|}} \; \underset{\mathsf{j}=1}{\overset{\mathsf{n}_2}{\|}} \mathsf{a}_\mathsf{i} \xrightarrow{\mathsf{m}} \mathsf{a}_\mathsf{j}$$

The problem is that the two indices i and j come from two different applications of the concurrent composition operator, yet they are both attached to the name a. Hence, the result of the projection onto a participant a_k depends on where the value of k lies in the sub-intervals defined by 1, n_1 and n_2. Since our projection function does not take external restrictions into account, we omit the treatment of such cases in this thesis.

5.4 Actor Calculus

The types defined in the previous sections specify the permissible sequences of messages that participants may exchange. However, types by themselves provide no implementation of the protocol. This section presents a programming language designed for protocols expressible in the type syntax given on pages 66 and 72. We define this language such that there is an one-to-one correspondence between its constructs and the syntax of local types. Note that, in order to capture the strict protocol adherence implied by the use of global types, the calculus defined here differs from that of Section 2.1. First, it disallows the dynamic introduction of actor names, notwithstanding the values of indices. That is, the index i in $\mathsf{a_i}$ can take different values at runtime; as a result, code can introduce actors a_1, a_2, et cetera, depending on the value of i. Second, actor behaviors do not include message handlers; rather, the language supports explicit receive statements—similar in fashion to basic Erlang [86] (without OTP), and the actor calculus of Agha et al. [3].

The calculus syntax is given in Figure 5.8. A program P includes parameter definitions $\overline{\mathsf{p}}$, message structure definitions $\overline{\mathsf{s}}$, and actor definitions $\overline{\mathtt{A}}$. Program parameters correspond to type parameters, as they appear in global and local types (pages 66 and 72, respectively). Message structure definitions are user-defined types similar to structs in the C family of programming languages [128], Pascal records [113], et cetera, and we omit their formal syntax. Actor definitions include the commands L that each actor will execute at runtime. The syntax for both actor and message structure definitions includes an optional parameter in brackets after their name. In the case of actors, the parameter controls the number of actors spawned. In the case of message structures, the parameter controls the number of message *types* declared. This allows the implementation of protocols where actor names and message types are both parameterized, such as the global type $\|_{\mathsf{i}=1}^{\mathsf{n}}(\mathsf{a_i} \xrightarrow{\mathsf{m_i}} \mathsf{b_i})$. Internal choice is implemented via the **select** $\{\mathsf{L}_1 \dots \mathsf{L}_\kappa\}$ statement. Assuming a **select** statement is executed in an actor b, the L_ψ taken starts with a command **recv**$(a, \mathtt{x})$ where a, b and the sort m of $\mathtt{x}$ match some message $a \xrightarrow{u:m} b$ pending to be delivered. The value u (of type m) in the message will be stored in $\mathtt{x}$, accessible in subsequent statements. External choice, on the other hand, is implemented via a **case** statement which selects the branch to be taken according to the value of the supplied expression $\mathtt{e}$.

To give a taste of the language, Figure 5.9 provides an implementation of the sliding window example of page 64. The size of the window is captured by the program's only parameter, n, declared on line 1. The sorts m and ack, declared on lines 3 and 4, represent the message and acknowledgment sorts, respectively. The sender then spawns n parallel processes, each proceeding to send a message $\mathsf{m\_}$ of sort m, and expecting an acknowledgment

Figure 5.8: Calculus syntax.

$\mathtt{P}$	$::=$	$\overline{\mathtt{p}}\,\overline{\mathtt{s}}\,\overline{\mathtt{A}}$					
$\mathtt{p}$	$::=$	$\textbf{param}\ n$		n	$::=$	parameter names	
$\mathtt{s}$	$::=$	$\textbf{struct}\ \mathsf{m}\ \{\ldots\}$		m	$::=$	message sort names	
	$\mid$	$\textbf{struct}\ \mathsf{m}[n]\ \{\ldots\}$					
$\mathtt{A}$	$::=$	$\textbf{actor}\ \mathsf{a}\ \{\mathtt{L}\}$		a	$::=$	actor names	
	$\mid$	$\textbf{actor}\ \mathsf{a}[n]\ \{\mathtt{L}\}$					
$\mathtt{L}$	$::=$	$\mathtt{V}\,;\mathtt{L} \mid \textbf{send}(a,\mathtt{x}) \mid \textbf{recv}(a,\mathtt{x})$		a	$::=$	$\mathsf{a} \mid \mathsf{a}_i$	
	$\mid$	$\mathtt{x} := \mathsf{e}$		$\mathtt{x}$	$::=$	variables	
	$\mid$	$\mathtt{L}_1\,;\ldots;\mathtt{L}_\kappa$		$\mathsf{e}, \mathsf{e}_1, \ldots$	$::=$	expressions	
	$\mid$	$\textbf{for}\ i = 1..n\,\{\mathtt{L}\}$		i	$::=$	indices	
	$\mid$	$\textbf{case}\ \mathsf{e}\ \textbf{of}\ \{\mathsf{e}_1 : \mathtt{L}_1 \ldots \mathsf{e}_\kappa : \mathtt{L}_\kappa\}$		u	$::=$	values	
	$\mid$	$\mathtt{operator}\ \{\mathtt{L}_1 \ldots \mathtt{L}_\kappa\}$		m	$::=$	$\mathsf{m} \mid \mathsf{m}_i$	
	$\mid$	$\mathtt{operator}\ i = 1..n\ \{\mathtt{L}\}$					
	$\mid$	$\textbf{while}\ \mathsf{e}\ \textbf{do}\ \{\mathtt{L}\}$					
	$\mid$	$\textbf{ready}$				[runtime syntax]	
$\mathtt{V}$	$::=$	$\textbf{var}\ \mathtt{x} : \mathtt{T}$	variable declaration				
$\mathtt{T}$	$::=$	$\mathsf{Int} \mid \mathsf{Boolean} \mid m \mid \ \ldots$					
$\mathtt{operator}$	$::=$	$\textbf{select} \mid \textbf{spawn} \mid \textbf{shuffle}$					
$\mathtt{C}$	$::=$	$(\mathtt{M}, A)$	configuration	[runtime syntax]			
$\mathtt{M}$	$::=$	$\{a_1 \xrightarrow{u_1:m_1} b_1, \ldots, a_\kappa \xrightarrow{u_\kappa:m_\kappa} b_\kappa\}$	multiset of messages	[runtime syntax]			
A	$::=$	$\{\langle \mathtt{L}_1\rangle^{V_1}_{a_1}, \ldots, \langle \mathtt{L}_\kappa\rangle^{V_\kappa}_{a_\kappa}\}$	multiset of actors	[runtime syntax]			
V	$::=$	$\{\mathtt{x}_1 \mapsto u_1{:}m_1, \ldots, \mathtt{x}_\kappa \mapsto u_\kappa{:}m_\kappa\}$	local store	[runtime syntax]			

Figure 5.9: The sliding window example (page 63) in the language of Figure 5.8.

```
param n

struct m { ... }
struct ack { ... }

// the sender                            // the receiver
actor a {                                actor b {
  spawn i = 1..n {                         spawn i = 1..n {

    var m_ : m;                              var m_ : m;
    var a_ : ack;                            var ack_ : ack;
    var NotDone : Boolean;                   var NotDone : Boolean;
    NotDone := true;

    while NotDone do {                       while NotDone do {
      m_ := ...                                recv(a, m_);
      send(b, m_);                             send(a, ack_);
      recv(b, ack_);                           NotDone := ...
      NotDone := ...                         }
    }                                      }
  }                                      }
}
```

in variable a_ of sort ack. Correspondingly, the receiver spawns n parallel processes, each expecting a message and replying with an acknowledgment.

We will later discuss the inference of local types from such programs. As a prequel to that presentation, we note that applying the type inference rules of Figure 5.12 onto the example code, we get the local types $\|_{i=1}^{n}\big((b!m\,; b?ack)^*\big)$ and $\|_{i=1}^{n}\big((a?m\,; a!ack)^*\big)$, which respectively correspond to actors a and b. These are exactly the results of the projections from the global type $\|_{i=1}^{n}\big((a \xrightarrow{m} b\,; b \xrightarrow{ack} a)^*\big)$, performed on page 77.

5.4.1 Actor Calculus Semantics

Structural congruence on programs is defined as the least equivalence relation $\equiv$ that includes the rules of Figure 5.10. These rules are very similar to the definition of structural congruence on local types, given on page 73. Mirroring Lemma 5.1, which applies to local types, the following statement captures the fact that structural congruence is compositional on set union:

Figure 5.10: Structural congruence on programs in the syntax of page 80.

$$\textsc{Commutativity}\ \frac{\texttt{op} \in \{\textbf{shuffle}, \textbf{select}, \textbf{spawn}\}}{\texttt{op}\{\texttt{L}_1\,\texttt{L}_2\} \equiv \texttt{op}\{\texttt{L}_2\,\texttt{L}_1\}}$$

$$\textsc{Associativity}\ \frac{\texttt{op} \in \{\textbf{select}, \textbf{spawn}\}}{\texttt{op}\{\texttt{L}_1\ \texttt{op}\{\texttt{L}_2\,\texttt{L}_3\}\} \equiv \texttt{op}\{\texttt{op}\{\texttt{L}_1\,\texttt{L}_2\}\,\texttt{L}_3\}}$$

$\textsc{Distributivity}$

$$\begin{array}{rcll}
\texttt{L}_1 \,;\, \textbf{case}\ \texttt{e}\ \textbf{of}\ \{\texttt{e}_2 : \texttt{L}_2\quad \texttt{e}_3 : \texttt{L}_3\} & \equiv & \textbf{case}\ \texttt{e}\ \textbf{of}\ \{\texttt{e}_2 : \texttt{L}_1 \,;\, \texttt{L}_2\quad \texttt{e}_3 : \texttt{L}_1 \,;\, \texttt{L}_3\} & \\
\texttt{L}_1 \,;\, \textbf{select}\{\texttt{L}_2\ \ \texttt{L}_3\} & \equiv & \textbf{select}\{\texttt{L}_1 \,;\, \texttt{L}_2\ \ \texttt{L}_1 \,;\, \texttt{L}_3\} & \\
\textbf{case}\ \texttt{e}\ \textbf{of}\ \{\texttt{e}_1 : \texttt{L}_1\quad \texttt{e}_2 : \texttt{L}_2\} \,;\, \texttt{L}_3 & \equiv & \textbf{case}\ \texttt{e}\ \textbf{of}\ \{\texttt{e}_1 : \texttt{L}_1 \,;\, \texttt{L}_3\quad \texttt{e}_2 : \texttt{L}_2 \,;\, \texttt{L}_3\} & \\
\textbf{select}\ \{\texttt{L}_1\ \ \texttt{L}_2\} \,;\, \texttt{L}_3 & \equiv & \textbf{select}\ \{\texttt{L}_1 \,;\, \texttt{L}_3\ \ \texttt{L}_2 \,;\, \texttt{L}_3\} & \\
\textbf{spawn}\{\texttt{L}_1\ \ \textbf{case}\ \texttt{e}\ \textbf{of}\ \{\texttt{e}_2 : \texttt{L}_2\ \ \texttt{e}_3 : \texttt{L}_3\}\} & \equiv & \textbf{case}\ \texttt{e}\ \textbf{of}\ \{\texttt{e}_2 : \textbf{spawn}\{\texttt{L}_1\,\texttt{L}_2\}\ \ \texttt{e}_3 : \textbf{spawn}\{\texttt{L}_1\,\texttt{L}_3\}\} & \\
\textbf{spawn}\{\texttt{L}_1\ \ \textbf{select}\ \{\texttt{L}_2\ \ \texttt{L}_3\}\} & \equiv & \textbf{select}\ \{\textbf{spawn}\ \{\texttt{L}_1\,\texttt{L}_2\}\ \ \textbf{spawn}\ \{\texttt{L}_1\,\texttt{L}_3\}\} & \\
\textbf{select}\ i = 1..n\ \{\texttt{L}_1\} \,;\, \texttt{L}_2 & \equiv & \textbf{select}\ i = 1..n\ \{\texttt{L}_1 \,;\, \texttt{L}_2\} & i \notin \text{free-names}(\texttt{L}_2) \\
\texttt{L}_1 \,;\, \textbf{select}\ i = 1..n\ \{\texttt{L}_1\} & \equiv & \textbf{select}\ i = 1..n\ \{\texttt{L}_1 \,;\, \texttt{L}_2\} & i \notin \text{free-names}(\texttt{L}_1) \\
\textbf{spawn}\ \{\texttt{L}_1\quad \textbf{select}\ i = 1..n\ \{\texttt{L}_2\}\} & \equiv & \textbf{select}\ i = 1..n\ \{\textbf{spawn}\ \{\texttt{L}_1\ \ \texttt{L}_2\}\} & i \notin \text{free-names}(\texttt{L}_1)
\end{array}$$

$$\textsc{Composition}\ \left[\begin{array}{c}
\dfrac{\texttt{L}_1 \equiv \texttt{L}_1' \quad \texttt{L}_2 \equiv \texttt{L}_2' \quad \texttt{op} \in \{\textbf{shuffle}, \textbf{select}, \textbf{spawn}\}}{\texttt{op}\{\texttt{L}_1\ \ \texttt{L}_2\} \equiv \texttt{op}\{\texttt{L}_1'\ \ \texttt{L}_2'\}} \\[2ex]
\dfrac{\texttt{L}_1 \equiv \texttt{L}_1' \quad \texttt{L}_2 \equiv \texttt{L}_2'}{\textbf{case}\ \texttt{e}\ \textbf{of}\{\texttt{e}_1 : \texttt{L}_1\ \ \texttt{e}_2 : \texttt{L}_2\} \equiv \textbf{case}\ \texttt{e}\ \textbf{of}\{\texttt{e}_1 : \texttt{L}_1'\ \ \texttt{e}_2 : \texttt{L}_2'\}} \\[2ex]
\dfrac{\texttt{L} \equiv \texttt{L}' \quad \texttt{op} \in \{\textbf{select}, \textbf{shuffle}, \textbf{spawn}, \textbf{for}\}}{\texttt{op}\ i = 1..n\ \{\texttt{L}\} \equiv \texttt{op}\ i = 1..n\ \{\texttt{L}'\}}
\end{array}\right.$$

$$\textsc{Expansion}\ \left[\begin{array}{c}
\dfrac{u \in \mathbb{N} \quad \texttt{op} \in \{\textbf{shuffle}, \textbf{select}, \textbf{spawn}\}}{\texttt{op}\ i = 1..u\ \{\texttt{L}\} \equiv \texttt{op}\ \{\texttt{L}[1/i] \ldots \texttt{L}[u/i]\}} \\[2ex]
\dfrac{u \in \mathbb{N}}{\textbf{for}\ i = 1..u\ \{\texttt{L}\} \equiv \texttt{L}[1/i] \,;\, \ldots \,;\, \texttt{L}[u/i]}
\end{array}\right.$$

$$\textsc{Shuffle}\ \frac{f_1 \ldots f_{\kappa!} : \text{bijective } \{1..\kappa\} \mapsto \{1..\kappa\} \qquad f_1 \ldots f_{\kappa!} \text{ distinct}}{\textbf{shuffle}\{\texttt{L}_1 \ldots \texttt{L}_\kappa\} \equiv \textbf{select}\{\texttt{L}_{f_1(1)} \,;\, \ldots \,;\, \texttt{L}_{f_1(\kappa)}\ \cdots\ \texttt{L}_{f_{\kappa!}(1)} \,;\, \ldots \,;\, \texttt{L}_{f_{\kappa!}(\kappa)}\}}$$

$$\textsc{Runtime}\ \left[\begin{array}{cc}
\dfrac{\texttt{L}_1 \equiv \texttt{L}_2}{\langle \texttt{L}_1 \rangle_a^V \equiv \langle \texttt{L}_2 \rangle_a^V} & \dfrac{\langle \texttt{L}_1 \rangle_a^V \equiv \langle \texttt{L}_2 \rangle_a^V}{\left\{\langle \texttt{L}_1 \rangle_a^V\right\} \equiv \left\{\langle \texttt{L}_2 \rangle_a^V\right\}} \\[2ex]
\dfrac{A_1 \equiv A_2}{A \cup A_1 \equiv A \cup A_2} & \dfrac{A_1 \equiv A_2}{(\texttt{M}, A_1) \equiv (\texttt{M}, A_2)}
\end{array}\right.$$

Figure 5.11: The transition relation on programs in the syntax of page 80.

$$\textsc{Lang-Send}\ \frac{V(\mathtt{x}) = u : m}{\mathtt{M},\ A \cup \{\langle \textsf{\textbf{send}}(b, \mathtt{x})\rangle_a^V\} \xrightarrow{\tau} \mathtt{M} \cup \{a \xrightarrow{u:m} b\},\ A \cup \{\langle \textsf{\textbf{ready}}\rangle_a^V\}}$$

$$\textsc{Lang-Seq}\ \frac{\mathtt{M},\ A \cup \{\langle \mathtt{L}_1\rangle_a^V\} \xrightarrow{e} \mathtt{M}',\ A \cup \{\langle \mathtt{L}_1'\rangle_a^{V'}\}}{\mathtt{M},\ A \cup \{\langle \mathtt{L}_1 ; \mathtt{L}_2 ; \ldots ; \mathtt{L}_\kappa\rangle_a^V\} \xrightarrow{e} \mathtt{M}',\ A \cup \{\langle \mathtt{L}_1' ; \mathtt{L}_2 ; \ldots ; \mathtt{L}_\kappa\rangle_a^{V'}\}}$$

$$\textsc{Lang-Ready}\ \mathtt{M},\ A \cup \{\langle \textsf{\textbf{ready}} ; \mathtt{L}_1 ; \ldots ; \mathtt{L}_\kappa\rangle_a^V\} \xrightarrow{\tau} \mathtt{M},\ A \cup \{\langle \mathtt{L}_1 ; \ldots ; \mathtt{L}_\kappa\rangle_a^V\}$$

$$\textsc{Lang-Spawn}\ \frac{\mathtt{M},\ A \cup \{\langle \mathtt{L}_\psi\rangle_a^V\} \xrightarrow{e} \mathtt{M}',\ A \cup \{\langle \mathtt{L}_\psi'\rangle_a^{V'}\}}{\mathtt{M},\ A \cup \{\langle \textsf{\textbf{spawn}}\{\mathtt{L}_1 \ldots \mathtt{L}_\psi \ldots \mathtt{L}_\kappa\}\rangle_a^V\} \xrightarrow{e} \mathtt{M}',\ A \cup \{\langle \textsf{\textbf{spawn}}\{\mathtt{L}_1 \ldots \mathtt{L}_\psi' \ldots \mathtt{L}_\kappa\}\rangle_a^{V'}\}}$$

$$\textsc{Lang-Spawn-Ready}\ \mathtt{M},\ A \cup \{\langle \textsf{\textbf{spawn}}\, \{\textsf{\textbf{ready}} \ldots \textsf{\textbf{ready}}\}\rangle_a^V\} \xrightarrow{\tau} \mathtt{M},\ A \cup \{\langle \textsf{\textbf{ready}}\rangle_a^V\}$$

$$\textsc{Lang-Ext-Choice}\ \frac{\mathrm{eval}(V, \mathtt{e}) = \mathrm{eval}(V, \mathtt{e}_\psi)}{\mathtt{M},\ A \cup \{\langle \textsf{\textbf{case}}\ \mathtt{e}\ \textsf{\textbf{of}}\ \{\mathtt{e}_1 : \mathtt{L}_1 \ldots \mathtt{e}_\psi : \mathtt{L}_\psi \ldots \mathtt{e}_\kappa : \mathtt{L}_\kappa\}\rangle_a^V\} \xrightarrow{\tau} \mathtt{M},\ A \cup \{\langle \mathtt{L}_\psi\rangle_a^V\}}$$

$$\textsc{Lang-Int-Choice}\ \frac{\begin{array}{c}\text{“}a_\psi \xrightarrow{u:m} b\text{”} \in \mathtt{M} \\ \mathrm{first}(\mathtt{L}_\psi) = \textsf{\textbf{recv}}(a_\psi, \mathtt{x}_\psi) \quad \mathrm{typeof}(V, \mathtt{x}_\psi) = m\end{array}}{\mathtt{M},\ A \cup \{\langle \textsf{\textbf{select}}\{\mathtt{L}_1 \ldots \mathtt{L}_\psi \ldots \mathtt{L}_\kappa\}\rangle_b^V\} \xrightarrow{\tau} \mathtt{M},\ A \cup \{\langle \mathtt{L}_\psi\rangle_b^V\}}$$

$$\textsc{Lang-Recv}\ \frac{\mathrm{typeof}(V, \mathtt{x}) = m \quad V' = V[\mathtt{x} \mapsto u : m] \quad e = \text{“}a \xrightarrow{m} b\text{”}}{\mathtt{M} \cup \{a \xrightarrow{u:m} b\},\ A \cup \{\langle \textsf{\textbf{recv}}(a, \mathtt{x})\rangle_b^V\} \xrightarrow{e} \mathtt{M},\ A \cup \{\langle \textsf{\textbf{ready}}\rangle_b^{V'}\}}$$

$$\textsc{Lang-Assignment}\ \frac{\mathrm{typeof}(V, \mathtt{x}) = \mathrm{typeof}(V, \mathtt{e}) = m \quad \mathrm{eval}(V, \mathtt{e}) = u \quad V' = V[\mathtt{x} \mapsto u : m]}{\mathtt{M},\ A \cup \{\langle \mathtt{x} := \mathtt{e}\rangle_a^V\} \xrightarrow{\tau} \mathtt{M},\ A \cup \{\langle \textsf{\textbf{ready}}\rangle_a^{V'}\}}$$

$$\textsc{Lang-VarDecl}\ \frac{V' = V[\mathtt{x} \mapsto :\mathtt{T}]}{\mathtt{M},\ A \cup \{\langle \textsf{\textbf{var}}\ \mathtt{x} : \mathtt{T}\rangle_a^V\} \xrightarrow{\tau} \mathtt{M},\ A \cup \{\langle \textsf{\textbf{ready}}\rangle_a^{V'}\}}$$

$$\textsc{Lang-While}\ \left[\begin{array}{c} \dfrac{\mathit{eval}(V, \mathtt{e}) = \mathit{true}}{\mathtt{M},\ A \cup \{\langle \textsf{\textbf{while}}\ \mathtt{e}\ \{\mathtt{L}\}\rangle_a^V\} \xrightarrow{\tau} \mathtt{M},\ A \cup \{\langle \mathtt{L} ; \textsf{\textbf{while}}\ \mathtt{e}\ \{\mathtt{L}\}\rangle_a^V\}} \\[2ex] \dfrac{\mathit{eval}(V, \mathtt{e}) = \mathit{false}}{\mathtt{M},\ A \cup \{\langle \textsf{\textbf{while}}\ \mathtt{e}\ \{\mathtt{L}\}\rangle_a^V\} \xrightarrow{\tau} \mathtt{M},\ A \cup \{\langle \textsf{\textbf{ready}}\rangle_a^V\}} \end{array}\right.$$

$$\textsc{Lang-Expansion}\ \left[\begin{array}{c} \dfrac{V(n) = u : \mathit{param}}{\mathtt{M},\ A \cup \{\langle \textsf{\textbf{for}}\ i = 1..n\ \{\mathtt{L}\}\rangle_a^V\} \xrightarrow{\tau} \mathtt{M},\ A \cup \{\langle \mathtt{L}[1/i] ; \ldots ; \mathtt{L}[u/i]\rangle_a^V\}} \\[2ex] \dfrac{V(n) = u : \mathit{param} \quad \textsf{op} \in \{\textsf{\textbf{shuffle}}, \textsf{\textbf{spawn}}, \textsf{\textbf{select}}\}}{\mathtt{M},\ A \cup \{\langle \mathtt{op}\ i = 1..n\ \{\mathtt{L}\}\rangle_a^V\} \xrightarrow{\tau} \mathtt{M},\ A \cup \{\langle \mathtt{op}\, \{\mathtt{L}[1/i] \ldots \mathtt{L}[u/i]\}\rangle_a^V\}} \end{array}\right.$$

$$\textsc{Lang-Shuffle}\ \frac{f_1 \ldots f_{\kappa!} : \text{bijective } \{1..\kappa\} \mapsto \{1..\kappa\} \quad f_1 \ldots f_{\kappa!} \text{ distinct}}{\begin{array}{l}\mathtt{M},\ A \cup \{\langle \textsf{\textbf{shuffle}}\{\mathtt{L}_1 \ldots \mathtt{L}_\kappa\}\rangle_a^V\} \\ \qquad \xrightarrow{\tau} \mathtt{M},\ A \cup \{\langle \textsf{\textbf{select}}\{\mathtt{L}_{f_1(1)} ; \ldots ; \mathtt{L}_{f_1(\kappa)} \ \cdots\ \mathtt{L}_{f_{\kappa!}(1)} ; \ldots ; \mathtt{L}_{f_{\kappa!}(\kappa)}\}\rangle_a^V\end{array}}$$

Lemma 5.4 (Congruence is compositional on set union)**.** Let $\mathtt{L}_1$ and $\mathtt{L}_2$ be program statements, a be an actor name, $\mathtt{M}$ a message multiset, V a local store and A a multiset of running actors, all as in Figure 5.8.
Then, $\mathtt{L}_1 \equiv \mathtt{L}_2$ iff $A \cup \{\langle \mathtt{L}_1 \rangle_a^V\} \equiv A \cup \{\langle \mathtt{L}_2 \rangle_a^V\}$.
Equivalently, $\mathtt{L}_1 \equiv \mathtt{L}_2$ iff $(\mathtt{M},\ A \cup \{\langle \mathtt{L}_1 \rangle_a^V\}) \equiv (\mathtt{M},\ A \cup \{\langle \mathtt{L}_2 \rangle_a^V\})$.

Proof. Directly from the Runtime rules in Figure 5.10. □

The transition relation $\longrightarrow$ on programs is the least such relation that includes the rules in Figure 5.11. The rules transform runtime configurations of the form $(\mathtt{M}, A)$ where $\mathtt{M}$ contains pending messages, and A is the multiset of concurrently executing actors. Elements of $\mathtt{M}$ have the form $a \xrightarrow{u:m} b$, where a is the sender, u is the sent value, m is the message sort, and b is the receiving actor. Elements of A have the form $\langle \mathtt{L} \rangle_a^V$, where a is the executing actor, $\mathtt{L}$ is the code it executes, and V is the associated value store. The latter maps variable names to their values and types. For example, $V(\mathtt{x}) = u : m$ means that the variable $\mathtt{x}$ has value u and sort m (in this case, per our conventions, $\mathtt{x}$ is a message).

The initial configuration of a program $\mathtt{P}$ is a pair consisting of the empty message set, and a multiset of concurrently running actors A. The latter is generated according to definitions of the form "**actor** $\mathtt{a}[n]\{\mathtt{L}\}$" in the program. When execution starts, parameters n are replaced by their runtime values, and so such a definition generates running actors $\langle \mathtt{L} \rangle_{\mathtt{a}_1}^V \dots \langle \mathtt{L} \rangle_{\mathtt{a}_n}^V$. In the initial configuration, all actors are associated with a copy of the same store V, which maps parameter names to their values. During execution, individual actor stores are updated independently; for example, when dealing with a variable assignment $\mathtt{x} := \mathtt{e}$, rule Lang-Assignment evaluates the expression $\mathtt{e}$ and updates V so that it maps $\mathtt{x}$ to $\mathit{eval}(V, \mathtt{e})$. Since each actor has its own store, such an assignment is local to the actor where the assignment statement appears. Variable declarations **var** $\mathtt{x} : \mathtt{T}$ update the store to "remember" that the type of $\mathtt{x}$ is $\mathtt{T}$, i.e., $V'(\mathtt{x}) = \, : \mathtt{T}$, in rule Lang-VarDecl.

The reduction rules for this language are similar to the reduction rules for local types, given on page 74. They share the same compositional structure; for example, rule Lang-Seq reduces sequentially composed statements $\mathtt{L}_1 ; \dots ; \mathtt{L}_\kappa$ to $\mathtt{L}'_1 ; \dots ; \mathtt{L}_\kappa$, provided that $\mathtt{L}_1$ reduces to $\mathtt{L}'_1$. The rule for **case** statements evaluates the expression $\mathtt{e}$ and branches off to the code segment $\mathtt{L}_\psi$ that is guarded by the same value. Correspondingly, **select** statements choose the code segment $\mathtt{L}_\psi$ that starts with a message receipt command which matches a message available in the set $\mathtt{M}$. Lang-Spawn reduces statements of the form **spawn**$\{\mathtt{L}_1 \dots \mathtt{L}_\kappa\}$, implementing concurrent semantics for the contained statements $\mathtt{L}_1 \dots \mathtt{L}_\kappa$.

Upon reducing a runtime configuration $\mathtt{C}$, the rules in Figure 5.11 produce events, i.e., $\mathtt{C} \xrightarrow{e} \mathtt{C}'$. Events can have the form $a \xrightarrow{m} b$, produced by rule Lang-Recv to signify the receipt of a message. In all other cases, the generated event is τ, as in the behavior of local types, i.e., events $a \xrightarrow{m} b$ are generated at the point of message receipt. The statements that follow mirror the respective results for local types:

Lemma 5.5 (Reduction is compositional on set union)**.** Let $\mathtt{L}$ and $\mathtt{L}'$ be program statements, $\mathtt{M}$ and $\mathtt{M}'$ be multisets of pending messages, V and V' be local stores, a be some actor name, and A be a multiset of running actors. Then,
$(\mathtt{M},\ \{\langle\mathtt{L}\rangle_a^V\}) \longrightarrow (\mathtt{M}',\ \{\langle\mathtt{L}'\rangle_a^{V'}\})$ iff $(\mathtt{M},\ A \cup \{\langle\mathtt{L}\rangle_a^V\}) \longrightarrow (\mathtt{M}',\ A \cup \{\langle\mathtt{L}'\rangle_a^{V'}\})$.
As a direct consequence,
$(\mathtt{M},\ \{\langle\mathtt{L}\rangle_a^V\}) \longrightarrow^* (\mathtt{M}',\ \{\langle\mathtt{L}'\rangle_a^{V'}\})$ iff $(\mathtt{M},\ A \cup \{\langle\mathtt{L}\rangle_a^V\}) \longrightarrow^* (\mathtt{M}',\ A \cup \{\langle\mathtt{L}'\rangle_a^{V'}\})$.

Proof. By induction on the size of A, and using the definition of the reduction relation in Figure 5.11. □

Lemma 5.6 (Congruence follows semantics—single element)**.** Let $\mathtt{L}_1$, $\mathtt{L}_1'$ and $\mathtt{L}_2$ be running actors, $\mathtt{M}$ and $\mathtt{M}'$ be multisets of pending messages, V and V' be local stores, and a be an actor name. If $\mathtt{L}_1 \equiv \mathtt{L}_2$ and $(\mathtt{M}, \{\langle\mathtt{L}_1\rangle_a^V\}) \longrightarrow^* (\mathtt{M}', \{\langle\mathtt{L}_1'\rangle_a^{V'}\})$, then there exists $\mathtt{L}_2'$ such that $(\mathtt{M}, \{\langle\mathtt{L}_2\rangle_a^V\}) \longrightarrow^* (\mathtt{M}', \{\langle\mathtt{L}_2'\rangle_a^{V'}\})$ with $\mathtt{L}_1' \equiv \mathtt{L}_2'$.

Proof. By induction on the structural congruence and reduction relations, and using the fact that structural congruence and the reduction relation are both compositional on set union (lemmas 5.4 and 5.5). As done for Lemma 5.3 on page 75. □

Our intuition that structural congruence preserves the semantics of the calculus is captured in the following theorem:

Theorem 5.3. Let A_1, A_1', A_2, $\mathtt{M}$, and $\mathtt{M}'$ be as in Figure 5.8. If $A_1 \equiv A_2$ and $(\mathtt{M}, A_1) \longrightarrow^* (\mathtt{M}', A_1')$ then there exists A_2' such that $(\mathtt{M}, A_2) \longrightarrow^* (\mathtt{M}', A_2')$ and $A_1' \equiv A_2'$.

Proof. As for Theorem 5.2 on page 76, using lemmas 5.4 and 5.5. □

5.5 Typing

This section presents an algorithm for assigning local types to programs written in the syntax of Figure 5.8, and provides standard sanity checks. For reasons of presentation clarity, we distinguish between value- and local- types, the latter abstracting actor behaviors. We use the term *sort* for the types of messages, variables and expressions, while the term *type* is used exclusively to refer to local types.

The type inference relation is defined in Figure 5.12. Given a program $\mathtt{P}$, the rules derive a map $\{a : L_a\}_{a \in \mathtt{P}}$, which associates actors in the program with their local types. We use

Figure 5.12: Rules for assigning local types to programs. The variables a, b, m etc. are as in Figure 5.1. Program information is carried along in Δ, accessed by the auxiliary functions $\mathit{sort}(\cdot,\cdot)$, which returns the sort of variables, and $\mathit{behavior}(\cdot,\cdot)$, which returns the code corresponding to the given actor's behavior.

$$\textsc{Inf-Send}\;\frac{a \in \mathrm{dom}(\Delta) \qquad \mathit{sort}(\Delta, \mathtt{x}) = m}{\Delta \vdash \textbf{send}(a, \mathtt{x}) : a!m} \qquad \textsc{Inf-Recv}\;\frac{a \in \mathrm{dom}(\Delta) \qquad \mathit{sort}(\Delta, \mathtt{x}) = m}{\Delta \vdash \textbf{recv}(a, \mathtt{x}) : a?m}$$

$$\textsc{Inf-Seq}\;\frac{\Delta \vdash \mathtt{L}_1 : L_1 \;\ldots\; \Delta \vdash \mathtt{L}_\kappa : L_\kappa}{\Delta \vdash \mathtt{L}_1 \,;\ldots;\, \mathtt{L}_2 : (L_1 \,;\ldots;\, L_\kappa)} \qquad \textsc{Inf-For}\;\frac{\Delta \vdash \mathtt{L} : L \qquad \mathit{sort}(\Delta, n) = \mathit{param}}{\Delta \vdash \textbf{for}\; i = 1..n\,\{\mathtt{L}\} : \bigodot_{i=1}^{n} L}$$

$$\textsc{Inf-Spawn}\;\frac{\Delta \vdash \mathtt{L}_1 : L_1 \;\ldots\; \Delta \vdash \mathtt{L}_\kappa : L_\kappa}{\Delta \vdash \textbf{spawn}\,\{\mathtt{L}_1 \ldots \mathtt{L}_\kappa\} : (L_1 \| \ldots \| L_\kappa)} \qquad \textsc{Inf-Spawn-N}\;\frac{\Delta \vdash \mathtt{L} : L \qquad \mathit{sort}(\Delta, n) = \mathit{param}}{\Delta \vdash \textbf{spawn}\; i = 1..n\,\{\mathtt{L}\} : \big\|_{i=1}^{n} L}$$

$$\textsc{Inf-Shuffle}\;\frac{\Delta \vdash \mathtt{L}_1 : L_1 \;\ldots\; \Delta \vdash \mathtt{L}_\kappa : L_\kappa}{\Delta \vdash \textbf{shuffle}\,\{\mathtt{L}_1 \ldots \mathtt{L}_\kappa\} : (L_1 \otimes \ldots \otimes L_\kappa)} \qquad \textsc{Inf-Shuffle-N}\;\frac{\Delta \vdash \mathtt{L} : L \qquad \mathit{sort}(\Delta, n) = \mathit{param}}{\Delta \vdash \textbf{shuffle}\; i = 1..n\,\{\mathtt{L}\} : \bigotimes_{i=1}^{n} L}$$

$$\textsc{Inf-Select}\;\frac{\Delta \vdash \mathtt{L}_1 : L_1 \;\ldots\; \Delta \vdash \mathtt{L}_\kappa : L_\kappa}{\Delta \vdash \textbf{select}\,\{\mathtt{L}_1 \ldots \mathtt{L}_\kappa\} : (L_1 \oplus \ldots \oplus L_\kappa)} \qquad \textsc{Inf-Select-N}\;\frac{\Delta \vdash \mathtt{L} : L \qquad \mathit{sort}(\Delta, n) = \mathit{param}}{\Delta \vdash \textbf{select}\; i = 1..n\,\{\mathtt{L}\} : \bigoplus_{i=1}^{n} L}$$

$$\textsc{Inf-Case}\;\frac{\Delta \vdash \mathtt{L}_1 : L_1 \;\ldots\; \Delta \vdash \mathtt{L}_\kappa : L_\kappa}{\Delta \vdash \textbf{case}\;\mathtt{e}\;\textbf{of}\,\{\mathtt{e}_1 : \mathtt{L}_1 \;\ldots\; \mathtt{e}_\kappa : \mathtt{L}_\kappa\} : (L_1 \oplus \ldots \oplus L_\kappa)}$$

$$\textsc{Inf-While}\;\frac{\Delta \vdash \mathtt{L} : L \qquad \mathit{sort}(\Delta, \mathtt{e}) = \mathit{Boolean}}{\Delta \vdash \textbf{while}\;\mathtt{e}\;\textbf{do}\,\{\mathtt{L}\} : (L^*)}$$

$$\textsc{Inf-Ready}\;\Delta \vdash \textbf{ready} : \tau \qquad\qquad \textsc{Inf-Var}\;\Delta \vdash \mathtt{V} : \tau$$

$$\textsc{Inf-Program}\;\frac{\Delta = \mathit{info}(\mathtt{P}) \qquad \forall a.(\mathit{behavior}(\Delta, a) = \mathtt{L}_a \implies \Delta \vdash \mathtt{L}_a : L_a)}{\vdash \mathtt{P} : \{a : L_a \mid \mathit{behavior}(\Delta, a) = \mathtt{L}_a\}}$$

Δ for static program information, i.e., $\Delta = \mathit{info}(\mathtt{P})$ as initiated in rule Inf-Program. This information is carried up the deduction chain, and consulted by functions *sort* and *behavior*. These extract, respectively, the sorts of expressions and the code corresponding to an actor behavior. We omit the formal definition of these functions for the sake of brevity.

The rules produce judgments of the form $\Delta \vdash \mathtt{L} : L$, meaning that with program information Δ, the code block $\mathtt{L}$ is assigned the type L. For example, rule Inf-Seq composes the types $L_1 \ldots L_\kappa$ corresponding to actions $\mathtt{L}_1 \ldots \mathtt{L}_\kappa$ to derive the type of $\mathtt{L}_1 ; \ldots ; \mathtt{L}_\kappa$ to be $L_1 ; \ldots ; L_\kappa$. Similarly, rule Inf-Shuffle-N associates the type $\bigotimes_{i=1}^{n} L$ with the code segment **shuffle** $i = 1..n\{\mathtt{L}\}$ given that $\Delta \vdash \mathtt{L} : L$ and that n is a parameter, i.e., $\mathit{sort}(\Delta, n) = \mathit{param}$.

Syntax-Directedness. The rules of Figure 5.12 form an algorithm, because they are syntax-directed: there is exactly one rule for each of the syntax rules (given in Figure 5.8); and the premises of each rule apply inductively to the structure of the current program segment. As a result, with regard to the inference mechanism, (i) exactly one rule applies at each step, and (ii) the procedure terminates.

The types of runtime configurations. It is possible to extend the above mechanism to runtime configurations. Let $\mathtt{C} = (\mathtt{M}, A)$ be a runtime configuration where $A = \{\langle \mathtt{L}_\psi \rangle_{a_\psi}^{V_\psi}\}_{\psi=1..\kappa}$. Using the rules of Figure 5.12, we can assign a local type L_ψ to each $\mathtt{L}_\psi$ such that $\Delta \vdash \mathtt{L}_\psi : L_\psi$. These local types form a set $\Lambda = \{\langle L_\psi \rangle_{a_\psi} \mid \Delta \vdash \mathtt{L}_\psi : L_\psi\}_{\psi=1..\kappa}$. We therefore extend typing to runtime configurations, such that a configuration $(\mathtt{M}, A)$ is assigned the type $(s(\mathtt{M}), \Lambda)$ where $s(\mathtt{M})$ is the same as $\mathtt{M}$ but with only sorting information retained. Formally,

$$s(\mathtt{M}) = \begin{cases} \emptyset & \text{if } \mathtt{M} = \emptyset \\ s(\mathtt{M}') \cup \{a \xrightarrow{m} b\} & \text{if } \mathtt{M} = \mathtt{M}' \cup \{a \xrightarrow{u:m} b\} \end{cases}$$

To clarify, $s(\mathtt{M})$ follows the syntax of M on page 72. The type inference rules for runtime configurations are shown in Figure 5.13. In judgments $\Delta \vdash (\mathtt{M}, A) : (M, \Lambda)$, the syntax of $(\mathtt{M}, A)$ is that of program configurations $\mathtt{C}$ given on page 80, while the syntax of (M, Λ) is that of local type configurations C on page 72.

5.5.1 Meta-theory

We argue that $\Delta \vdash \mathtt{C} : C$ implies that the local type configuration C reduces in a manner that mirrors the messaging semantics of the program configuration $\mathtt{C}$. Formally, the close relationship between the reduction rules for local types and programs (Figures 5.6 and 5.11 respectively) directly suggests the following standard type preservation results:

Figure 5.13: Assigning types to runtime configurations.

$$\Delta \vdash \{a \xrightarrow{u:m} b\} : \{a \xrightarrow{m} b\}$$

$$\frac{\Delta \vdash \mathtt{M}_1 : M_1 \qquad \Delta \vdash \mathtt{M}_2 : M_2}{\Delta \vdash \mathtt{M}_1 \cup \mathtt{M}_2 : M_1 \cup M_2} \qquad \frac{\Delta \vdash \mathtt{L} : L}{\Delta \vdash \{\langle \mathtt{L} \rangle_a^V\} : \{\langle L \rangle_a\}}$$

$$\frac{\Delta \vdash A_1 : \Lambda_1 \qquad \Delta \vdash A_2 : \Lambda_2}{\Delta \vdash A_1 \cup A_2 : \Lambda_1 \cup \Lambda_2} \qquad \frac{\Delta \vdash \mathtt{M} : M \qquad \Delta \vdash A : \Lambda}{\Delta \vdash (\mathtt{M}, A) : (M, \Lambda)}$$

Lemma 5.7 (Single-Element Subject Reduction)**.** Let $\mathtt{M}$ and $\mathtt{M}'$ be message multisets, and $M = s(\mathtt{M})$. Let V and V' be value stores, $\mathtt{L}$ and $\mathtt{L}'$ be program statements, L be a local type, a be an actor name, and Δ be program information.
If $\Delta \vdash (\mathtt{M}, \{\langle \mathtt{L} \rangle_a^V\} : (M, \{\langle L \rangle_a\})$ and $(\mathtt{M}, \{\langle \mathtt{L} \rangle_a^V\}) \longrightarrow^* (\mathtt{M}', \{\langle \mathtt{L}' \rangle_a^{V'}\})$, then there exist M' and L' such that $(M, \{\langle L \rangle_a\}) \longrightarrow^* (M', \{\langle L' \rangle_a\})$ and $\Delta \vdash (\mathtt{M}', \{\langle \mathtt{L}' \rangle_a^{V'}\}) : (M', \{\langle L' \rangle_a\})$.

Proof. By induction on the structure of configurations, and the relations $\longrightarrow$ and $\vdash$.

Sequencing

Assume the following configurations and associated reduction:

$$(\mathtt{M},\ \{\langle \mathtt{L}_1\,;\mathtt{L}_2\,;\ldots;\mathtt{L}_\kappa \rangle_a^V\}) \longrightarrow (\mathtt{M}',\ \{\langle \mathtt{L}_1'\,;\mathtt{L}_2\,;\ldots;\mathtt{L}_\kappa \rangle_a^{V'}\}) \tag{5.8}$$

Assume $M = s(\mathtt{M})$ and the local types $L_1, L_2, \ldots, L_\kappa$, such that

$$(\mathtt{M},\ \{\langle \mathtt{L}_1\,;\mathtt{L}_2\,;\ldots;\mathtt{L}_\kappa \rangle_a^V\}) : (M,\ \{\langle L_1\,; L_2\,;\ldots; L_\kappa \rangle_a\}) \tag{5.9}$$

The above relations are as in the premises of this lemma. From 5.8 and the transition rule Lang-Seq (page 83), we have

$$(\mathtt{M},\ \{\langle \mathtt{L}_1 \rangle_a^V\}) \longrightarrow (\mathtt{M}',\ \{\langle \mathtt{L}_1' \rangle_a^{V'}\}) \tag{5.10}$$

From 5.9 and the typing rule for sequencing (Inf-Seq on page 86), we get

$$\Delta \vdash (\mathtt{M},\ \{\langle \mathtt{L}_1 \rangle_a^V\}) : (M,\ \{\langle L_1 \rangle_a\}) \tag{5.11}$$

Applying the inductive hypothesis onto 5.10 and 5.11, we have that there exist M' and L'_1 such that

$$\big(M,\ \{\langle L_1\rangle_a\}\big) \longrightarrow^* \big(M',\ \{\langle L'_1\rangle_a\}\big) \tag{5.12}$$

$$\Delta \vdash \big(\mathtt{M}',\ \{\langle \mathtt{L}'_1\rangle_a^{V'}\}\big) : \big(M',\ \{\langle L'_1\rangle_a\}\big) \tag{5.13}$$

From 5.12 and sequence reduction for local types (rule LS-SEQ on page 74), we get

$$\big(M,\ \{\langle L_1 \,;\, L_2 \,;\, \dots \,;\, L_\kappa\rangle_a\}\big) \longrightarrow^* \big(M',\ \{\langle L'_1 \,;\, L_2 \,;\, \dots \,;\, L_\kappa\rangle_a\}\big) \tag{5.14}$$

From 5.13 and rule INF-SEQ (page 86), we get

$$\Delta \vdash \big(\mathtt{M}',\ \{\langle \mathtt{L}'_1 \,;\, \mathtt{L}_2 \,;\, \dots \,;\, \mathtt{L}_\kappa\rangle_a^{V'}\}\big) : \big(M',\ \{\langle L'_1 \,;\, L_2 \,;\, \dots \,;\, L_\kappa\rangle_a\}\big) \tag{5.15}$$

The result consists of statements 5.14 and 5.15.

We omit the rest of the cases (concurrent composition, choice, etc.) because they pose no additional technical difficulties, and can be shown in the same manner as the case of sequencing. □

Theorem 5.4 (Subject Reduction). Let $\mathtt{C}$ and $\mathtt{C}'$ be program runtime configurations, C be a configuration of local types, and Δ be program information as above. If $\Delta \vdash \mathtt{C} : C$ and $\mathtt{C} \longrightarrow^* \mathtt{C}'$, then there exists C' such that $C \longrightarrow^* C'$ and $\Delta \vdash \mathtt{C}' : C'$.

Proof. Directly from Lemma 5.7 and the compositional nature of the rules in Figure 5.13. □

The converse is also true; that is, program configuration reductions predicted by the semantics of the respective types are guaranteed to take place:

Theorem 5.5 (Session Fidelity). Let $\mathtt{C}$ be a program configuration, and let C be its assigned type, i.e., $\Delta \vdash \mathtt{C} : C$ for some Δ. Then, $C \longrightarrow^* C'$ implies that there exists $\mathtt{C}'$ such that $\mathtt{C} \longrightarrow^* \mathtt{C}'$ with $\Delta \vdash \mathtt{C}' : C'$.

Proof. By induction on the structure of types, and the relations $\longrightarrow$ and $\vdash$, mirroring the proof of subject reduction. □

5.6 Extensions

In this section, we consider a few possible extensions to the system. First, we deal with the problem of whether a given global type is *realizable*. That is, given a global type G, the question of whether there exists a program in the syntax of Figure 5.8 that produces the same set of traces as those prescribed by G. We then proceed with a short discussion on index sets, and the problem outlined on page 78. Finally, we touch upon certain notions from the literature that a future version of the system can incorporate.

5.6.1 On the Realizability of Global Types

In this section, we are concerned with static, structural checks to decide whether a global type is realizable; that is, the question of whether a protocol given in the language of Figure 5.1 is implementable by a program in the syntax of Figure 5.8.

To bootstrap the discussion, we need to define the acceptable message sequences that a type implies for a protocol implementation. We remind the reader that a runtime configuration of local types C has the form (M, Λ), where M is the set of pending messages, and Λ is a set of processes which reduce local types concurrently. Per the reduction relation of Figure 5.6, such configurations reduce producing events of the form $a \xrightarrow{m} b$ when a message is received, and τ in all other cases. In this section, we are only interested in events of the first kind, and the possible sequences of such events that a set of local types can produce are called the *traces* of the set:

Definition 5.1 (Local Type Traces)**.** Let Λ be as defined in Figure 5.4. The set of traces producible by Λ, written $\mathrm{tr}(\Lambda)$, is the set of possible non-τ event sequences producible from the initial configuration $(\emptyset, \Lambda)$ by application of the rules in Figure 5.6 until termination. We say that Λ terminates when it has been reduced to a set of processes of the form $\langle\tau\rangle_a$

The respective definition for programs follows. We remind the reader that for a given program $\mathtt{P}$, the initial configuration contains one actor of the form $\langle\mathtt{L}\rangle_a^V$ for each actor definition in the program, where a is the actor's name, $\mathtt{L}$ is its behavior, and V is its value store. Program runtime configurations $\mathtt{C}$ have the form $(\mathtt{M}, A)$ where $\mathtt{M}$ is the multiset of pending messages, and A contains the running actors. We write $\mathtt{C} \xrightarrow{e} \mathtt{C}'$ to say that $\mathtt{C}$ reduces to $\mathtt{C}'$ producing event e. Events are of the form $a \xrightarrow{m} b$ upon message receipt (rule Lang-Recv), and τ in all other cases. As before, we are only interested in events of the first kind.

Definition 5.2 (Program Traces)**.** The set of traces producible by a program $\mathtt{P}$, written $\mathrm{tr}(\mathtt{P})$, is the set of possible non-τ event sequences producible from the initial configuration $\mathit{init}(P)$, by application of the rules in Figure 5.11.

It is important to observe that events of interest, i.e., of the form $a \xrightarrow{m} b$ and $a \xrightarrow{m:u} b$, are produced at the point of message receipt (see rules LS-Recv and Lang-Recv on pages 74 and 83, respectively).

Figure 5.14: The auxiliary functions first and last, used to develop structural realizability criteria for global types.

$$\mathrm{first}(G) = \begin{cases} \{a \xrightarrow{m} b\} & \text{if } G \equiv a \xrightarrow{m} b \\ \mathrm{first}(G_1) & \text{if } G \equiv (G_1\,;G_2) \\ \mathrm{first}(G_1) & \text{if } G \equiv \overset{n}{\underset{i=1}{\mathtt{OP}}}(G_1) & \mathtt{OP} \in \{\circledcirc, \bigoplus, \bigotimes, \|\} \\ \mathrm{first}(G_1) & \text{if } G \equiv (G_1^*) \\ \bigcup_{\psi=1}^{\kappa} \mathrm{first}(G_\psi) & \text{if } G \equiv (G_1\,\mathtt{op}\ldots\mathtt{op}\,G_\kappa) & \mathtt{op} \in \{\oplus, \otimes, \|\} \end{cases}$$

$$\mathrm{last}(G) = \begin{cases} \{a \xrightarrow{m} b\} & \text{if } G \equiv a \xrightarrow{m} b \\ \mathrm{last}(G_2) & \text{if } G \equiv (G_1\,;G_2) \\ \mathrm{last}(G_1) & \text{if } G \equiv \overset{n}{\underset{i=1}{\mathtt{OP}}}(G_1) & \mathtt{OP} \in \{\circledcirc, \bigoplus, \bigotimes, \|\} \\ \mathrm{last}(G_1) & \text{if } G \equiv (G_1^*) \\ \bigcup_{\psi=1}^{\kappa} \mathrm{last}(G_\psi) & \text{if } G \equiv (G_1\,\mathtt{op}\ldots\mathtt{op}\,G_\kappa) & \mathtt{op} \in \{\oplus, \otimes, \|\} \end{cases}$$

$$\mathrm{first}(L) = \begin{cases} \{a!m\} & \text{if } L \equiv a!m \\ \{a?m\} & \text{if } L \equiv a?m \\ \mathrm{first}(L_1) & \text{if } L \equiv L_1\,;L_2 \\ \mathrm{first}(L_1) & \text{if } L \equiv \overset{n}{\underset{i=1}{\mathtt{OP}}}(L_1) & \mathtt{OP} \in \{\circledcirc, \bigoplus, \bigotimes, \|\} \\ \mathrm{first}(L_1) & \text{if } L \equiv (L_1^*) \\ \bigcup_{\psi=1}^{\kappa} \mathrm{first}(L_\psi) & \text{if } L \equiv (L_1\,\mathtt{op}\ldots\mathtt{op}\,L_\kappa) & \mathtt{op} \in \{\oplus, \otimes, \|\} \end{cases}$$

$$\mathrm{last}(L) = \begin{cases} \{a!m\} & \text{if } L \equiv a!m \\ \{a?m\} & \text{if } L \equiv a?m \\ \mathrm{last}(L_2) & \text{if } L \equiv L_1\,;L_2 \\ \mathrm{last}(L_1) & \text{if } L \equiv \overset{n}{\underset{i=1}{\mathtt{OP}}}(L_1) & \mathtt{OP} \in \{\circledcirc, \bigoplus, \bigotimes, \|\} \\ \mathrm{last}(L_1) & \text{if } L \equiv (L_1^*) \\ \bigcup_{\psi=1}^{\kappa} \mathrm{last}(L_\psi) & \text{if } L \equiv (L_1\,\mathtt{op}\ldots\mathtt{op}\,L_\kappa) & \mathtt{op} \in \{\oplus, \otimes, \|\} \end{cases}$$

In order to guarantee that projection maintains the semantics of global types, we impose restrictions on their structure. To facilitate the discussion, Figure 5.14 defines the auxiliary functions first and last, which return the first and last element of a type, respectively. The functions first and last are extended to traces (i.e., event sequences) in the obvious way. Because the well-formedness of global types depends at times on the form of the respective projections, we first discuss well-formedness criteria for local types. These have the purpose of ensuring that choice constructs implement either internal, or external choice—guaranteeing the correct functioning of rules LS-INT-CHOICE and LS-EXT-CHOICE (page 74). We subsequently discuss examples of global types for which the projected local types do not produce the same traces, and propose well-formedness criteria to avoid the problems.

Well-Formed Local Types

The syntax given on page 72 allows choice branches to begin with both sending and receiving actions. In order to ensure choice semantics are implementable, we demand that local types be *well-formed*, which in the case of $L \equiv L_1 \oplus L_2$ is equivalent to one of the following two statements being true:

$$\begin{aligned}\mathrm{int}(L) &\stackrel{\text{def}}{=} \mathrm{first}(L_1) = \{a_1?m_1\} \wedge \mathrm{first}(L_2) = \{a_2?m_2\} \wedge (a_1 \neq a_2 \vee m_1 \neq m_2)\\ \mathrm{ext}(L) &\stackrel{\text{def}}{=} \mathrm{first}(L_1) = \{a_1!m_1\} \wedge \mathrm{first}(L_2) = \{a_2!m_2\} \wedge (a_1 \neq a_2 \vee m_1 \neq m_2)\end{aligned}$$

In other words, $L_1 \oplus L_2$ is well-formed iff either (a) both L_1 and L_2 start with a distinct receive action; or (b) they both start with a distinct send action. The two statements capture the concepts of internal and external choice, respectively. The relation wf$(\cdot)$ on local types captures well-formedness,[1] and is defined inductively in Figure 5.15. It holds up to structural congruence. Notice that the last four rules in the figure re-state the int and ext relations above.

Well-Formed Global Types

This section discusses how to maintain the intended meaning of global types after projection, identifying three key cases where things can go wrong.

Preserving the semantics of sequencing. The type $\mathsf{a} \xrightarrow{\mathsf{x}} \mathsf{b}\,;\, \mathsf{c} \xrightarrow{\mathsf{y}} \mathsf{d}$ can be projected onto the participants a, b, c and d to give the local types $\mathsf{b!x}$, $\mathsf{a?x}$, $\mathsf{d!y}$ and $\mathsf{c?y}$ respectively. However, composing these types does not maintain the semantics of the ; operator, i.e., c is

[1]Using the rules in Figure 5.12 as the basis of a type system, one can envision incorporating the checks mentioned to ensure the correct functioning of rules LANG-INT-CHOICE and LANG-EXT-CHOICE (page 83) for programs. We leave this as future work.

Figure 5.15: The relation wf of well-formed local types.

$$\tau \in \text{wf} \qquad a!m \in \text{wf} \qquad a?m \in \text{wf}$$

$$\frac{\text{wf}(L_1) \quad \text{wf}(L_2) \quad \big(\text{int}(L_1 \oplus L_2) \vee \text{ext}(L_1 \oplus L_2)\big)}{\text{wf}(L_1 \oplus L_2)}$$

$$\frac{\text{wf}(L_1) \;\ldots\; \text{wf}(L_\kappa) \quad \mathtt{op} \in \{;\,, \|\}}{\text{wf}(L_1\,\mathtt{op} \ldots \mathtt{op}\, L_\kappa)}$$

$$\frac{\mathtt{OP} = \text{parameterized}(\mathtt{op}) \quad \mathtt{op} \in \{;\,, \oplus, \otimes, \|\} \quad \text{wf}(L[1/i]\ \mathtt{op}\ L[2/i])}{\text{wf}\Big(\mathop{\mathtt{OP}}\limits_{i=1}^{n} L\Big)}$$

$$\frac{\text{first}(L_1) = \{a_1?m_1\} \quad \text{first}(L_2) = \{a_2?m_2\} \quad a_1 \neq a_2}{\text{int}(L_1 \oplus L_2)}$$

$$\frac{\text{first}(L_1) = \{a_1?m_1\} \quad \text{first}(L_2) = \{a_2?m_2\} \quad m_1 \neq m_2}{\text{int}(L_1 \oplus L_2)}$$

$$\frac{\text{first}(L_1) = \{a_1!m_1\} \quad \text{first}(L_2) = \{a_2!m_2\} \quad a_1 \neq a_2}{\text{ext}(L_1 \oplus L_2)}$$

$$\frac{\text{first}(L_1) = \{a_1!m_1\} \quad \text{first}(L_2) = \{a_2!m_2\} \quad m_1 \neq m_2}{\text{ext}(L_1 \oplus L_2)}$$

Figure 5.16: Well-Formedness sub-relations.

$$\begin{aligned}
\text{sp}(G_1\,;G_2) \quad &\overset{\text{def}}{=} \quad (X,Y) \in \big(\text{last}(G_1),\, \text{first}(G_2)\big) \implies \exists\, a,b,c,m_1,m_2. \big(X = \text{“}a \xrightarrow{m_1} b\text{”} \;\wedge\; Y = \text{“}b \xrightarrow{m_2} c\text{”}\big) \\
\text{cp}(G_1 \oplus G_2) \quad &\overset{\text{def}}{=} \quad \forall p.\, \text{wf}\big((G_1 \oplus G_2) \rhd p\big) \;\wedge\; \exists p\, \forall p' \neq p.\, \big(\text{int}(G \rhd p') \;\wedge\; \text{ext}(G \rhd p)\big) \\
\text{pp}\big((G_{11} \oplus G_{12}) \| G_2\big) \quad &\overset{\text{def}}{=} \quad \big(\, \text{first}(G_{11} \rhd p) = \{a_1?m_1\} \;\wedge\; \text{first}(G_{12} \rhd p) = \{a_2?m_2\} \\
&\qquad\quad \implies \text{“}p!m_1\text{”} \notin (G_2 \rhd a_1) \;\wedge\; \text{“}p!m_2\text{”} \notin (G_2 \rhd a_2)\,\big) \\
&\qquad \vee \\
&\qquad \big(\, \text{first}(G_{11} \rhd p) = \{a_1?m_1\} \;\wedge\; G_{12} \rhd p = \bot \\
&\qquad\quad \implies \text{“}p!m_1\text{”} \notin (G_2 \rhd a_1)\,\big)
\end{aligned}$$

required to send y to d after b has received x, but c has no way of knowing when this has happened. Contrast this with the type $\mathsf{a} \xrightarrow{\mathsf{x}} \mathsf{b}\,;\mathsf{b} \xrightarrow{\mathsf{y}} \mathsf{c}$, whose projection into $\mathsf{b!x}$, $\mathsf{a?x}\,;\mathsf{c!y}$ and $\mathsf{b?y}$ (respectively for a, b and c) does indeed sequence the events of receiving x and sending y. The criterion thus can be stated as: $G_1\,; G_2$ maintains the semantics of sequencing if every action ending G_1 can be sequenced with every action starting G_2. More formally, we require that $\mathrm{sp}(G_1\,; G_2)$, defined in Figure 5.16.

Preserving the semantics of choice. We require that the choice of branch in $G_1 \oplus G_2$ is guided by a single actor. This means that projecting onto the participants should give well-formed local types, and that of these types, exactly one satisfies the relation ext (Figure 5.15), and the rest all satisfy the relation int. Observe that the requirement boils down to all actors implementing internal choice, except for one who acts as the guiding oracle, implementing external choice. For a global type $G \equiv G_1 \oplus G_2$, the requirement is captured by the relation $\mathrm{cp}(G)$, defined in Figure 5.16.

Preserving the semantics of choice in presence of concurrency. For the concurrent composition $G_1 \| G_2$ to work as expected, messaging in G_2 should not affect choices in G_1. That is, messages that guide the selection of branches in G_1 should not overlap with messages sent in G_2. This is captured by the relation pp, formally defined in Figure 5.16.

Putting it All Together

The well-formedness conditions outlined above are formally captured by the relation wf on global types, defined in Figure 5.17. The relations sp, cp, pp and wf hold up to structural congruence.

The purpose of the well-formedness conditions is to ensure that the semantics of global types is maintained after projection. A well-formed global type G produces the same traces as the multiset of local types resulting from the projections of G onto the participating actors. The Conjecture that follows captures this property; it can be proven by induction on the structure of well-formed global types, using subject reduction (Theorem 5.4) and session fidelity (Theorem 5.5).

Conjecture 5.1. It is $\mathrm{tr}(G) = \mathrm{tr}(\{G \rhd p \,|\, p \in \mathrm{actors}(G)\})$ for all well-typed global types G.

It is easy to see that the language semantics mirrors that of local types. However, local type branches, as captured by operator $\oplus$, are non-deterministic at the points of external choice and Kleene Star (rules LS-Ext-Choice and LS-KleeneStar on page 74). This is not the case for programs, as **case** statements and **while** loops execute according to the values of the supplied expressions (rules Lang-Ext-Choice and Lang-While on page 83).

Figure 5.17: The relation wf of well-formed global types.

$$a \xrightarrow{m} b \;\in \mathrm{wf} \qquad\qquad \tau \;\in \mathrm{wf}$$

$$\frac{\mathrm{wf}(G_1) \quad \mathrm{wf}(G_2) \quad \mathrm{sp}(G_1\,;G_2)}{\mathrm{wf}(G_1\,;G_2)} \qquad \frac{\mathrm{wf}(G_1) \quad \mathrm{wf}(G_2) \quad \mathrm{cp}(G_1 \oplus G_2)}{\mathrm{wf}(G_1 \oplus G_2)}$$

$$\frac{\mathrm{wf}(G_{11}) \quad \mathrm{wf}(G_{12}) \quad \mathrm{wf}(G_2) \quad \mathrm{pp}((G_{11} \oplus G_{12}) \parallel G_2)}{\mathrm{wf}((G_{11} \oplus G_{12}) \parallel G_2)}$$

$$\frac{\mathtt{OP} = \mathrm{parametric}(\mathtt{op}) \quad \mathtt{op} \in \{;, \oplus, \otimes\} \quad \mathrm{wf}(G[1/i]\;\mathtt{op}\;G[2/i])}{\mathrm{wf}\left(\overset{n}{\underset{i=1}{\mathtt{OP}}} G\right)}$$

Thus, assuming that a program $\mathtt{P}$ implements a set of local types Λ, i.e., rule Inf-Program (page 86) gives $\vdash \mathtt{P} : \Lambda$, we have $\mathrm{tr}(\mathtt{P}) \subseteq \mathrm{tr}(\Lambda)$. This implies that the correctness criteria for global types are directly applicable to programs that implement them.

5.6.2 Towards More Expressive Types

In its present form, this system does not allow the arbitrary use of parameters. First of all, only indices that range over continuous integers are supported. Secondly, some all-to-all types of communication are not supported, due to the issues discussed in Section 5.3.2. These limitations can be overcome by allowing index sets, and making restrictions on them explicit. This would, for example, enable types with constraints, such as

$$\underset{i \in I}{\parallel} \; \underset{j \in J}{\parallel} a_i \xrightarrow{m} a_j \quad I \cap J = \emptyset.$$

One can then reason about how these restrictions affect projection, which will produce different outputs, depending on the provided side-conditions; such as, "project onto a_ψ where $\psi \in I$".

Future work considerations include support for exception handling (in the sense of Carbone et al. [28]), which seems possible with the addition of a special construct to capture the exception handling code. It may also be possible to transfer the more precise realizability results [11] for choreographies [114] to our parameterized specifications, a topic we touch upon in Section 5.6.1. Finally, other possible extensions concern the runtime monitoring application domain [39]. In particular, adding support for global assertions [15] can form the basis of a powerful theory for deriving local restrictions for each participant, which an asynchronous observer [38] can then enforce.

5.7 Discussion

The purpose of this section is to offer insights on the limitations of the actor typing presented here, and how those relate to the shortcomings of other systems in the literature. For example, the session types in this chapter do not cover issues of actor creation—however, much of the already existing work on session types [14, 25, 126] deals with process creation naturally, in the context of extensions for the π-calculus. We argue that systems following the paradigm of Honda et al. [70] disassociate process creation from session creation, which makes the former possible to capture in the resulting calculi. As argued later in Section 5.7.2, this same disassociation does not seem possible, or at least natural, in actor systems.

Much like actor creation, it is difficult to support session delegation in session types for an actor system. Even though session delegation is supported by much of the related work in this domain, that seems to come primarily from the fact that those systems are extensions of the π-calculus. Actor systems are unique in that sessions, as captured by global types, are not associated with channels. In the π-calculus, channels are syntactic entities over which values—including channel names—are communicated. Hence, global types for the π-calculus can naturally capture the sending of a protocol, i.e., one associated with a channel, over another channel. Consider a group of π-calculus processes in a session over some channel c. If one more process needs to partake in the session, it merely needs to become aware of c. The respective behavior in actor development would be to send a message with the names of the participants over to the new actor, which is not a useful programming pattern.

We begin the section with an analysis of the concept of session delegation, which develops the ideas necessary to argue the above positions. In short, the discussed shortcomings with regard to session delegation and actor creation are not unique to this system; rather, they reflect fundamental limitations of the original proposals on session types for π-derived calculi. The interested reader might find the work of Masini and Francalanza [98] to complement the discussion in this section.

5.7.1 Session Delegation

In the context of session type systems for the π-calculus, *session delegation* refers to the communication of channels as values. The term is used to capture the fact that the communication protocol associated with the channel is itself *delegated* from the sending

Figure 5.18: The global types of Honda et al. [70].

Global	G	$::=$	$p \longrightarrow\!\!\!\!\rightarrow p' : k\langle U \rangle$	values
		$\mid$	$p \longrightarrow\!\!\!\!\rightarrow p' : k\{l_j : G_j\}_{j \in J}$	branching
		$\mid$	G, G'	parallel
		$\mid$	$\mu t.G$	recursive
		$\mid$	t	variable
		$\mid$	$\mathtt{end}$	end
Value	U	$::=$	$\overline{S} \mid T@p$	
Sort	S	$::=$	$\mathtt{bool} \mid \mathtt{nat} \mid \ldots \mid \langle G \rangle$	

process to the receiving one. Much of the related work in this area derives from Honda et al.'s original papers on multiparty asynchronous session types [69, 70], and they all allow global types to include constructs of the form $p_1 \longrightarrow\!\!\!\!\rightarrow p_2 : k\langle s \rangle$. The latter captures the event of a process p_1 sending a protocol (a.k.a. session type) s to another process p_2 via channel k. The full grammar for Honda et al.'s global types is given in Figure 5.18, where we use $\longrightarrow\!\!\!\!\rightarrow$ for communication; not to be confused with $\longrightarrow$, which is used for reductions throughout this thesis.

In their grammar, the sending of a value U from p to p' via the channel k is captured by the construct $p \longrightarrow\!\!\!\!\rightarrow p' : k\langle U \rangle$. The sending of labels l_j from p to p', each selecting a different interaction G_j, is denoted with $p \longrightarrow\!\!\!\!\rightarrow p' : k\{l_j : G_j\}_{j \in J}$. Recursion is achieved by operator μ, binding t to the same type in the construct $\mu t.G$. As usual, it is $\mu t.G \equiv G[\mu t.G/t]$. Values U include vectors of sorts $\overline{S}$, and local types $T@p$. In Honda et al.'s work, the latter construct means a local type T taken by process p. $T@p$ is allowed as the type of a communicable value in global types so as to capture the sending of process names. Notice that the grammar of sorts S includes global types: sending $\langle G \rangle$ over a channel k corresponds to the sending of another channel, k', which has been assigned the (global) type G.

Honda et al.'s syntax for global types is shared by many well-established theories; prime examples include the initial work of Deniélou et al. [51] on parameterized session types, and the role extensions by Deniélou and Yoshida [49]. Most importantly, this syntax is shared unchanged in its crucial parts by all works deriving from these theories.

In the aforementioned calculi, one can write $p!k_1\langle k_2 \rangle$ to mean the sending of channel k_2 over channel k_1 to process p. This way, the protocol associated with k_2 is "delegated" to the recipient p. In this manner, the aforementioned works treat sessions (or at least their restrictions onto individual channels) as first-class citizens. However, actors do not communicate via the use of channels. To simulate this concept in an extension of the work presented in this chapter, one would have to communicate the names of the participants in a sub-interaction. Doing so would establish that the related sub-protocol now takes place with the message recipient as a new end point. That is not a common pattern in actor programming.

To capture the concept of session delegation in a seamless manner, similar to that of session types for the π-calculus, we need to equip our global types with sub-protocols. Since these would involve a subset of the actors in the system, it is not clear how one would delegate the session to some recipient without sending (and even knowing) all of the actor names involved. A possible solution would be to consider some actors as "representatives" of the involved groups, say one per group, and treat the communication of these names as delegating the represented session.

It is worth noting that the benefits of such extensions are not clear at this point. One would have to determine whether common actor programming patterns call for such treatment—as well as evaluate whether programmers would be willing to deal with the complexities of the methodology.

5.7.2 Actor Creation

Most session-based extensions to the π-calculus support dynamic process creation, with the notable exception of the work of Deniélou et al. [51], and all works deriving from it. In those works, the construct to create n actors has the form $(R\ P_0\ \lambda i.P)\ n$ and creates processes $P_0, P[1/i], P[2/i], \ldots, P[n-1/i]$. The syntax for P does not allow process creation in the sense of the standard π-calculus or actors; for instance, the construct $!P$ (which replicates process P) is missing. Observe that the way the parameter n binds the number of created processes is similar to our bracket notation in Figure 5.8: the construct **actor** a[n] {L} creates n actors executing L. In this regard, the system of Chapter 5 is similar to the work of Deniélou et al.: dynamic process creation is not supported outside the scope of the parameterization described here.

While the extensive use of parameters in the type system complicates matters, it is not the only obstacle in the way of incorporating dynamic actor creation in the system of this chapter. As analyzed in Section 5.7.1, sessions are not first-class entities in our system; rather, our types describe communication protocols that are not tied to initiation and termination instructions on a specific channel.

In contrast, calculi deriving from the work of Takeuchi et al. [126] support statements of the form request $a(c)$ in S_1 and accept $a(c)$ in S_1. The concurrent composition of such code implements two processes that enter a session which uses channel c for communication; this name is bound in both $\mathtt{S}_1$ and $\mathtt{S}_2$, which implement the interactions prescribed by the global type of a. The name a identifies the session, and is assumed to be one of a list of unique, shared names used for this purpose. However, these names are statically known; that is, the related calculi do not contain recursive constructs that would allow the dynamic introduction of session identifiers.

Session identifiers such as a above are each statically assigned a global type. Furthermore, the global type syntax of Figure 5.18 does not support the dynamic creation of global types in the syntax. To clarify, the creation of a session (global) type is not a describable behavior in the global types of Honda et al. [69], which is an extension of the original system of Takeuchi et al. [126] from which much of the related work derives. This observation is important, because actor creation can be seen as the creation of a session—the protocol in which the new actors participate. As such, we make the argument that in terms of dynamic actor creation, the theory of this chapter suffers from the same drawback as session type systems based on the π-calculus: even though many of those works support dynamic process creation, the equivalent notion in an actor-based system is not dynamic actor creation; rather, it is dynamic session creation.

It can be argued that the work of Honda et al. (and all derivatives) captures dynamic session creation via the creation of channels. However, (a) their global types do not express this as an action, and (b) channels do not have a behavior per se; i.e., a channel does not itself have the capability to create other channels, whereas actors do. The first point hints at the difficulty of expressing actor creation in our global types. The second point, however, seems inadequate at first: one can argue that a channel can be used to send a message that causes a recipient to introduce another process dynamically. Nevertheless, this new process cannot join any already running session—at least in the works cited in this thesis. Even systems that use parameterized session types[2] fix the number of participants to some parameter n, the value of which cannot change at runtime.

The problem seems to lie in the way processes are addressed within sessions, i.e., the use of the request/accept constructs discussed in Section 2.2.2. With these constructs in play, processes are typechecked according to the behavior corresponding to the index they pick upon joining a session. This index acts as a name that is used to address processes in communication, and it cannot change dynamically. Hence, systems deriving from the work of Takeuchi et al. [126] disallow processes from dynamically joining and leaving sessions; they also disallow dynamic changes in a participant's index within a session. Alas, allowing participants to dynamically switch the roles they are typechecked against seems to require runtime support.

[2]all based on the work of Yoshida et al. [138]

To understand the nature of the problem, it is useful to look at the work of Deniélou and Yoshida [49]. Although being presented as an extension of the π-calculus, their system assigns both a unique identity, and a *role* to each process, giving their calculus an actor flavor. An example from the original paper [49] is that of a map-reduce program [47], where the two roles (local types) can be described as such:

$$L_{\text{client}} = \mu t.?\langle \text{server}, \text{Map}\rangle \,;\, !\langle \text{server}, \text{Reduce}\rangle \,;\, t$$
$$L_{\text{server}} = \mu t.\forall x : \text{client}.\{!\langle x, \text{Map}\rangle \,;\, ?\langle x, \text{Reduce}\rangle\} \,;\, t$$

In the above, client processes receive Map messages from the server, to which they reply with a Reduce. Server processes do the converse, but with all client processes x—hence the universal quantifier. Both roles repeat their behavior indefinitely, expressed through the use of the μt construct. Processes (i.e., implementations of roles such as the above) follow a syntax that mirrors that of roles.

The work of Deniélou and Yoshida allows dynamic process creation by supporting both a recursive construct $\mu X.P$ and concurrent composition $P|P$ in their calculus syntax. Furthermore, it allows processes to join and leave roles dynamically. However, to implement the polling mechanism introduced with the $\forall$ operator, the runtime keeps track of the role of each process—a trade-off we chose not to take in this chapter. Furthermore, their mechanism does not allow the expression of patterns as complex as our work, or the work of Deniélou et al. on parameterized session types [51] discussed before. For example, the butterfly network topology, used for fast Fourier transforms, is expressible through certain operations on process indices; simple assignment of roles to processes as in the map-reduce example above is not enough.

Concluding, it seems that supporting dynamic process creation in session types becomes harder as one increases the accuracy with which protocols are expressed in the type language. Newly created actors/processes are to participate in protocols that session types aspire to capture, and making this connection requires an extra step that is not obvious. The work of Deniélou and Yoshida [49] on roles solves the problem (at a cost to expressiveness and runtime overhead) via a type system that knows how every dynamically introduced process behaves with regard to every other dynamically introduced process a priori. This seems difficult to achieve in works such as the one in this chapter, or that of Deniélou et al. [51], due to the heavy use of dependent local and global types.

Chapter 6

Conclusions and Open Problems

Statically guaranteeing interesting properties in concurrent programs is a difficult problem that admits solutions inspired from many different viewpoints. This thesis has presented work deriving from two such angles: tracking typestates, and session types.

The two approaches differ in significant ways, not the least of which being the expressiveness of the resulting calculi. It is evident that an approach based on typestates can be applied to an actor calculus that is very close to the original proposals of Hewitt [65] and Agha et al. [3]. This is because each name in the program can be associated with a typestate, on a per-scope basis—similar to the way types are associated with basic data elements in languages such as C. On the other hand, an approach based on session types offers more precise control over the communication protocol, but seems to require unconventional language constructs. Such deviations from model calculi have been proposed for both the π-calculus, and actors—in exchange for an increase in the precision with which protocols are typechecked.

Particularly for the case of actors, a session type applies to an implied entity, i.e., the communication itself. This is in contrast to π- derived calculi, where session types are associated with communication channels. Consequently, the system of Chapter 5 assumes that each program has a fixed set of actors, in order to enable typechecking against a protocol where the names of the participants are mentioned explicitly. In fact, the actor names mentioned in the global type have to match those in the respective implementation. This clearly breaks basic code design principles, such as the separation of a component's interface from its implementation.

It seems possible to overcome this limitation by attaching session (global) types to specific names in the program, which would act as session identifiers. This could allow, for example, referring to global types as constructs of the form $G(\overline{a}, \overline{n})$ parameterized in actor names $\overline{a}$ and arithmetic parameters $\overline{n}$. Invoking such a construct with arguments $\overline{\alpha}$ and $\overline{\mathsf{n}}$ would instantiate the session with actors $\overline{\alpha}$ and numeric arguments $\overline{\mathsf{n}}$. The latter would be as in Chapter 5, the work of Deniélou et al. [51], et cetera. Such extensions can allow the system to capture the dynamic creation of actors, whose exact role in the protocol may be determined by their position in the argument list passed to G.

This approach would closely resemble the session advertising and joining constructs of Honda et al. [70]. These are meant to solve the related problem of matching participants to their types in a session-typed extension of the π-calculus, as discussed in Section 2.2.2. Much of the related work in session types for the π-calculus uses constructs which advertise a session on a name, say s, which participants can join by declaring their role in the session—usually by specifying their index explicitly. This is necessary in order to enable the typechecking of participants against local types projected from the (global) type of s. Communication in s takes place over a set of channels $\overline{c}$ that are also advertised, and that conceptually belong to the same session.

However, the resulting calculi allow sending operations to specify the recipient, along with the channel. For instance, $c!\langle 1, e\rangle$ will send the value of the expression e to the participant with index 1, over channel c. Our position is that this is a major deviation from the way the π-calculus was intended to express communication. The original Milner et al. papers [103, 104] discuss the π-calculus and its merits with examples where each channel "connects" exactly two processes. This is in line with our intuition for how channels connect distributed communicating processes, and their implementation in practice. A good example would be the ever-present Transmission Control Protocol (TCP) [32, 33].

On the other hand, the typestate-based approach of Chapter 4 is less invasive. Types are naturally attached to process names, without additional constructs to aid in the matching of a protocol to its participants. Programmers tend to be very reluctant to the adoption of new ideas, especially those that pose overhead on already established practices. This makes our typestate approach especially attractive for implementation in a language such as Erlang, or an actor framework such as Akka—both widely used.

This is not to suggest that typestates are panacea for the world of statically enforced properties in concurrent programming; or that the properties captured by the typestates in this thesis are the most useful ones. It is a topic of future research to come up with a set of properties that combine to offer both accurate, and succinct control over a program's communication structure. The problem is both interesting and not trivial, since one has to consider the trade-offs regarding programmability, the nature of the desired properties, the extend to which these can be decided statically, and at what performance penalty.

Thank you for taking interest in my research.

Bibliography

[1] Gul A. Agha. 1990. *ACTORS – a model of concurrent computation in distributed systems. MIT Press series in artificial intelligence*. MIT Press. ISBN: 978-0-262-01092-4.

[2] Gul Agha and Prasanna Thati. 2004. *An algebraic theory of actors and its application to a simple object-based language.* In *From Object-Orientation to Formal Methods, Essays in Memory of Ole-Johan Dahl* (Lecture Notes in Computer Science). Olaf Owe, Stein Krogdahl, and Tom Lyche, (Eds.) Volume 2635. Springer, 26–57. ISBN: 3-540-21366-X. DOI: `10.1007/978-3-540-39993-3_4`.

[3] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. 1997. *A foundation for actor computation. J. Funct. Program.*, 7, 1, 1–72. DOI: `10.1017/S095679689700261X`.

[4] Gul Agha, Christopher R. Houck, and Rajendra Panwar. 1991. *Distributed execution of actor programs.* In *Languages and Compilers for Parallel Computing, Fourth International Workshop, Santa Clara, California, USA, August 7-9, 1991, Proceedings* (Lecture Notes in Computer Science). Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David A. Padua, (Eds.) Volume 589. Springer, 1–17. ISBN: 3-540-55422-X. DOI: `10.1007/BFb0038654`.

[5] Alfred V. Aho and Jeffrey D. Ullman. 1977. *Principles of Compiler Design.* Addison Wesley. ISBN: 978-0201100730.

[6] Bowen Alpern and Fred B. Schneider. 1985. *Defining liveness. Inf. Process. Lett.*, 21, 4, 181–185. DOI: `10.1016/0020-0190(85)90056-0`.

[7] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. 2005. *Realizability and verification of MSC graphs. Theor. Comput. Sci.*, 331, 1, 97–114. DOI: `10.1016/j.tcs.2004.09.034`.

[8] Sandra Alves, Maribel Fernández, Mário Florido, and Ian Mackie. 2010. *Gödel's system tau revisited. Theor. Comput. Sci.*, 411, 11-13, 1484–1500. DOI: `10.1016/j.tcs.2009.11.014`.

[9] Stephanie Balzer and Frank Pfenning. 2015. *Objects as session-typed processes.* In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015, Pittsburgh, PA, USA, October 26, 2015.* Elisa Gonzalez Boix, Philipp Haller, Alessandro Ricci, and Carlos Varela, (Eds.) ACM, 13–24. ISBN: 978-1-4503-3901-8. DOI: `10.1145/2824815.2824817`.

[10] Hendrik Pieter Barendregt, Wil Dekkers, and Richard Statman. 2013. *Lambda Calculus with Types. Perspectives in logic.* Cambridge University Press. ISBN: 978-0-521-76614-2.

[11] Samik Basu, Tevfik Bultan, and Meriem Ouederni. 2012. *Deciding choreography realizability.* In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012.* John Field and Michael Hicks, (Eds.) ACM, 191–202. ISBN: 978-1-4503-1083-3. DOI: 10.1145/2103656.2103680.

[12] Andi Bejleri. 2012. *Parameterised session types communication patterns: through the looking glass of session types.* Ph.D. Dissertation. Imperial College London, UK.

[13] Andi Bejleri. 2010. *Practical parameterised session types.* In *Formal Methods and Software Engineering - 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, 2010. Proceedings* (Lecture Notes in Computer Science). Jin Song Dong and Huibiao Zhu, (Eds.) Volume 6447. Springer, 270–286. ISBN: 978-3-642-16900-7. DOI: 10.1007/978-3-642-16901-4_19.

[14] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. 2008. *Global progress in dynamically interleaved multiparty sessions.* In *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings* (Lecture Notes in Computer Science). Franck van Breugel and Marsha Chechik, (Eds.) Volume 5201. Springer, 418–433. ISBN: 978-3-540-85360-2. DOI: 10.1007/978-3-540-85361-9_33.

[15] Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. 2010. *A theory of design-by-contract for distributed multiparty interactions.* In *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings* (Lecture Notes in Computer Science). Paul Gastin and François Laroussinie, (Eds.) Volume 6269. Springer, 162–176. ISBN: 978-3-642-15374-7. DOI: 10.1007/978-3-642-15375-4_12.

[16] Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. 2017. *Monitoring networks through multiparty session types. Theor. Comput. Sci.*, 669, 33–58. DOI: 10.1016/j.tcs.2017.02.009.

[17] Eduardo Bonelli and Adriana B. Compagnoni. 2007. *Multipoint session types for a distributed calculus.* In *Trustworthy Global Computing, Third Symposium, TGC 2007, Sophia-Antipolis, France, November 5-6, 2007, Revised Selected Papers* (Lecture Notes

in Computer Science). Gilles Barthe and Cédric Fournet, (Eds.) Volume 4912. Springer, 240–256. ISBN: 978-3-540-78662-7. DOI: 10.1007/978-3-540-78663-4_17.

[18] Eduardo Bonelli, Adriana B. Compagnoni, and Elsa L. Gunter. 2005. *Correspondence assertions for process synchronization in concurrent communications. Journal of Functional Programming*, 15, 2, 219–247. DOI: 10.1017/S095679680400543X.

[19] Eduardo Bonelli, Adriana B. Compagnoni, and Elsa L. Gunter. 2005. *Typechecking safe process synchronization. Electr. Notes Theor. Comput. Sci.*, 138, 1, 3–22. DOI: 10.1016/j.entcs.2005.05.002.

[20] Gérard Boudol. 1992. *Asynchrony and the Pi-calculus*. Research Report. https://hal.inria.fr/inria-00076939

[21] Daniel Brand and Pitro Zafiropulo. 1983. *On communicating finite-state machines. J. ACM*, 30, 2, 323–342. DOI: 10.1145/322374.322380.

[22] Roberto Bruni and Jürgen Dingel, (Eds.) *Formal Techniques for Distributed Systems - Joint 13th IFIP WG 6.1 International Conference, FMOODS 2011, and 31st IFIP WG 6.1 International Conference, FORTE 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings*, volume 6722 of *Lecture Notes in Computer Science*, (2011). Springer. ISBN: 978-3-642-21460-8. DOI: 10.1007/978-3-642-21461-5.

[23] Luís Caires and Frank Pfenning. 2010. *Session types as intuitionistic linear propositions.* In *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings* (Lecture Notes in Computer Science). Paul Gastin and François Laroussinie, (Eds.) Volume 6269. Springer, 222–236. ISBN: 978-3-642-15374-7. DOI: 10.1007/978-3-642-15375-4_16.

[24] Marco Carbone and Fabrizio Montesi. 2013. *Deadlock-freedom-by-design: multiparty asynchronous global programming.* In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. Roberto Giacobazzi and Radhia Cousot, (Eds.) ACM, 263–274. ISBN: 978-1-4503-1832-7. DOI: 10.1145/2429069.2429101.

[25] Marco Carbone, Nobuko Yoshida, and Kohei Honda. 2009. *Asynchronous session types: exceptions and multiparty interactions.* In *Formal Methods for Web Services, 9th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2009, Bertinoro, Italy, June 1-6, 2009, Advanced Lectures* (Lecture Notes in Computer Science). Marco Bernardo, Luca Padovani, and Gianluigi Zavattaro, (Eds.) Volume 5569. Springer, 187–212. ISBN: 978-3-642-01917-3. DOI: 10.1007/978-3-642-01918-0_5.

[26] Marco Carbone, Ornela Dardha, and Fabrizio Montesi. 2014. *Progress as compositional lock-freedom*. In *Coordination Models and Languages - 16th IFIP WG 6.1 International Conference, COORDINATION 2014, Held as Part of the 9th International Federated Conferences on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings* (Lecture Notes in Computer Science). eva Kühn and Rosario Pugliese, (Eds.) Volume 8459. Springer, 49–64. ISBN: 978-3-662-43375-1. DOI: `10.1007/978-3-662-43376-8_4`.

[27] Marco Carbone, Kohei Honda, and Nobuko Yoshida. 2007. *Structured communication-centred programming for web services*. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings* (Lecture Notes in Computer Science). Rocco De Nicola, (Ed.) Volume 4421. Springer, 2–17. ISBN: 978-3-540-71314-2. DOI: `10.1007/978-3-540-71316-6_2`.

[28] Marco Carbone, Kohei Honda, and Nobuko Yoshida. 2008. *Structured interactional exceptions in session types*. In *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings* (Lecture Notes in Computer Science). Franck van Breugel and Marsha Chechik, (Eds.) Volume 5201. Springer, 402–417. ISBN: 978-3-540-85360-2. DOI: `10.1007/978-3-540-85361-9_32`.

[29] Luca Cardelli and Peter Wegner. 1985. *On understanding types, data abstraction, and polymorphism. ACM Comput. Surv.*, 17, 4, 471–522. DOI: `10.1145/6041.6042`.

[30] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. 2012. *On global types and multi-party session. Logical Methods in Computer Science*, 8, 1. DOI: `10.2168/LMCS-8(1:24)2012`.

[31] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. 2011. *On global types and multi-party sessions*. In *Formal Techniques for Distributed Systems - Joint 13th IFIP WG 6.1 International Conference, FMOODS 2011, and 31st IFIP WG 6.1 International Conference, FORTE 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings* (Lecture Notes in Computer Science). Roberto Bruni and Jürgen Dingel, (Eds.) Volume 6722. Springer, 1–28. ISBN: 978-3-642-21460-8. DOI: `10.1007/978-3-642-21461-5_1`.

[32] V. Cerf and R. Kahn. 1974. *A protocol for packet network intercommunication. IEEE Transactions on Communications*, 22, 5, 637–648. ISSN: 0090-6778. DOI: `10.1109/TCOM.1974.1092259`.

[33] Vinton G. Cerf and Robert E. Kahn. 2005. *A protocol for packet network intercommunication. Computer Communication Review*, 35, 2, 71–82. DOI: 10.1145/1064413.1064423.

[34] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. *Bigtable: A distributed storage system for structured data. ACM Trans. Comput. Syst.*, 26, 2, 4:1–4:26. DOI: 10.1145/1365815.1365816.

[35] Minas Charalambides, Peter Dinges, and Gul Agha. 2012. *Parameterized concurrent multi-party session types.* In *Proceedings 11th International Workshop on Foundations of Coordination Languages and Self Adaptation, FOCLASA 2012, Newcastle, U.K., September 8, 2012.* (EPTCS). Natallia Kokash and António Ravara, (Eds.) Volume 91, 16–30. DOI: 10.4204/EPTCS.91.2.

[36] Minas Charalambides, Peter Dinges, and Gul A. Agha. 2016. *Parameterized, concurrent session types for asynchronous multi-actor interactions. Science of Computer Programming*, 115-116, 100–126. DOI: 10.1016/j.scico.2015.10.006.

[37] Minas Charalambides, Karl Palmskog, and Gul Agha. 2017. *Types for progress in actor programs.* In *Proceedings of the Workshop on Actors and Active Objects, IFM Satellite Event, Torino, Italy, September 18, 2017* number 2.

[38] Tzu-Chun Chen and Kohei Honda. 2012. *Specifying stateful asynchronous properties for distributed programs.* In *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings* (Lecture Notes in Computer Science). Maciej Koutny and Irek Ulidowski, (Eds.) Volume 7454. Springer, 209–224. ISBN: 978-3-642-32939-5. DOI: 10.1007/978-3-642-32940-1_16.

[39] Tzu-Chun Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, and Nobuko Yoshida. 2011. *Asynchronous distributed monitoring for multiparty session enforcement.* In *Trustworthy Global Computing - 6th International Symposium, TGC 2011, Aachen, Germany, June 9-10, 2011. Revised Selected Papers* (Lecture Notes in Computer Science). Roberto Bruni and Vladimiro Sassone, (Eds.) Volume 7173. Springer, 25–45. ISBN: 978-3-642-30064-6. DOI: 10.1007/978-3-642-30065-3_2.

[40] Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. 2014. *On the preciseness of subtyping in session types.* In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury,*

United Kingdom, September 8-10, 2014. Olaf Chitil, Andy King, and Olivier Danvy, (Eds.) ACM, 135–146. ISBN: 978-1-4503-2947-7. DOI: 10.1145/2643135.2643138.

[41] Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. 2016. *On the preciseness of subtyping in session types*. *CoRR*, abs/1610.00328. arXiv: 1610.00328.

[42] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. *PNUTS: yahoo!'s hosted data serving platform*. *PVLDB*, 1, 2, 1277–1288.

[43] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. 2007. *Asynchronous session types and progress for object oriented languages*. In *Formal Methods for Open Object-Based Distributed Systems, 9th IFIP WG 6.1 International Conference, FMOODS 2007, Paphos, Cyprus, June 6-8, 2007, Proceedings* (Lecture Notes in Computer Science). Marcello M. Bonsangue and Einar Broch Johnsen, (Eds.) Volume 4468. Springer, 1–31. ISBN: 978-3-540-72919-8. DOI: 10.1007/978-3-540-72952-5_1.

[44] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. 1999. *Parallel computer architecture – a hardware/software approach*. Morgan Kaufmann. ISBN: 978-1-55860-343-1.

[45] Robert DeLine and Manuel Fähndrich. 2001. *Enforcing high-level protocols in low-level software*. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*. Michael Burke and Mary Lou Soffa, (Eds.) ACM, 59–69. ISBN: 1-58113-414-2. DOI: 10.1145/378795.378811.

[46] Robert DeLine and Manuel Fähndrich. 2004. *Typestates for objects*. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings* (Lecture Notes in Computer Science). Martin Odersky, (Ed.) Volume 3086. Springer, 465–490. ISBN: 3-540-22159-X. DOI: 10.1007/978-3-540-24851-4_21.

[47] Jeffrey Dean and Sanjay Ghemawat. 2008. *Mapreduce: simplified data processing on large clusters*. *Commun. ACM*, 51, 1, (January 2008), 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492.

[48] Gero Decker, Oliver Kopp, Frank Leymann, and Mathias Weske. 2007. *Bpel4chor: extending BPEL for modeling choreographies*. In *2007 IEEE International Conference on Web Services (ICWS) 2007), July 9-13, 2007, Salt Lake City, Utah, USA*. IEEE Computer Society, 296–303. ISBN: 0-7695-2924-0. DOI: 10.1109/ICWS.2007.59.

[49] Pierre-Malo Deniélou and Nobuko Yoshida. 2011. *Dynamic multirole session types.* In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011.* Thomas Ball and Mooly Sagiv, (Eds.) ACM, 435–446. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926435.

[50] Pierre-Malo Deniélou and Nobuko Yoshida. 2012. *Multiparty session types meet communicating automata.* In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings* (Lecture Notes in Computer Science). Helmut Seidl, (Ed.) Volume 7211. Springer, 194–213. ISBN: 978-3-642-28868-5. DOI: 10.1007/978-3-642-28869-2_10.

[51] Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. 2012. *Parameterised multiparty session types. Logical Methods in Computer Science*, 8, 4. DOI: 10.2168/LMCS-8(4:6)2012.

[52] Mariangiola Dezani-Ciancaglini, Elena Giachino, Sophia Drossopoulou, and Nobuko Yoshida. 2006. *Bounded session types for object oriented languages.* In *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures* (Lecture Notes in Computer Science). Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, (Eds.) Volume 4709. Springer, 207–245. ISBN: 978-3-540-74791-8. DOI: 10.1007/978-3-540-74792-5_10.

[53] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. 2006. *Session types for object-oriented languages.* In *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings* (Lecture Notes in Computer Science). Dave Thomas, (Ed.) Volume 4067. Springer, 328–352. ISBN: 3-540-35726-2. DOI: 10.1007/11785477_20.

[54] *Fourth Gravitational Wave Found, Blue Waters Supercomputer.* https://tinyurl.com/ncsa-fourth-grav-wave

[55] Nissim Francez. 1986. *Fairness. Texts and Monographs in Computer Science.* Springer. ISBN: 978-3-540-96235-9. DOI: 10.1007/978-1-4612-4886-6.

[56] Paul Gastin and François Laroussinie, (Eds.) *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3,*

2010. Proceedings, volume 6269 of *Lecture Notes in Computer Science*, (2010). Springer. ISBN: 978-3-642-15374-7. DOI: `10.1007/978-3-642-15375-4`.

[57] Simon J. Gay and Malcolm Hole. 2005. *Subtyping for session types in the pi calculus. Acta Inf.*, 42, 2-3, 191–225. DOI: `10.1007/s00236-005-0177-z`.

[58] Simon J. Gay and Vasco Thudichum Vasconcelos. 2010. *Linear type theory for asynchronous session types. J. Funct. Program.*, 20, 1, 19–50. DOI: `10.1017/S0956796809990268`.

[59] Simon J. Gay, Vasco Thudichum Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. 2010. *Modular session types for distributed object-oriented programming*. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. Manuel V. Hermenegildo and Jens Palsberg, (Eds.) ACM, 299–312. ISBN: 978-1-60558-479-9. DOI: `10.1145/1706299.1706335`.

[60] Simon J. Gay, Nils Gesbert, and António Ravara. 2014. *Session types as generic process types*. In *Proceedings Combined 21st International Workshop on Expressiveness in Concurrency and 11th Workshop on Structural Operational Semantics, EXPRESS 2014, and 11th Workshop on Structural Operational Semantics, SOS 2014, Rome, Italy, 1st September 2014*. (EPTCS). Johannes Borgström and Silvia Crafa, (Eds.) Volume 160, 94–110. DOI: `10.4204/EPTCS.160.9`.

[61] Simon Gay and Vasco Vasconcelos. 2007. *Asynchronous Functional Session Types*. Technical report. `http://www.dcs.gla.ac.uk/~simon/publications/TR-2007-251.pdf`

[62] Simon Gay, Vasco Vasconcelos, and António Ravara. 2003. *Session Types for Inter-Process Communication*. Technical report. `http://www.dcs.gla.ac.uk/~simon/publications/TR-2003-133.pdf`

[63] Kurt Gödel. 1958. *Über eine bisher noch nicht benützte erweiterung des finiten standpunktes. Dialectica*, 280–287.

[64] Luciano Lavagno, Grant Martin, and Bran Selic, (Eds.) 2003. *Message sequence charts. UML for Real: Design of Embedded Real-Time Systems*. Springer US, Boston, MA, 77–105. ISBN: 978-0-306-48738-5. DOI: `10.1007/0-306-48738-1_4`.

[65] Carl Hewitt. 1977. *Viewing control structures as patterns of passing messages. Artif. Intell.*, 8, 3, 323–364. DOI: `10.1016/0004-3702(77)90033-9`.

[66] C. A. R. Hoare. 1985. *Communicating Sequential Processes*. Prentice-Hall. ISBN: 0-13-153271-5.

[67] Kohei Honda. 1993. *Types for dyadic interaction.* In *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings* (Lecture Notes in Computer Science). Eike Best, (Ed.) Volume 715. Springer, 509–523. ISBN: 3-540-57208-2. DOI: 10.1007/3-540-57208-2_35.

[68] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. *Language primitives and type discipline for structured communication-based programming.* In *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings* (Lecture Notes in Computer Science). Chris Hankin, (Ed.) Volume 1381. Springer, 122–138. ISBN: 3-540-64302-8. DOI: 10.1007/BFb0053567.

[69] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. *Multiparty asynchronous session types. J. ACM*, 63, 1, 9:1–9:67. DOI: 10.1145/2827695.

[70] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. *Multiparty asynchronous session types.* In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008.* George C. Necula and Philip Wadler, (Eds.) ACM, 273–284. ISBN: 978-1-59593-689-9. DOI: 10.1145/1328438.1328472.

[71] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. 2011. *Scribbling interactions with a formal foundation.* In *Distributed Computing and Internet Technology - 7th International Conference, ICDCIT 2011, Bhubaneshwar, India, February 9-12, 2011. Proceedings* (Lecture Notes in Computer Science). Raja Natarajan and Adegboyega K. Ojo, (Eds.) Volume 6536. Springer, 55–75. ISBN: 978-3-642-19055-1. DOI: 10.1007/978-3-642-19056-8_4.

[72] Raymond Hu, Nobuko Yoshida, and Kohei Honda. 2008. *Session-based distributed programming in java.* In *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings* (Lecture Notes in Computer Science). Jan Vitek, (Ed.) Volume 5142. Springer, 516–541. ISBN: 978-3-540-70591-8. DOI: 10.1007/978-3-540-70592-5_22.

[73] Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. 2010. *Type-safe eventful sessions in java.* In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings* (Lecture Notes in Computer Science). Theo D'Hondt, (Ed.) Volume 6183. Springer, 329–353. ISBN: 978-3-642-14106-5. DOI: 10.1007/978-3-642-14107-2_16.

[74] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. 2016. *Foundations of session types and behavioural contracts. ACM Comput. Surv.*, 49, 1, 3:1–3:36. DOI: 10.1145/2873052.

[75] Atsushi Igarashi and Naoki Kobayashi. 2004. *A generic type system for the pi-calculus. Theor. Comput. Sci.*, 311, 1-3, 121–163. DOI: 10.1016/S0304-3975(03)00325-6.

[76] Keigo Imai, Nobuko Yoshida, and Shoji Yuen. 2017. *Session-ocaml: A session-based library with polarities and lenses.* In *Coordination Models and Languages - 19th IFIP WG 6.1 International Conference, COORDINATION 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017, Proceedings* (Lecture Notes in Computer Science). Jean-Marie Jacquet and Mieke Massink, (Eds.) Volume 10319. Springer, 99–118. ISBN: 978-3-319-59745-4. DOI: 10.1007/978-3-319-59746-1_6.

[77] Guha Jayachandran, V. Vishal, and Vijay S. Pande. 2006. *Using massively parallel simulation and markovian models to study protein folding: examining the dynamics of the villin headpiece. The Journal of Chemical Physics.* DOI: 10.1063/1.2186317.

[78] Simon L. Peyton Jones, Andrew D. Gordon, and Sigbjorn Finne. 1996. *Concurrent haskell.* In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996.* Hans-Juergen Boehm and Guy L. Steele Jr., (Eds.) ACM Press, 295–308. ISBN: 0-89791-769-3. DOI: 10.1145/237721.237794.

[79] Naoki Kobayashi. 1998. *A partially deadlock-free typed process calculus. ACM Trans. Program. Lang. Syst.*, 20, 2, 436–482. DOI: 10.1145/276393.278524.

[80] Naoki Kobayashi. 2002. *A type system for lock-free processes. Information and Computation*, 177, 2, 122–159. ISSN: 0890-5401. DOI: 10.1006/inco.2002.3171.

[81] Naoki Kobayashi, Shin Saito, and Eijiro Sumii. 2000. *An implicitly-typed deadlock-free process calculus.* In *CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings* (Lecture Notes in Computer Science). Catuscia Palamidessi, (Ed.) Volume 1877. Springer, 489–503. ISBN: 3-540-67897-2. DOI: 10.1007/3-540-44618-4_35.

[82] Vijay Anand Korthikanti. 2012. *Towards energy-performance trade-off analysis of parallel applications*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign, (February 2012).

[83] Vijay Anand Korthikanti, Gul Agha, and Mark R. Greenstreet. 2011. *On the energy complexity of parallel algorithms*. In *International Conference on Parallel Processing, ICPP 2011, Taipei, Taiwan, September 13-16, 2011*. Guang R. Gao and Yu-Chee Tseng, (Eds.) IEEE Computer Society, 562–570. ISBN: 978-1-4577-1336-1. DOI: 10.1109/ICPP.2011.84.

[84] Dimitrios Kouzapas, Nobuko Yoshida, and Kohei Honda. 2011. *On asynchronous session semantics*. In *Formal Techniques for Distributed Systems - Joint 13th IFIP WG 6.1 International Conference, FMOODS 2011, and 31st IFIP WG 6.1 International Conference, FORTE 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings* (Lecture Notes in Computer Science). Roberto Bruni and Jürgen Dingel, (Eds.) Volume 6722. Springer, 228–243. ISBN: 978-3-642-21460-8. DOI: 10.1007/978-3-642-21461-5_15.

[85] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. *Imagenet classification with deep convolutional neural networks*. *Commun. ACM*, 60, 6, 84–90. DOI: 10.1145/3065386.

[86] Ericsson Computer Science Laboratory. *Erlang*. https://www.erlang.org

[87] Leslie Lamport. 1977. *Proving the correctness of multiprocess programs*. *IEEE Trans. Software Eng.*, 3, 2, 125–143. DOI: 10.1109/TSE.1977.229904.

[88] Ivan Lanese, Claudio Guidi, Fabrizio Montesi, and Gianluigi Zavattaro. 2008. *Bridging the gap between interaction- and process-oriented choreographies*. In *Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa, 10-14 November 2008*. Antonio Cerone and Stefan Gruner, (Eds.) IEEE Computer Society, 323–332. ISBN: 978-0-7695-3437-4. DOI: 10.1109/SEFM.2008.11.

[89] Julien Lange and Emilio Tuosto. 2012. *Synthesising choreographies from local session types (extended version)*. *CoRR*, abs/1204.2566.

[90] Julien Lange, Emilio Tuosto, and Nobuko Yoshida. 2015. *From communicating machines to graphical choreographies*. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. Sriram K. Rajamani and David Walker, (Eds.) ACM, 221–232. ISBN: 978-1-4503-3300-9. DOI: 10.1145/2676726.2676964.

[91] Xavier Leroy, Jérôme Vouillon, Damien Doligez, Didier Rémy, Ascánder Suárez, et al. *OCaml*. https://ocaml.org/

[92] Eugene Letuchy. *Facebook Chat*. https://tinyurl.com/facebook-actors

[93] Lightbend. *Akka*. http://akka.io

[94] *Linear Algebra Package*. http://www.netlib.org/lapack/

[95] Barbara Liskov. 1987. *Keynote address - data abstraction and hierarchy.* In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum)* (OOPSLA '87). ACM, Orlando, Florida, USA, 17–34. ISBN: 0-89791-266-7. DOI: 10.1145/62138.62141.

[96] Barbara Liskov. 1988. *Keynote address - data abstraction and hierarchy. SIGPLAN Not.*, 23, 5, 17–34. ISSN: 0362-1340. DOI: 10.1145/62139.62141.

[97] Joseph Masini and Adrian Francalanza. 2013. *Typing actors using behavioural types.* In *Proceedings of the Second International Workshop on Behavioral Types (BEAT).*

[98] Joseph Masini and Adrian Francalanza. 2015. *Typing Actors using Behavioural Types.* Technical report. https://tinyurl.com/masini2015typing

[99] Satoshi Matsuoka and Akinori Yonezawa. 1993. *Research directions in concurrent object-oriented programming.* In Gul Agha, Peter Wegner, and Akinori Yonezawa, (Eds.) MIT Press, Cambridge, MA, USA. Chapter Analysis of Inheritance Anomaly in Object-oriented Concurrent Programming Languages, 107–150. ISBN: 0-262-01139-5.

[100] *Message Passing Interface.* http://mpi-forum.org/

[101] Robin Milner. 1999. *Communicating and mobile systems – the Pi-calculus.* Cambridge University Press. ISBN: 978-0-521-65869-0.

[102] Robin Milner and Davide Sangiorgi. 1992. *Barbed bisimulation.* In *Automata, Languages and Programming, 19th International Colloquium, ICALP92, Vienna, Austria, July 13-17, 1992, Proceedings* (Lecture Notes in Computer Science). Werner Kuich, (Ed.) Volume 623. Springer, 685–695. ISBN: 3-540-55719-9. DOI: 10.1007/3-540-55719-9_114.

[103] Robin Milner, Joachim Parrow, and David Walker. 1992. *A calculus of mobile processes, I. Inf. Comput.*, 100, 1, 1–40. DOI: 10.1016/0890-5401(92)90008-4.

[104] Robin Milner, Joachim Parrow, and David Walker. 1992. *A calculus of mobile processes, II. Inf. Comput.*, 100, 1, 41–77. DOI: 10.1016/0890-5401(92)90009-5.

[105] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. *Human-level control through deep reinforcement learning*. *Nature*, 518, 7540, 529–533. DOI: `10.1038/nature14236`.

[106] Peter D. Mosses. 2006. *Formal semantics of programming languages: - an overview -*. *Electr. Notes Theor. Comput. Sci.*, 148, 1, 41–73. DOI: `10.1016/j.entcs.2005.12.012`.

[107] Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. 2009. *Global principal typing in partially commutative asynchronous sessions*. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings* (Lecture Notes in Computer Science). Giuseppe Castagna, (Ed.) Volume 5502. Springer, 316–332. ISBN: 978-3-642-00589-3. DOI: `10.1007/978-3-642-00590-9_23`.

[108] Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. 2017. *Timed runtime monitoring for multiparty conversations*. *Formal Aspects of Computing*, 1–34. ISSN: 1433-299X. DOI: `10.1007/s00165-017-0420-8`.

[109] Oscar Nierstrasz. 1993. *Regular types for active objects*. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Eighth Annual Conference, Washington, DC, USA, September 26 - October 1, 1993, Proceedings*. Timlynn Babitsky and Jim Salmons, (Eds.) ACM, 1–15. ISBN: 0-89791-587-9. DOI: `10.1145/165854.167976`.

[110] Oracle. *Java Threads Tutorial*. `http://tinyurl.com/java-thread-tutorial`

[111] Luca Padovani. 2017. *Deadlock-Free Typestate-Oriented Programming*. Technical report. `https://hal.archives-ouvertes.fr/hal-01628801`

[112] Luca Padovani. 2013. *From lock freedom to progress using session types*. In *Proceedings 6th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2013, Rome, Italy, 23rd March 2013.* (EPTCS). Nobuko Yoshida and Wim Vanderbauwhede, (Eds.) Volume 137, 3–19. DOI: `10.4204/EPTCS.137.2`.

[113] *Pascal records*. `http://wiki.freepascal.org/Record`

[114] Chris Peltz. 2003. *Web services orchestration and choreography. IEEE Computer*, 36, 10, 46–52. DOI: 10.1109/MC.2003.1236471.

[115] Benjamin C. Pierce and Davide Sangiorgi. 1993. *Typing and subtyping for mobile processes.* In *Proceedings of the Eighth Annual Symposium on Logic in Computer Science (LICS) '93), Montreal, Canada, June 19-23, 1993.* IEEE Computer Society, 376–385. ISBN: 0-8186-3140-6. DOI: 10.1109/LICS.1993.287570.

[116] Benjamin C. Pierce and Davide Sangiorgi. 1996. *Typing and subtyping for mobile processes. Mathematical Structures in Computer Science*, 6, 5, 409–453.

[117] Robert L. Probert and Kassem Saleh. 1991. *Synthesis of communication protocols: survey and assessment. IEEE Trans. Computers*, 40, 4, 468–476. DOI: 10.1109/12.88466.

[118] Riccardo Pucella and Jesse A. Tov. 2008. *Haskell session types with (almost) no class.* In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008.* Andy Gill, (Ed.) ACM, 25–36. ISBN: 978-1-60558-064-7. DOI: 10.1145/1411286.1411290.

[119] Franz Puntigam. 2000. *Concurrent Object-Oriented Programming with Process Types.* Habilitationsschrift. Der Andere Verlag, Osnabrück, Germany.

[120] Franz Puntigam. 1997. *Coordination requirements expressed in types for active objects.* In *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings* (Lecture Notes in Computer Science). Mehmet Aksit and Satoshi Matsuoka, (Eds.) Volume 1241. Springer, 367–388. ISBN: 3-540-63089-9. DOI: 10.1007/BFb0053387.

[121] Franz Puntigam. 1996. *Types for active objects based on trace semantics.* In *Proceedings FMOODS '96.* Chapman & Hall, 4–19. DOI: 10.1.1.48.9370.

[122] Franz Puntigam and Christof Peter. 2001. *Types for active objects with static deadlock prevention. Fundam. Inform.*, 48, 4, 315–341.

[123] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. *Mastering the game of go with deep neural networks and tree search. Nature*, 529, 7587, 484–489. DOI: 10.1038/nature16961.

[124] Robert E. Strom and Shaula Yemini. 1986. *Typestate: A programming language concept for enhancing software reliability*. *IEEE Trans. Software Eng.*, 12, 1, 157–171. DOI: 10.1109/TSE.1986.6312929.

[125] Eijiro Sumii and Naoki Kobayashi. 1998. *A generalized deadlock-free process calculus*. *Electr. Notes Theor. Comput. Sci.*, 16, 3, 225–247. DOI: 10.1016/S1571-0661(04)00144-6.

[126] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. 1994. *An interaction-based language and its typing system*. In *PARLE '94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings* (Lecture Notes in Computer Science). Constantine Halatsis, Dimitris G. Maritsas, George Philokyprou, and Sergios Theodoridis, (Eds.) Volume 817. Springer, 398–413. ISBN: 3-540-58184-7. DOI: 10.1007/3-540-58184-7_118.

[127] Gerard Tel. 2000. *Introduction to Distributed Algorithms*. (2nd ed.). Cambridge University Press. DOI: 10.1017/CBO9781139168724.

[128] *The C++ standard reference, struct declarations*. http://en.cppreference.com/w/c/language/struct

[129] *The Scala Programming Language*. https://www.scala-lang.org/

[130] *The WhatsApp Architecture Facebook Bought for $19 Billion*. https://tinyurl.com/whatsapp-actors

[131] Franck van Breugel and Marsha Chechik, (Eds.) *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings*, volume 5201 of *Lecture Notes in Computer Science*, (2008). Springer. ISBN: 978-3-540-85360-2.

[132] Vasco T Vasconcelos, Simon J Gay, António Ravara, Nils Gesbert, and Alexandre Z Caldeira. 2009. *Dynamic interfaces*. In *FOOL 2009 - International Workshop on Foundations of Object-Oriented Languages, Savannah, Georgia, January 24, 2009, Proceedings*. (January 2009).

[133] Vasco Thudichum Vasconcelos, Simon J. Gay, and António Ravara. 2006. *Type checking a multithreaded functional language with session types*. *Theor. Comput. Sci.*, 368, 1-2, 64–87. DOI: 10.1016/j.tcs.2006.06.028.

[134] W3C. *Extensible Markup Language*. https://www.w3.org/XML/

[135] W3C. *The Web Services Choreography Description Language*. http://www.w3.org/TR/ws-cdl-10/

[136] Philip Wadler. 1990. *Linear types can change the world!* In *IFIP TC*. Volume 2, 347–359. DOI: 10.1.1.31.5002.

[137] Nobuko Yoshida and Vasco Thudichum Vasconcelos. 2007. *Language primitives and type discipline for structured communication-based programming revisited: two systems for higher-order session communication. Electr. Notes Theor. Comput. Sci.*, 171, 4, 73–93. DOI: 10.1016/j.entcs.2007.02.056.

[138] Nobuko Yoshida, Pierre-Malo Deniélou, Andi Bejleri, and Raymond Hu. 2010. *Parameterised multiparty session types.* In *Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings* (Lecture Notes in Computer Science). C.-H. Luke Ong, (Ed.) Volume 6014. Springer, 128–145. ISBN: 978-3-642-12031-2. DOI: 10.1007/978-3-642-12032-9_10.

[139] M. C. Yuang. 1988. *Survey of protocol verification techniques based on finite state machine models.* In *[1988] Proceedings. Computer Networking Symposium*, 164–172. DOI: 10.1109/CNS.1988.4993.

[140] Martin Zinkevich, Markus Weimer, Alexander J. Smola, and Lihong Li. 2010. *Parallelized stochastic gradient descent.* In *Advances in Neural Information Processing Systems 23: 24th Annual Conference on Neural Information Processing Systems 2010. Proceedings of a meeting held 6-9 December 2010, Vancouver, British Columbia, Canada.* John D. Lafferty, Christopher K. I. Williams, John Shawe-Taylor, Richard S. Zemel, and Aron Culotta, (Eds.) Curran Associates, Inc., 2595–2603.

[141] Ugo de'Liguoro and Luca Padovani. 2018. *Mailbox types for unordered interactions. CoRR*, abs/1801.04167. arXiv: 1801.04167.