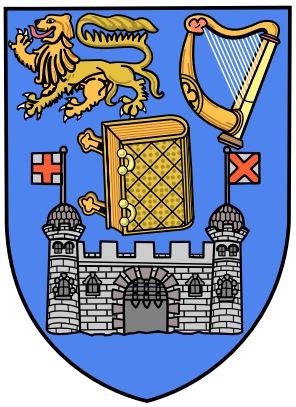
Language Support for Communicating Transactions

A thesis submitted to the University of Dublin, Trinity College
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

School of Computer Science and Statistics, Trinity College Dublin



09 2015

Carlo Spaccasassi

Declaration

This thesis has not been submitted as an exercise for a degree at this or any other university. It is entirely the candidates own work. The candidate agrees that the Library may lend or copy the thesis upon request. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgement.

Carlo Spaccasassi

Because it is filled with truth, charity can be understood in the abundance of its values, it can be shared and communicated. Truth, in fact, is *lógos*, which creates *dia-lógon*, and hence communication and communion. Truth, by enabling men and women to let go of their subjective opinions and impressions, allows them to move beyond cultural and historical limitations and to come together in the assessment of the value and substance of things. Truth opens and unites our minds in the lógos of love: this is the Christian proclamation and testimony of charity.

—Pope Benedict XVI, *Caritas in Veritate*

The Christian God acts with Lógos. [. . .] "In the beginning was the *lógos*, and the *lógos* is God," says the Evangelist. The marriage of Hebrew scripture and Greek philosophy that begat Christianity and subsequently Europe is not mere coincidence, nor is Greek philosophy some adulteration of an otherwise pure Gospel. Europe means Biblical faith plus Greek thought: Europe is based on Lógos.

— E. Michael Jones, *The Jewish Revolutionary Spirit*

History is God's plan.

— E. Michael Jones

Abstract

In the context of databases, a transaction is a sequence of data operations that are executed *atomically* and in *isolation*: either all operations are executed or none is; and their effects are visible to the environment only after the transaction commits. Database transactions are well-established abstractions that simplify concurrent data access in computer programming. Recently Software Transactional Memory (STM) has been proposed to extend the notion of transactions from databases to concurrent programming in shared-memory systems. STM has been likened to garbage collection for memory management in terms of cost and benefits: both of them provide modular abstractions that separate the specification of memory operations, which is delegated to the programmer, from its actual implementation and inherent complexity, which is delegated to an underlying management system.

Few such abstractions currently exist for concurrent programming in distributed systems, where communication is based on message-passing rather than shared memory. *Communicating transactions* is one such abstraction that has been recently proposed to model automatic fault recovery in distributed systems. Dropping the isolation requirement from traditional transactions, communicating transactions allow groups of concurrent processes to interact and commit after reaching a consensus. When they fail to reach a consensus for any reason, including deadlock and unforeseen failures, all interactions are annulled and all side-effects are rolled back.

This thesis focuses on the application of communicating transactions to concurrent programming languages for distributed systems. We explore the expressivity of communicating transactions in TCML, a novel transactional concurrent functional language inspired to CML. We review a powerful proof method based on bisimulations to reason about communicating transactions in the context of a new transactional calculus, TCCS^{m}, and we present a new, equivalent bisimulation method that allows bisimulation equivalences between finite-state TCCS^{m} processes to be calculated algorithmically.

We study the performance of a TCML implementation using naïve transaction scheduling policies. Our study shows that under such policies the creation of consensus groups decreases exponentially in proportion to the number of participants contending for the same resources. This points to the need of more sophisticated transaction analysis tools. A promising tool is session types, a recently proposed mechanism to statically verify the communication protocols employed by concurrent processes. We propose the first fully automatic, sound and complete session type inference algorithm that supports session delegation and recursion. We believe this mechanism to be the foundation for further development and performance enhancement in the realm of transactions scheduling.

Acknowledgements

First of all, I would like to thank my supervisors, Matthew Hennessy and Vasileios Koutavas, for their precious guidance throughout the phd, for their untiring availability after hours and hours of discussions, and for teaching me the ropes of science. This thesis would never have seen the light of day without their support, and I am very grateful to both Matthew and Vassilis for introducing me to fascinating topics of formal methods, such as bisimulations and type and effect systems. I would like to thank the viva examiners, Nick Benton and Andrew Butterfield, for their insights and suggestions, and for the engaging discussions during the defense. I would also like to thank Microsoft Research Cambridge for their generosity and for allowing me to take this fantastic journey in science.

I feel very fortunate for the very many great people I have met in Dublin. My life as a postgraduate would not have been the same without my phd colleagues, with a friendship that extended well beyond the lab. First of all I would like to thank Giovanni Bernardi and Andrea Cerone, not only for helping me with any questions I had on concurrency theory, but also for being great craic all around. Alessandro Vaccaro and Cristina De Persis have been great partners in many a crime. Riccardo Bresciani provided the best audio set in the lab, which I jealously protected. Colm Bhandal has been a steady stream of interesting mathematical puzzles; one day we will solve the tromino problem with Aimee Borda.

Trinity College Societies have greatly enriched my experience during my studies, especially The Hist's debates. I had the greatest fun as chair of the Italian Society together with Andrea Droghetti, Alessandro and Valeria Di Cosmo, without mentioning the many great friends I met through the society. Cisco's concert and Dotti's talk were unforgettable. I was also lucky to witness the resurrection of the Laurentian Society thanks to the efforts of a good few (friends): Alvaro Paul, Ziggy Zdziarski, Aoife Beglin and Ryan Connolly. May all the Laurentians keep strong in the time to come.

Very special thanks go to the Dominican Fathers at St. Saviour's Priory, especially Fr. John Harris and Fr. Brian Doyle, and to Finche Ryan and Cornelius J. Casey of the Loyola Institute, for sharing their wisdom and knowledge, and bringing me closer to Christ. May God bless your works!

Last but not least, I thank my family for their unconditional encouragement, for bearing with me throughout the highs and lows of the doctorate, and for providing me the determination to finish.

Carlo Spaccasassi

Trinity College Dublin

02 2016

Contents

Chapter 1

Introduction

In the context of databases, a transaction is a sequence of data operations that are executed *atomically* and in *isolation*: either all operations are executed or none are; and their effects are visible to the environment only after the transaction commits. Database transactions are well-established abstractions that simplify concurrent data access in computer programming. Software Transactional Memory (STM) has been recently proposed to extend the notion of transactions from databases to concurrent programming in shared-memory systems. STM has been likened in [Grossman, 2007] to garbage collection for memory management in terms of cost and benefits: both of them provide modular abstractions that separate the specification of memory operations, which is delegated to the programmer, from its actual implementation and inherent complexity, which is delegated to an underlying management system.

Programming language support for consensus [Kshemkalyani and Singhal, 2008] has been limited in distributed systems. Achieving consensus between concurrent processes is a ubiquitous problem in multicore and distributed programming [Herlihy and Shavit, 2008]. Among the classic instances of consensus is leader election and synchronous multi-process communication. There are well known algorithms to solve particular consensus problems, such as the Paxos algorithm [Lamport, 2002]. On the one hand, such solutions can be quite complicated to program in low-level languages, scaling up to several thousands of lines of C++ code for example [Chandra et al., 2007]. On the other hand, implementations in high-level languages tend to lack modularity and not to integrate well with the rest of the language. For example, it is not possible to write a three-way rendezvous as an event in CML [Reppy, 1999, Ch. 5, pg. 128], as we will explain in more detail in Section 1.3.2.

These shortcomings point to the need for programming language support to address consensus problems. Borrowing from the traditional concept of transactions, many transactional constructs have been proposed to address this issue, such as cJoin [Bruni et al., 2015], communicating memory transactions [Lesani and Palsberg, 2011] and transactional events [Donnelly and Fluet, 2006]. This thesis focuses on *communicating transactions*, a construct proposed in [de Vries et al., 2010] to model automatic system recovery in distributed systems.

Communicating transactions model inter-process communication in distributed system, therefore they contain concurrency operations, such as send and receive operations over public channels, instead of memory operations on shared memory. Communicating transactions have an all-or-nothing

behaviour too, meaning that either all concurrent operations succeed or none of them do. However, transactions are not isolated by design in this setting: transactions can collaborate and form a consensus group. When all parties are ready to commit, then their communications become *permanent*; before that point all communications are *tentative*. Transactions in this setting can be automatically aborted by the system at any time. When this happens, all tentative communications are cancelled, and each communication partner is automatically *rolled back*.

In order to better illustrate the difficulty of implementing consensus problems, Section 1.1 presents the *Saturday Night Out* (SNO) problem, a concrete problem about consensus in a group of friends. Section 1.2 presents the communicating transactions construct and how the SNO problem can be easily implemented using it.

The use of communicating transactions is not restricted to consensus problems though. They are expressive enough to implement general constructs such as guarded commands and external choice from CSP [Hoare, 1978a], the aforementioned three-way rendezvous, and examples of speculative programming such as the n-queens problem [Lanese et al., 2013a] or depth-first graph search, which is shown in Section 1.3. This generality raises significant challenges for the adoption of communicating transactions in a programming language, both in theory and practice, as explained in Section 1.4.

On the one hand this thesis focuses on the development of formal methods, in particular bisimulation equivalences [Milner, 1982], to reason about transactional programs. On the other hand the thesis explores practical approaches for efficient transaction scheduling. Naïve scheduling policies are found to yield unsatisfactory performances in an experimental study exposed in Chapter 5. The communication patterns of a transaction, however, contain significant information to predict which consensus groups are more likely to be successful. In order to allow for more sophisticated scheduling policies using this information, a significant topic is *session types* [Honda, 1993], a typing discipline that exposes the communication protocols of a process.

Section 1.5 concludes the chapter by presenting the thesis contributions addressing these challenges, and the thesis outline.

1.1 The Saturday Night Out problem

Let us consider a hypothetical scenario of generalized consensus, which we will call the *Saturday Night Out* (SNO) problem. In this scenario, a number of friends (representing computer processes) are seeking partners for various activities on Saturday night. Each has a list of desired activities to attend in a certain order, and will only agree for a night out if there is a partner for each individual activity. A set S of participants forms a *consensus group* when, for each participant, all of its partners belong to S. A solution to the SNO problem is a set of disjoint consensus groups.

Alice, for example, is looking for company to go out for dinner and then a movie (not necessarily with the same person). To find partners for these events in this order she may attempt to synchronize on the "handshake" channels `dinner` and `movie`:

$$\mathsf{Alice} \stackrel{\text{def}}{=} \texttt{sync dinner; sync movie}$$

Here `sync` is a two-party synchronization operator, similar to CSP synchronization. Bob, on the other

hand, wants to go for dinner and then for dancing:

Bob $\stackrel{\text{def}}{=}$ `sync dinner; sync dancing`

Alice and Bob can agree on dinner but they need partners for a movie and dancing, respectively, to commit to the night out. Their agreement is *tentative*.

Let Carol be another friend in this group who is only interested in dancing:

Carol $\stackrel{\text{def}}{=}$ `sync dancing`

Once Bob and Carol agree on dancing they are both happy to commit to going out. However, Alice has no movie partner and she can still cancel her agreement with Bob. Therefore Alice, Bob and Carol do not form a consensus group, and they are not a valid solution. If Alice cancels her agreement with Bob, Bob and Carol need to be notified to cancel their agreement and everyone starts over their search for partners.

An implementation of the SNO scenario between concurrent processes would need to have a specialized way of reversing failed tentative synchronizations. Suppose David is also a participant in this set of friends.

David $\stackrel{\text{def}}{=}$ `sync dancing; sync movie`

After the partial agreement between Alice, Bob, and Carol is canceled, David together with the first two can synchronize on `dinner`, `dancing`, and `movie` and agree to go out (leaving Carol at home).

Notice that when Alice raised an objection to the agreement that was forming between her, Bob, and Carol, all three participants were forced to restart. If, however, Carol was taken out of the agreement (even after she and Bob were happy to commit their plans), David would have been able to take Carol's place and the work of Alice and Bob until the point when Carol joined in would not need to be repeated.

Programming SNO between an arbitrary number of processes (which can form multiple agreement groups) in CML is complicated. Especially if we consider that the participants are allowed to perform arbitrary computations between synchronizations affecting control flow, and can communicate with other parties not directly involved in the SNO. For example, Bob may want to go dancing only if he can agree with the babysitter to stay late:

Bob $\stackrel{\text{def}}{=}$ `sync dinner; if babysitter() then sync dancing`

In this case Bob's computation has side-effects outside of the SNO group of processes. To implement this we would require code for dealing with the SNO protocol to be written in the Babysitter (or any other) process, breaking any potential modular implementation.

1.2 The communicating transactions construct

In order to make the presentation more concrete, we introduce a simple concurrent functional language inspired by CML [Reppy, 1999]. The formal semantics of this language, called TCML, is presented in Chapter 5; for the moment we only give an informal explanation of it in order to give a better intuition about the semantics of communicating transactions.

Values v in this language are integers $n \in \{0, 1, 2 \ldots\}$, booleans $b \in \{\texttt{tt}, \texttt{ff}\}$, the unit value $()$. Variables x, y, z range over an infinite set of variables, and channels c, d from an infinite set of channel names. An expression e comprises the standard sequential constructs of let expressions ($\texttt{let}\, x = e\, \texttt{in}\, e$), functions ($\texttt{fun} f(x) = e$) and conditionals ($\texttt{if}\, b\, \texttt{then}\, e\, \texttt{else}\, e$). The expression $e_1; e_2$ is syntactic sugar for $\texttt{let}\, x = e_1\, \texttt{in}\, e_2$ when x is not free in e_2.

We only consider as concurrent primitives $\texttt{send}\, c\, v$ and $\texttt{recv}\, c$ to exchange values over a channel c, and `newChan` to create private channels. A process P or Q is either an expression e, the parallel composition $P \mid Q$ and the channel restriction $\nu c.P$. Communication is synchronous: a $\texttt{send}\ c\ v$ expression in a process P blocks the evaluation of P until another parallel process Q can evaluate the complementary $\texttt{recv}\, c$ expression, and viceversa. Channel restriction $\nu c.P$ guarantees that the channel c cannot be used for communications outside the scope of P.

Communicating transactions extend a concurrent language, such as ML with concurrency primitives for example, with two constructs: $[\![P \triangleright_k Q]\!]$ and $\texttt{co}\ k$, where k is a transaction name (or identifier). In $[\![P \triangleright_k Q]\!]$, P is called the *default* process and Q is the *alternative* process. Processes in P are free to execute and communicate with each other in transaction k. For example, if $P = \texttt{send}\, c\, v \mid \texttt{recv}\, c$, the following is a valid transition:

$$[\![\texttt{send}\, c\, v \mid \texttt{recv}\, c\ \triangleright_k\ Q]\!] \rightarrow [\![() \mid v\ \triangleright_k Q]\!]$$

The alternative Q cannot be run instead; in order to model faults in distributed systems, transaction k can abort the default process P at any time, and replace P with Q, which is then free to run. In the previous example, the following are both valid transitions caused by aborts:

$$[\![\texttt{send}\, c\, v \mid \texttt{recv}\, c\ \triangleright_k Q]\!] \rightarrow Q$$
$$[\![() \mid v\ \triangleright_k Q]\!] \rightarrow Q$$

Since P is replaced by Q, any interaction that happened inside transaction k is cancelled. Interactions in P are thus *tentative*. When a transaction contains the *commit* process $\texttt{co}\ k$, the transaction is ready to be committed:

$$[\![P \mid \texttt{co}\ k\ \triangleright_k Q]\!] \rightarrow P$$

Since Q and the transactional construct are discarded, process P cannot be aborted any more, and it has become *permanent*. In order for another process R outside of k to interact with P, process R has to be *embedded* into k:

$$[\![P \triangleright_k Q]\!] \mid R \rightarrow [\![P \mid R\ \triangleright_k Q \mid R]\!]$$

When R is embedded, a copy of R is stored in the alternative, so that in case of abort R is restored to its original state in parallel with Q, as if no interaction had taken place at all, and k alone was aborted instead. Process R is therefore effectively *rolled-back*.

Just like software transactional memory and database transactions in general, communicating

transactions have an *all-or-nothing* behaviour. Consider for example the following processes:

- $P \stackrel{\text{def}}{=} [\![\texttt{send}\ c\ v_1;\ \texttt{send}\ d\ v_2;\ \texttt{co}\ k\ \triangleright_k\ ()]\!]$
- $Q \stackrel{\text{def}}{=} \texttt{recv}\ c$

In the system $P \mid Q$, process Q can be embedded in k and a synchronization over channel c can take place. However transaction k cannot be committed yet. Therefore its effects are not permanent yet. In fact, the only transition left available is to abort k:

$$\begin{aligned} P \mid Q &\to [\![\texttt{send}\ c\ v_1;\ \texttt{send}\ d\ v_2;\ \texttt{co}\ k \mid Q\ \triangleright_k\ () \mid Q]\!] \\ &= [\![\texttt{send}\ c\ v_1;\ \texttt{send}\ d\ v_2;\ \texttt{co}\ k \mid \texttt{recv}\ c\ \triangleright_k\ () \mid Q]\!] \\ &\to [\![\texttt{send}\ d\ v_2;\ \texttt{co}\ k \mid v_1\ \triangleright_k\ () \mid Q]\!] \\ &\to () \mid Q \end{aligned}$$

If process $R \stackrel{\text{def}}{=} \texttt{recv}\ d$ is added to the system, then the group composed of $P \mid Q \mid R$ can indeed commit after performing two embeddings:

$$\begin{aligned} P \mid Q \mid R &\to [\![\texttt{send}\ c\ v_1;\ \texttt{send}\ d\ v_2;\ \texttt{co}\ k \mid Q\ \triangleright_k\ () \mid Q]\!] \mid R \\ &= [\![\texttt{send}\ c\ v_1;\ \texttt{send}\ d\ v_2;\ \texttt{co}\ k \mid \texttt{recv}\ c\ \triangleright_k\ () \mid Q]\!] \mid R \\ &\to [\![\texttt{send}\ d\ v_2;\ \texttt{co}\ k \mid v_1\ \triangleright_k\ () \mid Q]\!] \mid R \\ &\to [\![\texttt{send}\ d\ v_2;\ \texttt{co}\ k \mid v_1 \mid \texttt{recv}\ d\ \triangleright_k\ () \mid Q \mid R]\!] \\ &\to [\![\texttt{co}\ k \mid v_1 \mid v_2\ \triangleright_k\ () \mid Q \mid R]\!] \\ &\to v_1 \mid v_2 \end{aligned}$$

Two transactions can be embedded into each other in order to allow them to communicate. However embeddings in TCML are not driven by the need of processes to communicate, but they are performed entirely non-deterministically.

The SNO problem finds a very simple encoding using communicating transactions. We first define *restarting* transactions, as a transaction T that has some default process P and transaction T itself as the alternative:

$$\texttt{atomic}[\![P]\!] \stackrel{\text{def}}{=} \texttt{rec}\ f(\_) \Rightarrow [\![P \blacktriangleright f()]\!]$$

A restarting transaction performs the body P until it commits, or it aborts and tries to perform P again.

The SNO problem is now encoded as follows:

$$\begin{aligned} \mathit{Alice} &\stackrel{\text{def}}{=} \texttt{atomic}[\![\mathit{sync\ dinner};\ \mathit{sync\ movie};\ \texttt{co}]\!] \\ \mathit{Bob} &\stackrel{\text{def}}{=} \texttt{atomic}[\![\mathit{sync\ dinner};\ \mathit{sync\ dancing};\ \texttt{co}]\!] \\ \mathit{Carol} &\stackrel{\text{def}}{=} \texttt{atomic}[\![\mathit{sync\ dinner};\ \texttt{co}]\!] \\ \mathit{David} &\stackrel{\text{def}}{=} \texttt{atomic}[\![\mathit{sync\ movie};\ \mathit{sync\ dancing};\ \texttt{co}]\!] \end{aligned}$$

where *dinner*, *movie* and *dancing* are synchronous public channels.

The encoding is very intuitive; each participant tries to synchronize with other partners for one or more activities and then commits. Because of the semantics of communicating transactions, unsuccessful groups such as *Alice* | *Carol* can never commit and they are automatically aborted. More importantly, no exception handling is required on the part of the programmer. This is also a modular implementation, since the implementations of each participant are independent from each other. For example if we introduced a *Babysitter* process to the group, the implementation of the four participants in the SNO is unchanged.

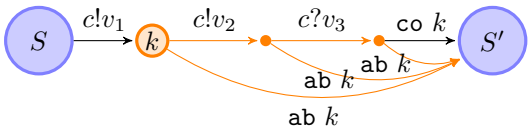
1.3 Expressiveness of communicating transactions

We will now explore the expressiveness of communicating transactions through examples. In order to better illustrate the behaviour of each system that we are going to present, we informally introduce a graphical notation for a Labelled Transition System (LTS) over single transactions. The examples we are going to show do not contain parallel transactions running concurrently, but at most a single transaction running independently. An LTS is a triple $< S, L, Act >$, where S is a set of states, which in our case is a subset of the set of all processes P; L is a relation over $S \times Act \times S$ that indicates labelled transitions from a state to another state; and Act is the set of transition labels.

The set Act comprises labels $c!v, c?v, \texttt{ab}\ k$ and $\texttt{co}\ k$. The first two labels annotate transitions in which the initial state sends or receives a value v over channel c by evaluating the expression $\texttt{send}\ c\ v$ or $\texttt{recv}\ c$. The $\texttt{ab}\ k$ label indicates that transaction k in the initial state is aborted and replaced by its alternative process. The $\texttt{co}\ k$ label indicates that transaction k is committed in the initial state. Intuitively, these transitions model the interaction of the system with an external agent performing a complementary receive or send action. Sequences of tentative actions that terminate with a `co` label become permanent; intermediate sequences can always be rolled back by an `ab` abort action.

In all the examples shown in this section, there will only be one active transaction at a time in a process. When a process is a transaction k, we represent it with an orange state k; all tentative transitions $c!v$ and $c?v$ are drawn with an orange arrow. All other states are blue, and permanent transition are black. A transaction $[\![P \triangleright_k Q]\!]$ can be aborted at any time, therefore all tentative states derived from k can transition to state Q using an $\texttt{ab}\ k$ action. In order to keep the LTS visually simple, we only draw one abort transition from the first state k where transaction k can perform any actions. Thus each orange node has an implicit *abort* arrow, that points to same state as the explicit abort arrow.

For example, process $S = \texttt{send}\ c\ v_1;\ [\![\texttt{send}\ c\ v_2;\ \texttt{recv}\ c;\ \texttt{co}\ k \triangleright_k ()]\!]$ gives rise to the following LTS:



where $S' = ()$. However, we will represent it as follows:

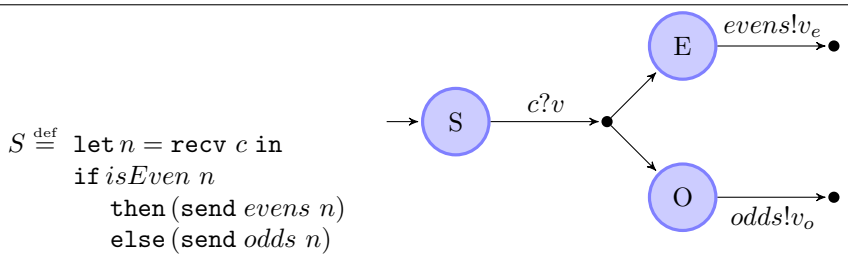
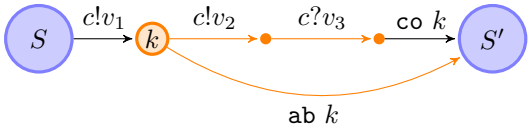


Fig. 1.1: Even/Odd filter specification and LTS.



This informal LTS definition is sufficient to give a visual aid for the transactional code shown in this section. A more detailed account can be found in [Spaccasassi, 2013].

1.3.1 Even-Odd filter

The *Even-Odd filter* describes a system containing three public channels, c, *evens* and *odds*. In the finite case, a single integer number is input into the system system from channel c. Even numbers are output on channel *evens*, and odd numbers on channel *odds*. The technical report in [Spaccasassi, 2013] discusses the more interesting case with a stream of numbers, rather than a single one. The goal of this section is to introduce an encoding of external choice using transactions, therefore we limit ourselves to the case where the system can process exactly one number.

We discuss two versions of the Even-Odd filter: a *specification* and an *implementation*. The specification describes the expected behaviour of the system without using communicating transactions. The implementation will use communicating transactions instead, and we will argue that the specification and implementation describe two systems that cannot be distinguished by an external agents, judging by the set of **permanent** actions that the system can perform. Uncommitted tentative actions are ignored.

Figure 1.1 shows the specification's code and its resulting LTS. The specification is very simple: process S receives an integer n from c, tests whether it is even using the $isEven$ function, and then sends it over the appropriate channel, either *evens* or *odds*. State S in the LTS refers to the system in this initial state. State E is the state of the resulting system when n is an even number (`send` *evens* n); O is the resulting state when n is odd (`send` *odds* n).

These are all possible sequences of actions that an external agent can observe when interacting with the system. If the transactional implementation of the filter can display the behaviour (namely receving a value on c and sending it either on *evens* or *odds*), the transactional system would appear indistinguishable from the specification to an external observer. In that case, we can say that the transactional version implements the specification.

Figure 1.2 shows the implementation of the Even-Odd filter. The implementation S' only features

$$
\begin{aligned}
S' \stackrel{\text{def}}{=} [\![& \texttt{let}\ v = \texttt{recv}\ c\ \texttt{in} \\
& \ \texttt{if}\ isEven\ v \\
& \quad \texttt{then co}\ k;\ \texttt{send}\ evens\ v \\
& \quad \texttt{else}\ () \\
& \triangleright_k\ [\![\texttt{let}\ v = \texttt{recv}\ c\ \texttt{in} \\
& \quad \texttt{if}\ isOdd\ v \\
& \quad\quad \texttt{then co}\ l;\ \texttt{send}\ odds\ v \\
& \quad\quad \texttt{else}\ () \\
& \triangleright_l\ S']\!]
\end{aligned}
$$

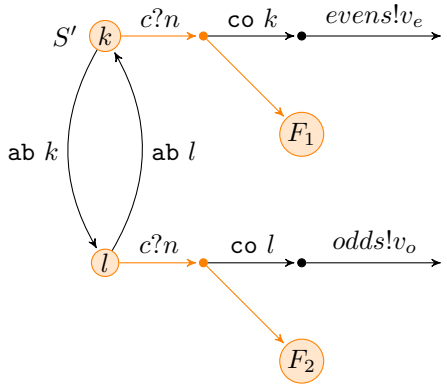


Fig. 1.2: Second version of transactional Even/Odd consumer implementation. F_1 and F_2 are uncommittable states.

a single transactional process that combines the three agents in the specification. The body of this process is transaction k, whose alternative is to start another transaction l, the alternative of which recursively starts transaction k again. Therefore either transaction k is active or transaction l is. Aborting k will activate l, and viceversa. The behaviour of the system alternates between that of transaction k and that of transaction l.

The default process of k receives a number v on public channel c and tests whether v is even. If it is, then it sends v on channel *evens*. Otherwise the system enters the "failed" state $F_1 = [\![() \triangleright_k [\![\ldots \triangleright_l S']\!]]\!]$, where it can only abort k and activate transaction l. When transaction l is active, its default process receives a value from v from c, and tests whether it is odd. If it is, v is sent over the *odds* channel. Otherwise the system enters in the "failed" state $F_2 = [\![() \triangleright_l S']\!]$, where it can only abort l and therefore effectively restart system S'.

If we only consider these kind of paths and ignore the paths that lead to an abort, since they are not definitive and whatever effect was performed on the system is rolled-back, we can easily verify that the system will output even numbers on the *evens* channel and odd numbers on the *odds* channel, after having received it from channel c. The failed states F_1 and F_2 never lead to a commit, therefore they never become permanent and are invisible to an external agent. It is also possible to abort transaction k and l for an indefinite number of times. However these transitions do not bear any permanent action $c!v$ or $c?v$, therefore they are not visible to an external agent and can be ignored for now.

The alternating behaviour of this system is quite interesting. It is reminiscent of *guarded commands* from [Dijkstra, 1975], which is a vector of *guards* of the form $(b_i \rightarrow e_i)$. All boolean conditions b_i are evaluated, and all guarded commands are discarded except for the first command j with $b_j = true$; expression e_j is then executed. The case with just two guarded commands can be implemented as follows using transactions:

$$
\begin{aligned}
gCommand\ b_1\ b_2\ e_1\ e_2 \stackrel{\text{def}}{=} & [\![\texttt{if}\ b_1\ \texttt{then co}\ k;\ e_1\ \texttt{else}\ () \\
& \triangleright_k\ [\![\texttt{if}\ b_2\ \texttt{then co}\ l;\ e_2\ \texttt{else}\ () \\
& \triangleright_l\ gCommand\ b_1\ b_2\ e_1\ e_2]\!]
\end{aligned}
$$

Guards in guarded commands will be tried until one of them can be committed; any eventual effects resulting from the evaluation of b_1 or b_2 is always rolled back.

Similarly, it is possible to exploit the alternating behaviour of the same construct to have external choice (CSP's deterministic choice from [Hoare, 1978b]). If expression e_1 is chosen only after communicating on channel c_1, and expression e_2 is chosen only after communicating on channel c_2, then external choice can be encoded as:

$$\begin{aligned} \mathit{eChoice}\ (c_1, e_1)\ (c_2, e_2) \stackrel{\text{def}}{=}\ & [\![\, \texttt{let}\, v_1 = \texttt{recv}\, c_1\ \texttt{in co}\ k;\ (e_1\ v_1) \\ & \rhd_k\ [\![\, \texttt{let}\, v_2 = \texttt{recv}\, c_2\ \texttt{in co}\ l;\ (e_2\ v_2) \\ & \rhd_l\ \mathit{eChoice}\ (c_1, e_1)\ (c_2, e_2)]\!]]\!] \end{aligned}$$

If a matching communication is available on one of the two branches of the external choice, the construct can abort its transactions until the matching branch is activated.

1.3.2 Three-way rendezvous

In the following section we will demonstrate that TransCCS is expressive enough to define the *three-way rendezvous* (TWR), a construct that synchronizes three processes simultaneously (from an external point of view). Just like two processes can simultaneously exchange some values through a swap channel, three processes swap values by synchronizing on a public three-way rendezvous channel. The solution presented in this section is inspired by the implementation of TWR in Transactional Events from [Donnelly and Fluet, 2006].

Theorem 6.1 from [Reppy, 1999] implies that the three-way rendezvous cannot be implemented modularly in CML:

Theorem 6.1 Given the standard CML event combinators and an n-way rendezvous base-event constructor, one cannot implement an (n+1)-way rendezvous operation abstractly (i.e. as an event value).

CML offers two 2-way rendezvous event constructors, namely sendEvt and recvEvt, which are primitives to send and receive values over typed channels. When provided with appropriate arguments, both constructs have type event, which is a special type reserved for values that have a concurrent effect when applied to the sync primitive.

This theorem does not state that no implementation of an (n+1)-way rendezvous exists in CML, but that there is no such implementation of type event. It is in fact possible to implement a three-way rendezvous that does not have type event. The interested reader can find a discussion about such a CML implementation of TWR in [Spaccasassi, 2013]. The problem with such ad-hoc solutions is that they do not compose well with other primitives. For example, the primitive choose combines two events into a single event that behaves as the external choice of the two events. Because of Theorem 6.1, the external choice of two three-way rendezvous functions cannot be constructed natively using choose, but it must be implemented manually, potentially from scratch if the ad-hoc implementation

```
rendezvous(c, v1) ≝
    let lead = fun l() =
        let c2 = recv c
            c3 = recv c
            v2 = recv c2
            v3 = recv c3
        in send c2 (v1, v3);
           send c3 (v1, v2);
           (v2, v3)
    let flw = fun f() =
        let c1 = newChan () in
        send c c1; send c1 v1; recv c1
    in
      atomic_k[[(let v2,3 = lead v1 in commit; v2,3)
              ⊕ (let v2,3 = flw v1 in commit; v2,3)]]
```

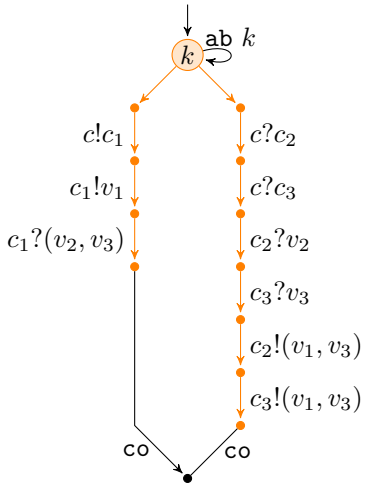


Fig. 1.3: Three-way rendezvous code and LTS. The "⊕" operator stands for internal choice.

of a single rendezvous is not general enough.

Figure 1.3 provides a modular implementation of the three-way rendezvous using communicating transactions, together with its LTS. Using the `rendezvous` function, a participant can exchange his own value v_1 for the other participants' values v_2 and v_3 over public channel c. A participant can play one of two roles in the exchange, the *follower* (encoded by the *flw* function) or the *leader* (encoded by the *lead* function). A follower creates a private channel c_1 and sends it over public channel c, where the leader will be listening. The follower then sends its own value v_1 over the private channel, and receives back the values of the other participants from the leader. Conversely, the leader first receives two private channels over c, one for each follower synchronizing with the leader. The leader then receives the follower's values and exchanges them to all parties accordingly.

There is an obvious complementarity between follower and leader: each action performed by the former is matched by the latter. When the follower sends its own private channel over c, the leader receives it; then when the follower can send its own value, the leader can receive it, and so on. The leader interleaves and coordinates the requests of the two followers in order to complete the rendezvous.

The code responsible for assigning a role to a participant is in the restarting transaction at the bottom of the `rendezvous` program. The internal choice operator $e_1 \oplus e_2$ evaluates randomly to either e_1 or e_2[1]. By using the internal choice operator, the participant chooses its own role completely randomly, and independently from the choices that the other participants make. Figure 1.4 lists all the combination of rendezvous roles. At the start, the three participants have not decided which role to fulfill in the rendezvous. A process in the "undecided" state, drawn as a black dot, can non-deterministically decide to become either a follower (blue dot) or a leader (red dot).

In light of the complementarity of leader and follower, it should be clear that the rendezvous can only succeed if two participants happen to be followers, and one leader. All other combinations result in the participants becoming stuck (for example, if two participants are leaders and one is a follower, the follower will start the rendezvous with one leader, but the other leader will be unable to join them).

[1]It can be encoded as `if` $randomBool()$ `then` e_1 `else` e_2, where $randomBool()$ is a function that returns non-deterministically either `tt` (true) or `ff` (false). It can also be encoded using concurrency primitives by `send` c $()$ | `recv` c; e_1 | `recv` c; e_2, where c is a private channel.

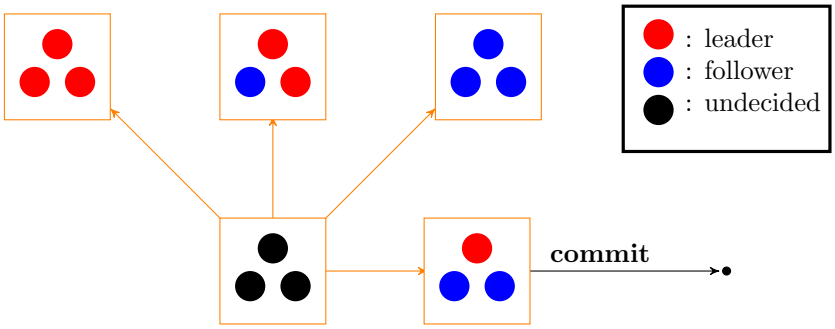


Fig. 1.4: Possible consensus groups of three participants in the three-way rendezvous.

However, since the *lead* and *flw* functions are enclosed in restarting transactions, stuck configurations can be aborted and a new configuration can be tried out.

Notice that it is not necessary to abort the whole rendezvous in case of an erroneous configuration. For example, if all the processes become followers, aborting a single transaction might be sufficient to have a successful rendezvous. The aborted process has a second chance to become leader next, and the rendezvous might succeed. Note also the transactional TWR scales well with the number of participants, because if there are four or more agents engaging in a rendezvous, any stuck configuration they might get into will be aborted, and the participants will synchronize in groups of three. These considerations should be taken into account in the practical implementation of communicating transactions, where we are interested in maximizing the number of committing transactions, and minimizing the number of aborts due to stuck configurations.

1.3.3 Graph search

We will now examine a simple form of transaction nesting and a similarity we have found with the Prolog programming language and its backtracking capabilities.

A classical example in Prolog is graph searching. Given an acyclic directed graph, such as the one shown in Figure 1.5, the problem is to find a path between a starting node and an ending node in the graph, if such a path exists. For example, we might want to find a path between node a and node c, drawn in green and red colour respectively.

One standard solution in Prolog is to perform a depth-first search on the graph, as described in the following Prolog code:

```
gs(X, X, [X]).
gs(X, Y, [X|T]) :-
  link(X, Z),
  gs(Z, Y, T).
```

The first argument in the `gs` clause is the current node being evaluated in the graph. The second argument is the final node that needs to be reached, and the third one is the list of traversed nodes. The `link` predicate is true if an only if an edge exists between the nodes passed to it as arguments. According to the Prolog implementation, there exists a path between the starting and ending node either if they are the same node (first clause), or if there exists a path from a node to which the

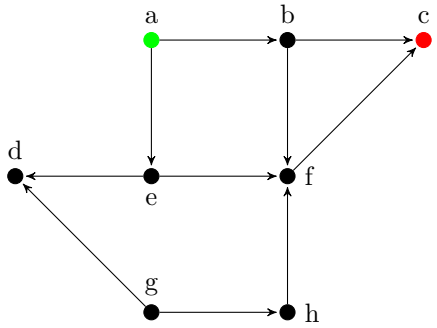


Fig. 1.5: An example graph. The start node 'a' is green, the end node 'c' is red.

starting node is linked to, and the ending node (second clause). This intermediate node becomes the new starting node, and the `gs` clause is invoked recursively on it.

At execution time, there might be many edges departing from the starting node. A Prolog machine will pick any edge satisfying the `link` clause and will try to find a path from there to the ending node, using the `gs` clause recursively. After each recursive step, the Prolog machine will pick edge after edge until the ending node is found; because of this, this style of searching is called *depth-first*. If at some point a node is not connected to any other node, the Prolog machine will stop and it will try to revert the last decision it made when picking edges, and try another link; this step is called *backtracking*. No edge is tried twice, so the algorithm is guaranteed to terminate after having examined all possible paths in the graph.

A Prolog machine would find the following solutions:

```
X = [a,e,f,c]
X = [a,b,f,c]
X = [a,b,c]
```

Communicating transactions can display the same backtracking behaviour using restarting transactions, as shown by the graphSearch program in Fig. 1.6. Let us also model each edge from node n to node n' in Fig. 1.5 as a process that is trying to send channel name n' on channel n. Given the channel name relative to a node, we can non-deterministically retrieve one of the nodes that is directly connected to it. We will use the term node and channel interchangeably in the following discussion.

The *gs* function in the TCML code will verify whether starting node x and ending node e are the same. If they are, then it will output node x on channel *result* and commit the transaction, as in the first Prolog clause. Otherwise, it will receive one of the nodes y connected to node x first, send x to the *result* channel and invoke *gs* on the new node, as in the second clause.

First of all, note that the whole *gs* function is inside a restarting transaction, thus all communications are tentative until a path to the end node is found. If the depth-first search performed by the *gs* function encounters a node that is not connected to any other node, then the transaction can be aborted and another path can be tried.

From the resulting LTS in Figure 1.6, we can recognize three sequences of actions that lead to a commit point: [path!a, path!e, path!f, path!c], [path!a, path!b, path!f, path!c] and [path!a, path!b, path!c]. These sequences represent the same set of answers as the Prolog program. There are a couple of points to keep in mind though. Even though the set of answers is the same, a Prolog machine would

```
graphSearch(x, end) ≝
 atomic_k⟦ rec gs(x, end) ⇒
  if x = end
   then send path x; co k
   else send path x; gs (recv x, end)⟧
```

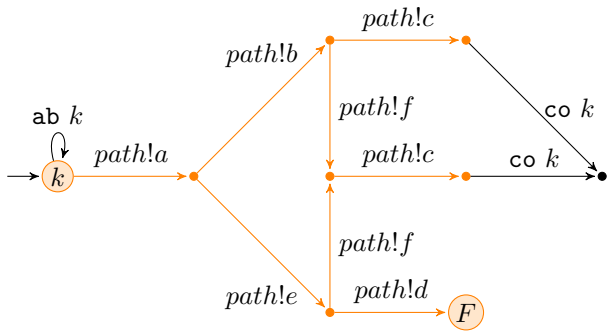


Fig. 1.6: Transactional graph search algorithm and its LTS.

exhaustively test all possible paths in the graph, in depth-first order. In particular, no two equivalent paths are ever tried twice. Even though the TCML example follows the depth-first style to explore the graph, nodes connected to the current node are chosen non-deterministically. Thus the same path may be tried several times. Note also that a Prolog machine has the ability to say that there are no more paths after finding the first three paths. On the contrary, a TCML program is wrapped into a restarting transaction, which can be aborted indefinitely. Thus a restarting transaction will run indefinitely if no solution exists.

With the TCML version transactions can be aborted at any time, even if a correct solution was being found; it could even just abort indefinitely. In a practical environment, an implementation of TCML would have to be particularly *smart* to achieve both efficiency and faithfulness to the reduction semantics, maybe exploiting knowledge on the behaviour of the system accrued over time, such as which transactions aborted most often, or on the currently active processes in the system, and wisely orchestrate all running transactions.

1.4 Challenges

Communicating transactions provide a modular and intuitive abstraction to easily implement consensus problems, such as the SNO problem and the three-way rendezvous, which can be viewed as a leader election problem. It is also a very expressive construct that allows us to build more complicated abstractions, from external choice to examples of speculative computing, such as graph search. It is also possible to encode the n-queens problem, as shown in [Lanese et al., 2013b], and complicated flight booking scenarios in [Bruni et al., 2015].

There are two main challenges for the fruitful introduction of transactions in a programming language: one is theoretical and the other is practical. The first challenge that we need to address is to provide a solid theoretical foundation for communicating transactions. First of all we need formal methods to be able to establish that, for example, the implementation of the Even-Odd filter from Sec. 1.3.1 is correct w.r.t its specification. This can only be established by a formal behavioural theory for communication. Ideally an algorithm would verify whether implementation and specification coincide automatically.

The second challenge is providing an efficient implementation for communicating transactions. As hinted in the discussion about graph search in Sec. 1.3.3, communicating transactions are very power-

ful constructs that can encode hard problems such as the n-queens problem; [Donnelly and Fluet, 2006] shows how to encode 3-SAT problems into transactions, which means that the programmer might encode an NP-complete problem and require a communicating transaction scheduler to solve it. It is therefore virtually impossible to have an efficient implementation of transactions in general, but it might still be possible to achieve good performances in more standard scenarios such as in the SNO problem using some heuristics.

It is unclear what heuristics would work best in practice. As discussed in Sec. 1.1, there is only one grouping of partners in the SNO problem that can succeed, which is *Alice*, *Bob* and *David*. For all practical purposes, *Carol* can be considered as non-existent; any group that includes *Carol* is doomed to abort. Because of this, no consensus group can include Carol, and therefore any solution to the SNO problem has to prevent Carol from joining any group. An efficient implementation of communicating transactions should ideally take advantage of this fact and avoid uncommmittable groups, because any computation performed in such groups is ultimately rolled back, and therefore a waste of time and a performance penalty.

1.5 Contributions and thesis outline

This thesis provides the following contributions:

1. a new mechanism that replaces embedding (see Sec. 1.2) with a simpler construct based on transaction name merging, presented in [Koutavas et al., 2014]. The process calculus that includes this new mechanism is called $TCCS^m$. This innovation greatly simplifies the technical treatment of the language and permits a much easier definition of bisimulation, called *history bisimulation*;

2. a new bisimulation method for TCCS^{m} that permits the definition of an algorithm to decide bisimulation equivalence between finite-state TCCS^{m} processes. Taking history bisimulations as a starting point, a new proof method called *historyless bisimulations* is presented and proved to have the same distinguishing power as history bisimulations;

3. the identification of a sub-language of TCCS^{m} which only includes finite-state programs. Bisimulation equivalence is decidable for this sub-language;

4. the definition of a new concurrent functional language with transactions, called TCML;

5. an empirical study of scheduling policies for transactions in TCML;

6. a session type discipline that guarantees progress under specific conditions;

7. to our knowledge, the first sound and complete, fully automatic inference algorithm for *session types* with delegation in a concurrent functional language à la ML. Session type inference forms the foundation for efficient scheduling of communicating transactions.

The thesis is organized as follows. Chapter 2 introduces the history bisimulations and the new transaction name merging mechanism presented in [Koutavas et al., 2014]. The renaming mechanism is my personal main contribution to that publication. In the history bisimulation setup, each process

is attached with a *history*, an ordered list that records all the actions that the process has performed. As explained in Chapter 3, histories create infinite state LTSs out of any TCCS^{m} process by design. Bisimulation algorithms terminate only in the presence of finite-state LTS, therefore we introduce *historyless bisimulations*, a new proof method that replaces histories with *dependency sets*. In Chapter 4 we show that historyless bisimulations are finite for a certain class of TCCS^{m} processes. We describe a bisimulation algorithm, which has also been implemented and for which a sample output is provided.

Regarding the use of transactions in a practical setting, Chapter 5 presents TCML, an ML-like concurrent functional language that incorporates concurrent transactions. Through experimentation, we have discovered in [Spaccasassi and Koutavas, 2013] that contention of shared channels drastically penalizes performance: the number of committing groups decreases exponentially as the number of participants in consensus groups increases. The generality and expressiveness of communicating transactions, as exposed in Sec. 1.3, are accountable for this. Given this negative result, we concluded that a promising approach to improve performance is through more refined code analysis tools, by statically predicting the transactional and concurrent behaviour of a program.

A promising approach to formally specify and verify the communication protocols of a system is *session types*, as mentioned earlier. Session types are a typing discipline recently proposed to model protocol interactions in [Honda, 1993]. For example, consider the following process P:

$$P \stackrel{\text{def}}{=} \texttt{send}\ c\ ();\ \texttt{if}\ \texttt{recv}\ c\ \texttt{then}\ \texttt{send}\ c\ 1\ \texttt{else}\ \texttt{send}\ c\ 2$$

Process P first sends the unit value over channel c, then it receives a boolean value, and finally it either sends the integer 1 or 2.

Regardless of what action is taken in the conditional expression and of the actual values exchanged over c, we can describe abstractly the protocol that P implements with the following sequence of typed actions:

$$\eta = !\mathsf{Unit}\,.\,?\mathsf{Bool}\,.\,!\mathsf{Int}$$

Protocol η is called a session type; the session type codifies the exchange of typed values that P has with the external environment over c (P sends a unit value, receives a boolean and sends an integer).

Consider now the three way rendezvous from Section 1.3.2. We have already argued that there exists an obvious complementarity between the protocol followed by the leader and the followers. Session types can give an explicit and formal description of such protocol. By analyzing the protocol, it is possible to establish that a consensus group can only be formed by a leader and two followers. Through static analysis means, it is therefore possible to equip a transaction scheduler with session types information and enhance its ability to predict which group is more likely to reach a consensus.

Chapter 6 introduces ML_{S}, a concurrent functional language equipped with session type primitives, polymorphism and higher-order session communications. A peculiar property of our typing discipline is that, in the absence of divergent processes and in the presence of enough participants to each communication protocol, any system is guaranteed to terminate. Chapter 7 introduces the type inference algorithm for ML_{S}, together with proofs of termination, soundness and completeness.

Chapter 9 summarizes the topics presented in the thesis. We introduce a number of languages in this thesis: TCCS^{m} , TCML and ML_{S}. We envision a version of TCML that uses transaction renaming instead of embeddings, and session types. Future directions of work are discussed, such as bisimulation methods for nested transactions and how session types and transactions can be combined to improve transaction scheduling performances, as outlined for the three-way rendezvous example.

Chapter 2

Bisimulations for communicating transactions

A central notion in the formal analysis of concurrent systems is the study of observable behaviour, which is the kind of interactions that a program is capable of with its environment. Of particular interest is proving that two systems have the same *extensional* behaviour, meaning that it is impossible for an external agent to find a difference in the communication patterns of the two systems only by interacting with them. This notion of equivalence would justify the claim that, for example, the transactional implementation of the three-way rendezvous in Section 1.3.2 is indeed indistinguishable from its non-transactional specification.

Such behavioural theories are usually expressed in terms of some form of environment or context. A context might be a hole in the abstract syntax tree of a system [Sangiorgi and Walker, 2001], or a concurrent system to be run in parallel with one of the systems under test [Honda and Yoshida, 1995]. In general two systems are defined extensionally equivalent if there is no observable difference in the behaviour between the two systems when put in a context, *for any context*. Since contexts can be arbitrarily complex, direct proofs of equivalence are generally hard to find.

Bisimulations [Milner, 1982] provide an elegant and effective proof technique for proving contextual equivalences between processes. A bisimulation can be considered as a challenger/defender game played by two opponents, where the challenger tries to discover a difference in the extensional behaviour of two processes, while the defender tries to refute these attempts [Stirling, 1998].

For example, consider the following processes from Milner's Calculus of Communicating Systems (CCS):

$$P \stackrel{\text{def}}{=} a.(b.0 + c.0) \qquad\qquad Q \stackrel{\text{def}}{=} a.b.0 + a.c.0$$

where P performs an action a and then chooses to perform either b or c; and Q chooses one of two sequences of actions to play (namely $a.b.0$ and $a.c.0$), and then proceeds to play them. In a bisimulation game the challenger can show that P and Q have indeed a different observable behaviour. The challenger can first play play a from P. This forces the opponent to play a matching action from Q, which entails choosing one of the two sequences of actions $a.b.0$ or $a.c.0$ in order to play action

a. In case the defender chooses the branch $a.b.0$, the challenger can play c from $P' = b.0 + c.0$. The defender cannot reply to this attack, since it can only play b, and therefore loses the bisimulation game. Viceversa, if Q chooses the other branch, then the challenger can play c and win the game. There is no winning strategy for the defender, and therefore the two processes are said to be not bisimilar, and therefore not equivalent.

This chapter focuses on the development of a behavioural theory of CCS extended with communicating transactions, called $TCCS^m$, for the sake of simplicity. The development of a theory for a functional programming language is left for future work. Before elaborating further, we must note that embeddings introduce several technical difficulties in the development of a behavioural theory. Consider the following two CCS processes enclosed in communicating transactions:

$$P \stackrel{\text{def}}{=} [\![a.\texttt{co}\ k \triangleright_k 0]\!] \qquad\qquad Q \stackrel{\text{def}}{=} [\![\bar{a}.\texttt{co}\ l \triangleright_l 0]\!]$$

and consider P and Q to be in parallel. Clearly P and Q can synchronize on a and commit. However, in order for this to happen, a double embedding has to take place. Either k is embedded into l, and the default process of l is

$$P \mid Q \rightarrow\!\!\!\!\rightarrow [\![[\![a.\texttt{co}\ k \mid \bar{a}.\texttt{co}\ l \triangleright_k 0 \mid \bar{a}.\texttt{co}\ k]\!] \triangleright_l P \mid 0]\!]$$

or the opposite happens, namely that l is embedded into k first, and then the default of k is embedded in l:

$$P \mid Q \rightarrow\!\!\!\!\rightarrow [\![[\![a.\texttt{co}\ k \mid \bar{a}.\texttt{co}\ l \triangleright_l a.\texttt{co}\ k \mid 0]\!] \triangleright_k 0 \mid Q]\!]$$

First of all, this example shows that there are multiple ways of embedding transactions into each other, however it is unclear whether the order of embedding makes any difference. In this case it does not, and ideally it should not, since the purpose of embedding is just to allow communication to happen between two partners. It is unclear whether the order of embeddings makes a difference in general. Secondly, notice that transaction names are binders in TransCCS. After an embedding takes place, the commit point $\texttt{co}\ k$ of a transaction k is copied in the alternatives of some other transaction, and it becomes hard to track the process that can commit k. Thirdly, embeddings are not communication driven. A process can be embedded in a transaction unnecessarily, which further complicates reasoning about transactions.

Because of these reasons, Section 2.1 introduces $TCCS^m$, a transactional version of CCS where embeddings are replaced by a new mechanism called *transaction merging*. We only consider *flat* transactions, i.e. transactions that do not contain nested transactions in their default processes. We believe that the theory presented in this chapter can be extended to the nested case as well. Section 2.2 discusses *reduction barbed equivalence*, a natural notion of contextual equivalence from [Honda and Yoshida, 1995] with some adaptations to the transactional world. Finally Section 2.3 discusses *history bisimulations*, a proof technique based on the notion of bisimilarity [Milner, 1982]. History bisimulations are sound and complete with regards to reduction barbed equivalence. We do not present the proof of these results, which are outside the scope of the thesis. The material presented

in this chapter has been published in [Koutavas et al., 2014], where the interested reader is referred to for more technical details about the soundness and completeness proofs.

2.1 The language $TCCS^m$

The syntax for terms of $TCCS^m$ is given in Fig. 2.1 where $\mu \in \mathsf{Act}_\tau \uplus \Omega$ and X ranges over a collection of *recursion variables*. Here Ω is a special set of actions which will be used to define contextual equivalence while $(\bar{\cdot}) : \mathsf{Act} \to \mathsf{Act}$ is a total bijection over Act, used in the standard manner to formalise CCS synchronisation between processes.

The language $TCCS^m$ is essentially obtained from CCS [Milner, 1982] by introducing three new constructs: $[\![P \rhd_k Q]\!]$ and $[\![P \blacktriangleright Q]\!]$ for communicating transactions, and a new command `co` for committing them. The construct $[\![P \rhd_k Q]\!]$ is called an *active* transactions, that is a transaction is ready to interact with its environment. Transaction has name k, which we assume to belong to an infinite set of names. The construct $[\![P \blacktriangleright Q]\!]$ is called a *dormant* transactions, which can become active after it is allocated a fresh transaction name k. Before that time, a dormant transaction cannot interact with its environment.

We assume the standard notion of free and bound occurrence of recursion variables, and only consider closed terms, those which contain no free occurrences of variables. We use the standard abbreviations associated with CCS, and write $s \sharp s'$ when the transaction names of the syntax object s are fresh from those in s'; $\mathit{ftn}(s)$ denotes the transaction names in s. Note that unlike previous work in TransCCS [de Vries et al., 2010, De Vries et al., 2010], transaction names are never bound and we do not require that all transaction names used in a term are distinct. Thus, we allow terms of the form $[\![P_1 \rhd_k P_2]\!] \,|\, R \,|\, [\![Q_1 \rhd_k Q_2]\!]$. Here k should be looked upon as a *distributed* transaction whose behaviour will be approximately the same as the centralised $[\![P_1 \,|\, Q_1 \rhd_k P_2 \,|\, Q_2]\!]$. The use of these distributed transactions will simplify considerably the exposition of the reduction semantics.

Definition 2.1.1 (Well-formed terms)**.** *A closed term is called* well-formed *if in every occurrence of* $[\![P \rhd_k Q]\!]$, $[\![P \blacktriangleright Q]\!]$, *and* $\mathtt{rec}X.P$, *the subterms* P *and* Q *do not contain named transactions of the form* $[\![- \rhd_- -]\!]$. *We refer to well-formed terms as* processes.

Dormant transactions *can* appear within other transactions and under recursion, but the semantics of $TCCS^m$ will allow them to be activated only when they end up at top-level. In the sequel we only consider well-formed terms.

The reduction semantics of the language is given as a binary relation between processes $P \to Q$. However this is defined indirectly in terms of three auxiliary relations, which will also be used in the formulation of bisimulations:

$$P \to Q \qquad \text{when} \qquad P \xrightarrow{\tau}_\sigma Q \qquad \text{or} \qquad P \xrightarrow{k(\tau)}_\sigma Q \qquad \text{or} \qquad P \xrightarrow{\beta} Q$$

The first, $P \xrightarrow{\tau} Q$, is essentially synchronisation between pure CCS processes. The second, $P \xrightarrow{k(\tau)}_\sigma Q$, is synchronization between a transaction and another $TCCS^m$ process. More generally, we write $P \xrightarrow{k(\mu)}_\sigma Q$ where $\mu \in \mathsf{Act} \cup \{\tau\}$ for the tentative interaction of P with an environment. The third,

$TCCS^m$ Syntax

$$P,Q,R \quad ::= \quad \sum \mu_i.P_i \;\mid\; P \,|\, Q \;\mid\; \nu a.P \;\mid\; X \;\mid\; \mathtt{rec}X.P \;\mid\; [\![P \rhd_k Q]\!] \;\mid\; \mathtt{co}.P \;\mid\; [\![P \blacktriangleright Q]\!]$$

CCS Transitions

$$\textsc{CCSsum}\ \frac{}{\Sigma\mu_i.P_i \xrightarrow{\mu_i}_\varepsilon P_i} \qquad \textsc{CCSsync}\ \frac{P \xrightarrow{a}_\varepsilon P' \qquad Q \xrightarrow{\overline{a}}_\varepsilon Q'}{P \,|\, Q \xrightarrow{\tau}_\varepsilon P' \,|\, Q'} \qquad \textsc{CCSrec}\ \frac{}{\mu X.P \xrightarrow{\tau}_\varepsilon P[\mu X.P/X]}$$

Transactional Transitions

$$\textsc{TrTau}\ \frac{P \xrightarrow{\tau}_\varepsilon P'}{[\![P \rhd_k Q]\!] \xrightarrow{\tau}_\varepsilon [\![P' \rhd_k Q]\!]} \qquad \textsc{TrSum}\ \frac{}{\Sigma\mu_i.P_i \xrightarrow{k(a)}_{\varepsilon \mapsto k} [\![P_j \,|\, \mathtt{co} \rhd_k \Sigma\mu_i.P_i]\!]}\ \mu_j = a$$

$$\textsc{TrAct}\ \frac{P \xrightarrow{a}_\varepsilon P'}{[\![P \rhd_l Q]\!] \xrightarrow{k(a)}_{l \mapsto k} [\![P' \rhd_k Q]\!]}\ k \sharp l \qquad \textsc{TrSync}\ \frac{P \xrightarrow{k(a)}_{\sigma_1} P' \qquad Q \xrightarrow{k(\overline{a})}_{\sigma_2} Q'\ \ \sigma_1 = \widetilde{l}_1 \mapsto k}{P \,|\, Q \xrightarrow{k(\tau)}_{(\widetilde{l}_1,\widetilde{l}_2) \mapsto k} P'\sigma_2 \,|\, Q'\sigma_1\ \ \sigma_2 = \widetilde{l}_2 \mapsto k}$$

Propagation Transitions

$$\textsc{Restr}\ \frac{P \xrightarrow{\alpha}_\sigma P'}{\nu a.P \xrightarrow{\alpha}_\sigma \nu a.P'}\ a \sharp \alpha \qquad \textsc{ParL}\ \frac{P \xrightarrow{\alpha}_\sigma P'}{P \,|\, Q \xrightarrow{\alpha}_\sigma P' \,|\, Q\sigma}\ range(\sigma) \sharp Q$$

Fig. 2.1: Communication and internal transitions (omitting symmetric rules)

$P \xrightarrow{\beta} Q$, where β ranges over $\mathtt{co}\ k$, $\mathtt{ab}\ k$ and $\mathtt{new}\ k$, encode commit/abort transitions which eliminate transactions and the creation of new named transactions. These need to be *broadcast actions*, i.e. actions performed synchronously by each single process in P, because of the presence of distributed transactions: for example, a transaction can be committed only if all its distributed parts are ready to commit.

We first concentrate on the second relation. Action $P \xrightarrow{k(a)}_\sigma Q$ should be viewed as the synchronisation between P and some transaction named k in the environment which can perform the complementary $\overline{a}$. Because this transaction is external we will always assume that the name k is fresh. Also, the effect of this synchronisation is that the future behaviour of P, or at least any transactions involved in the execution of a, is dependent on the eventual committing of k. This dependency is implemented by σ, a substitution renaming the responsible transaction in P to k. The essential rule in the generation of these judgements is TrAct in Fig. 2.1. For example, this rule ensures that we can derive: $[\![a.P_1 \rhd_{l_1} Q_1]\!] \xrightarrow{k(a)}_{l_1 \mapsto k} [\![P_1 \rhd_k Q_1]\!]$ for any fresh k. The substitution recorded in the action is propagated by ParL into contexts. Note that by TrSum, even pure CCS processes with no transactions can perform a $k(a)$ action; e.g., $a.P \xrightarrow{k(a)}_{\varepsilon \mapsto k} [\![P \,|\, \mathtt{co} \rhd_k a.P]\!]$. This encloses P into the k-transaction; the distributed part of the k-transaction surrounding P is always ready to commit (hence the introduction of co). Note that this is a *communication-driven embedding*, which reduces the nondeterminism of embeddings.

The conjoining of transactions is implemented in TrSync. Using it we infer:

$$[\![a.P_1 \rhd_{l_1} Q_1]\!] \mid [\![\overline{a}.P_2 \rhd_{l_2} Q_2]\!] \xrightarrow{k(\tau)}_{(l_1,l_2) \mapsto k} [\![P_1 \rhd_k Q_1]\!] \mid [\![P_2 \rhd_k Q_2]\!]$$

$$\textsc{TrNew}\ \frac{}{[\![P \blacktriangleright Q]\!] \xrightarrow{\texttt{new}\ k} [\![P \triangleright_k Q]\!]} \qquad \textsc{TrAb}\ \frac{}{[\![P \triangleright_k Q]\!] \xrightarrow{\texttt{ab}k} Q} \qquad \textsc{TrCo}\ \frac{P \rightsquigarrow_{\texttt{co}} P'}{[\![P \triangleright_k Q]\!] \xrightarrow{\texttt{co}k} P'}$$

$$\textsc{TrBroadcast}\ \frac{P \xrightarrow{\beta} P' \qquad Q \xrightarrow{\beta} Q'}{P \,|\, Q \xrightarrow{\beta} P' \,|\, Q'} \beta \in \{\texttt{co}k, \texttt{ab}k\} \qquad \textsc{TrIgn}\ \frac{P \xrightarrow{\beta} P'}{P \,|\, Q \xrightarrow{\beta} P' \,|\, Q} \beta \sharp Q \qquad \textsc{TrRestr}\ \frac{P \xrightarrow{\beta} P'}{\nu a.P \xrightarrow{\beta} \nu a.P'}$$

Fig. 2.2: Transactional reconfiguration transitions

for any fresh k. Here the previously independent transactions l_1, l_2 have been merged into the transaction k (recorded in the substitution $(l_1, l_2) \mapsto k$). Note that this new transaction is distributed, in that its activity is divided in two. In order for it to commit the rules in Fig. 2.2 ensure that *both* components commit.

Consider for example the process $P = \nu p.\, [\![a.p.\texttt{co}.R \triangleright_{l_1} 0]\!] \,|\, [\![b.\overline{p}.\texttt{co}.S \triangleright_{l_2} 0]\!]$. Two applications of TrAct followed by rule Restr gives the reduction:

$$P \to^* \nu p.\, [\![p.\texttt{co}.R \triangleright_{k_1} 0]\!] \,|\, [\![\overline{p}.\texttt{co}.S \triangleright_{k_2} 0]\!] = P'$$

where k_1 and k_2 are fresh names. Now an application of the synchronisation rule TrSync and the structural rule gives

$$P' \to \nu p.\, [\![\texttt{co}.R \triangleright_k 0]\!] \,|\, [\![\texttt{co}.S \triangleright_k 0]\!]$$

where k is an arbitrary fresh name. Here the residual is a single transaction named k, albeit distributed. For it to commit both components have to commit: using TrBroadcast this leads to the process $R \,|\, S$.

Broadcast actions $P \xrightarrow{\beta} Q$ are described in Fig. 2.2; we first explain the effect of commits on a single active transaction k. An active transaction can initiate a commit when it contains a top level co keyword. In this case the alternative process and the transactional construct itself are removed as per Rule TrCo. Additionally, this operation removes all occurrences of the co keyword from the body of the k-transaction, referring to the transaction. Any co inside an inner dormant transaction are retained, since they refer to that transaction.

The definition of local commits is defined by structural induction as follows:

$$\frac{}{\texttt{co}.P \rightsquigarrow_{\texttt{co}} P'\{\!|\tau \backslash \texttt{co}|\!\}} \qquad \frac{P \rightsquigarrow_{\texttt{co}} P'}{\nu a.P \rightsquigarrow_{\texttt{co}} \nu a.P'}$$

$$\frac{P \rightsquigarrow_{\texttt{co}} P' \qquad Q \rightsquigarrow_{\texttt{co}} Q'}{P \,|\, Q \rightsquigarrow P' \,|\, Q'} \qquad \frac{P \rightsquigarrow_{\texttt{co}} P'}{P \,|\, Q \rightsquigarrow P' \,|\, Q\{\!|\tau \backslash \texttt{co}|\!\}} \nexists Q'.Q \not\rightsquigarrow_{\texttt{co}} Q'$$

where the term $Q\ \{\!|\tau \backslash \texttt{co}|\!\}$ substitutes all occurrences of co in Q with τ-prefixes, except for those inside dormant transactions. The obvious symmetric rules are omitted.

Commits are propagated by Rule TrBroadcast and Rule TrIgn. In order for a distributed transaction k to commit, all its distributed parts must be able to commit locally (Rule TrBroadcast). Otherwise Rule TrIgn guarantees that a process can broadcast a co k action only when transaction k

does not occur in it. For example transaction $[\![\texttt{co} \rhd_k 0]\!] \mid [\![0 \rhd_k 0]\!]$ cannot commit, because the second distributed part cannot commit locally; Rule TrIgn cannot be applied on the right-hand side of the parallel construct.

When a transaction $[\![P \rhd_k Q]\!]$ is aborted by Rule TrAb, its default process P is removed and its alternative Q is released. In order to ensure that a distributed transaction k is not aborted partially, abort actions $\texttt{ab}\ k$ is broadcast in the same manner as commits by Rule TrBroadcast and Rule TrIgn. Dormant transactions $[\![P \blacktriangleright Q]\!]$ are activated by Rule TrNew into the active transaction $[\![P \rhd_k Q]\!]$, for some name k. The side condition of Rule TrIgn guarantees that a transaction is activated (with action $\texttt{new}\ k$) only if the name k is fresh with respect to other parallel processes. All broadcast actions are propagated through restrictions by Rule TrRestr.

The semantics has a number of properties: it preserves well-formedness, generates only fresh transaction names and is equivariant. The properties about transaction names are important because they give us the liberty to pick fresh enough transaction names in proofs. To state these properties we use *renamings*, ranged over by r, which are bijective substitutions of the form $[l_1 \mapsto k_1], \ldots, [l_n \mapsto k_n]$. The following lemma states that names generated by either a transaction merging or from a $\texttt{new}\ k$ broadcast action are always fresh:

Lemma 2.1.2 (Freshness of generated names)**.** *Let P and Q be well-formed TCCS^m terms. If $P \xrightarrow{\alpha}_\sigma Q$, then $\alpha, rg(\sigma) \sharp P$. If $P \xrightarrow{\texttt{new}\ k} Q$, then $k \sharp P$.*

Proof. By rule induction. □

We conclude this section by showing a minor property of the LTS, namely that the shape of a substitution σ depends on the kind of action ($\tau, k(\tau)$ or $k(a)$) that a process performs. This property will be useful for the technical development of later chapters:

Lemma 2.1.3 (Substitution inversion)**.** *Let P and Q be well-formed TCCS^m terms. If $P \xrightarrow{\alpha} \sigma Q$, then:*

- $\sigma = \epsilon$ *when* $\alpha = \tau$
- $\sigma = [l, l' \mapsto k]$ *when* $\alpha = k(\tau)$
- $\sigma = [\epsilon \mapsto k]$ *or* $\sigma = [l \mapsto k]$ *when* $\alpha = k(a)$

Proof. By rule induction. □

2.2 Reduction barbed equivalence

Based on this semantics we give a natural contextual equivalence, based on the notion of *reduction barbed equivalence* [Honda and Yoshida, 1995]. We write $\Rightarrow$ for the reflexive transitive closure of $\rightarrow$. We define a *barb*, which are the actions externally observable by the context:

Definition 2.2.1 (Barb)**.** *$P \Downarrow \omega$ if there exist Q, Q' such that $P \Rightarrow Q \xrightarrow{\omega}_\epsilon Q'$.*

Notice that barbs are never tentative, but must be played by a top-level process, i.e. one that is not inside a transaction. Reduction barbed equivalence is defined as follows:

Definition 2.2.2 (Reduction Barbed Equivalence ($\cong_{\mathsf{rbe}}$)). *$(\cong_{\mathsf{rbe}}) \subseteq \mathcal{L}_{\mathsf{Act}\uplus\Omega} \times \mathcal{L}_{\mathsf{Act}\uplus\Omega}$ is the largest relation for which $P \cong_{\mathsf{rbe}} Q$ when:*

1. *$P{\Downarrow}\omega$ iff $Q{\Downarrow}\omega$,*
2. *if $P \to P'$ then there exists Q' such that $Q \Rightarrow Q'$ and $P' \cong_{\mathsf{rbe}} Q'$,*
3. *if $Q \to Q'$ then there exists P' such that $P \Rightarrow P'$ and $P' \cong_{\mathsf{rbe}} Q'$,*
4. *$P \mid R \cong_{\mathsf{rbe}} Q \mid R$ for any $R \in \mathcal{L}_{\mathsf{Act}\uplus\Omega}$ with $R \sharp P, Q$.*

Here we consider contexts with fresh transaction names to enforce that observer transactions are distinct from processes transactions before communication occurs. If this was not the case then transaction names would be observable: $[\![P \rhd_k Q]\!]$ would not be equivalent to $[\![P \rhd_l Q]\!]$ because by introducing the context $R = [\![0 \rhd_k 0]\!]$ the k-transaction can no longer commit (whereas l still can).

To see why in the above definition we use barbs from a distinct Ω consider:

$$P = [\![a.\mathsf{co} \rhd_k 0]\!] \qquad\qquad Q = a.0 + \tau.0$$

Intuitively, we would expect P to have exactly the same behaviour as Q, eventually executing the single action a, or failing with a τ step. But if we allowed the barb $\Downarrow a$ in Def. 2.2.2 then they would not be equivalent because $P \not\Downarrow a$ and $Q \Downarrow a$.

While it is hard to prove directly from Def. 2.2.2 that $P \cong_{\mathsf{rbe}} Q$ holds for some arbitrary P and Q, the proof of the contrary is sometimes viable. Consider for example the following:

$$P = [\![a.b.\mathsf{co} \;+\; a.c.\mathsf{co} \rhd_k 0]\!] \qquad\qquad Q = [\![a.(b.\mathsf{co} \;+\; c.\mathsf{co}) \rhd_l 0]\!]$$

and take $C_1 = \bar{a}$ as a context. Then we have:

$$P \mid C_1 \xrightarrow{l(\tau)} [\![b.\mathsf{co} \rhd_l 0]\!] \mid [\![\mathsf{co} \rhd_l C_1]\!] = P' \qquad \text{and thus} \qquad P \mid C_1 \to P'$$

where P' is the residual of P after choosing its left-hand side branch. Transaction l in P' can commit only if its context can synchronize on b.

If $P \cong_{\mathsf{rbe}} Q$ was true, then either $Q \mid C_1$ or one of its successors (namely $Q' = [\![b.\mathsf{co} + c.\mathsf{co} \rhd_m 0]\!] \mid [\![0 \rhd_m C_1]\!]$ after a synchronization, or $Q'' = 0 \mid C_1$ after aborting l) would be reduction barbed equivalent to P'. However this is not the case, because for each successor of $Q \mid C_1$ there exists a context C_2 that distinguishes it from P'. If we take $C_2 = \bar{c}.\omega$, it is easy to see that $Q \mid C_1 \mid C_2 {\Downarrow}\omega$ holds (after Q synchronizes with C_1 first and then C_2), but this is not possible for P', which cannot communicate over c. The same reasoning applies in the case of Q'. Conversely, if we take $C_2' = b.\omega$, then $P' \mid C_2' {\Downarrow}\omega$ holds but $Q'' \mid C_2'$ cannot perform the same barb, because transaction l has been aborted.

2.3 History bisimulations

The standard bisimulation game outlined in the introduction to this chapter cannot be easily extended to $TCCS^m$. When a process inside a transaction k performs an action a, the resulting action $k(a)$ is

tentative and can be aborted at any time by transaction k. If k never commits, all its tentative action $k(\mu)$ are never going to become permanent, and therefore they will never be observable to an external observer.

To underline this consider the processes:

$$P_1 = [\![a.(b.\mathtt{co} + c.0) \rhd_k 0]\!] \qquad\qquad Q_1 = [\![a.b.\mathtt{co} \rhd_l 0]\!] \tag{2.1}$$

Here P_1 commits only after performing a and b. It would be unreasonable for the challenger, after the a action, to demand a response to c. Not only is this action tentative on the completion of transaction k but also if c is performed then k will *never* commit; so the defender should be able to ignore this challenge.

The definition of bisimulation game is based on *configurations*, of the form $\mathcal{C} = (H \rhd P)$ where P is a system and H a *history* of all the tentative actions taken so far in the game by P. These are of the form $k(a)$, where k is the name of a transaction which needs to commit before the action becomes a permanent a. When playing bisimulation moves, the histories of both systems being scrutinized must remain *consistent*, in that permanent actions in the respective histories must match exactly. The crucial aspect of this new game is that when a system commits a transaction k, and only then, all tentative actions in its history dependent on k are made permanent. This consistency requirement then forces a response in which the corresponding actions match exactly.

For example consider the following variation on processes $P \stackrel{\text{def}}{=} a.(b.0 + c.0)$ and $Q \stackrel{\text{def}}{=} a.b.0 + a.c.0$ from the introduction, using transactions:

$$P_2 = [\![a.b.\mathtt{co} \ + \ a.c.\mathtt{co} \rhd_k 0]\!] \qquad\qquad Q_2 = [\![a.(b.\mathtt{co} \ + \ c.\mathtt{co}) \rhd_l 0]\!] \tag{2.2}$$

Replaying the game from the introduction, where the challenger first chooses a from P and then c from Q with the same responses, we reach the configurations:

$$\mathcal{C}_2 = (k(a), k(c) \rhd [\![\mathtt{co} \rhd_k 0]\!]) \qquad\qquad \mathcal{D}_2 = (l(a), l(b) \rhd [\![\mathtt{co} \rhd_l 0]\!])$$

At this stage the two histories are still consistent as they contain no permanent actions. However, now there is no possible response when the challenger chooses the commit move: $\mathcal{C}_2 \to \mathcal{C}_2' = (a, c \rhd 0)$. This is a silent move from $\mathcal{C}_2$ in which transaction k commits, making the two actions in the history permanent. There are various ways in which $\mathcal{D}_2$ can try to respond but all lead to an inconsistent history. Thus, with our version of bisimulations P_2 and Q_2 are not bisimilar.

Note that such a successful attack by the challenger cannot be mounted for (2.1) above. After one round in the game we have the configurations:

$$\mathcal{C}_1 = (k(a) \rhd [\![b.\mathtt{co} + c.0 \rhd_k 0]\!]) \qquad\qquad \mathcal{D}_1 = (l(a) \rhd [\![b.\mathtt{co} \rhd_l 0]\!])$$

and since $\mathcal{D}_1$ has no possible c actions the challenger might request a response to the action: $\mathcal{C}_1 \to \mathcal{C}_1' = (k(a), k(c) \rhd [\![0 \rhd_k 0]\!])$. However the tentative $k(c)$ recorded in the history will never become permanent and thus the defender can successfully respond with any tentative action of $\mathcal{D}_1$ which will

also never become permanent. To ensure that such responses can always be made, our bisimulations will allow *any* configuration to make the degenerate silent move: $(H \rhd P) \rightarrow (H, k(\star) \rhd P)$, where $\star$ is a reserved symbol. So the move above by $\mathcal{C}_1$ can be successfully matched by $\mathcal{D}_1$ playing this degenerate move followed by the abort of the transaction. Later we prove that P_1 and Q_1 are indeed bisimilar.

Using recursion we can write restarting transactions as $\mathtt{rec}X.[\![P \blacktriangleright X]\!]$. Here $[\![P \blacktriangleright X]\!]$ is an uninitiated transaction which has yet to be allocated a name. These transactions can abort and re-run internal steps, thus branching differences in initial silent actions can be hidden from the challenger. Consider a compiler that performs common subexpression elimination, transforming P_3 to Q_3:

$$P_3 = \mathtt{rec}X.[\![\tau.b.\mathtt{co} + \tau.c.\mathtt{co} \blacktriangleright X]\!] \qquad\qquad Q_3 = \mathtt{rec}X.[\![\tau.(b.\mathtt{co} + c.\mathtt{co}) \blacktriangleright X]\!] \tag{2.3}$$

As we will show, all moves of the challenger can be matched by the defender in the bisimulation game. The interesting scenario is when (after initiating the transactions) the challenger picks the right τ action from P_3 and the defender responds with the τ action from Q_3. We then get the configurations:

$$\mathcal{C}_3 = (\varepsilon \rhd [\![c.\mathtt{co} \rhd_k P_3]\!]) \qquad\qquad \mathcal{D}_3 = (\varepsilon \rhd [\![b.\mathtt{co} + c.\mathtt{co} \rhd_l Q_3]\!])$$

The challenger then picks the b action from $\mathcal{D}_3$. The defender responds with a silent abort of $\mathcal{C}_3$ which will reinstate P_3, re-initialize the transaction, and select the left τ action in $\mathcal{P}_3$ followed by the b action. This would lead to the configurations: $\mathcal{C}'_3 = (k'(b) \rhd [\![\mathtt{co} \rhd_{k'} P_3]\!])$ and $\mathcal{D}'_3 = (l(b) \rhd [\![\mathtt{co} \rhd_l Q_3]\!])$. We will in fact prove that this optimisation is sound in our setting, although it is not sound in the case of P_4, Q_4, even if we used restarting transactions.

We now give a formal account of history bisimulations. Our bisimulations will be over configurations $(H \rhd P)$ with a process P and a history H of the tentative interactions of P with its environment. An element of such a history can be a $k(a)$, a, $\mathtt{ab}$, $k(\star)$, or $\star$. A past tentative action $k(a)$ that has not been committed or aborted is recorded as is in the history. If the k-transaction that performed this action has committed, the action is recorded as a; if the transaction has aborted, the action is recorded as $\mathtt{ab}$. Histories also record the trivial actions $k(\star)$ which can be performed by any process. If these actions have been committed, they are recorded as $\star$; if they have been aborted they are recorded again as $\mathtt{ab}$. For technical convenience elements in a history are uniquely indexed.

Definition 2.3.1 (History)**.** *A* history *H is a partial function from objects i of a countable set I to the set $\{a, \star, k(a), k(\star), \mathtt{ab} \mid a \in \mathsf{Act}\}$.*

We can think of a history as a sequence of the aforementioned actions indexed by the set I. We often write histories as *lists*, omitting the indices of their elements. History composition, written as H_1, H_2, is defined when $dom(H_1) \cap dom(H_2) = \emptyset$. We also let $\hat{a}$ and $\hat{b}$ range over $\mathsf{Act} \cup \{\star\}$ and $\hat{\mu}$ range over $\mathsf{Act} \cup \Omega \cup \{\tau, \star\}$. To express the effect of commits and aborts to histories we define the following operations.

Definition 2.3.2. *$H \backslash_{\mathsf{co}} k$ and $H \backslash_{\mathsf{ab}} k$ are the lifting to lists of the operations:*

$$
\begin{array}{llll}
(i \mapsto k(\hat{a})) \backslash_{\mathsf{co}} k & = (i \mapsto \hat{a}) & (i \mapsto k(\hat{a})) \backslash_{\mathsf{ab}} k & = (i \mapsto \mathtt{ab}) \\
(i \mapsto l(\hat{a})) \backslash_{\mathsf{co}} k & = (i \mapsto l(\hat{a})) & (i \mapsto l(\hat{a})) \backslash_{\mathsf{ab}} k & = (i \mapsto l(\hat{a})) \qquad \textit{when } k \sharp l \\
(i \mapsto \hat{a}) \backslash_{\mathsf{co}} k & = (i \mapsto \hat{a}) & (i \mapsto \hat{a}) \backslash_{\mathsf{ab}} k & = (i \mapsto \hat{a}) \\
(i \mapsto \mathtt{ab}) \backslash_{\mathsf{co}} k & = (i \mapsto \mathtt{ab}) & (i \mapsto \mathtt{ab}) \backslash_{\mathsf{ab}} k & = (i \mapsto \mathtt{ab})
\end{array}
$$

For the reasons we explained in the Introduction, weak bisimulations for $TCCS^m$ require configurations to agree on the committed actions in their histories, and only those actions. Soundness of our technique will establish this as a sufficient requirement for contextual equivalence between processes.

Definition 2.3.3 (Well-formed configurations)**.** *A configuration $(H \rhd P)$ is* well-formed *when P is a process.*

Definition 2.3.4 (Consistency)**.** *H_1 and H_2 are* consistent *when they have the same domain and for all $i \in I$, $a \in \mathsf{Act}$: $H_1(i) = a$ iff $H_2(i) = a$.*

History consistency is one of the two main requirements for weakly bisimilar configurations; the other is to have the same barbs. Thus the weak bisimulation game for $TCCS^m$ will be over transitions with three simple labels: $\zeta ::= \tau \mid k \mid \omega$ annotating internal (τ), tentative synchronization (k) and barbs (ω).

Definition 2.3.5 (Bisimulation Transitions)**.** *$\mathcal{C} \xrightarrow{\zeta} \mathcal{C}'$ is derived by the rules:*

$$
\begin{array}{llll}
(H \rhd P) \xrightarrow{\tau} (\sigma(H) \rhd Q) & \textit{if} \quad P \xrightarrow{\tau}_{\sigma} Q & (\mathrm{LTS}\tau) \\
(H \rhd P) \xrightarrow{\tau} (\sigma(H) \rhd Q) & \textit{if} \quad P \xrightarrow{k(\tau)}_{\sigma} Q \textit{ and } k \sharp H & (\mathrm{LTS}k(\tau)) \\
(H \rhd P) \xrightarrow{\tau} (H \rhd Q) & \textit{if} \quad P \xrightarrow{\mathtt{new}\, k} Q \textit{ and } k \sharp H & (\mathrm{LTS}\mathtt{new}) \\
(H \rhd P) \xrightarrow{\tau} (H \backslash_{\mathsf{co}} k \rhd Q) & \textit{if} \quad P \xrightarrow{\mathtt{co}\, k} Q & (\mathrm{LTS}\mathtt{co}) \\
(H \rhd P) \xrightarrow{\tau} (H \backslash_{\mathsf{ab}} k \rhd Q) & \textit{if} \quad P \xrightarrow{\mathtt{ab}\, k} Q & (\mathrm{LTS}\mathtt{ab}) \\
(H \rhd P) \xrightarrow{k} (\sigma(H), i \mapsto k(a) \rhd Q) & \textit{if} \quad P \xrightarrow{k(a)}_{\sigma} Q, k \sharp H \textit{ and } i \sharp dom(H) & (\mathrm{LTS}k(a)) \\
(H \rhd P) \xrightarrow{k} (H, i \mapsto k(\star) \rhd P) & \textit{if} \quad k \sharp H, P \textit{ and } i \sharp dom(H) & (\mathrm{LTS}\star) \\
(H \rhd P) \xrightarrow{\omega} (\sigma(H) \rhd Q) & \textit{if} \quad P \xrightarrow{\omega}_{\sigma} Q & (\mathrm{LTS}\omega)
\end{array}
$$

We define $\stackrel{\zeta}{\Rightarrow}$ to be $\xrightarrow{\tau}{}^*$ when $\zeta = \tau$, and $\stackrel{\tau}{\Rightarrow}\xrightarrow{\zeta}\stackrel{\tau}{\Rightarrow}$ otherwise.

The first five rules encode the $TCCS^m$ reduction semantics $P \to Q$ presented in Section 2.1, updating the history of the configurations accordingly. LTS$k(a)$ encodes the synchronization between a transaction in the process and its environment, yielding a fresh transaction k; this tentative action is recorded in the history using a fresh index i. LTSω encodes top-level barbs and LTS$\star$ records a trivial defender synchronization move on a fresh index i. Weak τ- and k-transitions can always be performed by the defender in the bisimulation game. Moreover, there are no top-level a-transitions because they can always be simulated by a $k(a)$-transition followed by the commit of k.

The challenger of this bisimulation game performs all-but-$\star$ transitions.

Definition 2.3.6. *$\mathcal{C}_1 \xrightarrow{\zeta} \mathcal{C}_2$ is a* challenger move *if it is derived without using* LTS$\star$.

We now give the definition for a weak bisimulation over the above transitions.

Definition 2.3.7 (Weak Bisimulation)**.** *A binary relation* $\mathcal{R} \subseteq \mathsf{Conf}_{\mathsf{Act} \uplus \Omega} \times \mathsf{Conf}_{\mathsf{Act} \uplus \Omega}$ *is a weak bisimulation when for all* $\mathcal{C}_1 \; \mathcal{R} \; \mathcal{C}_2$*:*

1. *hist*$(\mathcal{C}_1)$ *and hist*$(\mathcal{C}_2)$ *are consistent,*
2. *if* $\mathcal{C}_1 \xrightarrow{\zeta} \mathcal{C}_1'$ *is a challenger move and* $\zeta \sharp C_2$ *then* $\exists \; \mathcal{C}_2'$*:* $\mathcal{C}_2 \stackrel{\zeta}{\Rightarrow} \mathcal{C}_2'$ *and* $\mathcal{C}_1' \; \mathcal{R} \; \mathcal{C}_2'$*,*
3. *the converse of the preceding condition.*

We define weak bisimilarity ($\approx$) as the largest weak bisimulation, and extend it to processes by letting $P \approx Q$ if $(\emptyset \rhd P) \approx (\emptyset \rhd Q)$. Similarly to process transitions in Sec. 2.1, bisimulation transitions and weak bisimulations are unaffected by fresh renaming. Thus, the name selected in a challenger move is unimportant, and we can consider only one convenient fresh name for each transition.

We now explain how to prove bisimilarity between two $TCCS^m$ processes by example. Recall P_1 and Q_1 from (2.1). In order to show that $P_1 \approx Q_1$ (i.e., $(\emptyset \rhd \mathcal{P}_1) \approx (\emptyset \rhd \mathcal{Q}_1)$), it suffices to check that $\mathcal{R}_1$ in Fig. 2.3 is a weak bisimulation.

Related histories in $\mathcal{R}_1$ are consistent. The interesting case is when $(k(a) \rhd [\![b.\mathtt{co} + c.0 \rhd_k 0]\!]) \xrightarrow{k'} (k'(a), k'(c) \rhd [\![0 \rhd_{k'} 0]\!])$. The defender responds with:

$$\begin{aligned}(k(a) \rhd [\![b.\mathtt{co} \rhd_k 0]\!]) &\xrightarrow{k'}_{\mathrm{LTS}\star} (k'(a), k'(\star) \rhd [\![b.\mathtt{co} \rhd_k 0]\!]) \\ &\xrightarrow{\tau}_{\mathrm{LTS}\mathtt{ab}} (\mathtt{ab}, \mathtt{ab} \rhd 0)\end{aligned}$$

and gets:

$$(k'(a), k'(c) \rhd [\![0 \rhd_{k'} 0]\!]) \; \mathcal{R}_1 \; (\mathtt{ab}, \mathtt{ab} \rhd 0)$$

The rest is trivial, thus the defender always wins, therefore $\mathcal{R}_1$ is a weak bisimulation and $P_1 \approx Q_1$.

Let us show $P_3 \approx Q_3$ from (2.3) by constructing relation $\mathcal{R}_3$ in Fig. 2.3. In this construction, H is a history with zero or more aborted actions; we add this to our configurations because restarting transactions can nondeterministically abort and restart. The proof that $\mathcal{R}_3$ is a weak bisimulation is again by an easy inspection of the moves of the challenger. The important move is when from the pair $((H \rhd [\![b.\mathtt{co} \rhd_k P_3]\!]), \; (H \rhd [\![(b.\mathtt{co} + c.\mathtt{co}) \rhd_l Q_3]\!]))$ the challenger picks the transition $(H \rhd [\![(b.\mathtt{co} + c.\mathtt{co}) \rhd_l Q_3]\!])) \xrightarrow{l'} (H, l'(c) \rhd [\![\mathtt{co} \rhd_{l'} Q_3]\!]))$ and the defender plays:

$$\begin{aligned}(H \rhd [\![b.\mathtt{co} \rhd_k P_3]\!]) \quad &\xrightarrow{\tau}_{(\mathrm{LTS}\mathtt{ab})} (H \rhd P_3) \\ &\xrightarrow{\tau}_{(\mathrm{LTS}\tau)} (H \rhd [\![\tau.b.\mathtt{co} + \tau.c.\mathtt{co} \blacktriangleright P_3]\!]) \\ &\xrightarrow{\tau}_{(\mathrm{LTS}\mathtt{new})} (H \rhd [\![\tau.b.\mathtt{co} + \tau.c.\mathtt{co} \rhd_k P_3]\!]) \\ &\xrightarrow{\tau}_{(\mathrm{LTS}k(\tau))} (H \rhd [\![c.\mathtt{co} \rhd_{k'} P_3]\!]) \\ &\xrightarrow{l'}_{(\mathrm{LTS}k(a))} (H, l'(c) \rhd [\![\mathtt{co} \rhd_{l'} P_3]\!])\end{aligned}$$

and gets:

$$(H, l'(c) \rhd [\![\mathtt{co} \rhd_{l'} Q_3]\!])) \; \mathcal{R}_3 \; (H, l'(c) \rhd [\![\mathtt{co} \rhd_{l'} P_3]\!])$$

which concludes the proof.

$$\mathcal{R}_1 \stackrel{\text{def}}{=} \{\ ((\varepsilon \rhd P_1),\ (\varepsilon \rhd Q_1)),\quad ((k(a) \rhd [\![b.\mathtt{co} + c.0 \rhd_k 0]\!]),\ (k(a) \rhd [\![b.\mathtt{co} \rhd_k 0]\!])),$$
$$((k(a), k(b) \rhd [\![\mathtt{co} \rhd_k 0]\!]),\ (k(a), k(b) \rhd [\![\mathtt{co} \rhd_k 0]\!])),\quad ((a, b \rhd 0),\ (a, b \rhd 0)),$$
$$((k(a), k(c) \rhd [\![0 \rhd_k 0]\!]),\ (\mathtt{ab}, \mathtt{ab} \rhd 0)),\quad ((\mathtt{ab}, \ldots \rhd 0),\ (\mathtt{ab}, \ldots \rhd 0))\ \mid \text{any } k\}$$

$$\mathcal{R}_3 \stackrel{\text{def}}{=} \{\ ((H \rhd P_3),\ (H \rhd Q_3)),\quad ((H, x \rhd 0),\ (H, x \rhd 0))$$
$$((H \rhd [\![\tau.b.\mathtt{co} + \tau.c.\mathtt{co} \blacktriangleright P_3]\!]),\ (H \rhd [\![\tau.(b.\mathtt{co} + c.\mathtt{co}) \blacktriangleright Q_3]\!])),$$
$$((H \rhd [\![\tau.b.\mathtt{co} + \tau.c.\mathtt{co} \rhd_k P_3]\!]),\ (H \rhd [\![\tau.(b.\mathtt{co} + c.\mathtt{co}) \rhd_l Q_3]\!])),$$
$$((H \rhd [\![b.\mathtt{co} \rhd_k P_3]\!]),\ (H \rhd [\![(b.\mathtt{co} + c.\mathtt{co}) \rhd_l Q_3]\!])),$$
$$((H \rhd [\![c.\mathtt{co} \rhd_k P_3]\!]),\ (H \rhd [\![(b.\mathtt{co} + c.\mathtt{co}) \rhd_l Q_3]\!])),$$
$$((H, k(x) \rhd [\![\mathtt{co} \rhd_k P_3]\!]),\ (H, k(x) \rhd [\![\mathtt{co} \rhd_k Q_3]\!])),$$
$$\mid \text{any } k, l, H = (\mathtt{ab}, \ldots),\ \text{and } x = a \text{ or } b\ \}$$

$$\mathcal{R}_4 \stackrel{\text{def}}{=} \{\ ((\varepsilon \rhd P_4),\ (\varepsilon \rhd Q_4)),\quad ((\mathtt{ab}, \ldots \rhd 0),\ (\mathtt{ab}, \ldots \rhd 0)),$$
$$(k_1(a) \rhd \nu p.\ [\![p.\mathtt{co}.R \rhd_{k_1} 0]\!] \mid [\![b.p.\mathtt{co}.S \rhd_{k_2} 0]\!]),\ (k_1(a) \rhd [\![b.\mathtt{co}.(R \mid S) \rhd_{k_1} 0]\!])),$$
$$(k_2(b) \rhd \nu p.\ [\![a.p.\mathtt{co}.R \rhd_{k_1} 0]\!] \mid [\![p.\mathtt{co}.S \rhd_{k_2} 0]\!]),\ (k_2(b) \rhd [\![a.\mathtt{co}.(R \mid S) \rhd_{k_2} 0]\!])),$$
$$((\boldsymbol{k_1}(a), \boldsymbol{k_2}(b) \rhd \nu p.\ [\![p.\mathtt{co}.R \rhd_{k_1} 0]\!] \mid [\![p.\mathtt{co}.S \rhd_{k_2} 0]\!]),$$
$$(k_2(a), k_2(b) \rhd [\![\mathtt{co}.(R \mid S) \rhd_{k_2} 0]\!])),$$
$$((\boldsymbol{k_2}(b), \boldsymbol{k_1}(a) \rhd \nu p.\ [\![p.\mathtt{co}.R \rhd_{k_1} 0]\!] \mid [\![p.\mathtt{co}.S \rhd_{k_2} 0]\!]),$$
$$(k_1(b), k_1(a) \rhd [\![\mathtt{co}.(R \mid S) \rhd_{k_1} 0]\!])),$$
$$((\boldsymbol{k}(x), \boldsymbol{k}(y) \rhd \nu p.\ [\![\mathtt{co}.R \rhd_k 0]\!] \mid [\![\mathtt{co}.S \rhd_k 0]\!]), (k(x), k(y) \rhd [\![\mathtt{co}.(R \mid S) \rhd_k 0]\!]))$$
$$((x, y, H \rhd \nu p.\ R \mid S), (x, y, H \rhd R \mid S))$$
$$\mid \text{any } k, k_1, k_2, R, S, H \text{ and } (x, y) = (a, b) \text{ or } (b, a)\ \}$$

Fig. 2.3: Relations used to prove the equivalences $P_1 \approx Q_1$, $P_3 \approx Q_3$ and $P_4 \approx Q_4$ presented in Section 2.3.

In our final example we prove a transactional variant of the parallel expansion law of CCS. In CCS, it holds true that $a.P \mid b.Q$ is bisimilar to $a.b.(P \mid Q) + b.a.(P \mid Q)$. Consider the following $TCCS^m$ processes:

$$P_4 \stackrel{\text{def}}{=} \nu p.\ \ [\![a.p.\mathtt{co}.R \rhd_k 0]\!] \mid [\![b.\overline{p}.\mathtt{co}.S \rhd_l 0]\!]$$
$$Q_4 \stackrel{\text{def}}{=} [\![a.b.\mathtt{co}.(R \mid S) + b.a.\mathtt{co}.(R \mid S) \rhd_m 0]\!]$$

We can prove that in $TCCS^m$ $P_4 \approx Q_4$ holds by constructing the relation $\mathcal{R}_4$ in Fig. 2.3.

It is easy to verify that all histories related in $\mathcal{R}_4$ are consistent and all challenger moves can be matched by the defender. Here it is noteworthy that the two tentative actions a and b are recorded in the left-hand history under different transaction names (k_1 and k_2, respectively) until the synchronization on p merges the two transactions; these history annotations are highlighted in bold typeface.

We conclude the chapter with the definition of committable action, committable transaction, and with some properties of the History LTS that will be used for the technical development of later chapters. A transition $k(a)$ is *committable* when there exists a series of transitions after which transaction k is committed:

Definition 2.3.8 (Commitable Transition)**.** *$\mathcal{C} \xrightarrow{\zeta} \mathcal{C}'$ with $\zeta \in \{\tau, \omega\}$ is committable; $\mathcal{C} \xrightarrow{k(\mu)} (H_1 \rhd P_1)$ is committable when there exists $(H_1 \rhd P_1) \xrightarrow{\zeta_1} \ldots \xrightarrow{\zeta_{(n+1)}} (H_n \rhd P_n)$ such that for any a and appropriate i:*

$$(H_1, (i \mapsto k(a)) \rhd P_1) \xrightarrow{\zeta_1} \ldots \xrightarrow{\zeta_{(n+1)}} (H_n, (i \mapsto a) \rhd P_n)$$

Notice that the item $i \to k(a)$ is added on top of H_1. Its role is to act as a probe for when transaction

k is committed, regardless of the renamings that might change the actual name of transaction k.

Similarly, a transaction k is committable when there exists a series of transitions such that k is committed:

Definition 2.3.9 (Commitable Transaction)**.** *A transaction k is* committable *in a configuration $\mathcal{C} = (H_0 \rhd P_0)$ when there exists a sequence of transitions $\mathcal{C} \xrightarrow{\zeta_1} (H_1 \rhd P_1) \ldots \xrightarrow{\zeta_n} (H_n \rhd P_n)$, where $n > 0$, and an index i such that $H_0(i) = k(\hat{a})$ and $H_n(i) = \hat{a}$.*

The only transaction names that can be merged, committed or aborted, are only those that occur in a configuration:

Lemma 2.3.10. *For any configuration $\mathcal{C} = (H \rhd P)$ and $\mathcal{C}'$, it holds that:*

1. *$\mathcal{C} \xrightarrow{k}_{\tilde{l} \mapsto k} \mathcal{C}'$ only if $\tilde{l} \in ftn(P)$*
2. *$P \xrightarrow{\mathtt{co}\, k} P'$ only if $k \in ftn(P)$.*
3. *$P \xrightarrow{\mathtt{ab}\, k} P'$ only if $k \in ftn(P)$.*

Proof. By inversion on the LTS rules. □

Finally, the following lemma shows that LTS transitions do not change action names in the history. For example, if $\mathcal{C} = (H, i \mapsto k(a) \rhd P)$ and $\mathcal{C} \xrightarrow{\zeta} (H' \rhd P')$ holds for some H' and P', then $H'(i)$ can never replace action a with another action $b \neq a$; $H'(i)$ can only range over the set $\{l(a), a, \mathtt{ab}\}$ for some transaction name l.

Lemma 2.3.11. *For any configuration $\mathcal{C} = (H \rhd P)$ and $\mathcal{C}' = (H' \rhd P')$ such that $\mathcal{C} \xrightarrow{\zeta} \mathcal{C}'$:*

1. *if $H(i) = a$, then $H'(i) = a$.*
2. *if $H(i) = k(a)$, then $H'(i) \in \{l(a), a, \mathtt{ab}\}$ for some transaction name l.*
3. *if $H(i) = \mathtt{ab}$, then $H'(i) = \mathtt{ab}$.*

Proof. By rule induction. □

Chapter 3

From history to historyless bisimulations

As shown in the previous chapter, History bisimulation is a proof technique to establish reduction barbed equivalence between flat TrancCCS processes. Having defined a proof method, we would like to have an algorithm to calculate the bisimulation relation automatically. However, it is not immediately clear how such an algorithm can be achieved.

The first and most obvious problem is that histories can grow indefinitely. The indefinite length of histories can cause even simple processes to generate an infinite state space in the History LTS. Therefore an algorithm based on histories seems unfeasible from the very start. Consider in fact the following example in standard CCS:

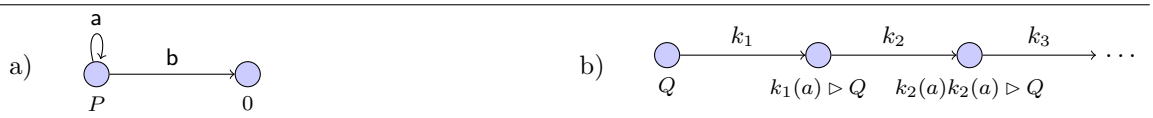
$$P = \texttt{rec}\ X.a.X + b$$

Process P performs a string of a actions, terminated by a b action. Figure 3.1.a shows the LTS generated by P, which is clearly finite. Consider now the following transactional version of P:

$$Q = \texttt{rec}\ X.[\![a.X + b.\texttt{co} \blacktriangleright b]\!]$$

Processes P and Q generate the same traces, the only difference being that Q's actions are tentative up to commitment. Figure 3.1.b shows the state space generated by Q, which is infinite because of its history. In fact, whenever Process Q performs an a action, a new item is added to the history. Since Q can perform recursively many a actions, there are infinitely many items that can be added, and therefore infinitely many states.

There is also another reason why Q's state space is infinite. At each transition, process Q picks a fresh transaction name. There are infinitely many fresh names available to Q, other than k_1, k_2 and k_3. Therefore a transition in Q can generate infinitely many different $k(a)$ items, one for each fresh name available. Chapter 4 presents a discipline for fresh names generation, that obviates this problem. The present chapter focuses on eliminating the first source of infinity, namely the indefinite length of histories, through a series of simplifications to the notion of history bisimulation. Since we



$P = \mathbf{rec}\ X.(a.X + b)$ $\qquad$ $Q = [\![\mathbf{rec}X.(a.X + b.\mathtt{co}) \triangleright_k nil]\!]$

Fig. 3.1: Process P yields a finite state LTS; Q has an infinite state LTS.

are not interested in results of contextual equivalence, we will ignore ω labels and Rule LTSω from Def. 2.3.5 for the rest of the chapter.

A naive first approach might be the following. By definition, the history bisimulation game can be played as long as the participants' histories are consistent: at the beginning of each round, all permanent actions in the participants' histories must match. Pairs of permanent actions remain consistent throughout a bisimulation game, because the LTS does not allow a configuration to modify permanent actions in the history. This observation suggests that the consistency check becomes redundant after the first time it is performed, and therefore we might be tempted to simply remove consistent permanent actions at each bisimulation round.

It is indeed the case that permanent actions can be safely removed. However this is not sufficient to obtain finite histories, because a transaction might never commit, and therefore its associated history might grow indefinitely with tentative actions. Instead of following this approach, a series of observations shows that there is no need for histories to store action names in the first place.

The first observation regards $\star$ actions. By definition of Rule LTS$\star$, a configuration $(H \triangleright P)$ introduces a $k(\star)$ action in the history only when k is fresh from H and P. By definition of the History LTS, process P can only abort, commit or rename tentative actions $l(\hat{a})$ in H only if l occurs in P. Since k is fresh from P by construction, the tentative $k(\star)$ cannot be modified anymore by P once it is introduced.

Tentative actions are irrelevant when checking if two histories are consistent; since $k(\star)$ actions are unmodifiable, they can be replaced by $\mathtt{ab}$ actions without losing distinguishing power in the bisimulation. Moreover, there is no need to introduce $k(\star)$ in the first place. Section 3.1 shows that Rule LTS$\star$ can safely introduce $\mathtt{ab}$ instead of a $k(\star)$ action, therefore making $\star$ actions entirely redundant. This modification greatly simplifies the technical development of this chapter, because it will allows us to assume that an history contains a tentative action $k(a)$ if and only if there exists a transaction k in P.

A second observation is that bisimilar processes must always play the same $k(a)$ action when k is committable, but they can play any action when k is uncommittable. Consider the following non-bisimilar for example:

$$P = [\![a.\mathtt{co} \triangleright_k 0]\!] \qquad Q = [\![b.\mathtt{co} \triangleright_k 0]\!]$$

In this example, transaction k and l are both committable. If P challenges Q with a tentative $k'(a)$ action and commits, there is no response that Q can play without rendering the histories inconsistent. Process Q can avoid losing the bisimulation game only responding if it can store a committable $k'(a)$ in its history. There would be no loss of generality if we forced Q to respond by storing either a $k'(a)$

or $\mathtt{ab}$ action in this case, instead of allowing Q to storing a committable $k'(b)$ action.

Consider now another example:

$$P = [\![a.b.c \rhd_k 0]\!] \qquad\qquad Q = [\![d.e.f \rhd_l 0]\!]$$

Processes P and Q are bisimilar because transaction k and l cannot commit. Even though they perform entirely different tentative actions, such actions never become permanent, and therefore they are never checked for consistency. Therefore, if P challenges Q with action $k'(a)$, it does not matter what action Q responds with. It is possible to deny Q the option to respond by storing tentative action $k'(d)$, and to force Q to store $\mathtt{ab}$ instead in this case too. There is no loss of generality, because k' is uncommittable.

These two considerations suggest that history bisimulation can actually restrict the range of action names allowed in a bisimulation game without loss of generality. When the attacker stores a $k(a)$ action, the defender can be forced to respond with either the same action $k(a)$ or with $\mathtt{ab}$, since playing other actions do not produce tangible differences. It can also be shown that, when the attacker plays a k move and the defender responds with $\mathtt{ab}$, the two players are bisimilar only if transaction k is uncommitable.

In order to prove this result, we show that aborting uncommittable transactions does not produce any observable difference in the behaviour of a configuration from the point of view of bisimulation. When a transaction k is uncommittable in a configuration $\mathcal{C}$, the defender can also transition into the *abort-k configuration* $\mathcal{C} \setminus_{\mathtt{ab}} k$, which is the resulting configuration after aborting k in $\mathcal{C}$. We prove in Section 3.2 that $\mathcal{C}$ and $\mathcal{C} \setminus_{\mathtt{ab}} k$ are bisimilar.

Notice that the histories produced by this version of bisimulation are now very uniform. For any index i, if one history stores $k(a)$ at index i, the other history either stores $l(a)$ or $\mathtt{ab}$. Action names are therefore redundant now, because any two histories will contain either the same action name, or a $\mathtt{ab}$ label for any given index. Section 3.3 introduces *uniform bisimulation*, the bisimulation with the aforementioned restrictions on the moves allowed in a bisimulation game, and where histories of bisimilar participants are *action consistent*, that is, action names of both permanent and tentative actions always match in the participants' histories.

Even though action names are redundant, we cannot eliminate histories yet. Consider the following example:

$$P = [\![a.b.\mathtt{co} + b.a.\mathtt{co} \rhd_k 0]\!] \qquad Q = [\![a.\mathtt{co} \rhd_{l_1} 0]\!] \parallel [\![b.\mathtt{co} \rhd_{l_2} 0]\!]$$

Process Q resembles the parallel expansion of process P, like the familiar parallel expansion law from CCS. Processes equivalent under the parallel expansion law are bisimilar in CCS. Therefore it is tempting to think that their transactional counterparts P and Q are bisimilar too. However, notice that the two transactions in Q are indendent from each other, since the first transaction is annotated by l_1 and the second one by l_2. This difference is sufficient to distinguish the two processes under bisimulation: process Q can perform a single permanent action a by committing l_1, but process P cannot do so, because it performs both actions a and b permanently, or none at all.

This discussion highlights that, even though action names can be dispensed with, histories keep track of dependencies among transactions, by matching transaction names occurring at the same index of attacker's and defender's history. Section 3.5 introduces the notion of *Historyless bisimulation*, which has an explicit parameter to track dependencies between attacker's and defender's transactions. The chapter is concluded by showing that Historyless bisimulation relates the same processes as Extended bisimulation, and therefore as History bisimulation.

3.1 Starless bisimulation

The first simplification to history bisimulation regards $k(\star)$ actions in the history. Once added to an history, $k(\star)$ elements cannot be aborted, committed or modified anymore. Consider in fact Rule LTS$\star$:

$$(H \rhd P) \quad \xrightarrow{k} (H, k(\star) \rhd P) \quad \text{if} \quad k \sharp H, P \quad (\text{LTS}\star)$$

According to Lem. 2.3.10, a transaction name k must occur in process P in order to be modified. Since k is fresh from P by definition of Rule LTS$\star$, no configuration $\mathcal{C}$ can modify $k(\star)$ elements in the history. This section shows that such actions can be substituted with the `ab` element without losing distinguishing power in the bisimulation, and therefore we can avoid introducing them in the first place in Rule LTS$\star$.

The previous observation about the unmodifiable nature of $k(\star)$ actions is formalized as follows:

Lemma 3.1.1. *Let $\mathcal{C} = (H, i \mapsto k(\hat{a}) \rhd P)$ and $k \sharp P$. For any configuration $(H' \rhd P')$ such that $\mathcal{C} \xrightarrow{\zeta} (H' \rhd P')$, $H'(i) = k(\hat{a})$ and $k \sharp P'$.*

Proof. By case analysis on the transition $\mathcal{C} \xrightarrow{\zeta} \mathcal{C}'$. Let $\mathcal{C} = (H \rhd P)$. By Lem. 2.3.10, if P aborts or commits a transaction, that transaction name must occur in the free transaction names of P. Therefore the abort and commit operators will not modify k in the history. If a new transaction l is activated by Rule LTS`new`, l is fresh from the history H by definition of the rule. Since H contains $k(\star)$, l is fresh from k as well, and therefore k is still fresh from P. If P performs any action involving a substitution σ, the domain of σ must be in P, and therefore it does not contain k, which remains unchanged. Since the range of σ is always a fresh variable from $\mathcal{C}$, k will also be fresh from P'. As already mentioned in the introduction, we do not consider ω transitions from Rule LTSω. □

Applied repeatedly, this lemma shows that $k(\star)$ elements in a history H are always fresh from the process P for any configuration $(H \rhd P)$. Because of Lem. 2.3.10, k can never be committed or aborted. Moreover, since k has to be fresh from H as well, all $k(\star)$ elements are unique in H. These properties are summarized in the following well-formedness condition:

Definition 3.1.2 (Star well-formedness)**.** *Let $\mathcal{C}$ be a configuration $(H \rhd P)$, where P is well-formed. A configuration $\mathcal{C}$ is $\star$-well formed when, for any index i and j such that $i \neq j$, if $H(i) = k(\star)$ then $k \sharp P$ and $H(j) \neq k(\star)$.*

The History LTS in Fig. 2.3.5 preserves $\star$-well formedness by Lem. 3.1.1. Because of this, and since two TCCS$^{\text{m}}$ processes P and Q are bisimilar only when they are history bisimilar starting from

$$
\begin{array}{llll}
(H \rhd P) & \xrightarrow{\tau}{}^{\mathtt{sl}}\ (\sigma(H) \rhd Q) & \text{if} \quad P \xrightarrow{\tau}_{\sigma} Q & (\mathrm{SL}\tau) \\
(H \rhd P) & \xrightarrow{\tau}{}^{\mathtt{sl}}\ (\sigma(H) \rhd Q) & \text{if} \quad P \xrightarrow{k(\tau)}_{\sigma} Q \text{ and } k \sharp H & (\mathrm{SL}k(\tau)) \\
(H \rhd P) & \xrightarrow{\tau}{}^{\mathtt{sl}}\ (H \rhd Q) & \text{if} \quad P \xrightarrow{\mathtt{new}\, k} Q \text{ and } k \sharp H & (\mathrm{SL}\mathtt{new}) \\
(H \rhd P) & \xrightarrow{\tau}{}^{\mathtt{sl}}\ (H \setminus_{\mathtt{co}} k \rhd Q) & \text{if} \quad P \xrightarrow{\mathtt{co}\, k} Q & (\mathrm{SL}\mathtt{co}) \\
(H \rhd P) & \xrightarrow{\tau}{}^{\mathtt{sl}}\ (H \setminus_{\mathtt{ab}} k \rhd Q) & \text{if} \quad P \xrightarrow{\mathtt{ab}\, k} Q & (\mathrm{SL}\mathtt{ab}) \\
(H \rhd P) & \xrightarrow{k}{}^{\mathtt{sl}}\ (\sigma(H), k(a) \rhd Q) & \text{if} \quad P \xrightarrow{k(a)}_{\sigma} Q \text{ and } k \sharp H & (\mathrm{SL}k(a)) \\
(H \rhd P) & \xrightarrow{k}{}^{\mathtt{sl}}\ (H, \mathtt{ab} \rhd P) & \text{if} \quad k \sharp H, P & (\mathrm{SL}\star)
\end{array}
$$

Fig. 3.2: Starless LTS.

the empty history, we assume that $\star$-well formedness holds for any configuration we consider in the remainder of this section.

Being uncommittable, a $k(\star)$ action never becomes the permanent action $\star$ in a bisimulation game, and therefore it plays the same role as $\mathtt{ab}$ as far as history consistency is concerned. The following lemma shows that, when a configuration has a $k(\star)$ element at a specific index i, it is bisimilar to itself after replacing $k(\star)$ with $\mathtt{ab}$:

Lemma 3.1.3. *If $\mathcal{C}_1 = (H, i \mapsto k(\star) \rhd P)$ and $\mathcal{C}_2 = (H, i \mapsto \mathtt{ab} \rhd P)$ be $\star$-well formed configurations, then $\mathcal{C}_1 \approx \mathcal{C}_2$.*

Proof. Let $\mathcal{R}$ be the relation between $\star$-well formed configurations defined as follows:

$$\mathcal{R} = \{((H_1 \rhd P), (H_2 \rhd P)) \mid H_1(i) = k(\star) \wedge H_2(i) = \mathtt{ab} \wedge \forall j \neq i. H_1(i) = H_2(i)\}$$

The lemma is proved by showing that $\mathcal{R}$ is a history bisimulation. Condition 1 of history bisimulation holds when H_1 and H_2 are consistent. Since H_1 and H_2 are identical for any index except i, where the fact $H_1(i) = k(\star)$ and $H_2(i) = \mathtt{ab}$. Since $k(\star)$ and $\mathtt{ab}$ are not permanent actions, H_1 and H_2 are consistent and Condition 1 is satisfied.

Condition 2 is satisfied too, because the challenger's configuration is identical to the defender's, except for their histories at index i. Since both configuration contain the same process P, if $(H_1 \rhd P) \xrightarrow{\zeta} (H_1' \rhd P')$ is a challenger move, then $(H_2 \rhd P)$ can perform the same transition to $(H_2' \rhd P')$. Since $H_1'(i) = k(\star)$ by Lem. 3.1.1 and $H_2'(i) = \mathtt{ab}$ by Lem. 2.3.11, then $(H_1' \rhd P')\mathcal{R}(H_2' \rhd P')$ holds and the lemma is proved.

Condition 3 is proved similarly. □

Since $k(\star)$ elements are indistinguishable from $\mathtt{ab}$ ones, this lemma suggests that the History LTS can avoid introducing $k(\star)$ elements in the histories altogether. Therefore we first define the *Starless LTS* in Fig. 3.2, which is identical to the History LTS in Fig. 2.3.5, except for Rule LTS$\star$, which is replaced by the following one:

$$(H \rhd P) \xrightarrow{k} (H, \mathtt{ab} \rhd P) \quad \text{if} \quad k \sharp H, P \quad (\mathrm{SL}\star)$$

Instead of adding a fresh $k(\star)$ element to the history, the Starless LTS adds an $\mathtt{ab}$ element in its place. We define a *challenger move* in the Starless LTS to be any transition not produce by Rule SL$\star$.

The definition of *starless history bisimulation* is identical to the definition of history bisimulation, except for the use of the Starless LTS. Although repetitive, we spell out these definitions for the sake of completeness. The starless history bisimulation is defined as follows:

Definition 3.1.4 (Starless history bisimulation)**.** *A binary relation* $\mathcal{R} \subseteq \mathsf{Conf}_{\mathsf{Act}} \times \mathsf{Conf}_{\mathsf{Act}}$ *is a* starless history bisimulation *when for all* $\mathcal{C}_1\ \mathcal{R}\ \mathcal{C}_2$*:*

1. *hist*$(\mathcal{C}_1)$ *and hist*$(\mathcal{C}_2)$ *are consistent,*
2. *if* $\mathcal{C}_1 \xrightarrow{\zeta}_{\mathtt{sl}} \mathcal{C}_1'$ *is a challenger move and* $\zeta \sharp C_2$ *then* $\exists\ \mathcal{C}_2'$*:* $\mathcal{C}_2 \stackrel{\zeta}{\Rightarrow}_{\mathtt{sl}} \mathcal{C}_2'$ *and* $\mathcal{C}_1'\ \mathcal{R}\ \mathcal{C}_2'$*,*
3. *the converse of the preceding condition.*

We let $\approx_{\mathsf{sl}}$ be the largest starless history bisimulation, and say that two TCCS$^{\mathrm{m}}$ processes P and Q are starless bisimilar, or $P \approx_{\mathsf{sl}} Q$, when $(\emptyset \rhd P) \approx_{\mathsf{sl}} (\emptyset \rhd Q)$ holds. From the definitions, it is easy to see that the result in Lemma 3.1.3 can be extended to starless bisimilar configurations:

Corollary 3.1.5. *If* $\mathcal{C}_1 = (H, i \mapsto k(\star) \rhd P)$ *and* $\mathcal{C}_2 = (H, i \mapsto \mathtt{ab} \rhd P)$ *be* $\star$*-well formed configurations, then* $\mathcal{C}_1 \approx_{\mathsf{sl}} \mathcal{C}_2$*.*

Proof. Similar to Lemma 3.1.3's proof. □

We can now show that history and starless bisimulation relate the same processes:

Theorem 3.1.6 (Starless and History bisimulation correspondence)**.** *Let* $\mathcal{C}_1$ *and* $\mathcal{C}_2$ *be* $\star$*-well formed configurations. If* $\mathcal{C}_1 \approx \mathcal{C}_2$*, then* $\mathcal{C}_1 \approx_{\mathsf{sl}} \mathcal{C}_2$*, and vice versa.*

Proof. In order to prove that $\mathcal{C}_1 \approx \mathcal{C}_2$ implies $\mathcal{C}_1 \approx_{\mathsf{sl}} \mathcal{C}_2$, we first show that $\approx$ is a starless history bisimulation. This implies that $\approx$ is included in $\approx_{\mathsf{sl}}$ because it is the largest starless history bisimulation. The reverse direction can be proved similarly to the forward direction and is therefore omitted.

Condition 1 in Def. 3.1.4 holds by hypothesis, since the histories of $\mathcal{C}_1$ and $\mathcal{C}_2$ are consistent by definition of history bisimulation.

Let us prove Condition 2 of starless history bisimulation. Suppose now that $\mathcal{C}_1 \xrightarrow{\zeta}_{\mathtt{sl}} \mathcal{C}_1'$ is a challenger move, and that $\zeta \sharp \mathcal{C}_2$. This transition can be translated into the History LTS as $\mathcal{C}_1 \xrightarrow{\zeta} \mathcal{C}_1'$, because the history and the starless LTS are identical except for Rule SL$\star$, and it is obviously also a challenger move. By History bisimulation it holds that $\mathcal{C}_2 \stackrel{\zeta}{\Rightarrow} \mathcal{C}_2'$ and $\mathcal{C}_1' \approx \mathcal{C}_2'$.

If $\zeta = \tau$, transition $\mathcal{C}_2 \stackrel{\zeta}{\Rightarrow} \mathcal{C}_2'$ can be directly translated into the Starless LTS as $\mathcal{C}_2 \stackrel{\zeta}{\Rightarrow}_{\mathtt{sl}} \mathcal{C}_2'$, which proves the theorem immediately.

If $\zeta = k$, the move $\mathcal{C}_2 \stackrel{\zeta}{\Rightarrow} \mathcal{C}_2'$ is equivalent to $\mathcal{C}_2 \stackrel{\tau}{\Rightarrow} \mathcal{C}_2^1 \xrightarrow{k} \mathcal{C}_2^2 \stackrel{\tau}{\Rightarrow} \mathcal{C}_2'$ by definition. We need to consider two cases: the case where $\mathcal{C}_2^1 \xrightarrow{k} \mathcal{C}_2^2$ is derived by Rule LTS$k(a)$, and when it is derived by Rule LTS$\star$. If Rule LTS$k(a)$ has been used, the theorem is proved as in the case $\zeta = \tau$. If Rule LTS$\star$ we cannot translate $\mathcal{C}_2 \stackrel{\zeta}{\Rightarrow} \mathcal{C}_2'$ into the Starless LTS, because Rule LTS$\star$ is different from Rule SL$\star$. In this case the theorem is proved by constructing a weak transition $\mathcal{C}_2 \stackrel{\zeta}{\Rightarrow}_{\mathtt{sl}} \mathcal{C}_2''$ such that $\mathcal{C}_1 \approx \mathcal{C}_2''$.

Let $\mathcal{C}_2^1 = (H \rhd P)$ for some H and P; by inversion on Rule LTS$\star$, let $\mathcal{C}_2^2 = (H, k(\star) \rhd P)$. By Rule SL$\star$ $\mathcal{C}_1^2 \xrightarrow{k}_{\mathtt{sl}} \mathcal{C}_2^{2\prime} = (H, \mathtt{ab} \rhd P)$ holds. Configuration $\mathcal{C}_2^{2\prime}$ is history bisimilar to $\mathcal{C}_2^2$ by Lem. 3.1.3. Because they are bisimilar, and since $\mathcal{C}_2^2$ can perform transition $\mathcal{C}_2^2 \stackrel{\tau}{\Rightarrow} \mathcal{C}_2'$, configuration $\mathcal{C}_2^{2\prime}$ can perform the transition $\mathcal{C}_2^{2\prime} \stackrel{\tau}{\Rightarrow} \mathcal{C}_2''$ such that $\mathcal{C}_2' \approx \mathcal{C}_2''$. As in the case for $\zeta = \tau$, transition $\mathcal{C}_2^{2\prime} \stackrel{\tau}{\Rightarrow} \mathcal{C}_2''$

can be translated straightforwardly into the Starless LTS. Since $\mathcal{C}_1' \approx \mathcal{C}_2'$ and $\mathcal{C}_2' \approx \mathcal{C}_2''$, by transitivity $\mathcal{C}_1' \approx \mathcal{C}_2''$ holds as well, which proves the theorem.

Condition 3 is proved similarly to Condition 2, by using Cor. 3.1.5 instead of Lem. 3.1.3. □

We conclude this section by remarking that, given a configuration $(H \rhd P)$, the Starless LTS never introduces transaction names fresh from P in H . Thanks to this property, we can assume that for any configuration $(H \rhd P)$, the set of free variables of H is a subset of the set of free variables of P. This well-formedness property is defined as follows:

Definition 3.1.7 (Configuration well-formedness)**.** *A configuration $\mathcal{C} = (H \rhd P)$ is well-formed when P is well-formed, and when $ftn(H) \subseteq ftn(P)$.*

3.2 Abort configurations

In order to further simplify the starless history bisimulation, we first need to develop some technical devices to deal with uncommittable transactions. In particular, the goal of this section is to prove that uncommittable transactions do not produce any observable behaviour in a bisimulation game, and therefore they can be safely aborted without altering the distinguishing power over a configuration.

To this end, we introduce *abort configurations*. The abort configuration $\mathcal{C} \setminus_{\mathsf{ab}} k$ is the resulting configuration after aborting transaction k in $\mathcal{C}$, if such a transaction exists. :

Definition 3.2.1 (Abort configuration)**.** *Let $\mathcal{C}$ be a well-formed configuration $(H \rhd P)$. The abort configuration $\mathcal{C} \setminus_{\mathsf{ab}} k$ is $(H \setminus_{\mathsf{ab}} k \rhd P')$ when $P \xrightarrow{\mathtt{ab}\ k} P'$ is defined.*

We now state some useful properties of $\mathrm{TCCS}^{\mathrm{m}}$ processes to reason about abort configurations. If Q is the process obtained by aborting k in P, then Q can simulate any α transition in P:

Lemma 3.2.2. *Let $P \xrightarrow{\alpha}_{\sigma} P'$ and $P \xrightarrow{\mathtt{ab}\ k} Q$. Then:*

1. *If $\sigma = [k \mapsto l]$, then $P' \xrightarrow{\mathtt{ab}\ l} Q$.*
2. *If $\sigma = [k, k' \mapsto l]$, then $Q \xrightarrow{\mathtt{ab}\ k'} Q'$ and $P' \xrightarrow{\mathtt{ab}\ l} Q'$.*
3. *If $k \notin dom(\sigma)$, then $Q \xrightarrow{\alpha}_{\sigma} Q'$ and $P' \xrightarrow{\mathtt{ab}\ k} Q'$.*

Proof. By rule induction on $P \xrightarrow{\alpha}_{\sigma} P'$. □

When P performs a β action, process Q can simulate it only if β does not involve k, because obviously transaction k has been aborted:

Lemma 3.2.3. *Let $P \xrightarrow{\beta} P', k \sharp \beta$ and $P \xrightarrow{\mathtt{ab}\ k} Q$. Then $Q \xrightarrow{\beta} Q'$ and $P' \xrightarrow{\mathtt{ab}\ k} Q'$.*

Proof. By rule induction on $P \xrightarrow{\beta} P'$. □

Substitutions distribute over the history abort operator $\setminus_{\mathsf{ab}} k$ as follows:

Lemma 3.2.4. *For any history H and substitution σ such that $rg(\sigma) \sharp H$:*

1. *if $\sigma = [k, k' \mapsto l]$ then $H \setminus_{\mathsf{ab}} k \setminus_{\mathsf{ab}} k' = \sigma(H) \setminus_{\mathsf{ab}} l$.*

2. if $\sigma = [k \mapsto l]$*, then* $H \setminus_{\mathtt{ab}} k = \sigma(H) \setminus_{\mathtt{ab}} l$*.*

Proof. We only prove the Property 1; the second property is proved similarly. Take an index i such that $H(i)$ is defined. There are two cases to consider: either $k, k' \sharp H(i)$, or either k or k' occurs in $H(i)$. If $k, k' \sharp H(i)$, then $l \sharp \sigma(H(i))$. By definition, $H(i) \setminus_{\mathtt{ab}} k \setminus_{\mathtt{ab}} k' = H(i)$ and $\sigma(H) \setminus_{\mathtt{ab}} l = H \setminus_{\mathtt{ab}} l = H$, and therefore the lemma is proved. If either k or k' occurs in $H(i)$, then $H(i) = k(a)$ or $H(i) = k'(\hat{a})$. In either case, $H \setminus_{\mathtt{ab}} k \setminus_{\mathtt{ab}} k' = \mathtt{ab}$ and $\sigma(H(i)) \setminus_{\mathtt{ab}} l = l(\hat{a}) \setminus_{\mathtt{ab}} l = \mathtt{ab}$, and the lemma is proved. □

These lemmas enable us to prove that, for any ζ fresh from k, a configuration $\mathcal{C}$ and its abort configuration $\mathcal{C} \setminus_{\mathtt{ab}} k$ are bisimilar. In order to do this, we must show that the abort configuration can simulate its source configuration. This property is a direct consequence of the more general lemma:

Lemma 3.2.5. *Let* k *occur in a well-formed configuration* $\mathcal{C}$*. If* $\mathcal{C} \xrightarrow{\zeta}_{\mathtt{sl}} \mathcal{C}'$ *is not derived by Rule* SL$\mathtt{co}$ *committing* k*, then either* $\mathcal{C}' = \mathcal{C} \setminus_{\mathtt{ab}} k$*, or there exists a transaction name* l *such that* l *occurs in* $\mathcal{C}'$ *and* $\mathcal{C} \setminus_{\mathtt{ab}} k \xrightarrow{\zeta}_{\mathtt{sl}} \mathcal{C}' \setminus_{\mathtt{ab}} l$*.*

Proof. The proof is by rule induction on $\mathcal{C} \xrightarrow{\zeta}_{\mathtt{sl}} \mathcal{C}'$. In each case, we will assume that $P \xrightarrow{\mathtt{ab}\ k} Q$ is defined, since k occurs in $\mathcal{C}$ and $\mathcal{C}$ is well-formed.

Suppose that Rule SLτ has been applied:

$$(H \rhd P) \quad \xrightarrow{\tau}_{\mathtt{sl}} (\sigma(H) \rhd P') \quad \text{if} \quad P \xrightarrow{\tau}_{\sigma} P' \quad (\text{SL}\tau)$$

and let $(H \rhd P) \setminus_{\mathtt{ab}} k = (H \setminus_{\mathtt{ab}} k \rhd Q)$. Then the following derivations hold:

1.	$P \xrightarrow{\tau}_{\sigma} P'$	by hypothesis
2.	$\sigma = \epsilon$	by inversion on the LTS
3.	$Q \xrightarrow{\tau}_{\sigma} Q'$	by Prop. 3 of Lem. 3.2.2 on (1)
4.	$P' \xrightarrow{\mathtt{ab}\ k} Q'$	”
5.	$(H \setminus_{\mathtt{ab}} k \rhd Q) \xrightarrow{\tau}_{\mathtt{sl}} (\sigma(H \setminus_{\mathtt{ab}} k) \rhd Q')$	by Rule SLτ on (3)
6.	$(\sigma(H \setminus_{\mathtt{ab}} k) \rhd Q') = (\sigma(H) \setminus_{\mathtt{ab}} k \rhd Q')$	because $\sigma = \epsilon$ by (1)
7.	$(\sigma(H) \setminus_{\mathtt{ab}} k \rhd Q') = (\sigma(H) \rhd P') \setminus_{\mathtt{ab}} k$	by def. of abort configuration

By inversion on transition $P \xrightarrow{\tau}_{\sigma} P'$ in the LTS from Fig. 2.1 (1), substitution σ can only be the empty substitution ϵ when a TCCS$^{\mathrm{m}}$ process performs a τ action (2). By Property 3 of Lemma 3.2.2, the abort configuration $(H \setminus_{\mathtt{ab}} k \rhd Q)$ can transition to $(\sigma(H \setminus_{\mathtt{ab}} k) \rhd Q')$ by Rule SLτ via a τ action (3, 5). Since Q' is the result of aborting k from P' and σ is ϵ, the final abort configuration $(H \setminus_{\mathtt{ab}} k \rhd Q')$ is equivalent to $(\sigma(H) \rhd P') \setminus_{\mathtt{ab}} k$. Therefore the lemma is proved by taking $l = k$ and by the transition $(H \rhd P) \setminus_{\mathtt{ab}} k \xrightarrow{\tau} (H \rhd P') \setminus_{\mathtt{ab}} k$.

Suppose that Rule SL$k(\tau)$ has been applied:

$$(H \rhd P) \quad \xrightarrow{\tau}_{\mathtt{sl}} (\sigma(H) \rhd P') \quad \text{if} \quad P \xrightarrow{l(\tau)}_{\sigma} P' \text{ and } l \sharp H \quad (\text{SL}k(\tau))$$

and let $(H \rhd P) \setminus_{\mathtt{ab}} k = (H \setminus_{\mathtt{ab}} k \rhd Q)$. If $k \notin dom(\sigma)$, the lemma is proved using Property 3 of Lem. 3.2.2 as in the first case. If $k \in dom(\sigma)$, then by inversion on the transition $P \xrightarrow{l(\tau)}_{\sigma} P'$, σ is

the substitution $[k, k' \mapsto l]$ for some k'. By Property 2 of Lem. 3.2.2, $P' \xrightarrow{\mathtt{ab}\ l} Q'$ and $Q \xrightarrow{\mathtt{ab}\ k'} Q'$. both hold. This case is proved as in the first case, using Lem. 3.2.4 to show that $H \setminus_{\mathtt{ab}} k \setminus_{\mathtt{ab}} k \setminus_{\mathtt{ab}} k' = \sigma(H) \setminus_{\mathtt{ab}} l$.

Suppose that Rule SL$k(a)$ is used:

$$(H \rhd P) \quad \xrightarrow{l}{}^{\mathtt{sl}} (\sigma(H), l(a) \rhd P') \quad \text{if} \quad P \xrightarrow{l(a)}_{\sigma} P' \text{ and } l \sharp H \quad (\text{SL}k(a))$$

and let $(H \rhd P) \setminus_{\mathtt{ab}} k = (H \setminus_{\mathtt{ab}} k \rhd Q)$. If $k \notin dom(\sigma)$, then the lemma is proved by Property 3 of Lem. 3.2.2 as in the first case.

If $k \in dom(\sigma)$, then the following derivations hold:

1. $\sigma = [k \mapsto l]$ — by inversion on the LTS
2. $P' \xrightarrow{\mathtt{ab}\ l} Q$ — by Property 1 of Lem. 3.2.2
3. $(\sigma(H), l(a) \setminus_{\mathtt{ab}} l \rhd P') \xrightarrow{\tau} (\sigma(H) \setminus_{\mathtt{ab}} l, \mathtt{ab} \rhd Q)$ — by Rule SL$\mathtt{ab}$ on (2)
4. $(H \setminus_{\mathtt{ab}} k \rhd Q) \xrightarrow{l}{}^{\mathtt{sl}} (H \setminus_{\mathtt{ab}} k, \mathtt{ab} \rhd Q)$ — by Rule SL$\star$
5. $H \setminus_{\mathtt{ab}} k = \sigma(H) \setminus_{\mathtt{ab}} l$ — by Prop. 2 of Lem. 3.2.4
6. $(H \setminus_{\mathtt{ab}} k \rhd Q) \xrightarrow{l}{}^{\mathtt{sl}} (\sigma(H) l(a) \setminus_{\mathtt{ab}} l \rhd Q)$ — by (4) and (6)
7. $\mathcal{C} \setminus_{\mathtt{ab}} k \xrightarrow{l}{}^{\mathtt{sl}} \mathcal{C}' \setminus_{\mathtt{ab}} l$ — by def. of abort configuration

By inversion on the premises of Rule SL$k(a)$, and since k is in the domain of σ, then σ must be $[k \mapsto l]$ (1). On the one hand, configuration $(\sigma(H), l(a) \setminus_{\mathtt{ab}} l \rhd P')$ can abort l and transition to $\xrightarrow{\tau} (\sigma(H) \setminus_{\mathtt{ab}} l, \mathtt{ab} \rhd Q)$ (3), where Q is the same process that P transitions to after aborting k by Lem. 3.2.2 (2). On the other hand the abort configuration cannot produce an $l(a)$ action, but it can store an $\mathtt{ab}$ action in its history by Rule SL$\star$ (4) without modifying process Q. By Lemma 3.2.4 the resulting history $H \setminus_{\mathtt{ab}} k$ is equivalent to $\sigma(H) \setminus_{\mathtt{ab}} l$ (5). Since $\mathtt{ab} = l(a) \setminus_{\mathtt{ab}} l$ holds by definition of abort operator $\setminus_{\mathtt{ab}} l$, the previous transition can be rewritten as $(H \setminus_{\mathtt{ab}} k \rhd Q) \xrightarrow{l}{}^{\mathtt{sl}} (\sigma(H) l(a) \setminus_{\mathtt{ab}} l, \mathtt{ab} \rhd Q)$, which proves the lemma (7).

When Rule SL$\star$ is used, the lemma is proved by transition $(H \rhd P) \setminus_{\mathtt{ab}} k \xrightarrow{\tau} (H\mathtt{ab} \rhd P) \setminus_{\mathtt{ab}} k$ by Rule SL$\star$, and by taking $l = k$.

If Rule SL$\mathtt{new}$ is used, then the lemma is proved as in the first case by Lem. 3.2.3, because the premise $l \sharp \mathcal{C}$ implies $k \sharp \mathtt{new}\ l$. If Rule SL$\mathtt{ab}$ is used, then either k or another transaction $l \neq k$ is aborted. In the first case the initial configuration $\mathcal{C}$ transitions to its abort configuration $\mathcal{C} \setminus_{\mathtt{ab}} k$, which proves the lemma immediately. In the second case the lemma follows by Lem. 3.2.3. If Rule SL$\mathtt{co}$ is used, the lemma follows by Lem. 3.2.3, because k is not committed by hypothesis.

□

Corollary 3.2.6. *Let k be uncommittable in $\mathcal{C}$. If $\mathcal{C} \xrightarrow{\zeta}{}^{\mathtt{sl}} \mathcal{C}'$, then either $\mathcal{C}' = \mathcal{C} \setminus_{\mathtt{ab}} k$ and $\zeta = \tau$, or there exists a transaction name l such that l is uncommittable in $\mathcal{C}'$ and $\mathcal{C} \setminus_{\mathtt{ab}} k \xrightarrow{\zeta}{}^{\mathtt{sl}} \mathcal{C}' \setminus_{\mathtt{ab}} l$.*

Proof. The lemma follows by Lem. 3.2.5, because k remains uncommittable in any successive configuration $\mathcal{C}'$ by definition. □

We conclude this section by showing that a configuration is starless history bisimilar to its abort k configuration, when k is uncommittable:

Proposition 3.2.7. *Let $\mathcal{C}$ be well-formed. If k is uncommittable in $\mathcal{C}$, then $\mathcal{C} \approx_{\mathsf{sl}} \mathcal{C} \setminus_{\mathsf{ab}} k$.*

Proof. By coinduction. Consider the following relation $\mathcal{R}$:

$$\mathcal{R} = \{(\mathcal{C}_1, \mathcal{C}_2) \mid \exists k.k \text{ is uncommittable in } \mathcal{C}_1 \text{ and } \mathcal{C}_2 = \mathcal{C}_1 \setminus_{\mathsf{ab}} k\} \cup \mathcal{I}d$$

where $\mathcal{I}d$ is the identity relation over configurations. The lemma is proved if we show that $\mathcal{R}$ is a starless history bisimulation, i.e. if the the three conditions of Def. 3.1.4 hold for $\mathcal{R}$.

When $\mathcal{R}$ relates two identical configurations because $\mathcal{I}d$, the proof is trivial. Suppose that $\mathcal{R}$ relates a configuration $\mathcal{C}$ and its abort configuration $\mathcal{C} \setminus_{\mathsf{ab}} k$, with k uncommittable in $\mathcal{C}$. Let $\mathcal{C} = (H \rhd P)$ and let $\mathcal{C} \setminus_{\mathsf{ab}} k = (H \setminus_{\mathsf{ab}} k \rhd Q)$. Since the $\setminus_{\mathsf{ab}} k$ operator does not modify permanent actions, the two histories H and $H \setminus_{\mathsf{ab}} k$ share the same permanent actions at the same indexes, and therefore Condition 1 holds.

Let $\mathcal{C}_1 \xrightarrow{\zeta} \mathcal{C}_1'$ be a challenger move and let $\zeta \sharp \mathcal{C}_1$. By Cor. 3.2.6, either $\mathcal{C}_1 = \mathcal{C}_1 \setminus_{\mathsf{ab}} k$ and $\zeta = \tau$, or $\mathcal{C}_1 \setminus_{\mathsf{ab}} k \xrightarrow{\zeta}_{\mathtt{sl}} \mathcal{C}_1' \setminus_{\mathsf{ab}} l$ and l is uncommittable in $\mathcal{C}_1'$. In the former case, the reflexive transition $\mathcal{C}_1 \setminus_{\mathsf{ab}} k \stackrel{\zeta}{\Rightarrow}_{\mathtt{sl}} \mathcal{C}_1 \setminus_{\mathsf{ab}} k$ proves the lemma, because $\mathcal{C}_1'$ and $\mathcal{C}_1 \setminus_{\mathsf{ab}} k$ are identical, and therefore related by $\mathcal{R}$ because of $\mathcal{I}d$. In the latter case, $\mathcal{R}$ relates $\mathcal{C}_1'$ and $\mathcal{C}_1' \setminus_{\mathsf{ab}} l$ by definition, because l is uncommittable in $\mathcal{C}_1'$. Therefore Condition 2 holds.

In order to prove Condition 3, suppose that $\mathcal{C}_2 \xrightarrow{\zeta} \mathcal{C}_2'$ is a challenger move. Since $\mathcal{C}_2 = \mathcal{C}_1 \setminus_{\mathsf{ab}} k$, $\mathcal{C}_1$ can abort k, become the same configuration as $\mathcal{C}_1$ and perform the same transition from $\mathcal{C}_2$ to $\mathcal{C}_2'$. Relation $\mathcal{R}$ relates $\mathcal{C}_1'$ and $\mathcal{C}_2'$ because the two configuration are identical, and therefore Condition 3 holds. □

3.3 Uniform history bisimulation

After dispensing with $k(\star)$ elements, this section presents an LTS and a notion of bisimulation that relates configurations with a stronger notion than history consistency, called *action consistency*, that is histories that match not only in their permanent actions, but also in their tentative ones.

Recall the example from the introduction, whereby configuration $\mathcal{C}_1 = (\emptyset \rhd [\![a.P \rhd_k 0]\!])$ is the challenger in a starless history bisimulation game, and configuration $\mathcal{C}_2 = (\emptyset \rhd [\![b.Q \rhd_l 0]\!])$ is the defender. If the challenger attacks by firing a tentative action $k'(a)$, the defender can respond by either firing $k'(b)$, by storing a dummy $\mathtt{ab}$ action by Rule SL$\star$ or by aborting l and then storing a dummy $\mathtt{ab}$ action. The resulting configurations will be $\mathcal{C}_1' = (k'(a) \rhd P)$ and either $\mathcal{C}_2' = (k'(b) \rhd Q), (\mathtt{ab} \rhd b.Q)$ or $(\mathtt{ab} \rhd 0)$. The defender loses regardless of the option it selects, because the challenger can commit k' and their history become inconsistent in all three cases.

This observation suggests that bisimilar configurations must have matching tentative actions $k(a)$ in their histories, as long as k can be committed:

Lemma 3.3.1. *Let $\mathcal{C}_1 = (H_1, i \mapsto k(a) \rhd P)$ and $\mathcal{C}_2 = (H_2, i \mapsto l(b) \rhd Q)$ for some index i, and let $\mathcal{C}_1 \approx \mathcal{C}_2$. If k is committable in $\mathcal{C}_1$, then $a = b$.*

Proof. A transaction k is committable in a configuration $\mathcal{C}$ if there exists a sequence of transitions $\mathcal{C} \xrightarrow{\zeta_1} \ldots \xrightarrow{\zeta_n} \mathcal{C}'$ such that, for some index i and action a, $hist(\mathcal{C})(i) = k(\hat{a})$, and $hist(\mathcal{C}')(i) = \hat{a}$. Let us take one such sequence of transitions from $\mathcal{C}_1$ to a configuration $\mathcal{C}_1'$. Since $\mathcal{C}_1$ and $\mathcal{C}_2$ are bisimilar, there exists another sequence of weak transitions $\mathcal{C}_2 \stackrel{\zeta_1}{\Longrightarrow} \ldots \stackrel{\zeta_n}{\Longrightarrow} \mathcal{C}_2'$ such that $\mathcal{C}_1' \approx \mathcal{C}_2'$. By definition of history bisimulation, the histories of the two final configurations $\mathcal{C}_1'$ and $\mathcal{C}_2'$ are consistent. Since configuration $\mathcal{C}_1'$ has permanent action a at index i by hypothesis, $\mathcal{C}_2'$ must have the same permanent action a at index i by definition of consistency. Thus the action b at index i in the history of $\mathcal{C}_2'$ must be equal to a, i.e. $b = a$, which proves the lemma. □

Conversely, if two configurations are bisimilar but their histories do not match for some $k(a)$ action, then k must be uncommittable:

Corollary 3.3.2. *Let $\mathcal{C}_1 = (H_1, i \mapsto k(a) \rhd P)$ and $\mathcal{C}_2 = (H_2, i \mapsto l(b) \rhd Q)$ for some index i, and let $\mathcal{C}_1 \approx \mathcal{C}_2$. If $a \neq b$, then k is uncommittable in $\mathcal{C}_1$.*

Proof. This result follows immediately by negating the logical implication in Lemma 3.3.1. □

The first result suggests that if the challenger attacks with a committable k action, then the defender must respond by storing the same $k(a)$ as the attacker in order to win the game. The second result suggest that if the defender can respond with a $k(b)$ action and win the game, then k must be uncommittable. However, when a transaction k is uncommittable, tentative actions are never checked for consistency in the histories, because they never become permanent. As shown in Section 3.2, there is no observable difference between an uncommittable transaction and an aborted one, therefore there is no harm in forcing the responder to use Rule SLab when it cannot produce the same action $k(a)$

These considerations suggest the notion of *uniform bisimulation*, where the defender's range of responses is restricted to either firing the same action $k(a)$ as the attacker, or to fire a dummy ab action without loss of generality. By construction, the histories in a uniform history bisimulation game cannot differ by action names. Such histories enjoy a stronger property than consistency, called *action consistency*:

Definition 3.3.3 (Action consistency)**.** *H_1 and H_2 are* action consistent *when they have the same domain and for all $i \in I$, $a \in \mathsf{Act}$:*

- *$H_1(i) = a$ iff $H_2(i) = a$.*
- *if $H_1(i) = k(a)$ and $H_2(i) = l(b)$, then $a = b$*

In summary, this definition implies that whenever two action consistent histories H_1 and H_2 contain an action at the same index i, they both contain the same action name a up to transaction names k and l.

The Uniform History LTS is presented in Fig. 3.3. Transitions have the form $\mathcal{C} \xrightarrow{\xi}{}^{\mathtt{u}}_{\sigma} \mathcal{C}'$, where $\xi ::= \tau \mid k(a)$. Therefore action names are now exposed, unlike the k labels in the history and starless History LTS. We define $\stackrel{\zeta}{\Rightarrow}^{\mathtt{u}}$ to be $\xrightarrow{\tau}{}^{\mathtt{u}^*}$ when $\zeta = \tau$, and $\stackrel{\tau}{\Rightarrow}^{\mathtt{u}} \xrightarrow{\zeta}{}^{\mathtt{u}} \stackrel{\tau}{\Rightarrow}^{\mathtt{u}}$ otherwise. A challenger move is any transition not produced by Rule Un-$\star$.

Apart from extending ζ labels, the LTS rules are very similar to the starless History LTS rules from Fig. 3.2. However, note that the freshness condition $k \sharp H$ has been dropped in Rules Un-$k(\tau)$,

$$
\begin{array}{llll}
(H \rhd P) & \xrightarrow{\tau}{}^{\mathtt{u}} (\sigma(H) \rhd Q) & \text{if} \quad P \xrightarrow{\tau}_{\sigma} Q & (\textsc{Un}\text{-}\tau) \\
(H \rhd P) & \xrightarrow{\tau}{}^{\mathtt{u}} (\sigma(H) \rhd Q) & \text{if} \quad P \xrightarrow{k(\tau)}_{\sigma} Q & (\textsc{Un}\text{-}k(\tau)) \\
(H \rhd P) & \xrightarrow{\tau}{}^{\mathtt{u}} (H \rhd Q) & \text{if} \quad P \xrightarrow{\mathtt{new}\, k} Q & (\textsc{Un}\text{-}\mathtt{new}) \\
(H \rhd P) & \xrightarrow{\tau}{}^{\mathtt{u}} (H \setminus_{\mathtt{co}} k \rhd Q) & \text{if} \quad P \xrightarrow{\mathtt{co}\, k} Q & (\textsc{Un}\text{-}\mathtt{co}) \\
(H \rhd P) & \xrightarrow{\tau}{}^{\mathtt{u}} (H \setminus_{\mathtt{ab}} k \rhd Q) & \text{if} \quad P \xrightarrow{\mathtt{ab}\, k} Q & (\textsc{Un}\text{-}\mathtt{ab}) \\
(H \rhd P) & \xrightarrow{k(a)}{}^{\mathtt{u}} (\sigma(H), k(a) \rhd Q) & \text{if} \quad P \xrightarrow{k(a)}_{\sigma} Q & (\textsc{Un}\text{-}k(a)) \\
(H \rhd P) & \xrightarrow{k(a)}{}^{\mathtt{u}} (H, \mathtt{ab} \rhd P) & \text{if} \quad k \sharp P & (\textsc{Un}\text{-}\star)
\end{array}
$$

Fig. 3.3: Uniform History LTS.

Un-**new**, Un-$k(a)$ and Un-$\star$. Thanks to configuration well-formedness, this check is now unnecessary. In fact, TCCS$^{\mathrm{m}}$ transitions $P \xrightarrow{\alpha}_{\sigma} Q$ and $P \xrightarrow{\beta} Q$ always generate names fresh from P. Since the set of transaction names in a history H is a subset of the transaction names in a process P by Def. 3.1.7, if $k \sharp P$ for some k in a well-formed configuration $(H \rhd P)$, then $k \sharp H$ as well.

Therefore we can establish the following correspondence between transitions in the uniform and Starless LTSs:

Lemma 3.3.4. *Let $\mathcal{C}$ be a well-formed configuration. Then:*

1. *$\mathcal{C} \xrightarrow{\tau} \mathcal{C}'$ if and only if $\mathcal{C} \xrightarrow{\tau}{}^{\mathtt{u}} \mathcal{C}'$.*
2. *$(H \rhd P) \xrightarrow{k} (H, k(a) \rhd Q)$ if and only if $(H \rhd P) \xrightarrow{k(a)}{}^{\mathtt{u}} (H, k(a) \rhd Q)$.*
3. *$(H \rhd P) \xrightarrow{k} (H, \mathtt{ab} \rhd P')$ if and only if $(H \rhd P) \xrightarrow{k(a)}{}^{\mathtt{u}} (H, \mathtt{ab} \rhd Q)$.*

Proof. By rule induction. Except for Rule SL$\star$ and Rule Un-$\star$, the inductive hypothesis provides a transition in the TCCS$^{\mathrm{m}}$ LTS of the form $P \xrightarrow{\alpha} \sigma Q$ or $P \xrightarrow{\beta} Q$. This transition alone is sufficient to prove the lemma in both directions when a transaction is aborted, committed, or a τ action is performed; it is also sufficient in the remaining cases when translating Starless LTS transitions into the Uniform History LTS. In the opposite direction, the Starless LTS requires some extra freshness conditions of the kind $k \sharp H$. Since $ftn(H) \subseteq ftn(P)$ by definition of well-formed configuration. these conditions hold by Lemma 2.1.2: namely that for any transition $P \xrightarrow{\alpha}_{\sigma} Q$ it holds that $\alpha, rg(\sigma) \sharp P$; and for any transition $P \xrightarrow{\mathtt{new}\, k} Q$, $k \sharp P$ holds. When translating Rule Un-$\star$ into Rule SL$\star$, the condition $k \sharp H$ is direct consequence of configuration well-formedness as well. □

Uniform history bisimulation is defined as follows:

Definition 3.3.5 (Uniform history bisimulation)**.** *A binary relation $\mathcal{R} \subseteq \mathsf{Conf}_{\mathsf{Act}} \times \mathsf{Conf}_{\mathsf{Act}}$ is a* uniform history bisimulation *when for all $\mathcal{C}_1 \mathcal{R} \mathcal{C}_2$:*

1. *$hist(\mathcal{C}_1)$ and $hist(\mathcal{C}_2)$ are action consistent,*
2. *if $\mathcal{C}_1 \xrightarrow{\zeta}{}^{\mathtt{u}} \mathcal{C}_1'$ is a challenger move and $\zeta \sharp \mathcal{C}_2$, then there exists $\mathcal{C}_2'$ such that $\mathcal{C}_2 \overset{\zeta}{\Rightarrow}{}^{\mathtt{u}} \mathcal{C}_2'$ and $\mathcal{C}_1' \mathcal{R} \mathcal{C}_2'$,*
3. *the converse of the preceding condition.*

We write $\approx_{\mathsf{un}}$ for the largest uniform history bisimulation. Two TCCS$^{\mathrm{m}}$ processes are uniform history bisimilar, or $P \approx_{\mathsf{un}} Q$, when $(\emptyset \rhd P) \approx_{\mathsf{un}} (\emptyset \rhd Q)$ holds.

By definition, participants in a uniform history bisimulation game must play matching $k(a)$ actions. By definition of Uniform History LTS and of challenger move, this means that if the challenger stores a $k(a)$ action, then the defender either stores $k(a)$ by Rule Un-$k(a)$ or $\mathtt{ab}$ by Rule Un-$\mathtt{ab}$. Because of this, participants in a uniform history bisimulation game preserve action consistency across histories by construction.

Starless bisimulation and uniform history bisimulation relate the same processes. In order to prove this, we must show that one bisimulation implies the other, and viceversa. Before proving this, we need a preliminary lemma to show that, for any challenger k move from $\mathcal{C}$ to $\mathcal{C}'$ in the History LTS, the same configuration can always transition to the abort configuration $\mathcal{C}' \setminus_{\mathsf{ab}} k$ in the extended History LTS.

Lemma 3.3.6. *Let $\mathcal{C}$ be a configuration. If $\mathcal{C} \xrightarrow{k} \mathcal{C}'$, then $\mathcal{C} \stackrel{k(a)}{\Longrightarrow}{}^{\mathtt{u}} \mathcal{C}' \setminus_{\mathsf{ab}} k$.*

Proof. The lemma is proved by constructing a weak transition $\mathcal{C} \stackrel{k(a)}{\Longrightarrow}{}^{\mathtt{u}} \mathcal{C}''$ that satisfies the lemma. Assuming that configurations are well-formed, the lemma follows by aborting k and by performing a $k(a)$ action by Rule Un-$\star$. □

We can now show that history bisimulation implies extended history bisimulation:

Lemma 3.3.7. *Let $\mathcal{R} \subseteq \mathsf{Conf}_{\mathsf{Act}} \times \mathsf{Conf}_{\mathsf{Act}}$ be a relation defined as follows:*

$$\mathcal{R} = \{\, ((H_1 \rhd P_1), (H_2 \rhd P_2)) \mid H_1 \text{ and } H_2 \text{ are action consistent, and } (H_1 \rhd P_1) \approx_{\mathsf{sl}} (H_2 \rhd P_2) \,\}$$

Relation $\mathcal{R}$ is a uniform history bisimulation.

Proof. A relation $\mathcal{R}$ over history configurations is an extended weak bisimulation when it satisfies the Condition 1 to 3 from Def. 3.3.5. Therefore we need to show that starless bisimulation satisfies these conditions. Condition 1 holds by definition of $\mathcal{R}$. In order to verify Condition 2, suppose that $\mathcal{C}_1 \xrightarrow{\xi}{}^{\mathtt{u}} \mathcal{C}_1'$ is a challenger move. We must show that $\mathcal{C}_2 \stackrel{\xi}{\Rightarrow}{}^{\mathtt{u}} \mathcal{C}_2'$ and $\mathcal{C}_1' \;\mathcal{R}\; \mathcal{C}_2'$. The proof follows by case analysis on label ξ

($\xi = \tau$): By Lemma 3.3.4 $\mathcal{C}_1 \xrightarrow{\tau}{}^{\mathtt{sl}} \mathcal{C}_1'$ is challenger move in the Starless LTS too. Since $\mathcal{C}_1 \approx_{\mathsf{sl}} \mathcal{C}_2$ holds by definition of $\mathcal{R}$, there exists a configuration $\mathcal{C}_2'$ such that $\mathcal{C}_2 \stackrel{\tau}{\Rightarrow}{}^{\mathtt{sl}} \mathcal{C}_2'$ and $\mathcal{C}_1' \approx_{\mathsf{sl}} \mathcal{C}_2'$. By repeated application of Lemma 3.3.4, configuration $\mathcal{C}_2$ can transition to $\mathcal{C}_2'$ in the uniform LTS as well (i.e. $\mathcal{C}_2 \stackrel{\tau}{\Rightarrow}{}^{\mathtt{u}} \mathcal{C}_2'$). Since the Uniform History LTS does not modify existing actions in an history, the histories in $\mathcal{C}_1'$ and $\mathcal{C}_2'$ are action consistent, and therefore $\mathcal{C}_1' \;\mathcal{R}\; \mathcal{C}_2'$ holds, which proves the lemma.

($\xi = k(a)$): By Lemma 3.3.4, $\mathcal{C}_1 \xrightarrow{k}{}^{\mathtt{sl}} \mathcal{C}_1'$ is a challenger move too, with $\mathcal{C}_1' = (\sigma_1(H_1)k(a) \rhd P')$ for some process P' and substitution σ_1. Since $\mathcal{C}_1$ and $\mathcal{C}_2$ are starless history bisimilar by hypothesis, it follows that $\mathcal{C}_2 \stackrel{k}{\Rightarrow}{}^{\mathtt{sl}} \mathcal{C}_2'$ and $\mathcal{C}_1' \approx \mathcal{C}_2'$ both hold. By definition of weak transition, $\mathcal{C}_2 \stackrel{k}{\Rightarrow}{}^{\mathtt{sl}} \mathcal{C}_2'$ can be written as $\mathcal{C}_2 \stackrel{\tau}{\Rightarrow}{}^{\mathtt{sl}} \mathcal{C}_2^1 \xrightarrow{k}{}^{\mathtt{sl}} \mathcal{C}_2^2 \stackrel{\tau}{\Rightarrow}{}^{\mathtt{sl}} \mathcal{C}_2'$. Let $\mathcal{C}_2^1 = (H_2^1 \rhd P_2^1)$. Transition $\mathcal{C}_2^1 \xrightarrow{k}{}^{\mathtt{sl}} \mathcal{C}_2^2$ can only be derived in one of the following three cases:

1. $\mathcal{C}_2^2 = (\sigma_2(H_2^1)k(a) \rhd P_2^2)$ by rule SL$k(a)$
2. $\mathcal{C}_2^2 = (\sigma_2(H_2^1), i \mapsto k(b) \rhd P_2^2)$, with $a \neq b$, by rule SL$k(a)$
3. $\mathcal{C}_2^2 = (\sigma_2(H_2^1)\mathtt{ab} \rhd P_2^2)$ by rule SL$\star$

In the first and third case, transition $\mathcal{C}_2 \stackrel{k(a)}{\Longrightarrow}{}^{\mathtt{u}} \mathcal{C}_2'$ can be derived by repeated application of Lemma 3.3.4. Since the Uniform History LTS preserves actions in the history, and since either $k(a)$ or $\mathtt{ab}$ are added to the history of $\mathcal{C}_2^2$, the histories of $\mathcal{C}_2'$ and $\mathcal{C}_1'$ are action consistent. Therefore $\mathcal{C}_1' \mathcal{R} \mathcal{C}_2'$ holds and the lemma is proved.

In the second case, configuration $\mathcal{C}_2^1$ can perform the transition $\mathcal{C}_2^1 \stackrel{\tau}{\rightarrow}{}^{\mathtt{sl}} \mathcal{C}_2^1 \setminus_{\mathtt{ab}} k$ by aborting k. Notice that no transition in $\mathcal{C}_2^2 \stackrel{\tau}{\Rightarrow}{}^{\mathtt{sl}} \mathcal{C}_2'$ can commit k, because $\mathcal{C}_1' \approx_{\mathsf{sl}} \mathcal{C}_2'$ implies that H_1' and H_2' are consistent: since $H_1'(i) = k(a)$ by hypothesis, either $H_2' = \mathtt{ab}$ or $H_2'(i) = l(b)$ for some l hold, since the Uniform History LTS perserves actions in the history. Therefore, by repeated application of Lem. 3.2.5, either $\mathcal{C}_2^1 \setminus_{\mathtt{ab}} k \stackrel{k}{\Rightarrow}{}^{\mathtt{sl}} \mathcal{C}_2'$ or $\mathcal{C}_2^1 \setminus_{\mathtt{ab}} k \stackrel{k}{\Rightarrow}{}^{\mathtt{sl}} \mathcal{C}_2^2 \setminus_{\mathtt{ab}} l$ for some l hold. The first transition proves the lemma directly, since $H_2'(i) = \mathtt{ab}$ and the histories of $\mathcal{C}_1'$ and $\mathcal{C}_2'$ are action consistent. In the case of the second transition, $H_1(i) = k(a)$ and $H_2'(i) = l(b)$ imply that l is uncommittable in $\mathcal{C}_2'$ by Cor. 3.3.2. Since l is uncommittable in $\mathcal{C}_2'$, $\mathcal{C}_2' \setminus_{\mathtt{ab}} l$ is starless bisimilar with $\mathcal{C}_2'$ by Prop. 3.2.7, and therefore the transition $\mathcal{C}_2 \stackrel{k}{\Rightarrow}{}^{\mathtt{sl}} \mathcal{C}_2' \setminus_{\mathtt{ab}} l$ proves the lemma.

The proof that Condition 3 of uniform history bisimulation holds for $\mathcal{R}$ is similar to the proof for Condition 2. □

The proof that extended bisimulation implies history bisimulation is more immediate. Since starless history bisimulation is more permissive than uniform history bisimulation, transitions in the uniform LTS can be mimicked straightforwardly in the Starless LTS.

Lemma 3.3.8. *Uniform history bisimulation $\approx_{\mathsf{un}}$ is a starless history bisimulation.*

Proof. We must show that, if $\mathcal{C}_1 \approx_{\mathsf{un}} \mathcal{C}_2$), then the three conditions from Definition 2.3.7 are satisfied. Condition 1, namely that the histories of the two configurations are consistent, is implied by the definition of action consistency. According to Condition 2, assume that $\mathcal{C}_1 \stackrel{\zeta}{\rightarrow}{}^{\mathtt{sl}} \mathcal{C}_2$ is a challenger move. If $\zeta = \tau$, the lemma is proved by repeated application of Lem. 3.3.4. If $\zeta = k$, then an element $k(a)$ is added to the history of $\mathcal{C}_1'$. By Lem. 3.3.4, $\mathcal{C}_1 \stackrel{k(a)}{\longrightarrow}{}^{\mathtt{u}} \mathcal{C}_1'$ holds, and therefore $\mathcal{C}_2 \stackrel{k(a)}{\Longrightarrow}{}^{\mathtt{u}} \mathcal{C}_2'$ holds too by uniform bisimulation; moreover the histories of $\mathcal{C}_1'$ and $\mathcal{C}_2'$ are uniform. The lemma is proved by repeated application of Lem. 3.3.4. Condition 3 is proved similarly.

□

We conclude this section by showing that starless and uniform history bisimulation relate the same TCCS$^{\mathrm{m}}$ processes.

Theorem 3.3.9. *If P and Q are two TCCS^m processes, then $P \approx_{\mathsf{sl}} Q$ if and only if $P \approx_{\mathsf{un}} Q$.*

Proof. Let $P \approx_{\mathsf{sl}} Q$; by definition, this case holds when $(\emptyset \rhd P) \approx_{\mathsf{sl}} (\emptyset \rhd Q)$. Since the empty histories are uniform, $(\emptyset \rhd P) \approx_{\mathsf{un}} (\emptyset \rhd Q)$ by Lem. 3.3.7. The opposite direction is a direct consequence of Lem. 3.3.8. □

3.4 Renaming bisimulation

All the definitions of bisimulations so far considered require the defender to respond with the same action ζ as the attacker's. If ζ is a tentative action $k(a)$, then the defender must contain a transaction with exactly the same name k to fire the $k(a)$ action. This restriction on transaction names is unnecessary, and it generates technical problems in the later developments of this chapter. This section presents a new bisimulation, called *renaming* bisimulation, where the defender is free to pick any transaction name l when responding.

We first introduce the notion of *(transaction) renaming*. A transaction renaming π is a partial injective function over $\mathcal{T}$ with the property that $\{\, k \in \mathcal{T} \mid \pi(k) \neq k \,\}$ is finite. We use $dom(\pi)$ to denote this finite set, with $range(\pi) = \{\, \pi(k) \mid k \in dom(\pi) \,\}$. For any syntactic object O, such as a process, history or configuration, we let $\pi(O)$ be the result of replacing every occurrence of a transaction name k with $\pi(k)$ when $k \in dom(\pi)$; no replacing is performed when $k \notin dom(\pi)$. If σ is the substitution $[\widetilde{k} \mapsto l]$, we let σ_π denote the substitution $[\widetilde{\pi(k_i)} \mapsto l]$; this last notion will be primarily used when $\pi(k)$ is k itself. If $\tilde{k}$ is a set of names $k_1, \ldots k_n$, we write $[k \widetilde{\mapsto \pi(k)}]$ for the renaming $[k_1 \mapsto \pi(k_1)] \cdot \ldots \cdot [k_n \mapsto \pi(k_n)]$. We write $\pi|_A$ for the restriction of the domain of π to the set A.

The composition of two renamings π_1 and π_2 is the partial function $\pi_1 \cdot \pi_2$ defined point-wise as follows:

$$\pi_1 \cdot \pi_2(k) = \begin{cases} \pi_2(k) & \text{if } k \in dom(\pi_2) \\ \pi_1(k) & \text{if } k \in dom(\pi_1) \end{cases}$$

Notice that, for any k in the domain of both π_1 and π_2, the renaming π_2 takes precedence over π_1 in $\pi_1 \cdot \pi_2$.

Any renaming π can always be split into two sub-renamings π_1 and π_2:

Lemma 3.4.1. *For any renaming π and set $\widetilde{k}$ such that $\widetilde{k} \subseteq dom(\pi)$, there exists a renaming π' such that $\pi = \pi' \cdot [k \widetilde{\mapsto \pi(k)}]$.*

Proof. We prove the lemma by taking $\pi' = \pi|_{dom(\pi) \setminus \widetilde{k}}$. Function π' is a renaming because the restriction of an injective function is an injective function. To show that $\pi = \pi' \cdot [k \widetilde{\mapsto \pi(k)}]$, we prove that for any $x \in dom(\pi)$, $\pi(x) = \pi' \cdot [k \widetilde{\mapsto \pi(k)}](x)$. Notice that $dom(\pi) = dom(\pi') \uplus \widetilde{k}$ holds by construction. If $x \in dom(\pi')$, then $\pi' \cdot [k \widetilde{\mapsto \pi(k)}](x) = \pi'(x)$ by definition of composition, and $\pi(x) = \pi'(x)$ by construction proves the lemma. If $x \in dom([k \widetilde{\mapsto \pi(k)}])$, then $\pi' \cdot [k \widetilde{\mapsto \pi(k)}](x) = \pi(x)$ by composition, which immediately proves the lemma. □

The following are simple properties of transaction renamings:

Lemma 3.4.2 (Actions under Renamings)**.** *Let π be a transaction renaming. If $l \sharp range(\pi)$ and $P \xrightarrow{l(\mu)}_{[\widetilde{k} \mapsto l]} Q$, then there exist a renaming π' and $\widetilde{k_1} \subseteq \widetilde{k}$ such that $\pi = \pi' \cdot [k_1 \widetilde{\mapsto \pi(k_1)}]$ and $\pi(P) \xrightarrow{l(a)}_{[\widetilde{\pi(k)} \mapsto l]} \pi'(Q)$.*

Proof. By taking $\widetilde{k_1} = dom(\pi) \cap \widetilde{k}$ and $\pi' = \pi|_{dom(\pi) \setminus \widetilde{k_1}}$ (which is a renaming by Lem. 3.4.1), and by rule induction.

If Rule CCSsum, CCSsync or CCSrec is used, no renaming takes place and the lemma follows trivially because $\pi' = \pi$. Suppose that Rule TrAct has been used:

$$\frac{P \xrightarrow{a}_{\varepsilon} P'}{[\![P \rhd_k Q]\!] \xrightarrow{l(a)}_{k \mapsto l} [\![P' \rhd_l Q]\!]}\ l \sharp k$$

By inductive hypothesis, $\pi(P) \xrightarrow{a}_{\varepsilon} \pi'(P')$. If $k \in dom(\pi)$, then $\pi([\![P \rhd_k Q]\!]) = [\![\pi(P) \rhd_{\pi(k)} \pi(Q)]\!]$. By application of Rule TrAct $[\![\pi(P) \rhd_{\pi(k)} \pi(Q)]\!] \xrightarrow{k(a)}_{\pi(k) \mapsto l} [\![\pi'(P') \rhd_l \pi(Q)]\!]$. Since Q does not contain active transactions by well-formedness, $\pi(Q) = \pi'(Q)$. By Lem. 3.4.1 $\pi = \pi' \cdot [k \mapsto \pi(k)]$, and therefore $[\![\pi(P) \rhd_{\pi(k)} \pi(Q)]\!] \xrightarrow{k(a)}_{\pi(k) \mapsto l} \pi'([\![P' \rhd_l Q]\!])$ proves the lemma. If $k \notin dom(\pi)$, then we can pick $\pi' = \pi$ and $\tilde{k}_1 = \emptyset$. Since k_1 is the empty set, $[\emptyset \widetilde{\mapsto \pi(k)}]$ is the empty substitution, and the lemma holds trivially by applying Rule TrAct on $\pi'([\![P \rhd_k Q]\!])$.

Suppose that Rule TrSync has been used:

$$\frac{P \xrightarrow{l(a)}_{\sigma_1} P' \qquad Q \xrightarrow{l(\overline{a})}_{\sigma_2} Q' \quad \sigma_1 = \widetilde{k}_1 \mapsto l}{P \,|\, Q \xrightarrow{l(\tau)}_{(\widetilde{k}_1, \widetilde{k}_2) \mapsto l} P'\sigma_2 \,|\, Q'\sigma_1 \quad \sigma_2 = \widetilde{k}_2 \mapsto l}$$

By inductive hypothesis on both premises, there exist renamings π_1 and π_2 such that $\pi = \pi_1 \cdot [k_1 \widetilde{\mapsto \pi(k_1)}]$ and $\pi = \pi_2 \cdot [k_2 \widetilde{\mapsto \pi(k_2)}]$. By Lem. 3.4.1 we can take π' such that $\pi = \pi' \cdot [k_i \widetilde{\mapsto \pi(k_i)}]$ with $i \in \{1, 2\}$, $\pi_1 = \pi' \cdot [k_2 \widetilde{\mapsto \pi(k_2)}]$ and $\pi_2 = \pi' \cdot [k_1 \widetilde{\mapsto \pi(k_1)}]$. Therefore by inductive hypothesis $\pi(P) \xrightarrow{l(a)}_{\pi(\widetilde{k_1}) \mapsto l} P''$ with $P'' = \pi' \cdot [k_2 \widetilde{\mapsto \pi(k_2)}](P')$ holds; similarly $\pi(Q) \xrightarrow{l(a)}_{\pi(\widetilde{k_2}) \mapsto l} Q''$ with $Q'' = \pi' \cdot [k_1 \widetilde{\mapsto \pi(k_1)}](Q')$. By applying Rule TrSync we have $\pi(P \,|\, Q) \xrightarrow{k(\tau)}_{\pi(\widetilde{k}_1, \widetilde{k}_2) \mapsto l} P''\sigma_2 \,|\, Q''\sigma_1$, with $\sigma_1 = [\pi(\widetilde{k_1}) \mapsto l]$ and $\sigma_2 = [\pi(\widetilde{k_2}) \mapsto l]$. By definition, $P''\sigma_2 = (\pi' \cdot [k_2 \widetilde{\mapsto \pi(k_2)}](P'))[\pi(\widetilde{k_2}) \mapsto l] = \pi'(P''[\widetilde{k_1} \mapsto l])$, and $Q''\sigma_1 = (\pi' \cdot [k_1 \widetilde{\mapsto \pi(k_1)}](Q'))[\pi(\widetilde{k_1}) \mapsto l] = \pi'(Q''[\widetilde{k_l} \mapsto l])$. Therefore $P''\sigma_2 \,|\, Q''\sigma_1 = \pi'(P'[\widetilde{k_1} \mapsto l] \,|\, Q'[\widetilde{k_2} \mapsto l])$, which proves the lemma.

The proof for Rule TrSum is similar to the proof of Rule TrAct. The remaining cases follow by inductive hypothesis □

Lemma 3.4.3 (Broadcasts under Renamings)**.** *Let π be a renaming satisfying $range(\pi) \sharp P$.*

1. *If $P \xrightarrow{\texttt{ab}\ k} Q$, then there exist renaming π' and k' such that $\pi = \pi' \cdot [k \mapsto k']$ and $\pi(P) \xrightarrow{\texttt{ab}\ k'} \pi'(Q)$*
2. *If $P \xrightarrow{\texttt{co}\ k} Q$, then there exist renaming π' and k' such that $\pi = \pi' \cdot [k \mapsto k']$ and $\pi(P) \xrightarrow{\texttt{co}\ k'} \pi'(Q)$*
3. *If $P \xrightarrow{\texttt{new}\ k} Q$, then there exist renaming π' and k' such that $\pi' = \pi \cdot [k \mapsto k']$ and $\pi(P) \xrightarrow{\texttt{new}\ k'} \pi'(Q)$*

Proof. By structural induction on P, by taking $k' = \pi(k)$ if $k \in dom(\pi)$, or by taking $k' = k$ if $k \notin dom(\pi)$. □

Lemma 3.4.4 (Histories under renamings)**.** *If $l \sharp range(\pi)$, then $\pi(H[\widetilde{k} \mapsto l]) = \pi(H)[\widetilde{\pi(k)} \mapsto l]$ for any history H*

Proof. By definition of renamings. □

Under the condition that $range(\pi) \sharp P$, these properties allow us to show that renamings do not introduce observable differences in a configuration. When the condition does not hold, transactions

might be accidentally merged and the observable behaviour of a configuration might change substantially. For example, consider configuration $\mathcal{C} = (\emptyset \rhd [\![a.\mathtt{co} \rhd_k 0]\!] \parallel [\![0 \rhd_l 0]\!])$. Transaction k is obviously committable in $\mathcal{C}$, and l is uncommittable. However, if $\pi = [l \mapsto k]$, k is not committable anymore in $\pi(\mathcal{C})$, and therefore it is not true that $\mathcal{C} \approx_{\mathsf{un}} \pi(\mathcal{C})$ in this case.

We first introduce a stronger notion of uniform history bisimulation than the weak one in Def. 3.3.5:

Definition 3.4.5 (Strong uniform history bisimulation)**.** *A binary relation $\mathcal{R} \subseteq \mathsf{Conf}_{\mathsf{Act}} \times \mathsf{Conf}_{\mathsf{Act}}$ is a* strong *uniform history bisimulation when for all $\mathcal{C}_1\ \mathcal{R}\ \mathcal{C}_2$:*

1. *$hist(\mathcal{C}_1)$ and $hist(\mathcal{C}_2)$ are action consistent,*
2. *if $\mathcal{C}_1 \xrightarrow{\zeta}_{\mathtt{u}} \mathcal{C}_1'$ is a challenger move and $\zeta \sharp \mathcal{C}_2$, then there exists $\mathcal{C}_2'$ such that $\mathcal{C}_2 \xrightarrow{\zeta}_{\mathtt{u}} \mathcal{C}_2'$ and $\mathcal{C}_1'\ \mathcal{R}\ \mathcal{C}_2'$,*
3. *the converse of the preceding condition.*

The largest strong uniform history bisimulation is denoted by $\sim_{\mathtt{u}}$. We now prove that a transaction renaming π in a configuration $\mathcal{C}$ is unobservable, when π does not introduce names already present in $\mathcal{C}$ (and therefore possibly merging some transaction names):

Proposition 3.4.6. *Let π be a transaction renaming. If $range(\pi) \sharp \mathcal{C}$, then $\mathcal{C} \sim_{\mathtt{u}} \pi(\mathcal{C})$.*

Proof. Let $\mathcal{R} = \{\, \mathcal{C}, \pi(\mathcal{C}) \mid range(\pi) \sharp \mathcal{C}, dom(\pi) \subseteq ftn(\mathcal{C}) \,\}$;

We need to show that the three condition of uniform history bisimulation from Def. 3.3.5 hold for any configuration $\mathcal{C} = (H \rhd P)$ such that . Condition 1 holds trivially, because renamings do not modify action names in the history.

To prove Condition 2, let $\mathcal{C} \xrightarrow{\zeta}_{\mathtt{u}} \mathcal{C}'$ be a challenger move, with $\zeta \sharp \pi(\mathcal{C})$. We need to show that there is configuration $\mathcal{C}_2$ such that $\pi(\mathcal{C}) \xrightarrow{\zeta}_{\mathtt{u}} \mathcal{C}''$ and $\mathcal{C}'' = \pi'(\mathcal{C}')$ for some π'. We proceed by case analysis on the challenger move.

If Rule Un-τ has been used, then $P \xrightarrow{\tau}_{\epsilon}^{\mathtt{u}} P'$ holds by Lem. 2.1.3, and the lemma is proved by taking $\pi' = \pi$, since no names are modified by the empty substitution ϵ.

If Rule Un-$k(\tau)$ has been used, then $(H \rhd P) \xrightarrow{\tau}_{\mathtt{u}} (\sigma(H) \rhd P')$ and $P \xrightarrow{k(\tau)}_{\sigma} P'$ hold, with $\sigma = [l_1, l_2 \mapsto k]$ by Lem. 2.1.3, k fresh from P and l_1, l_2 fresh from P'. By hypothesis, ζ is fresh from $\pi(\mathcal{C})$, which implies $k \sharp range(\pi)$. By Lem. 3.4.2 there exists π' such that $\pi = \pi' \cdot \widetilde{[l_i \mapsto k]}$ for $i \in \{1, 2\}$ and $\pi(P) \xrightarrow{k(\tau)}_{[\pi(l_1,l_2) \mapsto k]} \pi'(P')$ hold. By applying Rule Un-$k(\tau)$, we have $(\pi(H) \rhd \pi(P)) \xrightarrow{\tau}_{\mathtt{u}} ((\pi(H))[\pi(l_1, l_2) \mapsto k] \rhd \pi'(P'))$. By Lem. 3.4.4 $(\pi(H))[\pi(l_1, l_2) \mapsto k] = \pi(H[l_1, l_2 \mapsto k])$, which is equal to $\pi'(H[l_1, l_2 \mapsto k])$ because l_1 and l_2 are fresh from $H[l_1, l_2 \mapsto k]$. Therefore we can rewrite the previous transition $(\pi(H) \rhd \pi(P)) \xrightarrow{\tau}_{\mathtt{u}} ((\pi(H))[\pi(l_1, l_2) \mapsto k] \rhd \pi'(P'))$ as $(\pi(H) \rhd \pi(P)) \xrightarrow{\tau}_{\mathtt{u}} (\pi'(H[l_1, l_2 \mapsto k]) \rhd \pi'(P'))$, which proves the lemma. The case for Rule Un-$k(a)$ is proved similarly; the case for Rule Un-`new` uses Property 3 of Lem. 3.4.3.

If Rule Un-`co` has been used, then $(H \rhd P) \xrightarrow{\tau}_{\mathtt{u}} (H \setminus_{\mathtt{co}} k \rhd P')$ and $P \xrightarrow{\mathtt{co}\ k} P'$ hold. By Lem. 3.4.3 there exists π' such that $\pi = \pi' \cdot [k \mapsto k']$ and $\pi(P) \xrightarrow{\mathtt{co}\ k'} \pi'(P')$ hold. By applying Rule Un-`co` we have $(\pi(H) \rhd \pi(P)) \xrightarrow{\tau}_{\mathtt{u}} (\pi(H) \setminus_{\mathtt{co}} k' \rhd \pi'(P'))$. By definition of π and of the history commit

operation $\pi(H) \setminus_{\mathsf{co}} k' = \pi' \cdot [k \mapsto k'](H) \setminus_{\mathsf{co}} k' = \pi'(H \setminus_{\mathsf{co}} k)$. Therefore $(\pi(H) \rhd \pi(P)) \xrightarrow{\tau}{}^{\mathtt{u}} (\pi'(H \setminus_{\mathsf{co}} k) \rhd \pi'(P'))$ holds, and the lemma is proved. The case for Rule UN-ab is proved similarly.

Condition 3 is proved similarly to Condition 2, using the fact that the inverse renaming π^{-1} is injective because π is injective, and that $P = \pi^{-1}(\pi(P))$. □

We now introduce the notion of label similarity, which matches labels up to transaction names k, and the renaming bisimulation:

Definition 3.4.7 (Label similarity)**.** *A label ζ is action similar to ζ', or $\zeta \sim \zeta'$, when $\zeta = \zeta' = \tau$ or $\zeta = k(a)$ and $\zeta' = l(a)$ for some k and l.*

Definition 3.4.8. *A relation $\mathcal{R} \subseteq \mathsf{Conf}_{\mathsf{Act}} \times \mathsf{Conf}_{\mathsf{Act}}$ is a* renaming *history bisimulation when, for any $\mathcal{C}_1 \mathcal{R} \mathcal{C}_2$:*

- *$hist(\mathcal{C}_1)$ and $hist(\mathcal{C}_2)$ are action consistent*
- *if $\mathcal{C}_1 \xrightarrow{\zeta_1}{}^{\mathtt{u}} \mathcal{C}_1'$ is a challenger move, then there exist $\mathcal{C}_2'$ and ζ_2 such that $\mathcal{C}_2 \overset{\zeta_2}{\Rightarrow}{}^{\mathtt{u}} \mathcal{C}_2'$, $\zeta_1 \sim \zeta_2$ and $\mathcal{C}_1' \mathcal{R} \mathcal{C}_2'$,*
- *the converse of the preceding condition.*

The greatest renaming bisimulation is denoted by $\approx_{\mathsf{r}}$. Two TCCS$^{\mathrm{m}}$ processes P and Q are renaming bisimilar, or $P \approx_{\mathsf{r}} Q$, when $(\emptyset \rhd P) \approx_{\mathsf{r}} (\emptyset \rhd Q)$.

We conclude this section by showing that renaming and uniform history bisimulation relate the same processes. As usual, this is proved by showing that the two bisimulation relations are included in each other. The first inclusion, namely that the uniform history bisimulation $\approx_{\mathsf{un}}$ is also a renaming bisimulation, is trivial, because the moves played during the uniform history bisimulation game are allowed to be replicated directly in the renaming bisimulation game:

Lemma 3.4.9. *The uniform history bisimulation $\approx_{\mathsf{un}}$ is a renaming bisimulation.*

Proof. The three conditions of renaming bisimulation follow directly by the definition of uniform bisimulation. In particular, if $\mathcal{C}_1 \approx_{\mathsf{un}} \mathcal{C}_2$ and $\mathcal{C}_1 \xrightarrow{\zeta}{}^{\mathtt{u}} \mathcal{C}_1'$ is a challenger move, $\mathcal{C}_2 \overset{\zeta}{\Rightarrow}{}^{\mathtt{u}} \mathcal{C}_2'$ holds by uniform bisimulation, and Condition 2 is proved directly by taking $\zeta' = \zeta$ □

The proof of the opposite inclusion is complicated by the fact that, when the attacker plays a $k(a)$ move, the defender must play the same $k(a)$ action. Let $\mathcal{C}_1 \xrightarrow{k(a)}{}^{\mathtt{u}} \mathcal{C}_1'$ be the challenger's move, and let $\mathcal{C}_2 = (\emptyset \rhd [\![0 \rhd_{b.a.P} l_1]\!] \parallel [\![0 \rhd_{\bar{b}.Q} l_2]\!])$ be the defender. Before playing action a, transactions l_1 and l_2 must synchronize on b first, and therefore l_1 and l_2 must be merged. The definition of renaming bisimulation allows the defender to respond with the weak transition $\mathcal{C}_2 \xrightarrow{\tau}{}^{\mathtt{u}} (\emptyset \rhd [\![0 \rhd_{a.P} k]\!] \parallel [\![0 \rhd_{\bar{b}.Q} k]\!]) \xrightarrow{l(a)}{}^{\mathtt{u}} (\emptyset \rhd [\![0 \rhd_P l]\!] \parallel [\![0 \rhd_{\bar{b}.Q} l]\!])$, whereby l_1 and l_2 are merged into k. The problem is that the intermediate configuration $(\emptyset \rhd [\![0 \rhd_{a.P} k]\!] \parallel [\![0 \rhd_{\bar{b}.Q} k]\!])$ cannot perform a $k(a)$ action, because k is not fresh.

In order to resolve this problem, notice that k is fresh in the initial configuration $\mathcal{C}_2$; in general k has to be fresh in $\mathcal{C}_2$ by definition of uniform history bisimulation. The solution is to construct an alternative weak response that renames k with another fresh name k', whenever k is introduced.

By Proposition 3.4.6 the alternative response is strongly bisimilar to the original one. The following lemma shows that it is always possible to build such an alternative:

Lemma 3.4.10. *Let $\mathcal{C} \xrightarrow{\tau} \mathcal{C}'$.*

1. *If $k \sharp \mathcal{C}$, then there exists π such that $\mathcal{C} \xrightarrow{\tau} \pi(\mathcal{C}')$ and $k \sharp \pi(\mathcal{C}')$;*
2. *If $k \sharp \pi(\mathcal{C})$ for some π, then there exists π' such that $\pi(\mathcal{C}) \xrightarrow{\tau} \pi'(\mathcal{C}')$ and $k \sharp \pi'(\mathcal{C}')$.*

Proof. By rule induction, by respectively picking $\pi = [k \mapsto k']$ for Property 1 and $\pi' = \pi \cdot [k \mapsto k']$ for Property 2, when Rule UN-`new` or Rule UN-$k(a)$ is applied and k is chosen as the fresh name. □

We can now prove the reverse inclusion:

Lemma 3.4.11. *Bisimulation $\approx_{\mathsf{r}}$ is a uniform history bisimulation.*

Proof. We need to show that, for any $\mathcal{C}_1$ and $\mathcal{C}_2$ such that $\mathcal{C}_1 \approx_{\mathsf{r}} \mathcal{C}_2$, the three conditions in Def. 3.3.5 are satisfied.

Condition 1 follows immediately by definition of renaming bisimulation.

To prove Condition 2, let $\mathcal{C}_1 \xrightarrow{\zeta}{}^{\mathtt{u}} \mathcal{C}_1'$ be a challenger move, with $\zeta \sharp \mathcal{C}_2$. We need to show that there exists $\mathcal{C}_2'$ such that $\mathcal{C}_2 \stackrel{\zeta}{\Rightarrow}{}^{\mathtt{u}} \mathcal{C}_2'$ and $\mathcal{C}_1' \approx_{\mathsf{r}} \mathcal{C}_2'$. By definition of renaming bisimulation, $\mathcal{C}_2 \stackrel{\zeta_2}{\Rightarrow}{}^{\mathtt{u}} \mathcal{C}_2'$, $\zeta \sim \zeta_2$ and $\mathcal{C}_1' \mathcal{R} \mathcal{C}_2'$ hold.

If $\zeta = \tau$, then by renaming bisimulation there exists ζ_2 and $\mathcal{C}_2 \stackrel{\zeta_2}{\Rightarrow}{}^{\mathtt{u}} \mathcal{C}_2'$ and $\mathcal{C}_1' \approx_{\mathsf{un}} \mathcal{C}_2'$. By definition of action similarity $\zeta_2 = \tau$, and therefore $\mathcal{C}_2 \stackrel{\tau}{\Rightarrow}{}^{\mathtt{u}} \mathcal{C}_2'$ holds, which proves the lemma together with $\mathcal{C}_1' \approx_{\mathsf{r}} \mathcal{C}_2'$.

If $\zeta = k(a)$, the definition of action similarity implies that $\zeta_2 = l(a)$, where l might be different from k. Because of this, ζ might be different from ζ_2, and therefore we cannot use transition $\mathcal{C}_2 \stackrel{\zeta_2}{\Rightarrow}{}^{\mathtt{u}} \mathcal{C}_2'$ to prove the lemma as in the previous case. Instead, we construct a transition $\mathcal{C}_2 \stackrel{k(a)}{\Longrightarrow}{}^{\mathtt{u}} \mathcal{C}_2''$ such that $\mathcal{C}_2'' \approx_{\mathsf{r}} \mathcal{C}_2'$. The lemma is then proved by transitivity of $\approx_{\mathsf{r}}$, since $\mathcal{C}_1 \approx_{\mathsf{r}} \mathcal{C}_2'$ and $\mathcal{C}_2' \approx_{\mathsf{r}} \mathcal{C}_2''$ imply that $\mathcal{C}_1' \approx_{\mathsf{r}} \mathcal{C}_2''$.

By definition, $\mathcal{C}_2 \stackrel{\zeta_2}{\Rightarrow}{}^{\mathtt{u}} \mathcal{C}_2'$ stands for transition $\mathcal{C}_2 \stackrel{\tau}{\Rightarrow}{}^{\mathtt{u}} \mathcal{C}_2^1 \xrightarrow{l(a)}{}^{\mathtt{u}} \mathcal{C}_2^2 \stackrel{\tau}{\Rightarrow}{}^{\mathtt{u}} \mathcal{C}_2'$. Configuration $\mathcal{C}_2^1$ can perform a $k(a)$ action only if k is fresh from its history. Therefore we first have to construct a transition $\mathcal{C}_2 \stackrel{\tau}{\Rightarrow}{}^{\mathtt{u}} \pi_1(\mathcal{C}_2^1)$ such that $k \sharp \pi(\mathcal{C}_2^1)$. This is repeated application of Lem. 3.4.10 and by Prop. 3.4.6.

If $\mathcal{C}_2^1 \xrightarrow{l(a)}{}^{\mathtt{u}} \mathcal{C}_2^2$ is derived by Rule UN-$\star$, then the same rule can be applied to infer $\pi(\mathcal{C}_2^1) \xrightarrow{k(a)}{}^{\mathtt{u}} \pi(\mathcal{C}_2^2)$, since $k \sharp \pi(\mathcal{C}_2^1)$ holds and Rule UN-$\star$ does not rename transaction. The lemma is then proved by repeated application of Lem. 3.4.10 and by Prop. 3.4.6 on $\pi(\mathcal{C}_2^2)$.

If $\mathcal{C}_2^1 \xrightarrow{l(a)}{}^{\mathtt{u}} \mathcal{C}_2^2$ is derived by Rule UN-$k(a)$, then the same Rule can be applied to infer $\pi(\mathcal{C}_2^1) \xrightarrow{k(a)}{}^{\mathtt{u}} \pi'(\mathcal{C}_2^2)$ with $\pi' = \pi|_{dom(\pi) \setminus dom(\sigma)}$ for some σ, since $k \sharp \pi(\mathcal{C}_2^1)$ $\pi(\mathcal{C}_2^1) \xrightarrow{k(a)}{}^{\mathtt{u}} \mathcal{C}_2^2$. The lemma is then proved by repeated application of Lem. 3.4.10 and by Prop. 3.4.6 on $\pi(\mathcal{C}_2^2)$.

□

We conclude this section by showing that the uniform history and renaming bisimulations related the same processes:

Theorem 3.4.12. *For any TCCS^m process P and Q, $P \approx_{\mathsf{un}} Q$ if and only if $P \approx_{\mathsf{r}} Q$.*

Proof. By Lem. 3.4.9 and Lem. 3.4.11. □

3.5 Historyless bisimulation

Participants in the uniform history bisimulation game produce very closely related histories. Permanent and tentative elements of their histories always store matching action names, unless a transaction has been aborted. It would be tempting to eliminate elements with matching actions. However, such elements might have different transaction names, and this information is crucial to determine which tentative actions become permanent after a commit.

For example, consider the following processes:

$$P = [\![a.b.\mathtt{co} + b.a.\mathtt{co} \triangleright_k 0]\!] \qquad Q = [\![a.\mathtt{co} \triangleright_{l_1} 0]\!] \mid [\![b.\mathtt{co} \triangleright_{l_2} 0]\!]$$

Processes P and Q are an example of the familiar rule of parallel decomposition from CCS, in the transactional world. However, notice that process Q contains two distinct transactions l_1 and l_2, whereas process P is a single transaction k. This difference is sufficient to distingiush the two processes under bisimulation.

The initial configurations start with the empty history. During the first two rounds, process P attacks twice, by performing actions a and b tentatively:

$$\begin{array}{llll}
\emptyset \triangleright P & \xrightarrow{k_1(a)}{}^{\mathtt{u}} & k_1(a) \triangleright [\![b.\mathtt{co} \triangleright_{k_1} 0]\!] & \xrightarrow{k_2(b)}{}^{\mathtt{u}} \quad k_2(a)k_2(b) \triangleright [\![\mathtt{co} \triangleright_{k_1} 0]\!] \\
\emptyset \triangleright Q & \xrightarrow{k_1(a)}{}^{\mathtt{u}} & k_1(a) \triangleright [\![\mathtt{co} \triangleright_{k_1} 0]\!] \mid [\![b.\mathtt{co} \triangleright_{l_2} 0]\!] & \xrightarrow{k_2(b)}{}^{\mathtt{u}} \quad k_1(a)k_2(b) \triangleright [\![\mathtt{co} \triangleright_{k_1} 0]\!] \mid [\![b.\mathtt{co} \triangleright_{l_2} 0]\!]
\end{array}$$

Notice that the tentative elements in P's history are tagged by the same transaction name k_2: since there is only a single transaction in P, transaction name k_1 is renamed to k_2. Process Q responds by performing the same tentative actions, but without renaming k_1 to k_2, because the two transaction are distinct in Q.

At this point, Q attacks by committing only one of the two transactions it contains, namely k_2:

$$\begin{array}{lll}
k_2(a)k_2(b) \triangleright [\![\mathtt{co} \triangleright_{k_1} 0]\!] & \not\xrightarrow{\tau}{}^{\mathtt{u}} & \\
k_1(a)k_2(b) \triangleright [\![\mathtt{co} \triangleright_{k_1} 0]\!] \mid [\![b.\mathtt{co} \triangleright_{l_2} 0]\!] & \xrightarrow{\tau}{}^{\mathtt{u}} & k_1(a)b \triangleright [\![\mathtt{co} \triangleright_{k_1} 0]\!] \mid 0
\end{array}$$

Process P cannot produce an history consistent with Q's. It can commit k_2 and produce history $H_2 \setminus_{\mathtt{co}} k_2 = a, b$ or abort k_2 and produce history $H_2 \setminus_{\mathtt{ab}} k_2 = \mathtt{abab}$. However neither history is action consistent with $k_1(a)b$.

On the contrary, it is possible to show that P and Q are bisimilar, when $l_1 = l_2$ in Q (i.e. $[\![a.b.\mathtt{co} + b.a.\mathtt{co} \triangleright_k 0]\!] \approx_{\mathsf{un}} [\![a.\mathtt{co} \triangleright_l 0]\!] \mid [\![b.\mathtt{co} \triangleright_l 0]\!]$). Therefore this example shows that, even though action names are now irrelevant, the histories must still record transaction names. Transaction names in the history allow the bisimulation to distinguish whether a single transaction is simulated by multiple logically equivalent transactions, as in the example. More generally, transaction names in the histories track the inter-dependencies between groups of transactions across two configurations. Action names are redundant, but transaction names are not.

Capitalizing on this intuition, we can define the *historyless bisimulation*, a bisimulation that replaces histories with a single environment Δ, called the *dependency set*, that keeps track of dependencies across transactions during a bisimulation game. Let $\mathcal{D}$ be the set $\mathcal{T} \cup \{\mathtt{ab}, \mathtt{co}\}$, which is the set of transaction names $\mathcal{T}$ plus the ground terms $\mathtt{ab}$ and $\mathtt{co}$. A term $d \in \mathcal{D}$ is called a *dependency element*. The dependency set Δ is a binary relation over $\mathcal{D}$. The set of *left transaction names*

$$
\begin{array}{llll}
P & \xrightarrow{\tau}{}^{\mathtt{hl}}_{\sigma} Q & \text{if} \quad P \xrightarrow{\tau}_{\sigma} Q & (\text{HL-}\tau) \\
P & \xrightarrow{\tau}{}^{\mathtt{hl}}_{\sigma} Q & \text{if} \quad P \xrightarrow{k(\tau)}_{\sigma} Q & (\text{HL-}k(\tau)) \\
P & \xrightarrow{\tau}{}^{\mathtt{hl}}_{\epsilon} Q & \text{if} \quad P \xrightarrow{\mathtt{new}\ k} Q & (\text{HL-New}) \\
P & \xrightarrow{\tau}{}^{\mathtt{hl}}_{[k \mapsto \mathtt{co}]} Q & \text{if} \quad P \xrightarrow{\mathtt{co}\ k} Q & (\text{HL-Co}) \\
P & \xrightarrow{\tau}{}^{\mathtt{hl}}_{[k \mapsto \mathtt{ab}]} Q & \text{if} \quad P \xrightarrow{\mathtt{ab}\ k} Q & (\text{HL-Ab}) \\
P & \xrightarrow{k(a)}{}^{\mathtt{hl}}_{\sigma} Q & \text{if} \quad P \xrightarrow{k(a)}_{\sigma} Q & (\text{HL-}k(a)) \\
P & \xrightarrow{\mathtt{ab}(a)}{}^{\mathtt{hl}}_{\epsilon} P & \text{if} \quad k \sharp P & (\text{HL-}k^{\star})
\end{array}
$$

Fig. 3.4: Historyless LTS.

of Δ is the set $ftn_L(\Delta) = \{\, k \,|\, k\Delta d \text{ for any } d \,\}$; the set of *right transaction names* of Δ is the set $ftn_R(\Delta) = \{\, k \,|\, d\Delta k \text{ for any } d \,\}$.

When a transaction is aborted or committed, the abort and commit operations $\backslash_{\mathtt{ab}}\ k$ and $\backslash_{\mathtt{co}}\ k$ update the history of a configuration $\mathcal{C}$ accordingly. Instead of introducing similar operators for Δ, we generalize history substitutions to *historyless substitutions*. A historyless substitution is defined as a total function $\sigma^{\mathsf{hl}} :: \mathcal{D} \to \mathcal{D}$ of the form $[\widetilde{k \mapsto d}]$ such that $\sigma^{\mathsf{hl}}(\mathtt{ab}) = \mathtt{ab}$, $\sigma^{\mathsf{hl}}(\mathtt{co}) = \mathtt{co}$ and $\forall l \notin \widetilde{k}.\ \sigma^{\mathsf{hl}}(l) = l$. The domain of a historyless substitution $\sigma^{\mathsf{hl}} = [\widetilde{k \mapsto d}]$ is $dom(\sigma^{\mathsf{hl}}) = \widetilde{k}$, and its range is $range(\sigma^{\mathsf{hl}}) = \{\, l \,|\, l \in \widetilde{d} \,\}$. For ease of notation, we drop the superscript hl from σ^{hl} when it is clear from the context that σ denotes a historyless substitution.

Historyless substitutions generalize over history substitutions because they can replace transaction names with the $\mathtt{ab}$ or $\mathtt{co}$ term, in addition to renaming them. The composition of two historyless substitutions σ_1 and σ_2 is written $\sigma_1 \circ \sigma_2$, which is defined as the standard function composition $\sigma_2(\sigma_1(d))$ for any d. The application of a substitutions σ *to the left-hand side* of a dependency set Δ is defined as $\sigma\Delta = \{\, (\sigma(k), l) \,|\, (k, l) \in \Delta \,\}$; its application *to the right-hand side* of Δ is defined as $\Delta\sigma = \{\, (k, \sigma(l)) \,|\, (k, l) \in \Delta \,\}$. We abbreviate $(\sigma_1\Delta)\sigma_2$ with $\sigma_1\Delta\sigma_2$.

Figure 3.4 shows the Historyless LTS. A transitions in the Historyless LTs has the form $P \xrightarrow{\xi}{}^{\mathtt{hl}}_{\sigma} Q$, with label $\xi ::= \tau \mid d(a)$. Labels in the Historyless LTS are therefore a generalization of labels ζ from the Uniform History LTS from Fig. 3.3. The weak τ transition $\overset{\tau}{\Rightarrow}{}^{\mathtt{hl}}_{\sigma}$ is defined as $\xrightarrow{\tau}{}^{\mathtt{hl}}_{\sigma_1} \ldots \xrightarrow{\tau}{}^{\mathtt{hl}}_{\sigma_n}$ where σ stands for the composition $\sigma_1 \circ \ldots \circ \sigma_1$. The weak $k(a)$ transition $\overset{d(a)}{\Longrightarrow}{}^{\mathtt{hl}}_{\sigma}$ is defined as $\overset{\tau}{\Rightarrow}{}^{\mathtt{hl}}_{\sigma_1} \xrightarrow{d'(a)}{}^{\mathtt{hl}}_{\sigma_2} \overset{\tau}{\Rightarrow}{}^{\mathtt{hl}}_{\sigma_3}$ with $\sigma = \sigma_1 \circ \sigma_2 \circ \sigma_3$ and $d = \sigma_3(d')$. Rules HL-τ, HL-$k(\tau)$ and HL-$k(a)$ simply lift TCCS^{m} transitions to historyless transitions. Rule HL-New is the same as in the History LTS. Rules HL-Ab and HL-Co produce a τ as in the History LTS too; operations $\backslash_{\mathtt{ab}}\ k$ and $\backslash_{\mathtt{co}}\ k$ are respectively replaced by substitutions $[k \mapsto \mathtt{ab}]$ and $[k \mapsto \mathtt{co}]$. Finally, Rule HL-$k^{\star}$ lets a process P perform any action a, where a is paired by the abort dependency $\mathtt{ab}$ in $\mathtt{ab}(a)$.

As already mentioned, the set Δ keeps track of the dependencies between transactions in a process P and transactions in another process Q. It is reasonable to require that Δ only contains transaction names occurring in P and Q, just as $ftn(H) \subseteq ftn(P)$ holds for well-formed configurations $(H \rhd P)$. Therefore, we say that a triple (P, Δ, Q) is well-formed only when the transaction names in Δ are a subset of the transaction names in P and Q:

Definition 3.5.1 (Dependency set well-formedness)**.** *Let P and Q be well-formed terms, and let Δ*

be a dependency set. Processes P and Q are well-formed *in Δ, or alternatively the triple (P, Δ, Q) is well-formed, when $ftn_L(\Delta) \subseteq ftn(P)$ and $ftn_R(\Delta) \subseteq ftn(Q)$.*

Historyless transitions preserve well-formedness:

Lemma 3.5.2 (Preservation of well-formedness)**.** *For any well-formed triple (P, Δ, Q):*

1. *if $P \xrightarrow{\xi}{}^{\mathtt{hl}}_{\sigma_1} P'$, P' and Q are well-formed in $\sigma_1\Delta$.*
2. *if $Q \xrightarrow{\xi}{}^{\mathtt{hl}}_{\sigma_2} Q'$, P and Q' are well-formed in $\Delta\sigma_2$.*

Proof. Both properties are proved by rule induction; we only discuss the proof of Property 1, because the proof of Property 2 is very similar to it. Except for Rule HL-$k^\star$, transition in the Historyless LTS do not introduce fresh names unless a TCCS$^{\mathrm{m}}$ transition $P \xrightarrow{\alpha} \sigma P'$ or $P \xrightarrow{\beta} P'$ does so. In such cases the fresh name is in P' by Lem. 2.1.2, which proves the lemma. Rule HL-$k^\star$ is the only rule that generates fresh names. In particular, it allows the transition $P \xrightarrow{k(a)}{}^{\mathtt{hl}}_{[k\mapsto \mathtt{ab}]} P$ with $k \sharp P$. Since k is fresh from P, the substitution $[k \mapsto \mathtt{ab}]$ has no effect on any Δ from a well-formed triple (P, Δ, Q), i.e. $[k \mapsto \mathtt{ab}]\Delta = \Delta$. Since $P' = P$ and $[k \mapsto \mathtt{ab}]\Delta = \Delta$, the lemma holds trivially. □

Instead of recording action names a and $k(a)$, a dependency set records the element `co` in place of permanent actions a, and k in place of tentative actions $k(a)$. Consistent dependency sets therefore only relate the `co` element with the `co` element itself, and transaction names k with either another transaction name l or the `ab` element:

Definition 3.5.3 (Dependency set consistency)**.** *A dependency set Δ is consistent when, for all pairs $(x, y) \in \Delta$, $x = \mathtt{co}$ if and only if $y = \mathtt{co}$.*

A challenger move in historyless bisimulation is any transition not produced by Rule HL-$k^\star$. We are now ready to define historyless bisimulation:

Definition 3.5.4 (Historyless bisimulation)**.** *A ternary relation $\mathcal{R} :: \mathcal{P} \times \mathcal{D} \times \mathcal{P}$ is a historyless bisimulation when, for well-formed triple (P, Δ, Q):*

1. *Δ is consistent*
2. *if $P \xrightarrow{k(a)}{}^{\mathtt{hl}}_{\sigma_1} P'$, then $Q \xRightarrow{d(a)}{}^{\mathtt{hl}}_{\sigma_2} Q'$ and $(P', \Delta', Q') \in \mathcal{R}$, where $\Delta' = \sigma_1\Delta\sigma_2 \cup (k, d)$.*
3. *if $P \xrightarrow{\tau}{}^{\mathtt{hl}}_{\sigma_1} P'$, then $Q \xRightarrow{\tau}{}^{\mathtt{hl}}_{\sigma_2} Q'$ and $(P', \Delta', Q') \in \mathcal{R}$, where $\Delta' = \sigma_1\Delta\sigma_2$.*
4. *the converse of Conditions 2 and 3.*

Instead of adding actions in the histories, historyless bisimulation adds a dependency pair (k, k) to Δ whenever the challenger plays a $k(a)$ action. Transaction renamings, aborts and commits are recorded by historyless substitutions, which are respectively applied to the left and right-hand side of relation Δ. We say that two TCCS$^{\mathrm{m}}$ processes P and Q are historyless bisimilar, or $P \approx_{\mathsf{hl}} Q$, when there exists a relation $\mathcal{R}$ such that $\mathcal{R}$ is a historyless bisimulation and $(P, \emptyset, Q) \in \mathcal{R}$.

In order to show that historyless and uniform history bisimulation coincide, some technical definitions are required to relate histories and dependency sets on the one hand, and to relate the Historyless and the Uniform History LTS on the other hand. The *history join* operator $(H_1 \bowtie H_2)$ translates two histories H_1 and H_2 into a dependency set Δ as follows:

Definition 3.5.5 (History join)**.** *The* history join *operation* $(- \bowtie -) :: \mathcal{H} \times \mathcal{H} \to \mathcal{D}$ *is the partial function from histories* H_1, H_2 *of the same length to dependency sets defined as follows:*

$$(H_1 \bowtie H_2) = \{\, (d_1, d_2) \mid \exists i.\ d_1 = [\![H_1(i)]\!] \ \wedge\ d_2 = [\![H_2(i)]\!] \,\}$$

where $[\![-]\!]$ *is the total function from history elements to dependency elements defined as follows:*

$$[\![k(a)]\!] = k \qquad [\![a]\!] = \mathtt{co} \qquad [\![\mathtt{ab}]\!] = \mathtt{ab}$$

Historyless substitutions need to be translated into substitutions and $\backslash_{\mathsf{ab}}\ k$ and $\backslash_{\mathsf{co}}\ k$ operations on histories as well:

Definition 3.5.6 (Application of a historyless substitution to a history)**.** *Let* σ^{hl} *be a historyless substitution, and* H *be a history. The application of* σ^{hl} *to* H *is the lifting to lists of the operations:*

$$\begin{array}{lll} \sigma^{\mathsf{hl}}(a) = a & & \sigma^{\mathsf{hl}}(k(a)) = a \quad \textit{if } \sigma^{\mathsf{hl}}(k) = \mathtt{co} \\ \sigma^{\mathsf{hl}}(k(a)) = l(a) \quad \textit{if } \sigma^{\mathsf{hl}}(k) = l & & \sigma^{\mathsf{hl}}(k(a)) = k(a) \quad \textit{if } k \notin \mathit{dom}(\sigma^{\mathsf{hl}}) \\ \sigma^{\mathsf{hl}}(k(a)) = \mathtt{ab} \quad \textit{if } \sigma^{\mathsf{hl}}(k) = \mathtt{ab} & & \sigma^{\mathsf{hl}}(\mathtt{ab}) = \mathtt{ab} \end{array}$$

Notice that the application of a historyless substitution σ to a history H is always defined, and that action names a are never changed. A straightforward consequence of this definition is that historyless substitutions $[k \mapsto \mathtt{co}]$ and $[k \mapsto \mathtt{ab}]$ have the same effect as operations $\backslash_{\mathsf{co}}\ k$ and as $\backslash_{\mathsf{ab}}\ k$, respectively, on any history H:

Observation 3.5.7. *For any history* H, $H[k \mapsto \mathtt{ab}] = H \backslash_{\mathsf{ab}} k$ *and* $H[k \mapsto \mathtt{co}] = H \backslash_{\mathsf{co}} k$.

Historyless substitutions distribute over the translation function $[\![-]\!]$ of history elements:

Lemma 3.5.8. *For any history* H *and historyless substitution* σ, $\sigma([\![H_1(i)]\!]) = [\![\sigma(H(i))]\!]$.

Proof. By case analysis. □

As a corollary of the previous lemma, historyless substitutions distribute over history joins:

Proposition 3.5.9 (Distributivity of historyless substitutions over history joins)**.** *For any history* H_1, H_2 *and historyless substitution* σ_1, σ_2: $\sigma_1(H_1 \bowtie H_2)\sigma_2 = (\sigma_1(H_1) \bowtie \sigma_2(H_2))$.

Proof. By Lem. 3.5.8 and by definition of history join (Def. 3.5.5). □

With the aid of these translation functions, Uniform History LTS transitions can be translated into Historyless LTS transitions, and viceversa:

Proposition 3.5.10 (Translation between Uniform history and Historyless LTSs)**.** *For any well-formed configurations* $(H \rhd P)$:

1. *if* $(H \rhd P) \xrightarrow{\tau}{}^{\mathtt{u}} (H' \rhd Q)$, *then* $\exists \sigma. P \xrightarrow{\tau}{}^{\mathtt{hl}}_{\sigma} Q$ *and* $H' = \sigma(H)$.
2. *if* $P \xrightarrow{\tau}{}^{\mathtt{hl}}_{\sigma} Q$ *and* $H' = \sigma(H)$, *then* $(H \rhd P) \xrightarrow{\tau}{}^{\mathtt{u}} (H' \rhd Q)$.
3. *if* $(H \rhd P) \xrightarrow{k(a)}{}^{\mathtt{u}} (\sigma(H), k(a) \rhd Q)$ *and* $\sigma = [\widetilde{l} \mapsto k]$, *then* $P \xrightarrow{k(a)}{}^{\mathtt{hl}}_{\sigma} Q$.

4. *if* $P \xrightarrow{k(a)}{}^{\mathtt{hl}}_{\sigma} Q$ *and* $\sigma = [\widetilde{l} \mapsto k]$, *then* $(H \rhd P) \xrightarrow{k(a)}{}^{\mathtt{u}} (\sigma(H), k(a) \rhd Q)$.

5. *if* $(H \rhd P) \xrightarrow{k(a)}{}^{\mathtt{u}} (H\mathtt{ab} \rhd P)$, *then* $P \xrightarrow{k(a)}{}^{\mathtt{hl}}_{[k \mapsto \mathtt{ab}]} P$.

6. *if* $P \xrightarrow{k(a)}{}^{\mathtt{hl}}_{[k \mapsto \mathtt{ab}]} P$, *then* $(H \rhd P) \xrightarrow{k(a)}{}^{\mathtt{u}} (H\mathtt{ab} \rhd P)$.

Proof. Each property is proved by rule induction. Notice that each rule in one LTS has a corresponding rule in the other LTS with exactly the same premise, and vice versa. For example, Rule HL-τ in the Historyless LTS corresponds to Rule UN-τ in the Uniform History LTS, and both rules have the same premise $P \xrightarrow{\tau}_{\sigma} Q$.

In order to prove Property 1, the only difficulty here is relating the history H with the historyless substitution. When a transaction k is aborted or committed, the lemma follows by Observation 3.5.7. In the other cases, the substitution σ from the premise $P \xrightarrow{\tau}_{\sigma} Q$ has either the form ϵ or $[k, k' \mapsto l]$ by inversion on σ (Lemma 2.1.3). In either case, $H' = \sigma(H)$ follows by definition of applying a historyless substitution to a history (Def. 3.5.6), and therefore the lemma is proved.

Property 2 follows by inversion on the transition $P \xrightarrow{k(a)}_{\sigma} Q$, because such transition is only possible when $\sigma = [k \mapsto l]$ by Lem. 2.1.3. Property 3 follows straightforwardly by inversion. □

A similar result holds for weak actions:

Corollary 3.5.11. *For any well-formed configuration* $(H \rhd P)$*:*

1. *if* $(H \rhd P) \stackrel{\tau}{\Rightarrow}{}^{\mathtt{u}} (H' \rhd P')$, *then* $\exists \sigma. P \stackrel{\tau}{\Rightarrow}{}^{\mathtt{hl}}_{\sigma} P'$ *and* $H' = \sigma(H)$

2. *if* $P \stackrel{\tau}{\Rightarrow}{}^{\mathtt{hl}}_{\sigma} P'$ *and* $H' = \sigma(H)$, *then* $(H \rhd P) \stackrel{\tau}{\Rightarrow}{}^{\mathtt{u}} (H' \rhd P')$

3. *if* $(H \rhd P) \stackrel{k(a)}{\Longrightarrow}{}^{\mathtt{u}} (H' \rhd P')$, *then* $\exists \sigma, \sigma', k'. P \stackrel{(\sigma'(k'))(a)}{\Longrightarrow}{}^{\mathtt{hl}}_{\sigma} P'$ *and* $H' = \sigma(H), \sigma'(k(a))$

4. *if* $P \stackrel{(\sigma'(k))(a)}{\Longrightarrow}{}^{\mathtt{hl}}_{\sigma} P'$ *and* $H' = \sigma(H), \sigma'(k(a))$, *then* $\exists k. (H \rhd P) \stackrel{k(a)}{\Longrightarrow}{}^{\mathtt{u}} (H' \rhd P')$

Proof. Property 1 and 2 are consequences of Lem. 3.5.10 (1) and (2) respectively. Property 3 follows from Prop. 3 and 5 of the same lemma; and Property 4 from Prop. 4 and 6. □

Finally, as it can be easily expected, the history join of two action consistent histories produces a consistent dependency set. The converse is not true however: a consistent history join can be generated from two inconsistent histories. For example, when $H_1 = a$ and $H_2 = b$, $(H_1 \bowtie H_2) = \{(\mathtt{co}, \mathtt{co})\}$, but obviously H_1 and H_2 are not action consistent. Because of this issue, special care has to be taken when proving that a historyless bisimulation is an uniform history bisimulation.

We now have sufficient technical equipment to prove that historyless and uniform history bisimulation coincide. We start with the forward direction, namely that a historyless bisimulation is also a uniform history bisimulation:

Lemma 3.5.12. *Consider the following relation:*

$$\mathcal{R} = \{\ ((H_1 \rhd P_1), (H_2 \rhd P_2)) \mid H_1 \text{ and } H_2 \text{ are action consistent}, P_1 \approx_{\mathsf{hl}}^{(H_1 \bowtie H_2)} P_2\}.$$

$\mathcal{R}$ *is a renaming bisimulation.*

Proof. Let $C_1 = (H_1 \rhd P_1)$ and $C_2 = (H_2 \rhd P_2)$ be related by $\mathcal{R}$, and let $\Delta = (H_1 \bowtie H_2)$. By definition of $\mathcal{R}$, the histories H_1 and H_2 are action consistent and $P_1 \approx_{\mathsf{hl}}^{\Delta} P_2$ holds. We have to show that Condition 1, 2, and 3 of renaming bisimulation from Def. 3.4.8 hold for any $(\mathcal{C}_1, \mathcal{C}_2)$ in $\mathcal{R}$.

Condition 1, namely that histories H_1 and H_2 are action consistent, holds trivially by hypothesis. In order to prove Condition 2, let $\mathcal{C}_1 \xrightarrow{\zeta_1}{}^{\mathtt{u}} \mathcal{C}_1'$ be a challenger move. We have to find a transition $\mathcal{C}_2 \stackrel{\zeta_2}{\Rightarrow}{}^{\mathtt{u}} \mathcal{C}_2'$ in the Uniform History LTS such that $\zeta_1 \sim \zeta_2$ (i.e. if $\zeta_1 = \tau$ then $\zeta_2 = \tau$, and if $\zeta_1 = k(a)$ then $\zeta_2 = d(a)$ for some dependency element d) and $\mathcal{C}_1' \mathcal{R} \mathcal{C}_2'$. We prove the lemma by taking cases on ζ_1.

1. When $\zeta_1 = \tau$, the following derivations hold:

1.	$(H_1 \rhd P_1) \xrightarrow{\tau}{}^{\mathtt{u}} (H_1' \rhd P_1')$	by hypothesis
2.	$P_1 \xrightarrow[\sigma_1]{\tau}{}^{\mathtt{hl}} P_1'$ and $H_1' = H_1\sigma_1$	by Prop. 3.5.10
3.	$P_2 \stackrel{\tau}{\Rightarrow}{}^{\mathtt{hl}}_{\sigma_2} P_2'$	by $\approx_{\mathsf{hl}}$ and 1
4.	$P_1' \approx_{\mathsf{hl}}^{\Delta'} P_2'$	by $\approx_{\mathsf{hl}}$, where $\Delta' = \sigma_1 \Delta \sigma_2$
5.	$C_2 \stackrel{\tau}{\Rightarrow}{}^{\mathtt{u}} (H_2' \rhd P_2')$ and $H_2' = H_2\sigma_2$	by Cor. 3.5.11
6.	$\Delta' = \sigma_1(H_1 \bowtie H_2)\sigma_2$	because $\Delta = (H_1 \bowtie H_2)$
7.	$= (H_1\sigma_1 \bowtie H_2\sigma_2)$	by Prop. 3.5.9
8.	$= (H_1' \bowtie H_2')$	by 3 and 7
9.	H_1' and H_2' are action consistent	because H_1, H_2 are act. cons. and Δ' is cons.
10.	$(H_1' \rhd P_1') \mathcal{R} (H_2' \rhd P_2')$	by 4 and 9

   Assuming that $(H_1 \rhd P_1) \xrightarrow{\tau}{}^{\mathtt{u}} (H_1' \rhd P_1')$ is a challenger move (1), this transition can be replicated in the historyless LTS by Prop. 3.5.10 (2), and it is also a challenger move. Since P_1 and P_2 are historyless bisimilar under Δ, P_2 can match P_1's transition in the bisimulation game (3,4). P_2's move can be translated back in the History LTS (5) by Lem. 3.5.11. The final dependency set $\sigma_1(H_1 \bowtie H_2)\sigma_2$ can be rewritten as the join of histories H_1' and H_2' (8-10) by Prop. 3.5.9, since $H_1' = H_1\sigma_1$ (2) and $H_2' = H_2\sigma_2$ (5). Since H_1 and H_2 are action consistent by hypothesis; historyless substitutions do not change action names; and substitutions σ_1 and σ_2 produce a consistent dependency set Δ', then H_1' and H_2' are action consistent too (9). Derivations (4) and (9) prove the lemma (10).

2. When $\zeta = k(a)$, the following derivations hold:

1.	$(H_1 \rhd P_1) \xrightarrow{k(a)}{}^{\mathtt{u}} (H_1' \rhd P_1')$	by hypothesis
2.	$P_1 \xrightarrow{k(a)}{}^{\mathtt{hl}}_{\sigma_1} P_1'$ and $H_1' = \sigma_1(H_1), k(a)$	by Prop. 3.5.10
3.	$P_2 \xRightarrow{d(a)}{}^{\mathtt{hl}}_{\sigma_2} P_2'$	by $\approx_{\mathsf{hl}}$
4.	$P_1' \approx_{\mathsf{hl}}^{\Delta'} P_2$	with $\Delta' = \sigma_1(H_1 \bowtie H_2)\sigma_2 \cup \{\sigma_1(k), d\}$
5.	$\mathcal{C}_2 \xRightarrow{k'(a)}{}^{\mathtt{u}} (H_2' \rhd P_2')$ and $d = \sigma_2'(k')$	by Prop. 3.5.10 for some k'
6.	$H_2' = \sigma_2(H_2), \sigma_2'(k'(a))$	”
7.	$\Delta' = (\sigma(H_1), k(a) \bowtie \sigma_2(H_2), \sigma_2'(k'(a)))$	by Def. 3.5.5 and Prop 3.5.9
8.	H_1' and H_2' are action consistent	because H_1, H_2 are act. cons. and Δ' is cons.
9.	$(H_1' \rhd P_1') \mathcal{R} (H_2' \rhd P_2')$	by 4 and 8

The proof for the case $\zeta = k(a)$ is similar to the proof for the case $\zeta = \tau$.

□

Extended history bisimulation can be proved to be a historyless bisimulation too:

Lemma 3.5.13. *Consider the following relation $\mathcal{R}$:*

$$\mathcal{R} = \{\, (P, \Delta, Q) \,|\, (H_1 \rhd P) \approx_{\mathsf{r}} (H_2 \rhd Q), \Delta = (H_1 \bowtie H_2), \text{ for some } H_1, H_2 \,\}$$

$\mathcal{R}$ is a historyless bisimulation.

Proof. Let $C_1 = (H_1 \rhd P_1), C_2 = (H_2 \rhd P_2), \Delta = (H_1 \bowtie H_2)$, and let $C_1 \approx_{\mathsf{r}} C_2$. We have to show that the four conditions from Def. 3.5.4 of historyless bisimulation hold.

Condition 1, namely that, $\Delta = (H_1 \bowtie H_2)$ is consistent, is a straightforward consequence of the fact that H_1 and H_2 are action consistent. In order to prove Condition 2, we have to show that if $P_1 \xrightarrow{\tau}{}^{\mathtt{hl}}_{\sigma_1} P_1'$, then $P_2 \xRightarrow{\tau}{}^{\mathtt{hl}}_{\sigma_2} P_2'$ and $(P_1', \sigma_1\Delta\sigma_2, P_2') \in \mathcal{R}$.

The following derivation hold:

1.	$P_1 \xrightarrow{\tau}{}^{\mathtt{hl}}_{\sigma_1} P_1'$	by hypothesis
2.	$(H_1 \rhd P_1) \xrightarrow{\tau}{}^{\mathtt{u}} (H_1' \rhd P_1')$ and $H_1' = H_1\sigma_1$	by Prop. 3.5.10
3.	$(H_2 \rhd P_2) \xRightarrow{\tau}{}^{\mathtt{u}} (H_2' \rhd P_2')$	by $\mathcal{C}_1 \approx_{\mathsf{r}} \mathcal{C}_2$ and 2
4.	$(H_1' \rhd P_1') \approx_{\mathsf{r}} (H_2' \rhd P_2')$	”
5.	$P_2 \xRightarrow{\tau}{}^{\mathtt{hl}}_{\sigma_2} P_2'$ and $H_2' = H_2\sigma_2$	by Prop. 3.5.10 on 3
6.	$\sigma_1\Delta\sigma_2 = \sigma_1(H_1 \bowtie H_2)\sigma_2$	because $\Delta = (H_1 \bowtie H_2)$
7.	$= (H_1' \bowtie H_2')$	by Prop. 3.5.9
8.	$(P_1', \sigma_1\Delta\sigma_2, P_2') \in \mathcal{R}$	by 4, 7

The proof is similar to the proof of Condition 2 of Lemma 3.5.12.

Let us prove Condition 3, namely that if $P_1 \xrightarrow{k(a)}{}^{\mathtt{hl}}_{\sigma} P_1'$ is a challenger move and $k \sharp P_2, \Delta$, then $P_2 \xRightarrow{d(a)}{}^{\mathtt{hl}}_{\sigma_{123}} P_2'$ and $(P_1', \sigma_1\Delta\sigma_2 \cup \{(k, d\}, P_2') \in \mathcal{R}$.

The following derivation proves Condition 3:

1.	$P_1 \xrightarrow{k(a)}{}^{\mathtt{hl}}_{\sigma_1} P_1'$ and $k \sharp P_1$	by hypothesis
2.	$(H_1 \rhd P_1) \xrightarrow{k(a)}{}^{\mathtt{u}} (H_1' \rhd P_1')$	by Prop. 3.5.10
3.	$H_1' = H_1\sigma_1, k(a)$	”
4.	$(H_2 \rhd P_2) \stackrel{k(a)}{\Longrightarrow}{}^{\mathtt{u}} (H_2' \rhd P_2')$	because $\mathcal{C}_1 \approx_{\mathsf{r}} \mathcal{C}_2$ and 2
5.	$(H_1' \rhd P_1') \approx_{\mathsf{r}} (H_2' \rhd P_2')$	”
6.	$P_2 \stackrel{d(a)}{\Longrightarrow}{}^{\mathtt{hl}}_{\sigma_2} P_2'$	by Prop. 3.5.10
7.	$H_2' = H_2, \sigma_2'(k'(a))$ and $d = \sigma_2'(k')$	” for some σ_2' and k'
8.	$\Delta' = (H_1' \bowtie H_2')$	
9.	$= (H_1\sigma_1, k(a) \bowtie H_2\sigma_2, \sigma_2'(k'(a)))$	by 3 and 7
10.	$= (H_1\sigma_1 \bowtie H_2\sigma_2) \cup (k, d)$	by Def. 3.5.5
11.	$= \sigma_1(H_1 \bowtie H_2)\sigma_2 \cup (k, d)$	by Prop. 3.5.9
12.	$= \sigma_1\Delta\sigma_2 \cup (k, d)$	because $\Delta = (H_1 \bowtie H_2)$ by hypothesis
13.	$(P_1', \Delta', P_2') \in \mathcal{R}$	by 5, 12

The proof for Condition 4 is similar to the proof of Condition 2 and 3.

□

We conclude the chapter by proving that extended history bisimulation and historyless bisimulation coincide. Together with Theorem 3.3.9, this result proves that history and historyless bisimulation coincide too.

Theorem 3.5.14. *For any TCCS$^{\mathcal{M}}$ processes P and Q, $P \approx_{\mathsf{r}} Q$ if and only if $P \approx_{\mathsf{hl}} Q$.*

Proof. To prove the first direction of the theorem, suppose that $P \approx_{\mathsf{hl}} Q$. By definition, $P \approx_{\mathsf{hl}} Q$ holds if and only if $P \approx^{\emptyset}_{\mathsf{hl}} Q$ holds, which can also be reformulated as $P \approx^{(\emptyset \bowtie \emptyset)}_{\mathsf{hl}} Q$, since the join of two empty histories is the empty relation. The two empty histories are also action consistent, because they are empty. From these two facts, we can infer $(\emptyset \rhd P)\mathcal{R}(\emptyset \rhd Q)$ as per Lemma 3.5.12. Since $\approx_{\mathsf{r}}$ is the largest weak bisimilarity, it follows that $(\emptyset \rhd P) \approx_{\mathsf{r}} (\emptyset \rhd Q)$, and therefore $P \approx_{\mathsf{r}} Q$.

To prove the opposite direction of the theorem, suppose that $P \approx_{\mathsf{r}} Q$. By definition of history bisimulation, $P \approx_{\mathsf{r}} Q$ if and only if $(\emptyset \rhd P) \approx_{\mathsf{r}} (\emptyset \rhd Q)$. The history join of two empty sets is the empty set over pairs (i.e. $(\emptyset \bowtie \emptyset) = \emptyset$). Since this history join is trivially consistent and $(\emptyset \rhd P) \approx (\emptyset \rhd Q)$ holds by hypothesis, we have that $(P, (\emptyset \bowtie \emptyset), Q)$ belongs to $\mathcal{R}$ as per Lemma 3.5.13. Therefore $P \approx^{(\emptyset \bowtie \emptyset)}_{\mathsf{hl}} Q$ holds, which implies that $P \approx_{\mathsf{hl}} Q$ holds by definition of $\approx_{\mathsf{hl}}$. □

Chapter 4

Historyless bisimulation algorithm

The present chapter presents an algorithm to calculate a bisimulation equivalence between two TCCS$^{\mathrm{m}}$ processes, if one such relation exists. In order to guarantee termination, the bisimulation algorithms that we consider only terminate if their inputs are finite-state, but as discussed in the introduction to Chapter 2, histories can grow indefinitely in a history bisimulation game.

In Section 3.5 we described historyless bisimulation, where histories are replaced by a single environment Δ, the dependency set that relates one player's transactions to the other player's transactions and vice versa. Historyless bisimulation is not enough to guarantee a finite state LTS yet. There are two ways in which TCCS$^{\mathrm{m}}$ processes can yield an infinite LTS: by picking a new transaction name from an infinite supply of names, and by recursively spawning fresh processes or channels.

The first problem is illustrated by in Fig. 4.1.a. Let P be $[\![a.\mathtt{co} \triangleright_k 0]\!]$, a transaction whose body can only perform a tentative action a. By Rule HL-$k(a)$, the following are all valid transitions according to the LTS in Fig. 3.4:

$$P \xrightarrow{l_1(a)}{}^{\mathtt{hl}}_{[k \mapsto l_1]} [\![\mathtt{co} \triangleright_{l_1} 0]\!]$$
$$P \xrightarrow{l_2(a)}{}^{\mathtt{hl}}_{[k \mapsto l_2]} [\![\mathtt{co} \triangleright_{l_2} 0]\!]$$
$$P \xrightarrow{l_3(a)}{}^{\mathtt{hl}}_{[k \mapsto l_3]} [\![\mathtt{co} \triangleright_{l_3} 0]\!]$$
$$\ldots$$

The Historyless LTS allows transitions that rename a transaction with any fresh name, picked from an infinite set. Therefore the set of successors of P is infinite too; it contains a state $[\![P' \triangleright_{l_i} Q]\!]$ for each fresh transaction name l_i in the domain of all transaction names $\mathcal{T} \backslash \{k\}$.

Theorem 3.4.12 shows that the choice of a fresh name over another is not important as far as bisimulations are concerned. Section 4.1 shows that it is possible to reduce the choice of fresh labels from an infinite set of names to exactly one name, without losing distinguishing power.

Another source of infinity is due to recursive processes, as illustrated in Fig. 4.1.b.

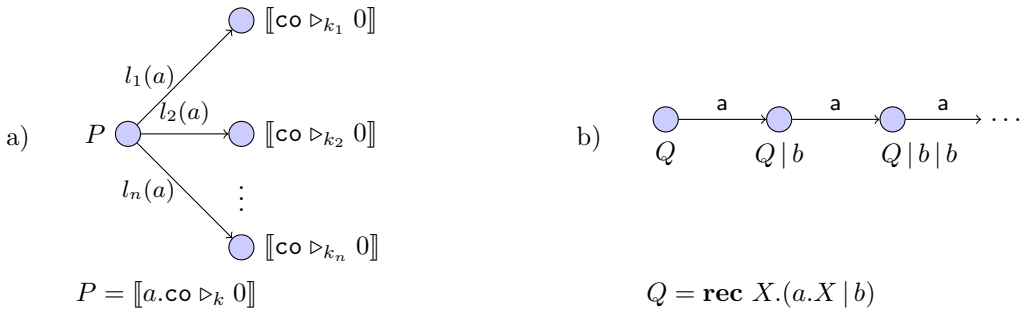


Fig. 4.1: Both processes P and Q yield infinite state LTSs.

Let $Q = \mathtt{rec}\ X.a.0 \mid X$. The following is a sequence of valid transitions from Q:

$$\begin{aligned} Q &\xrightarrow{\tau}{}^{\mathtt{hl}}_{\epsilon} Q \mid b \\ &\xrightarrow{\tau}{}^{\mathtt{hl}}_{\epsilon} Q \mid b \mid b \\ &\xrightarrow{\tau}{}^{\mathtt{hl}}_{\epsilon} Q \mid b \mid b \mid b \\ &\dots \end{aligned}$$

Each unfolding of the recursive process in Q spawns a new b process. The recursive process Q is still available at each unfolding, therefore Q can spawn an infinite number of b processes, and therefore Q yields an infinite state LTS.

Section 4.2 describes a class of processes that yield a finite state LTS by design. This is achieved by a syntactic restriction on TCCS$^{\mathrm{m}}$ processes, namely by disallowing spawning processes or creating fresh channels under a recursive process. Finally Section 4.3 describes an algorithm to calculate bisimulation equivalences between TCCS$^{\mathrm{m}}$ processes with finite state LTSs. The algorithm is an adaptation of the *on-the-fly* algorithm from [Bergstra, 2001] to the transactional case.

4.1 Nameless LTS

As mentioned in the introduction to this chapter, each transition in TCCS$^{\mathrm{m}}$ can substitute the name of one or more transactions with a fresh name. Since the set of transaction names is infinite, there is an infinite number of fresh names that transactions can be renamed with. This is not the only case where a process yields an infinite state LTS because of fresh names: the same problem arises when a dormant transaction is activated. For example, a simple process such as $Q = [\![0 \blacktriangleright 0]\!]$ generates an infinite state LTS, because Q can activate its dormant transaction by transition $[\![0 \blacktriangleright 0]\!] \xrightarrow{\mathtt{new}\ l} [\![0 \triangleright_l 0]\!]$ for any $k \in \mathcal{T}$.

In order to have finite state LTSs even for these simple processes, it is necessary to restrict the choice of fresh names to a single one. This can be achieved by imposing a total order $\leq_{\mathcal{T}}$ on $\mathcal{T}$, and by forcing the LTS to only pick the least fresh name available from $\mathcal{T}$ minus the transaction names currently used in a process. Each distinct transaction name k in $\mathcal{T}$ is assigned a unique natural number i; we indicate with k_i the i-th transaction name. The ordering $\leq_{\mathcal{T}}$ has the property that $k_i \leq k_j$ whenever $i \leq j$. The ordering $\leq_{\mathcal{T}}$ is a simple lifting of the natural number ordering to transaction

NL-Act $\dfrac{P \xrightarrow{\xi}_{\epsilon} Q}{P \xrightarrow{\xi}^{\mathtt{nl}}_{\epsilon} Q}$ NL-FreshAct $\dfrac{P \xrightarrow{\xi}_{\tilde{l}\mapsto k} Q}{P \xrightarrow{\xi}^{\mathtt{nl}}_{\tilde{l}\mapsto k} Q}$ $k = min(\mathcal{T}\backslash fn(P))$ NL-Co $\dfrac{P \xrightarrow{\mathtt{co}\,k} Q}{P \xrightarrow{\tau}^{\mathtt{nl}}_{[k\mapsto \mathtt{co}]} Q}$ NL-Ab $\dfrac{P \xrightarrow{\mathtt{ab}\,k} Q}{P \xrightarrow{\tau}^{\mathtt{nl}}_{[k\mapsto \mathtt{ab}]} Q}$ NL-New $\dfrac{P \xrightarrow{\mathtt{new}\ k} Q}{P \xrightarrow{\tau}^{\mathtt{nl}}_{\epsilon} Q}$ $k = min(\mathcal{T}\backslash fn(P))$

Fig. 4.2: Nameless LTS.

names, and therefore it is total by construction. The bottom element of $\leq_{\mathcal{T}}$ is k_0. For ease of notation we write $k_i + k_j$ for k_{i+j}.

The Historyless LTS from Fig. 3.4 is therefore refined into the *Nameless LTS* as shown in Fig. 4.2. When no fresh name is introduced, Rule NL-Act simply reproduces the transition performed in its premise. When a fresh transaction name is introduced (e.g. because of a $k(\mu)$ action), the Historyless LTS allows a process to pick any fresh name k from an infinite supply of names (namely $\mathcal{T}\backslash fn(P,Q)$, if P and Q are playing a bisimulation games).

Rule NL-FreshAct only allows processes to pick exactly one name k, that is the least element of $\mathcal{T}$ minus the transaction names in P. Since the order $\leq_{\mathcal{T}}$ on $\mathcal{T}$ is total and with bottom, there is always one and only one such name. Similarly, when a fresh transaction is activated, Rule NL-New allows process P to pick exactly one name k, which is the least $\mathcal{T}$ minus $fn(P)$. There is only one such transition for the same reason as for Rule NL-FreshAct. When a process P commits or aborts a transaction k, the name k does not occur anymore in P. Therefore Rule NL-Co and NL-Ab simply reproduce the TCCS$^{\mathrm{m}}$ transition in their premises.

Since Rule NL-FreshAct and NL-New restrict the choice of fresh transactions to exactly one, TCCS$^{\mathrm{m}}$ processes's LTSs are finite image, i.e. they do not support an infinite number of transitions. The definition of nameless bisimulation is similar to the definition of historyless bisimulation from Def. 3.5.4:

Definition 4.1.1. ***(Nameless bisimulation)*** *A ternary relation $\mathcal{R} \subseteq \mathcal{P}\times\mathcal{D}\times$ is a weak bisimulation when for all $(P,\Delta,Q) \in \mathcal{R}$:*

1. *Δ is consistent,*
2. *if $P \xrightarrow{k(a)}_{\sigma_1} P'$, then $Q \overset{d(a)}{\Longrightarrow}_{\sigma_2} Q'$ and $P' \approx_{\mathsf{hl}}^{\Delta'} Q'$, where $\Delta' = \sigma_1\Delta\sigma_2 \cup \{(k,d)\}$,*
3. *if $P \xrightarrow{\tau}_{\sigma_1} P'$, then $Q \overset{\tau}{\Rightarrow}_{\sigma_2} Q'$ and $P' \approx_{\mathsf{hl}}^{\sigma_1\Delta'\sigma_2} Q'$*
4. *the converse of the previous two conditions.*

We write $\approx_{\mathsf{nl}}$ for the largest nameless bisimulation. Two TCCS$^{\mathrm{m}}$ processes are nameless bisimilar, or $P \approx_{\mathsf{nl}} Q$, when $P \approx_{\mathsf{nl}}^{\emptyset} Q$ holds.

The proof that historyless and nameless bisimulation coincide is a consequence of the fact that processes are bisimilar up to renamings:

Lemma 4.1.2. *Let $P \approx_{\mathsf{hl}}^{\Delta} Q$. For any transaction renaming π_1 and π_2 such that $range(\pi_1) \sharp P$ and $range(\pi_2) \sharp Q$, $P \approx_{\mathsf{hl}}^{\Delta} Q$ implies $\pi_1(P) \approx_{\mathsf{hl}}^{\pi_1\Delta\pi_2} \pi_2(Q)$.*

Proof. By proving that the relation

$$\mathcal{R} = \{\ (\pi_1(P), \pi_1 \Delta \pi_2, \pi_2(Q)) \mid range(\pi_1) \sharp P \text{ and } range(\pi_2) \sharp Q\ \}$$

is a historyless bisimulation. The lemma is proved as Prop. 3.4.6. □

Lemma 4.1.3. *Let $P \approx_{\mathsf{nl}}^{\Delta} Q$. For any transaction renaming π_1 and π_2 such that $range(\pi_1) \sharp P$ and $range(\pi_2) \sharp Q$, $P \approx_{\mathsf{nl}}^{\Delta} Q$ implies $\pi_1(P) \approx_{\mathsf{nl}}^{\pi_1 \Delta \pi_2} \pi_2(Q)$.*

Proof. The lemma is proved as in the previous lemma, by proving that the relation

$$\mathcal{R} = \{\ (\pi_1(P), \pi_1 \Delta \pi_2, \pi_2(Q)) \mid range(\pi_1) \sharp P \text{ and } range(\pi_2) \sharp Q\ \}$$

is a nameless bisimulation. □

We can now prove that historyless and nameless bisimulation relate the same processes:

Theorem 4.1.4. *Let P and Q be $TCCS^m$ processes. Then $P \approx_{\mathsf{hl}} Q$ if and only if $P \approx_{\mathsf{nl}} Q$.*

Proof. We have to show that $\approx_{\mathsf{hl}}$ is a nameless bisimulation, and that $\approx_{\mathsf{nl}}$ is a historyless bisimulation. We will only show that $\approx_{\mathsf{nl}}$ is a historyless bisimulation, since the proof for the other case is similar.

Let $P \approx_{\mathsf{nl}}^{\Delta} Q$. In order to show that $\approx_{\mathsf{nl}}$ is a historyless bisimulation, we have to show the four conditions in Def. 3.5.4 hold for $\approx_{\mathsf{nl}}$. By definition, $P \approx_{\mathsf{nl}} Q$ holds when $P \approx_{\mathsf{nl}}^{\emptyset} Q$ holds, where $\emptyset$ is the empty dependency set. Condition 1 holds trivially, because the empty dependency is trivially consistent.

To prove Condition 2, suppose that $P \xrightarrow[\sigma_1]{k(a)}_{\mathtt{hl}} P'$ for some k. If k is not the least transaction name in $\mathcal{T} \backslash ftn(P)$, then there exists a transaction k' such that $P \xrightarrow[\sigma_1']{k'(a)}_{\mathtt{hl}} P''$, $P'' = \pi(P')$ and $\sigma_1' = \pi\sigma$, where $\pi = [k \mapsto k']$. By nameless bisimulation, there exists a weak transition $Q \overset{d(a)}{\Longrightarrow}_{\mathtt{nl}} \sigma_2 Q'$ such that $P'' \approx_{\mathsf{nl}}^{\Delta'} Q'$, where $\Delta' = \sigma_1' \Delta \sigma_2 \cup \{(k'', d)\} Q'$. Since $k'' = \pi(k)$ and $P'' = \pi(P')$, we can rewrite Δ' as $\Delta' = \pi\sigma_1 \Delta \sigma_2 \cup \pi\{(k, d)\} Q'$ and the theorem is proved by Lem. 4.1.3.

Condition 3 and 4 are proved similarly. □

4.2 Syntactic restrictions for a finite LTS

As already mentioned, bisimulation algorithms only terminate for processes with a finite LTS. In CCS, syntactic restrictions are a common method to identify classes of such processes (cfr. [Milner, 1982]): for any CCS processes P such that no parallel and restriction construct occurs within recursive definitions of the form $\mathtt{rec}\ X.Q$, the LTS of P is finite. Consider in fact the following process:

$$P = \mathtt{rec}\ X.\nu a.X$$

Process P recursively generates channel restrictions ad infinitum. Therefore LTS of P has an infinite number of states, because the set of states of P are processes P enclosed by a series of channel restrictions $\nu a.-$.

If transaction nesting was allowed, the same restriction would have to apply to dormant transactions under recursion. In fact, consider the following process:

$$Q = \texttt{rec}\ X.[\![X \blacktriangleright 0]\!]$$

Process Q does not contain the parallel or restriction constructs, however it recursively activates fresh transactions ad infinitum. As it was the case for process P, process Q would generate an infinite number of states, one for each nested transaction; and therefore dormant transactions should be disallowed syntactically too under the recursion construct, in order to have a finite state LTS.

Such restriction would unfortunately rule out restarting transactions. However in the case of flat transactions it is possible to both allow dormant transactions under recursive processes and obtain a finite state LTS. Section 4.2.1 shows that CCS processes with the aforementioned restrictions are indeed finite state. The proof of this property is omitted in [Milner, 1982], and I could not find a full proof for it in the literature; therefore we provide a full proof here. After showing the finite state LTS result for standard CCS, Section 4.2.2 shows that the same syntactic restriction yields finite state LTS for TCCS^{m} processes.

4.2.1 Finite State CCS

In this section we will prove that CCS processes have a finite set of states they can evolve to, when the parallel construct is forbidden within recursion. Our proof strategy is to provide an upper bound for the set of reachable states of a process P. The upper bound function is defined structurally on a process P, which simplifies the proof of finiteness. The core of the proof is then to show that the set of reachable states of a process is always contained in the upper bound.

Syntax

We call this restricted version of CCS *serial CCS*. Its syntax is defined as follows:

$$\begin{aligned} P, Q &::= \ 0 \mid X \mid \alpha.P \mid \texttt{rec}\ X.S \mid P + Q \mid \nu a.P \mid P \,|\, Q \\ S, T &::= \ 0 \mid X \mid a.S \mid \texttt{rec}\ X.S \mid S + S \end{aligned}$$

Standard CCS processes, such as choice, prefix and restriction, are indicated by the letters P, Q. Processes S and T are *serial* processes, that is, processes that cannot contain any occurrence of the parallel or restriction constructs under recursive definitions. Notice that closed serial processes are a subset of CCS processes.

We prove our results with reference to the following LTS semantics:

$$\frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \frac{}{\texttt{rec}\ X.P \xrightarrow{\tau} P[X \mapsto \texttt{rec}\ X.P]} \qquad \frac{P \xrightarrow{\alpha} P'}{\nu a.P \xrightarrow{\alpha} \nu a.P'}\ a \neq \alpha$$

$$\frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \qquad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\bar{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'} \qquad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$$

Label α ranges over the set $Act \cup \overline{Act} \cup \{\tau\}$. We indicate the set of processes with the letter $\mathcal{P}$, the set of serial processes with the letter $\mathcal{S}$, and the set of variables as Var.

The syntax allows terms with open recursion variables X. We are not interested in them, because we are only interested in terms that can be run in the operational semantics. Therefore we first define the usual notion of free variables:

Definition 4.2.1 (Free Variables)**.** *For a serial CCS process S, the set $FV(S)$ of free variables in S is defined inductively as follows:*

$$FV(0) = \emptyset \qquad FV(\alpha.S) = FV(S) \qquad FV(X) = \{X\}$$
$$FV(\texttt{rec}\ X.S) = FV(S) - \{X\} \qquad FV(P+Q) = FV(P) \cup FV(Q) \qquad FV(\nu a.P) = FV(P)$$

and stipulate that closed terms are not only well-formed, as per Def. 2.1.1, but also do not contain open recursion variables:

Definition 4.2.2 (Closed term)**.** *A term P is* closed *when $FV(P) = \emptyset$.*

We can now give a simple definition of substitution, that replaces an open variable X in a term P with a closed term Q:

Definition 4.2.3 (Term substitution)**.** *Term substitution $[X \mapsto R]$ is defined by the following equations:*

$$\begin{aligned}
0[X \mapsto R] &= 0 \\
X[X \mapsto R] &= R \\
Y[X \mapsto R] &= Y && \text{if } X \neq Y \\
\alpha.P[X \mapsto R] &= \alpha.(P[X \mapsto R]) \\
\texttt{rec}\ X.S[X \mapsto R] &= \texttt{rec}\ X.S \\
\texttt{rec}\ Y.S[X \mapsto R] &= \texttt{rec}\ Y.(S[X \mapsto R]) && \text{if } X \neq Y \\
\nu a.P[X \mapsto R] &= \nu a.(P[X \mapsto R]) \\
(P+Q)[X \mapsto R] &= P[X \mapsto R] + Q[X \mapsto R] \\
(P \,|\, Q)[X \mapsto R] &= P[X \mapsto R] \,|\, Q[X \mapsto R]
\end{aligned}$$

When R is a closed term and X is the only free variable occurring in P, $P[X \mapsto R]$ is a closed term. When R is closed, capture avoidance is unnecessary and therefore we do not need to work with syntactic terms up to renaming of bound variables [Pierce, 2002].

We conclude this section with a technical lemma that shows how to swap the application of two substitutions σ and σ' to the same process S:

Lemma 4.2.4. *Let T, T' be serial processes, and $\sigma = [X \mapsto T]$ and $\sigma' = [Y \mapsto T']$ be two substitutions. For any serial process S, if $X \neq Y$, T is closed and X is not free in T', then $S\sigma\sigma' = S\sigma'[X \mapsto T\sigma']$.*

Proof. We prove this lemma by structural induction on S.

$(S = 0)$: The equalities $0\sigma\sigma' = 0 = 0\sigma'[X \mapsto T\sigma']$ prove the lemma in this case.

$(S = Z)$: There are three cases to consider: $Z = X$, $Z = Y$ and Z is different from both X and Y. In the first case, the following equalities hold:

1. $S\sigma\sigma' = X[X \mapsto T]\sigma'$
2. $= T\sigma'$ by def. of substitution
3. $= X[X \mapsto T\sigma']$ "
4. $= X\sigma'[X \mapsto T\sigma']$ because $X[Y \mapsto T'] = X$
5. $= S\sigma'[X \mapsto T\sigma']$ because $S = X$ by hp.

When $Z = X$, $S\sigma = X[X \mapsto T] = T$, and therefore $S\sigma\sigma' = T\sigma'$ (1-2). By definition of substitution, $T\sigma'$ can be written as $X[X \mapsto T\sigma']$ (3). Since X and Y are different and $\sigma' = [Y \mapsto T']$, $X\sigma' = X$, and $S\sigma\sigma'$ can be rewritten as $X\sigma'[X \mapsto T\sigma']$ (4), which is equal to $S\sigma'[X \mapsto T\sigma']$ (5) because $S = X$ by hypothesis. The lemma is proved by (5).

In the second case, the following equalities hold:

1. $S\sigma\sigma' = Y\sigma[Y \mapsto T']$
2. $= T'$ because $\sigma = [X \mapsto T]$ and $X \neq Y$
3. $= T'[X \mapsto T\sigma']$ because $X \notin FV(T')$
4. $= Y[Y \mapsto T'][X \mapsto T\sigma']$ by def. of substitution
5. $= S\sigma'[X \mapsto T\sigma']$

When $Z = Y$, the equalities $S\sigma\sigma' = Y\sigma\sigma' = Y\sigma' = T'$ hold, since $\sigma = [X \mapsto T]$, $Y[X \mapsto T] = Y$ and $Y[Y \mapsto T']$ all hold by hypothesis and by definition of substitution (1-2). Since X is not free in T', $T' = T'\sigma XT$ holds (3). By definition of substitution we can also write that $T' = Y[Y \mapsto T']$, which gives $S\sigma\sigma' = Y[Y \mapsto T'][X \mapsto T\sigma']$ (4) when combined with the previous equalities (1-3). Since $S = Y$ and $\sigma' = [Y \mapsto T']$, this equality can be rewritten as $S\sigma\sigma' = S\sigma'[X \mapsto T\sigma']$, which proves the lemma.

Finally, let us consider the third case, where variable Z is different from both X and Y:

1. $S\sigma\sigma' = Z\sigma\sigma'$
2. $= Z$ because $Z \neq X, Y$
3. $= Z\sigma'[X \mapsto T\sigma']$ "
4. $= S\sigma'[X \mapsto T\sigma']$

In this case the lemma is proved by first applying σ and σ' to Z, which is equal to Z itself (1-2) since Z is different from both X and Y. By the same token, Z is also equal to $Z\sigma'[X \mapsto T\sigma']$ (3). By definition of substitution this implies $S\sigma\sigma' = S\sigma'[X \mapsto T\sigma']$ (5), which proves the lemma.

$(S = \alpha.S')$: This case is easily proved by the inductive hypothesis:

1. $S\sigma\sigma' = (\alpha.S')\sigma\sigma'$
2. $= \alpha.S'\sigma\sigma'$ by def. of substitution.
3. $= \alpha.S'\sigma'[X \mapsto T\sigma']$ by ind. hypothesis
4. $= (\alpha.S')\sigma'[X \mapsto T\sigma']$ by def. of substitution.
5. $= S\sigma'[X \mapsto T\sigma']$

($S = \texttt{rec}\ Z.S'$) : The proof of the lemma in this case depends on the value of variable Z. We have three cases: $Z = X$, $Z = Y$ and Z different from both X and Y.

If $Z = X$, the following holds:

1. $S\sigma\sigma' = (\texttt{rec}\ X.S')\sigma\sigma'$
2. $= (\texttt{rec}\ X.S')\sigma'$ by def. of substitution on `rec`
3. $= \texttt{rec}\ X.(S'\sigma'[X \mapsto T\sigma'])$
4. $= \texttt{rec}\ X.S'\sigma'[X \mapsto T\sigma']$
5. $= S\sigma'[X \mapsto T\sigma']$

The application of σ to S in (1) results in Equation (2) since $\texttt{rec}\ Z.S'$ is closed in X. Equation (3) follows by definition of substitution. Equality (4) is justified by definition of substitution extension, since variable Z is actually X, and the inner extension $[X \mapsto T\sigma']$ is overridden by the outer one. Equation (5) is obtained again by definition of substitution, and it can be rewritten as Equation (6), which proves the lemma.

If $Z = Y$, then the following holds:

1. $S\sigma\sigma' = (\texttt{rec}\ Y.S')[X \mapsto T]\sigma'$
2. $= (\texttt{rec}\ Y.S'[X \mapsto T])\sigma'$ by def. of substitution
3. $= \texttt{rec}\ Y'.S'[X \mapsto T]$ by def. of substitution on `rec`
4. $= (\texttt{rec}\ Y.S')\sigma'[X \mapsto T\sigma']$ because $\sigma' = [Y \mapsto T']$ and T is closed
5. $= S\sigma'[X \mapsto T\sigma']$

By definition of substitution, $\texttt{rec}\ Y.S'\sigma\sigma' = \texttt{rec}\ Y.S'\sigma\sigma'$ (1-2), which in turn is equal to $\texttt{rec}\ Y.S'\sigma$ (3) by definition of substitution, when Y is both the recursion variable and the domain of σ'. For the same reason, $\texttt{rec}\ Y.S'\sigma = (\texttt{rec}\ Y.S')\sigma = (\texttt{rec}\ Y.S')\sigma'\sigma$ holds. Since T is closed, $T = T\sigma'$ holds, and since $\sigma = [X \mapsto T]$, this implies that $(\texttt{rec}\ Y.S')\sigma'\sigma = (\texttt{rec}\ Y.S')\sigma'[X \mapsto T\sigma']$ (4). Therefore $S\sigma\sigma' = S\sigma'[X \mapsto T\sigma']$ holds, and the lemma is proved.

If Z is different from both X and Y, then the lemma follows straightforwardly by inductive hypothesis.

($S = S' + T$) : The lemma is proved by induction, as for the case $S = \alpha.S'$.

□

Proof strategy

To state the finiteness theorem more precisely, we need to define what the set of reachable states of a process is:

Definition 4.2.5 (Reachable states)**.** *The set of reachable states succ(P) of a finite CCS process P is:*

$$succ(P) = \{s \mid P \Rightarrow s\}$$

where $\Rightarrow = (\xrightarrow{\alpha})^*$ *is the reflexive and transitive closure of* $\xrightarrow{\alpha}$.

The finiteness theorem we are going to prove is the following:

Theorem 4.2.6 (Finite state LTS)**.** *For any process P in finite CCS, $succ(P)$ is a finite set.*

The semantic definition of $succ(P)$ makes a direct proof of this theorem difficult, especially because of the presence of loops in recursive processes. To overcome this problem, we can give an upper bound to the set of reachable states:

Definition 4.2.7. ***(State generating function)***
The state generator function $gen :: Proc \to \mathcal{P}(Proc)$ *is defined as follows:*

$$gen(P) = P \cup \begin{cases} \emptyset & \text{if } P = 0 \\ \{X\} & \text{if } P = X \\ gen(P') & \text{if } P = \alpha.P' \\ gen(S)[X \mapsto \texttt{rec } X.S] & \text{if } P = \texttt{rec } X.S \\ gen(P') \cup gen(Q) & \text{if } P = P' + Q \\ \{\ \nu a.s \mid s \in gen(Q)\ \} & \text{if } P = \nu a.Q \\ \{\ s|s' \mid s \in gen(P'), s' \in gen(Q) & \text{if } P = P'|Q \end{cases}$$

Although not very precise, the *gen* function contains all the possible states that a process can evolve to, plus many spurious ones, such as after forbidden communications over a restriction, or synchronizations over mismatching channels. But since we are only interested in proving the finiteness of restricted CCS processes, the upper bound provided by *gen* will be sufficient.

It is easy to see that *gen* only generates finite sets:

Lemma 4.2.8 (Finiteness of *gen*-sets)**.** *For any finite CCS process $P, gen(P)$ is a finite set.*

Proof. By structural induction on P. □

If we can show that $succ(P) \subseteq gen(P)$, then by Lemma 4.2.8 we can prove that $succ(P)$ is finite too, because the subset of a finite set is finite as well.

Finiteness of serial processes

Before proving that finiteness for restricted CCS processes, we prove finiteness for serial processes.

A serial process can only transition to another serial process:

Lemma 4.2.9. *Let S be a closed serial process such that $S \neq X$, and let σ be a substitution $[X \mapsto S_0]$ where S_0 is a closed term. If $S\sigma \xrightarrow{\alpha} P$, then there exists a serial process T such that $S \xrightarrow{\alpha} T$ and $P = T\sigma$.*

Proof. We prove this lemma by induction on the structure of S.

$(S = 0)$: This case is trivial, because process 0 cannot take any transitions.

$(S = Y)$: This case is trivial too, because S is different from X by hypothesis, and therefore $S\sigma = Y\sigma = Y$, and Y cannot take any transitions.

$(S = \alpha.S')$: By definition of substitution, $(\alpha.S')\sigma = \alpha.S'\sigma$. The only transition that this process can take is $\alpha.S'\sigma \to S'.\sigma$. The serial process S' proves the lemma.

$(S = \texttt{rec}\ Y.S')$: We must distinguish two cases: either $Y = X$, or $Y \neq X$.

$(X = Y)$: When $X = Y$, $(\texttt{rec}\ X.S')\sigma = \texttt{rec}\ X.S'$ holds by definition of substitution, and we have:

1. $\texttt{rec}\ X.S' \to S'[X \mapsto \texttt{rec}\ X.S']$ by def. of the LTS
2. $S'[X \mapsto \texttt{rec}\ X.S'] = S'[X \mapsto \texttt{rec}\ X.S']\sigma$ because $S'[X \mapsto \texttt{rec}\ X.S']$ is closed
3. $\texttt{rec}\ X.S'\sigma \to S'[X \mapsto \texttt{rec}\ X.S']\sigma$

By definition of the LTS for recursive processes, S can take the transition $\texttt{rec}\ X.S' \to S'[X \mapsto \texttt{rec}\ X.S']$ (1). Since S is a closed term, then $S'[X \mapsto \texttt{rec}\ X.S']$ is closed too, and $S'[X \mapsto \texttt{rec}\ X.S'] = S'[X \mapsto \texttt{rec}\ X.S']\sigma$ (2) holds because $\sigma = [X \mapsto S_0]$. Therefore, because of (2) and because $(\texttt{rec}\ X.S')\sigma = \texttt{rec}\ X.S'$, we can rewrite (1) as $\texttt{rec}\ X.S'\sigma \to S'[X \mapsto \texttt{rec}\ X.S']\sigma$ (3); taking $T = S'[X \mapsto \texttt{rec}\ X.S']$ proves the lemma.

$(X \neq Y)$: In the second case, when $S = \texttt{rec}\ Y.S'$ and $Y \neq X$, the following transitions hold:

1. $S \xrightarrow{\tau} S'[Y \mapsto \texttt{rec}\ Y.S']$ by def. of the LTS
2. $S\sigma \xrightarrow{\tau} S'\sigma[Y \mapsto \texttt{rec}\ Y.S'\sigma]$ “
3. $S'\sigma[Y \mapsto \texttt{rec}\ Y.S'\sigma] = S'[Y \mapsto \texttt{rec}\ Y.S']\sigma$ by Lem. 4.2.4
4. $S\sigma \xrightarrow{\tau} S'[Y \mapsto \texttt{rec}\ Y.S']\sigma$ by 2,3

In this case the substitution σ does apply to S, i.e. $(\texttt{rec}\ Y.S')\sigma = \texttt{rec}\ Y.S'\sigma$. According to the LTS, $\texttt{rec}\ Y.S'$ can take transition $S \xrightarrow{\tau} S'[Y \mapsto \texttt{rec}\ Y.S']$ (1); therefore $S\sigma$ can take transition $S\sigma \xrightarrow{\tau} S'\sigma[Y \mapsto \texttt{rec}\ Y.S'\sigma]$ too (2). Since $\sigma = [X \mapsto S_0]$, and both $\texttt{rec}\ Y.S'$ and S_0 are closed terms, the equality $S'\sigma[Y \mapsto \texttt{rec}\ Y.S'\sigma] = S'[Y \mapsto \texttt{rec}\ Y.S']\sigma$ (3) holds by Lemma 4.2.4. Thus we can rewrite the previous transition as $S\sigma \xrightarrow{\tau} S'[Y \mapsto \texttt{rec}\ Y.S']\sigma$ (4). The lemma is proved by taking $T = S'[Y \mapsto \texttt{rec}\ Y.S']$ as witness.

$(S = S' + S'')$: by inductive hypothesis on either branch of the choice.

□

From the previous lemma, we can infer the processes that a recursive process can evolve to, as stated in the following corollary:

Corollary 4.2.10. *Let S be a sequential process and σ be the substitution $[X \mapsto \texttt{rec}\ X.S]$. If $\texttt{rec}\ X.S \Rightarrow S'$, then either there exists a serial process T such that $S \Rightarrow T$ and $S' = T\sigma$, or $S' = \texttt{rec}\ X.S$.*

Proof. We prove this corollary by induction on the length n of the derivation $\texttt{rec}\ X.S \Rightarrow^n S'$.

(k = 0): If the length of the derivation is 0, then by reflexivity we have:

$$\texttt{rec}\ X.S \Rightarrow \texttt{rec}\ X.S$$

If we take $S' = \texttt{rec}\ X.S$, then S' is the witness that proves the lemma.

(n = k+1): If the length of the derivation is $k+1$, by inductive hypothesis we assume that $\texttt{rec}\ X.S \Rightarrow^k S'' \xrightarrow{\alpha} S'$, and either there exists a serial process T' such that $S \Rightarrow T'$ and $S'' = T'\sigma$, or S'' is $\texttt{rec}\ X.S$. Moreover, we can further divide the first case in two sub-cases: in the first sub-case, T' is X; in the second sub-case, T' is not the variable X, i.e. $T \neq X$. We prove the lemma for the three cases outlined above.

If $T' = X$, then $S'' = T'\sigma = \texttt{rec}\ X.S$. According to the LTS, the only transition that $\texttt{rec}\ X.S$ can take is $\texttt{rec}\ X.S \xrightarrow{\tau} S[X \mapsto \texttt{rec}\ X.S]$, which can be rewritten as $S'' \xrightarrow{\tau} S\sigma$. If we take $T = S$, then T is the witness that proves the lemma by the last derivation and the reflexive weak transition $S \Rightarrow S$.

Let us consider the case $S \neq X$. Since $S'' \xrightarrow{\alpha} S'$ and $S'' = T'\sigma$, by Lemma 4.2.9 $T'\sigma \xrightarrow{\alpha} S'$ implies that there exists a serial process T'' such that $T' \xrightarrow{\alpha} T''$ and $S' = T''\sigma$. The lemma is proved by the witness T'', since $S \Rightarrow T' \xrightarrow{\alpha} T''$ implies that $S \Rightarrow T''$, and $S' = T''\sigma$.

Lastly, the proof for the case $S'' = \texttt{rec}\ X.S$ is the same as in the first case, when $S'' = X$.

□

We can now show that the $gen(S)$ function is an upper bound to $succ(S)$ for any serial process S:

Lemma 4.2.11. *For any serial process S, $succ(S) \subseteq gen(S)$.*

Proof. We prove this lemma by structural induction on S.

$(S = 0)$: The lemma holds because $succ(0) = \{0\} = gen(0)$

$(S = \alpha.S')$: According to the LTS, the only transition that S can take is $\alpha.S' \xrightarrow{\alpha} S'$. Thus the only successors of $\alpha.S'$ are $\alpha.S'$ itself (by reflexivity) and the successors of S', i.e. $succ(\alpha.S') = \{\alpha.Q_s\} \cup succ(S')$. By definition of gen, $gen(\alpha.S') = \{\alpha.Q_s\} \cup gen(S')$. Since the singleton set $\{\alpha.S'\}$ is common to both $succ(S)$ and $gen(S)$, and $succ(S') \subseteq gen(S')$ holds by inductive hypothesis, the lemma is proved.

$(S = X)$: The lemma holds because $succ(X) = \{X\} = gen(X)$

$(S = \texttt{rec}\ X.S')$: We need to show that $succ(\texttt{rec}\ X.S') \subseteq gen(\texttt{rec}\ X.S')$. We prove this statement by showing that, for any serial process S_0 that $\texttt{rec}\ X.S'$ can evolve to, S_0 is contained in $gen(\texttt{rec}\ X.S')$. If we can prove this result for any successor S_0, it follows that all successors of S are contained in $gen(S)$, and thus that $succ(\texttt{rec}\ X.S') \subseteq gen(\texttt{rec}\ X.S')$, which proves the lemma.

We prove that, if $\texttt{rec}\ X.S' \Rightarrow S_0$, then $S_0 \in gen(\texttt{rec}\ X.S')$, by induction on the length of the derivation $\texttt{rec}\ X.S' \Rightarrow^n S_0$.

$(n = 0)$: By reflexivity, we have that $\texttt{rec}\ X.S' \Rightarrow \texttt{rec}\ X.S'$. By definition, $gen(\texttt{rec}\ X.S')$ is $\{\texttt{rec}\ X.S'\} \cup gen(S')[X \mapsto \texttt{rec}\ X.S']$, and thus $\texttt{rec}\ X.S'$ is contained in $gen(\texttt{rec}\ X.S')$.

$(n = k+1)$: Let us assume that $\texttt{rec}\ X.S' \Rightarrow^{k+1} S''$. Recall that, since the main lemma is being proved by structural induction, by inductive hypothesis we have that $succ(S') \subseteq gen(S')$. By Corollary 4.2.10, S'' is either equal to a serial process $T\sigma$ such that $S' \Rightarrow T$, or to $\texttt{rec}\ X.S'$. Let us analyze both cases.

In the first case, since T is a successor of S', we have that $T \in succ(S')$. By inductive hypothesis, we have that $succ(S') \in gen(S')$, and thus we can infer that T is contained in $gen(S')$. If we apply the substitution σ to both T and to the set $gen(S')$, we obtain that $T\sigma \in gen(S')\sigma$. Since $\{\texttt{rec}\ X.S'\} \cup gen(S')\sigma = gen(\texttt{rec}\ X.S')$, if $T\sigma$ is contained in $gen(S')\sigma$, then $T\sigma$ is contained in $gen(\texttt{rec}\ X.S')$ too. This last inclusion proves the lemma.

In the second case, where S''' is $\texttt{rec}\ X.S'$, the proof is the same as in the base case.

$(S = S' + S'')$: By induction on either branch of the choice.

□

Finiteness of CCS processes

Before proving Theorem 4.2.6, we need two auxiliary lemmas to infer the shape that processes under parallel and restriction constructs take, after taking transition steps.

Lemma 4.2.12. *Let $\nu a.P$ be a restricted CCS process. For any process Q such that $\nu a.P \Rightarrow Q$, there exists a process P' such that $P \Rightarrow P'$ and $Q = \nu a.P'$.*

Proof. By definition of the LTS, process $\nu a.P$ can make a transition to $\nu a.P'$ only if P transitions to P'. By repeatedly applying this observation, the lemma is proved. □

Lemma 4.2.13. *For any CCS processes P, Q and s, $P \,|\, Q \Rightarrow s$ implies that $s = P' \,|\, Q'$ for some process P' and Q' such that $P \Rightarrow P'$ and $Q \Rightarrow Q'$.*

Proof. By induction on the length of the derivation of $P \,|\, Q \Rightarrow s$. □

We prove next that *gen* is an upper bound to the successor states of any restricted CCS process P:

Lemma 4.2.14. *For any CCS process P, $succ(P) \subseteq gen(P)$.*

Proof. We prove this theorem by structural induction on P.

$(P = 0)$: $succ(0) = \{0\} = gen(0)$, which proves the case.

$(P = \alpha.P')$: The proof of this case is the same as for the corresponding case in Lemma 4.2.11. According to the LTS, the only transition that P can take is $\alpha.P' \xrightarrow{\alpha} P'$, and thus the only successors of P are P itself and the successors of P'. In other words, $succ(P) = \{P\} \cup succ(P')$. By definition of *gen*, we have $gen(P) = \{P\} \cup gen(P')$. Since, by inductive hypothesis, $succ(P') \subseteq gen(P')$, from the previous two deductions we can infer that $succ(S) \subseteq gen(S)$, which proves the lemma.

$(P = \texttt{rec}\ X.S)$: By Lemma 4.2.11.

$(P = \nu a.P)$: by Lemma 4.2.12 and structural induction.

$(P = P' + Q)$: by definition of $gen(P' + Q)$ and by structural induction.

$(P = P' \,|\, Q)$: by Lemma 4.2.13, structural induction and definition of $gen(P' \,|\, Q)$.

□

We can finally prove Theorem 4.2.6:

Theorem 4.2.6 *For any process P in finite CCS, $succ(P)$ is a finite set.*

Proof. From Lemma 4.2.14, we have that $succ(P) \subseteq gen(P)$. From Lemma 4.2.8, we have that $gen(P)$ is a finite set. Since $succ(P)$ is a subset of a finite set, then $succ(P)$ is a finite set too. □

4.2.2 Restrictions for finite-state TCCS$^{\mathrm{m}}$ processes

The same restrictions that makes the LTS derived from a CCS process finite, also makes a TCCS$^{\mathrm{m}}$ process finite. The proof strategy is the same as in the standard CCS case, by providing an upper bound for the set of states derivable from a TCCS$^{\mathrm{m}}$ process. However, the presence of transactions adds a complication to the proof strategy, because of transaction names. Since transactions can be renamed, activated, committed or aborted, we need to provide an upper bound to the highest transaction name according to the ordering $\leq_{\mathcal{T}}$. We show in this section that the total number of parallel processes that a process can spawn is an upper bound to the number of distinct transactions (and therefore distinct transaction names) that can be generated.

Syntactic restrictions on TCCS$^{\mathrm{m}}$

We present the grammar of restricted processes in TCCS$^{\mathrm{m}}$ that yield a finite state LTS. The restriction disallows the parallel and restriction constructs to occur within a recursive process. The grammar is as follows:

$$\begin{aligned} P, Q ::= &\ \sum \mu_i.P_i \mid X \mid \texttt{rec}\ X.S \mid \texttt{co}.P \mid [\![P \blacktriangleright Q]\!] \mid \nu a.P \mid P \,|\, Q \mid [\![P \triangleright_k Q]\!] \\ S, T ::= &\ \sum \mu_i.S_i \mid X \mid \texttt{rec}\ X.S \mid \texttt{co}.S \mid [\![S \blacktriangleright T]\!] \end{aligned}$$

Since restricted terms are a subset of TCCS$^{\mathrm{m}}$, the Nameless LTS from Fig. 4.2 is clearly applicable on them. For consistency with the previous section, we maintain label α to denote all kind of actions that the LTS from TCCS$^{\mathrm{m}}$ can perform, namely $a, k(a), \tau, \texttt{new}\ k, \texttt{ab}\ k$ and $\texttt{co}\ k$.

By Def. 2.1.1 of well-formedness, active transactions cannot occur under a recursive definition, a prefix or a transaction in a well-formed process. Therefore a serial process S cannot contain an active transaction, but only dormant ones.

Since active transactions can only occur inside the scope of a restriction or inside parallel processes, it is easy to show that the set of transaction names $ftn(P)$ is proportional to the number of top-level parallel processes in P. The number of parallel processes is defined as follows:

Definition 4.2.15 (Transaction name upper bound)**.** *Let P be well-formed. The maximum number*

of transaction names $tmax(P) :: Proc \to \mathcal{T}$ *is defined as follows:*

$$t(P) = \begin{cases} k_1 & \text{if } P \in \{0, X, \texttt{rec } X.S\} \\ max(t(P_i)) + k_1 & \text{if } P = \sum_{i \in I} \mu_i.P_i \text{ and } I \neq \emptyset \\ t(P') & \text{if } P = \texttt{co}.P' \\ max(t(P'), t(Q)) & \text{if } P = [\![P' \rhd_k Q]\!] \\ max(t(P'), t(Q)) & \text{if } P = [\![P' \blacktriangleright Q]\!] \\ t(P') & \text{if } P = \nu a.P' \\ t(P') + t(Q) & \text{if } P = P' \mid Q \end{cases}$$

Intuitively, there can be at most one active transaction per parallel process, because nested transactions are prohibited by well-formedness. Therefore the upper bound provides an estimate of the number of parallel processes that can be generated at any transition of an input process P. It is easy to see from the definition of the upper bound that if $t(P) = k_n$, then P contains at most $n - 1$ occurrences of the parallel construct "$\mid$". The rest of this section formalizes and proves this intuition.

We begin by proving that the upper bound is always greater than k_0, and that it is always an upper bound for the number of distinct active transactions in a process:

Lemma 4.2.16. *For any well-formed* P*:*

1. $k_1 \leq_{\mathcal{T}} t(P)$*;*
2. *if* $t(P) = k_n$ *then* $|ftn(P)| \leq n$*.*

Proof. Property 1 is proved straightforwardly by structural induction. Property 2 is proved by structural induction as well, using the fact that if $t(P) = k_n$, then $n \geq 1$ by Property 1. When P is either $0, X, \sum \mu_i.P_i, \texttt{rec } X.S, \texttt{co}.P'$ or $[\![P' \blacktriangleright Q]\!]$, then by well-formedness the set of free transaction names of P is the empty set, and therefore its size is 0; more formally, $|ftn(P)| = |\emptyset| = 0$. If $t(P) = k_n$, then n is greater than 0 by Property 1, and therefore $|ftn(P)| \leq n$ holds.

When $P = [\![P' \rhd_k Q']\!]$, then $|ftn(P)| = |\{k\}| = 1$ holds. Since n is greater or equal to 1 by Property 1, $|ftn(P)| \leq n$ holds and the lemma is proved. When $P = P' \mid Q$, $|ftn(P' \cup Q)| \leq |ftn(P')| + |ftn(Q)|$ holds by definition of set union. By inductive hypothesis if $t(P') = k_i$ then $|ftn(P')| \leq i$, and if $t(Q) = k_j$ then $|ftn(Q)| \leq j$. Therefore $|ftn(P' \cup Q)| \leq |ftn(P')| + |ftn(Q)| \leq i + j$ holds, and the lemma holds by transitivity. When $P = \nu a.P'$, the lemma is proved by the inductive hypothesis.

□

The following lemma shows that the upper bound never increases with transitions in the nameless LTS:

Lemma 4.2.17. *For any well-formed process* P*, if* $P \xrightarrow{\xi}{}^{\texttt{nl}}_{\sigma} Q$ *then* $t(P) \geq_{\mathcal{T}} t(Q)$*.*

Proof. By rule induction on $P \xrightarrow{\xi}{}^{\texttt{nl}}_{\sigma} Q$.

- Suppose that Rule NL-Act is used, then one of transitions rules from Fig. 2.2, namely one of the CCS rules (Rule CCSsum, CCSsync or CCSrec), transactional Rule TrTau, or one of the propagation rules (Rule Restr or ParL). It is easy to show by rule induction that in all of these

cases no transaction is renamed, and therefore $t(P) \leq_{\mathcal{T}} t(Q)$. For example, by Rule CCSsum the transition $\Sigma\mu_i.P_i \xrightarrow{\mu_i}_{\varepsilon} P_i$ holds. By definition of $t(\Sigma\mu_i.P_i) = max(t(P_i)) + k_1$, and therefore $t(P_j) \leq_{\mathcal{T}} t(P)$ holds by definition of maximum.

By Rule CCSsync, the following holds:

$$\frac{P \xrightarrow{a}_{\varepsilon} P' \qquad Q \xrightarrow{\overline{a}}_{\varepsilon} Q'}{P \mid Q \xrightarrow{\tau}_{\varepsilon} P' \mid Q'}$$

By definition $t(P \mid Q) = k_{i+j}$ if $t(P) = k_i$ and $t(Q) = k_j$; similarly $t(P' \mid Q') = k_{i'+j'}$ if $t(P') = k_{i'}$ and $t(Q') = k_{j'}$. Since $t(P) \leq_{\mathcal{T}} t(P')$ and $t(Q) \leq_{\mathcal{T}} t(Q')$ hold by inductive hypothesis, then $k_{i'+j'} \leq_{\mathcal{T}} k_{i+j}$ holds by transitivity.

When Rule CCSrec is used, the case follows immediately by definition of $t(-)$. If Rule TrTau is used, then $[\![P \triangleright_k Q]\!] \xrightarrow{\tau}_{\varepsilon} [\![P' \triangleright_k Q]\!]$ holds only if $P \xrightarrow{\tau}_{\varepsilon} P'$ holds. The alternative process Q in transaction k_i is unchanged, therefore the lemma follows by inductive hypothesis $t(P') \leq_{\mathcal{T}} t(P)$. When Rule Restr or ParL is used, the lemma follows by inductive hypothesis as well.

- Suppose that Rule NL-FreshAct is used, then either one of the transactional rules from Fig. 2.2 has been used (Rule TrSum, TrAct, or TrSync) or one of the propagation rules (Rule Restr or ParL).

  Suppose that Rule TrSum is used. Then $\Sigma\mu_i.P_i \xrightarrow{k_n(a)}_{\varepsilon \mapsto k_n} [\![P_j \mid \mathtt{co} \triangleright_{k_n} \Sigma\mu_i.P_i]\!]$ holds if $\mu_j = a$ for some j. By definition of $t(-)$, moreover $t([\![P_j \mid \mathtt{co} \triangleright_{k_n} \Sigma\mu_i.P_i]\!]) = max(t(P_j \mid \mathtt{co}), T(\Sigma\mu_i.P_i)) = max(k_1 + t(P_j), T(\Sigma\mu_i.P_i))$ holds. Since $t(\Sigma\mu_i.P_i) = k_1 + max(t(P_i))$, then $k_1 + t(P_j) \leq_{\mathcal{T}} k_1 + max(t(P_i))$ holds because the maximum on the right-hand side contains $t(P_j)$, and therefore the inequality $max(k_1 + t(P_j), T(\Sigma\mu_i.P_i)) \leq_{\mathcal{T}} T(\Sigma\mu_i.P_i)$ holds.

  When Rule TrAct is used, the lemma follows by inductive hypothesis, as for the case of Rule TrTau. When Rule TrSync is used, the proof is similar to that of Rule CCSsync. The remaining cases are proved by the inductive hypothesis.

- Suppose that Rule NL-New is used. The lemma is proved by rule induction on either Rule TrNew, TrIgn or TrRestr. In the first case $[\![P \blacktriangleright Q]\!] \xrightarrow{\mathtt{new}\, k} [\![P \triangleright_k Q]\!]$ holds, and the lemma holds because the upper bound for the dormant and active transaction is the same, by definition of $t(-)$. The remaining cases are proved by the inductive hypothesis.

- Suppose that Rule NL-Co or NL-Ab are used. The lemma is proved by rule induction on either Rule TrCo, TrAb, TrBroadcast, TrIgn or TrRestr. In the first case the lemma can be easily proved by rule induction on $\rightsquigarrow_{\mathtt{co}}$, since $t(\mathtt{co}.P) = t(P)$ for any P. The second case is straightforward, since $t(Q) \leq_{\mathcal{T}} max(t(P'), t(Q))$ holds for any P' and Q. The remaining cases are proved by inductive hypothesis.

□

We now show that nameless transitions preserve the limit provided by the upper bound $t(-)$:

Lemma 4.2.18. *Let P be a well-formed process such that $\forall l \in ftn(P).\ l \leq t(P)$. If $P \xrightarrow{\xi}{}^{\mathtt{nl}}_{\sigma} Q$ holds, then $\forall l \in ftn(Q).\ l \leq t(Q)$.*

Proof. By rule induction.

- When Rule NL-Act is used, no transaction is renamed or created, therefore the lemma follows straightforwardly by inductive hypothesis.

- Suppose that Rule NL-FreshAct is used. In order to prove this case, we first prove by rule induction a stronger property on transitions $\xrightarrow{k(a)}_{[\tilde{l}\mapsto k]}$ that introduce fresh transaction names:

$$\mathcal{P}(P \xrightarrow{k(a)}_{[\tilde{l}\mapsto k]} Q) = \text{for any set } K \subseteq \mathcal{T},\ k = min(\mathcal{T}\backslash(ftn(P) \cup K)) \\ \text{implies } \forall l \in ftn(Q).\ l \leq_{\mathcal{T}} t(Q) + k_m \text{ where } m = |K|$$

  If this stronger property holds, then the lemma follows by taking $K = \emptyset$.

  Suppose that Rule TrAct is used. Then $\Sigma\mu_i.P_i \xrightarrow{k_n(a)}_{\varepsilon\mapsto k_n} [\![P_j \mid \mathtt{co} \triangleright_{k_n} \Sigma\mu_i.P_i]\!]$ holds if $\mu_j = a$ for some j. Suppose also that $k_n = min(\mathcal{T}\backslash(ftn(\Sigma\mu_i.P_i) \cup K))$ for some K. Since $\Sigma\mu_i.P_i$ is well-formed by hypothesis, it contains no active transactions; therefore $ftn(\Sigma\mu_i.P_i) = \emptyset$ and $k_n = min(\mathcal{T}\backslash K)$.

  We need to show that $\forall l \in ftn(Q).\ l \leq_{\mathcal{T}} t(Q) + k_{|K|}$, which is equivalent to showing that $k \leq_{\mathcal{T}} t(Q) + k_{|K|}$, since the only free transaction name in Q is k itself. There are two cases to consider: either K is a sequence of contiguous names $k_0, k_1, \ldots k_n$, or it is not. If it is such a sequence, then $k = k_{n+1}$ because $\mathcal{T}\backslash K = \{k_{n+1}, k_{n+2} \ldots\}$, and $|K| = n + 1$. Because of this the inequation $k = k_{n+1} \leq_{\mathcal{T}} k_{n+1} = k_{|K|}$ holds, which implies $k_{n+1} \leq t(Q) + k_{|K|}$ holds by transitivity, and the lemma is proved. If K is not a sequence, then there exists an element in $\mathcal{T}$ that is not in K and that is lower than $k_{|K|}$. Since $k \leq k_{|K|}$ holds, then $k \leq t(Q) + max(K)$ also holds and the lemma is proved.

  Suppose now that Rule TrAct is used. Then $[\![P \triangleright_l Q]\!] \xrightarrow{k(a)}_{[l\mapsto k]} [\![P' \triangleright_k Q]\!]$ holds assuming that $P \xrightarrow{a}_{\varepsilon} P'$ and $k \sharp l$ hold. Let $k = min(\mathcal{T}\backslash(ftn([\![P \triangleright_l Q]\!]) \cup K))$ for some K. By definition of well-formedness P and Q have no nested transactions, therefore $ftn([\![P \triangleright_l Q]\!]) = \{l\}$ and $k = min(\mathcal{T}\backslash\{l\} \cup K)$ hold.

  We need to show that $\forall l \in ftn([\![P \triangleright_l Q]\!]).\ l \leq_{\mathcal{T}} t([\![P \triangleright_l Q]\!]) + k_{|K|}$, which is equivalent to showing that $k \leq_{\mathcal{T}} t([\![P \triangleright_l Q]\!]) + k_{|K|}$, since the only free transaction name in $[\![P \triangleright_l Q]\!]$ is k itself. There are two cases to consider: either $K \cup \{l\}$ is a sequence of contiguous names $k_0, k_1, \ldots k_n$, or it is not. If it is such a sequence, then $|K \cup \{l\}| \leq n + 1$, $|K| = n$, and $k = k_{n+1}$ hold, because $\mathcal{T}\backslash K \cup \{l\} = \{k_{n+1}, k_{n+2} \ldots\}$. Because of this, the inequality $k \leq_{\mathcal{T}} t([\![P \triangleright_l Q]\!]) + k_{|K|}$ can be rewritten as $k_{n+1} \leq t([\![P \triangleright_l Q]\!]) + k_n$; since $k_1 \leq_{\mathcal{T}} t([\![P \triangleright_l Q]\!])$ holds by Property 1 of Lem. 4.2.16, then $k_{n+1} \leq_{\mathcal{T}} k_1 + k_n \leq_{\mathcal{T}} t([\![P \triangleright_l Q]\!]) + k_n$ holds, and the lemma is proved by transitivity. If $K \cup \{l\}$ is not a sequence, then the lemma is proved as in the previous case.

  Suppose that Rule TrSync is used. Then the following holds:

$$\frac{P \xrightarrow{k(a)}_{\sigma_1} P' \qquad Q \xrightarrow{k(\overline{a})}_{\sigma_2} Q'}{P \mid Q \xrightarrow{k(\tau)}_{(\widetilde{l}_1,\widetilde{l}_2)\mapsto k} P'\sigma_2 \mid Q'\sigma_1} \quad \begin{array}{l}\sigma_1 = \widetilde{l}_1 \mapsto k \\ \sigma_2 = \widetilde{l}_2 \mapsto k\end{array}$$

where $k = min(\mathcal{T} \backslash (ftn(P \mid Q) \cup K))$ for any K. In order to apply the inductive hypothesis on the transitions $P \xrightarrow{k(a)}_{\sigma_1} P'$ and $Q \xrightarrow{k(\overline{a})}_{\sigma_2} Q'$, we need to show that $k = min$ By definition of free transaction names, $ftn(P \mid Q) = ftn(P) \cup ftn(Q)$, therefore $k = min(\mathcal{T} \backslash (ftn(P \mid Q) \cup K))$ can be rewritten as both $k = min(\mathcal{T} \backslash (ftn(P) \cup (ftn(Q) \cup K)))$ and $k = min(\mathcal{T} \backslash (ftn(Q) \cup (ftn(P) \cup K)))$ for any K. These equalities allow us to use the inductive hypothesis on the two aforementioned premises.

Let $ftn(P) = i$ and $ftn(Q) = j$. By inductive hypothesis $\forall l \in ftn(P).l \leq_{\mathcal{T}} t(P) + k_{|K \cup ftn(Q)|} = t(P) + k_{|K|} + k_j$ holds. By Property 2 of Lem. 4.2.16 $k_j \leq_{\mathcal{T}} t(Q)$ holds, and therefore $\forall l \in ftn(P).l \leq_{\mathcal{T}} t(P) + k_{|K|} + t(Q)$ holds too. Since $t(P \mid Q) = t(P) + t(Q)$ holds by definition of $t(-)$, the previous inequation can be rewritten as $\forall l \in ftn(P).l \leq_{\mathcal{T}} t(P \mid Q) + k_{|K|}$. By a similar reasoning, it can also be shown that $\forall l \in ftn(Q).l \leq_{\mathcal{T}} t(P \mid Q) + k_{|K|}$; the lemma is proved by combining the previous two statements.

When Rule Restr or ParL are used, the lemma follows by inductive hypothesis.

- When Rule NL-New is used, the lemma is proved by rule induction as in the case of Rule NL-FreshAct, using the same stronger inductive hypothesis.
- When Rule NL-Ab or NL-Co is used, a transaction k is removed from the initial process Q, and the lemma is proved straightforwardly by rule induction.

□

We conclude this section by showing that the function $t(-)$ provides an upper bound to the set of names generated by a restricted TCCS$^{\mathrm{m}}$ process:

Theorem 4.2.19. *Let P be a well-formed TCCS^m process such that $\forall k \in ftn(P).\ k \leq_{\mathcal{T}} t(P)$. If $P \xrightarrow{\xi_1}{}^{\mathtt{nl}}_{\sigma_1} \xrightarrow{\xi_2}{}^{\mathtt{nl}}_{\sigma_2} \ldots \xrightarrow{\xi_n}{}^{\mathtt{nl}}_{\sigma_n} Q$, then $\forall l \in ftn(Q).\ l \leq_{\mathcal{T}} t(P)$.*

Proof. By repeated application of Lem. 4.2.18 $\forall l \in ftn(Q).\ l \leq_{\mathcal{T}} t(Q)$. Since $t(Q) \leq_{\mathcal{T}} t(P)$ holds by repeated application of Lem. 4.2.17, the theorem holds by transitivity of $\leq_{\mathcal{T}}$. □

The theorem assumes that the active transactions in a process P are all less than $t(P)$. It is easy to show that, if this is not the case, there exists a renaming π such that this condition is satisfied in $\pi(P)$. The renamed process $\pi(P)$ is bisimilar to P by Thm. 3.4.12.

State generation

The state generating function *gen* is slightly more complicated in the transactional case, because of transaction names.

Definition 4.2.20. ***(State generating function)***

The state generator function $gen :: Proc \to \mathcal{P}(Proc)$ *is defined as follows:*

$$gen(P,n) = P \cup \begin{cases} \emptyset & \text{if } P = 0 \\ \{X\} & \text{if } P = X \\ gen(P',n) & \text{if } P = \alpha.P' \\ gen(S,n)[X \mapsto \texttt{rec } X.S] & \text{if } P = \texttt{rec } X.S \\ gen(P',n) \cup gen(Q,n) & \text{if } P = P' + Q \\ \{\, \nu a.s \mid s \in gen(P',n) \,\} & \text{if } P = \nu a.P' \\ \{\, s \mid s' \mid s \in gen(P',n), s' \in gen(Q,n) \,\} & \text{if } P = P' \mid Q \\ gen(P',n) & \text{if } P = \texttt{co}.P' \\ \{\, [\![P'' \rhd_k Q]\!] \mid P'' \in gen(P',n),\ k \leq n \,\} \cup gen(P',n) \cup gen(Q,n) & \text{if } P = [\![P' \blacktriangleright Q]\!] \\ \{\, [\![P'' \rhd_l Q]\!] \mid P'' \in gen(P',n),\ l \leq n \,\} \cup gen(P',n) \cup gen(Q,n) & \text{if } P = [\![P' \rhd_k Q]\!] \end{cases}$$

where $Q\backslash_{\texttt{co}}$ is Q' when $Q \rightsquigarrow_{\texttt{co}} Q'$.

As in the previous section, *gen* can be easily proved to generate only finite sets because it is defined inductively.

Lemma 4.2.21. *For any TCCS*$^{\mathcal{M}}$ *process* P *and transaction name* $n \in \mathcal{T}$*,* $gen(P,n)$ *is a finite set.*

Proof. By induction on the structure of P. □

Upper bound

We now prove that $gen(P, t(P))$ is an upper bound for $succ(P)$. As in the previous section, a few lemmas are needed to infer the states that parallel, recursive and transactional processes evolve to. We will have to extend Lemma 4.2.9, 4.2.11, 4.2.13 and Corollary 4.2.10:

Lemma 4.2.22. *Let* T, T' *be serial TCCS*$^{\mathcal{M}}$ *processes, and* $\sigma = [X \mapsto T]$ *and* $\sigma' = [Y \mapsto T']$ *be two substitutions. For any serial process* S*, if* $X \neq Y$*,* T *is closed and* X *is not free in* T'*, then* $S\sigma\sigma' = S\sigma'[X \mapsto T\sigma']$*.*

Proof. Similar to the proof of Lem. 4.2.4. □

Lemma 4.2.23. *Consider a TCCS*$^{\mathcal{M}}$ *process* S *and a substitution* $[X \mapsto S_0]$ *such that* $S \neq X$ *for any variable* X*. If* $S[X \mapsto S_0] \rightsquigarrow_{\texttt{co}} S'$*, then* $S \rightsquigarrow_{\texttt{co}} S''$ *and* $S' = S''[X \mapsto S_0]$*.*

Proof. By rule induction. □

Lemma 4.2.24. *Let* S *and* S_0 *be well-formed closed TCCS*$^{\mathcal{M}}$ *processes, and* $\sigma = [X \mapsto S_0]$*. If* $S \neq X$*, then* $S\sigma \xrightarrow{\xi}{}^{\texttt{nl}}_{\sigma'} Q$ *implies that there exists some* T *such that* $S \xrightarrow{\xi}{}^{\texttt{nl}}_{\sigma''} T$ *and either* $Q = T\sigma$ *or* $Q = [\![T \rhd_k T']\!]\,\sigma$ *for some* T' *and* k*.*

Proof. By structural induction as for Lem. 4.2.9 except when $S = [\![T_1 \blacktriangleright T_2]\!]$, in which case the lemma follows by definition of substitution. □

Corollary 4.2.25. *For any serial TCCS^{m} process S and variable X, $\texttt{rec}\ X.S \Rightarrow s$ implies that either $\exists T.S \Rightarrow T$ and $s = T[X \mapsto \texttt{rec}\ X.S]$ or $s = \texttt{rec}\ X.S$, or $S \Rightarrow [\![S_1 \triangleright_k S_2]\!]$ and $s = [\![S_1 \triangleright_k S_2]\!][X \mapsto \texttt{rec}\ X.S]$ for some k, S_1, S_2.*

Proof. The proof is the same as for Corollary 4.2.10. □

Before proving a result similar to Lemma 4.2.27, we need to explicate what shape a transactional process can take throughout its transitions.

Lemma 4.2.26. *For any serial processes P, Q and transaction name k, $[\![P \triangleright_k Q]\!] \Rightarrow s$ implies one of the following:*

1. *there exists P' and l such that $P \Rightarrow P'$ and $s = [\![P' \triangleright_l Q]\!]$, or*
2. *$P \Rightarrow \rightsquigarrow_{\texttt{co}} \Rightarrow P'$ and $s = P'$, or*
3. *$Q \Rightarrow Q'$ and $s = Q'$*

Proof. By induction on the length of the derivation $[\![P \triangleright_k Q]\!] \Rightarrow s$. In the base case $s = [\![P \triangleright_k Q]\!]$ by reflexivity, and the lemma is proved because s falls in the Case 1.

In the inductive case, suppose that $P \Rightarrow^n s' \rightarrow s$, and that s' falls in either Case 1, 2 or 3. In Case 1, if there exists P' and l such that $P \Rightarrow P'$ and $s' = [\![P' \triangleright_l Q]\!]$, then we need to take three cases on the transition $[\![P' \triangleright_l Q]\!] \rightarrow s$: either l is committed by Rule NL-Co, l is aborted by Rule NL-Ab, or P' performs a tentative action by Rule NL-Act or Rule NL-FreshAct. It is easy to verify that in the first case $P \Rightarrow P''$ and $s' = [\![P'' \triangleright_{l'} Q]\!]$ holds; in the second case $P' \rightsquigarrow_{\texttt{co}} P''$ holds, and therefore $P \Rightarrow \rightsquigarrow_{\texttt{co}} \Rightarrow P''$ holds; and in the third case $Q \Rightarrow Q'$ holds.

The lemma is straightforward for Case 2, because if there exists P' such that $P \Rightarrow \rightsquigarrow_{\texttt{co}} \Rightarrow s'$, $s' = P'$ and $P' \rightarrow s$, then $P \Rightarrow \rightsquigarrow_{\texttt{co}} \Rightarrow s$ follows by definition of $\Rightarrow$. The proof for Case 3 is similar to Case 2. □

We can now prove that $get(P, t(P))$ is an upper bound for $succ(P)$, provided that transaction names in P are lower than $t(P)$:

Lemma 4.2.27. *For any serial TCCS^{m} P such that $\forall l \in ftn(P).l \leq_{\mathcal{T}} t(P)$, $succ(P) \subseteq gen(P, t(P))$.*

Proof. The proof is by structural induction. The proofs for $0, \alpha.S$ are the same as for Lemma 4.2.11, since the definition of *gen* for TCCS$^{\mathrm{m}}$ does not change from the one for CCS. We have to develop a proof for the transactional constructs:

$(P = \texttt{rec}\ X.S)$: by Cor. 4.2.25.

$(P = \texttt{co}.T)$: this case is trivial, because $\texttt{co}.T$ cannot take transitions.

$P = [\![P \triangleright_k Q]\!]$ By definition of *succ*, for any $s \in succ([\![P \triangleright_k Q]\!])$, there exists a derivation $[\![P \triangleright_k Q]\!] \Rightarrow s$. By Lem. 4.2.26 is either $[\![P' \triangleright_l Q']\!]$ for some l, P', Q', or some P' such that $P \Rightarrow P'$, or some Q' such that $Q \Rightarrow Q'$. In the first case, by Thm. 4.2.19 $l \leq t([\![P \triangleright_k Q]\!])$ holds, and $[\![P' \triangleright_l Q']\!] \in gen([\![P \triangleright_k Q]\!], t([\![P' \triangleright_l Q]\!]))$ holds by inductive hypothesis. In the other two cases the lemma follows straightforwardly by inductive hypothesis.

$(P = [\![Q \blacktriangleright R]\!])$: This case is proved similarly to the case $P = [\![P \triangleright_k Q]\!]$.

□

Finally, we prove that processes from serial TCCS$^{\mathrm{m}}$ have a finite state LTS:

Theorem 4.2.28 (Finiteness of TCCS$^{\mathrm{m}}$)**.** *For any serial process P in TCCS^m , $succ(P)$ is a finite set.*

Proof. By Lem. 4.2.27, $succ(P) \subseteq gen(P, t(P))$ holds, where $gen(P, t(P))$ is a finite set by Lem. 4.2.21. Since $succ(P)$ is a subset of a finite set, then $succ(P)$ is a finite set too. □

4.3 Bisimulation algorithm

This section presents an algorithm to calculate bisimulation equivalence between TCCS$^{\mathrm{m}}$ agents. Our algorithm is an adaptation of the *on-the-fly* algorithm from Sokolsky and Cleaveland [Bergstra, 2001, Chp. 6, pg. 416], which operates on CCS agents. Given two processes P and Q, and their relative LTSs, a bisimulation algorithm calculates a bisimulation equivalence between P and Q, if one such relation exists. Many *global* bisimulation algorithms operate in two phases: in the first phase they pre-compute and store the entire LTSs from P and Q; in the second phase they calculate the bisimulation by manipulating this information.

For example, the second phase of the Paige-Tarjan algorithm [Paige and Tarjan, 1987] iteratively divides the entire state space into *blocks*. The algorithm starts with a single block B containing all the states derivable from P and Q. A state P_0, called *splitter*, is iteratively picked from a block B_i, and all the remaining states P_i are compared with the splitter. If there exists a transition that the splitter can perform but that P_i cannot, the block B_i is split into two blocks B_i and B_{i+1}: the former contains all the processes that can perform the same actions as the splitter, and the latter contains all the other processes. Blocks are split as much as possible; the algorithm terminates when no new blocks can be created. The final blocks constitute equivalence classes of bisimilar states. Processes P and Q are bisimilar if they belong to the same block.

As pointed out in [Bergstra, 2001], global bisimulation algorithms tend to perform poorly in practice, because the state space generated by P and Q can be large and therefore costly to store and manipulate. This situation is aggravated in the transactional case, because a historyless bisimulation is a ternary relation $\mathcal{P} \times \mathcal{D} \times \mathcal{P}$, where $\mathcal{D}$ is the set of dependency sets. If we were to adapt the Paige-Tarjan algorithm to TCCS$^{\mathrm{m}}$, then blocks would have to be parametric to some dependency set Δ. Suppose that the P can activate at most n different transactions, and that Q can activate at most m (i.e. $t(P) = k_n$ and $t(Q) = k_m$). The total number of possible dependency set is the power set of $\mathcal{D}$, which in this case amounts to $|\mathcal{P}(\mathcal{D})| = 2^{(n+2)(m+2)}$, where the constant 2 added to both n and m accounts for the labels `ab` and `co`. The adapted Paige-Tarjan algorithm would have to generate and store the entire state space of P and Q multiplied by a factor of $2^{(n+2)(m+2)}$, which can be a severe penalty.

In contrast to global algorithms, local or *on-the-fly* algorithms only comprise a single phase that combines state generation and bisimulation checking. The state space is generated and explored

dynamically starting from the input processes, and it terminates either after building a bisimulation equivalence, or after finding a counter-example. This approach has the advantage of generating only the portion of the input processes' LTSs that is necessary to build the equivalence. Moreover, if two processes are not bisimilar, chances are that local algorithms will terminate much more quickly. Global algorithms can find a counter-example only after the first phase is completed, whereas local algorithms can terminate after generating a smaller portion of the state space.

These advantages benefit the transactional setting too, in light of the previous discussion of the Paige-Tarjan algorithm. It is also not immediate how to adapt the Paige-Tarjan algorithm to transactions: since historyless bisimulations are ternary relations between two processes and a Δ dependency set, it is not possible to merge indistinctly all the states of P and Q into the one block, but care has to be taken to link transactions that belong to P and Q in Δ. The local algorithm described in [Bergstra, 2001] adapts more easily to the transactional setting.

The local bisimulation algorithm from Sokolsky and Cleveland builds a *bisimulation graph* $G = \langle V, E\rangle$ that represents a bisimulation relation. The vertices V are triples of the form (P, Q, Δ). The edges E, which we call *justification edges*, are triples of the form $((P, Q, \Delta), (\xi_i, \sigma), (P', Q', \Delta'))$ with $i \in \{1, 2\}$. A bisimulation graph G has the following properties:

- if $(P, Q, \Delta) \in V$, then Δ is consistent;
- whenever $(P, Q, \Delta) \in V$ and $P \xrightarrow{k(a)}{}^{\mathtt{nl}}_{\sigma_1} P'$, then $Q \stackrel{d(a)}{\Longrightarrow}{}^{\mathtt{nl}}_{\sigma_2} Q'$, $(P', Q', \sigma_1\Delta\sigma_2 \cup \{k, d\}) \in V$ and $((P, Q, \Delta), (k(a)_1, \sigma_1), (P', Q', \sigma_1\Delta\sigma_2 \cup \{k, d\})) \in E$;
- whenever $(P, Q, \Delta) \in V$ and $P \xrightarrow{\tau}{}^{\mathtt{nl}}_{\sigma_1} P'$, then $Q \stackrel{\tau}{\Rightarrow}{}^{\mathtt{nl}}_{\sigma_2} Q'$, $(P', Q', \sigma_1\Delta\sigma_2) \in V$ and $((P, Q, \Delta), (\tau_1, \sigma_1), (P', Q', \sigma_1\Delta\sigma_2 \cup \{k, d\})) \in E$;
- whenever $(P, Q, \Delta) \in V$ and $Q \xrightarrow{k(a)}{}^{\mathtt{nl}}_{\sigma_2} Q'$, then $P \stackrel{d(a)}{\Longrightarrow}{}^{\mathtt{nl}}_{\sigma_1} P'$ and $(P', Q', \sigma_1\Delta\sigma_2 \cup \{k, d\}) \in V$ and $((P, Q, \Delta), (k(a)_2, \sigma_2), (P', Q', \sigma_1\Delta\sigma_2 \cup \{k, d\})) \in E$;
- whenever $(P, Q, \Delta) \in V$ and $Q \xrightarrow{\tau}{}^{\mathtt{nl}}_{\sigma_2} Q'$, then $Q \stackrel{\tau}{\Rightarrow}{}^{\mathtt{nl}}_{\sigma_1} Q'$ and $(P', Q', \sigma_1\Delta\sigma_2) \in V$ and $((P, Q, \Delta), (\tau_2, \sigma_2), (P', Q', \sigma_1\Delta\sigma_2 \cup \{k, d\})) \in E$;

Two TCCS$^{\mathrm{m}}$ processes P and Q are bisimilar if there exists a bisimulation graph G containing the triple $(P, Q, \emptyset)$, where $\emptyset$ is the empty dependency set. If G exists, then its vertices V represent a nameless bisimulation between P and Q. The edges of the graph are triples of the form $((P, Q, \Delta), (\xi_i, \sigma), (P', Q', \Delta'))$ with $i \in \{1, 2\}$. The index i records which of the two players moved first in the bisimulation game: if i is 1, then P challenged Q with a transition $P \xrightarrow{\xi}{}^{\mathtt{nl}}_{\sigma} P'$, and Q responded with some weak transition to Q'; if i is 2, the converse holds.

The core bisimulation algorithm BISIM is presented in Fig. 4.3. The auxiliary search functions SEARCH_HIGH and SEARCH_LOW are shown in Fig. 4.4. The algorithm manipulates four main data structures: two *global* structures (shared across different invocations of BISIM , SEARCH_HIGH and SEARCH_LOW), and two *local* structures (visible only within the scope of a particular BISIM invocation).

The input of the algorithm is a triple (P, Q, Δ), an empty bisimulation graph G and an empty set R of rejected triples. If (P, Q, Δ) already belongs in R or Δ is not consistent, then the triple is added

```
1  R := ∅;                    -- the set of rejected triples
2  G := < ∅, ∅ >;             -- the empty bisimulation graph
3
4  related     = True
5  not_related = False
6
7  BISIM(p, q, Δ) =
8    if ⟨p, q, Δ⟩ ∈ R
9      then not_related
10     else if ⟨p, q, Δ⟩ ∈ vertices G
11       then return related
12       else
13         if not( isConsistent Δ)
14           then addNode R ⟨p, q, Δ⟩ -- reject the input triple
15                return not_related
16           else do
17         let G' = addNode G ⟨p, q, Δ⟩ -- assume that the triple is bisimilar
18         -- let P attack in the bisimulation game
19         for each (p', ξ, σ) ∈ {(p', ξ, σ1)|p -ξ->σ1 p'} while status = not_related:
20           status := not_related
21           if high_{p',ξ,σ,q} does not exist
22             then create high_{p',ξ,σ,q} -- creates a global variable
23                  high_{p',ξ,σ,q} := 1
24           status := SEARCH_HIGH(p -ξ->σ p', q, Δ)
25           if status = not_related
26             then A := A ∪{r -ξ->σi p, s, i, Δ'|((p, q, Δ), ξi, σi, (r, s, Δ')) ∈ edges(G)} }
   -- the input triple must be removed
27         -- let Q attack in the bisimulation game
28         for each (q', ξ, σ) ∈ {(q', ξ, σ2)|q -ξ->σ2 q'} while status = not_related:
29           status := not_related
30           if low_{q',ξ,σ,p} does not exist
31             then create low_{q',ξ,σ,p}
32                  low_{q',ξ,p} := 1
33           status := SEARCH_LOW(q -ξ-> q', p)
34           if status = not_related
35             then A := A ∪{s -ξ->σi q, r, i, Δ'|((p, q, Δ), ξi, σi(r, s, Δ')) ∈ edges(G)} }
36         if status = not_related
37           -- remove the input node and sanitize the bisimulation graph
38           G := removeNode G ⟨p, q, Δ⟩
39           R := R ∪ ⟨p, q, Δ⟩
40           while A ≠ ∅
41             choose (r -ξ-> r', s, type, Δ') from A
42             A := remove(A, (r -ξ->σ r', s, type, Δ'))
43             if type = 1
44               then high_{r',μ,s} := high_{r',μ,s} + 1
45                    status := SEARCH_HIGH(r -ξ->σ r', s)
46                    if status = not_related
47                      then A := addEdges(A, ⟨r, s, Δ'⟩, type)
48                           G := removeNode(G, ⟨r, s, Δ'⟩, type)
49                           R := addNode(R, ⟨r, s, Δ'⟩, type)
50               else low_{r',μ,s} := low_{r',μ,s} + 1
51                    status := SEARCH_LOW(r -ξ->σ r', s)
52                    if status = not_related
53                      then A := addEdges(A, ⟨r, s, Δ'⟩, type)
54                           G := removeNode(G, ⟨r, s, Δ'⟩, type)
55                           R := addNode(R, ⟨r, s, Δ'⟩, type)
56           return status
```

Fig. 4.3: Bisimulation algorithm.

```
1  SEARCH_HIGH( p -ξ->σ1 p', q, Δ)=
2    status := not_related
3    qs := {q -ξ'-> • | (ξ = τ ⇒ ξ' = τ) ∧ (ξ = k(a) ⇒ ξ' = d(a))}
4    while status ≠ related and high_{p',k(μ),q} ≤ |qs| (q -x->σ2 q') := qs[high_{p',k(μ),q}]
5      Δ' := σ1Δσ2
6      if ξ == k(a) ∧ ξ' == d(a)
7        then Δ' := Δ' ∪ {(k, d)}
8      if Δ' is consistent
9        then status := BISIM(p', q', Δ')
10     if status = related
11       then newEdge := < ⟨p, q, Δ⟩, -k(μ)->H_σ1 ⟨p', q', Δ'⟩ >
12            G := addEdge(G, newEdge)
13       else high_{p',ξ,q} := high_{p',k(μ),q} + 1
14   return status
15
16 SEARCH_LOW( q -ξ->σ2 q', p, Δ)=
17   status := not_related
18   ps := {p -ξ'-> • | (ξ = τ ⇒ ξ' = τ) ∧ (ξ = k(a) ⇒ ξ' = d(a))}
19   while status ≠ related and low_{q',l(μ),p} ≤ |ps|
20     (p -ξ'->σ1 p') := ps[low_{q',ξ,p}]
21     Δ' := σ1Δσ2
22     if ξ == k(a) ∧ ξ' == d(a)
23       then Δ' := Δ' ∪ {(k, d)}
24     if Δ' is consistent
25       status := BISIM(p', q', Δ')
26     if status = related
27       then newEdge := < ⟨p, q, Δ⟩ -l(μ)->L_σ2 ⟨p', q', Δ'⟩ >
28            G := addEdge(G, newEdge)
29       else low_{p',l(μ),q} := low_{p',l(μ),q} + 1
30   return status
```

Fig. 4.4: Search routines for the bisimulation players.

to R and the algorithm terminates with a negative answer. Otherwise, the algorithm greedily assumes that P and Q are related under Δ, adds the triple to the (initially empty) graph G and proceeds to verify that the two processes are indeed related.

For each transition $P \xrightarrow{\xi}{}^{\mathtt{nl}}_{\sigma_1} P'$, the algorithm enumerates all matching weak transitions $Q \stackrel{\xi'}{\Rightarrow}{}^{\mathtt{nl}}_{\sigma_2} Q'$ (ξ matches ξ' if they are both τ, or one is $d(a)$ and the other is $d'(a)$ for some d and d'). One of the weak transitions is picked and a new triple $(P', Q', \sigma_1\Delta\sigma_2)$ is created. The algorithm recursively checks if a bisimulation graph can be created from $(P', Q', \sigma_1\Delta\sigma_2)$, and greedily adds all such vertices as much as possible. If the new triple is indeed bisimilar, a new justification edge $((P, Q, \Delta), (\xi_1, \sigma_1), (P', Q', \sigma_1\Delta\sigma_2))$ is added to G, and a new transition from P is analysed.

If all transitions from P can be matched by Q and vice versa, the algorithm terminates with a positive answer and a bisimulation graph G. If a transition $P \xrightarrow{\xi}{}^{\mathtt{nl}}_{\sigma_1} P'$ cannot be matched by any weak transition $Q \stackrel{\xi'}{\Rightarrow}{}^{\mathtt{nl}}_{\sigma_2} Q'$ (for example because all the resulting triples $(P', Q', \sigma_1\Delta\sigma_2)$ sets are inconsistent), then P and Q are not related under Δ. The triple (P, Q, Δ) is removed from G and added to R, and all the justification edges connected to it, i.e. those of the form $((P_0, Q_0, \Delta_0), (\xi_i, \sigma), (P, Q, \Delta))$, are invalidated and must be recalculated. If i is 1, then the algorithm tries to find another weak

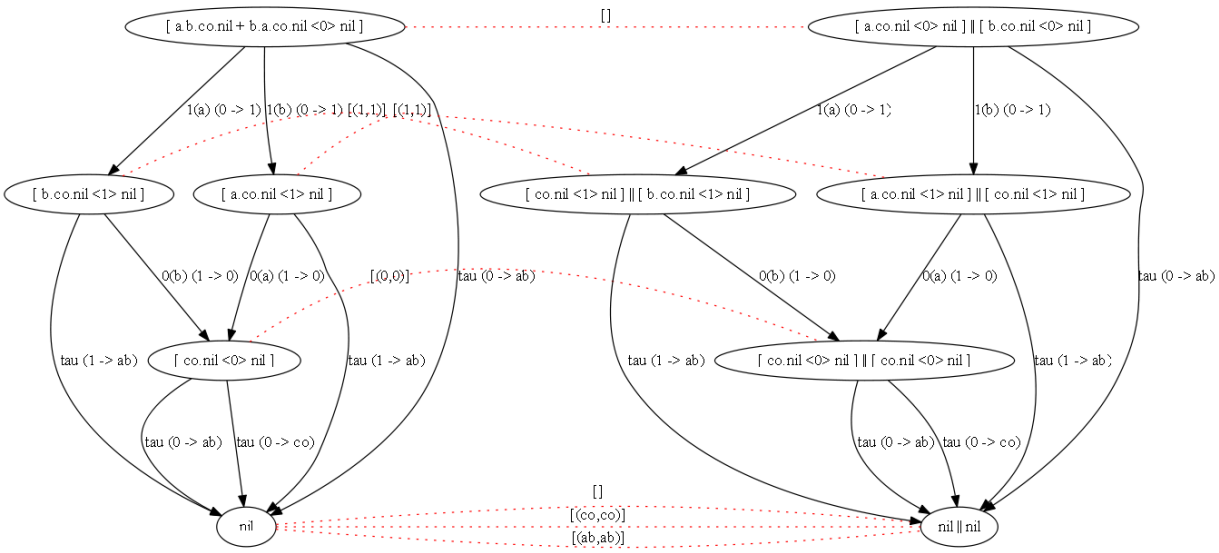


Fig. 4.5: Sample bisimulation of the parallel expansion law in TCCS^{m} .

transition from Q_0 to match $P_0 \xrightarrow{\xi}{}^{\mathtt{nl}}_{\sigma} P$. If i is 2, then the weak transition is picked from P_0 to match $Q_0 \xrightarrow{\xi}{}^{\mathtt{nl}}_{\sigma} Q$.

- the bisimulation graph G
- the set R of rejected triples (P, Q, Δ) such that P and Q are *not* bisimilar under Δ.

The local data structures are:

- a set of indexes $high_{P,\xi,\sigma,P',Q}$ and $low_{Q,\xi,\sigma,Q',P}$. For any transition $P \xrightarrow{\xi}{}^{\mathtt{nl}}_{\sigma} P'$, there exists a finite set $\mathcal{S}(P, \xi, \sigma, P', Q) = \{\, Q \overset{\xi'}{\Rightarrow}{}^{\mathtt{nl}}_{\sigma'} Q' \mid \exists Q'.Q \overset{\xi'}{\Rightarrow}{}^{\mathtt{nl}}_{\sigma'} Q', \xi \sim \xi' \,\}$ of weak transitions from Q that match the transition from P. We assume that the set $\mathcal{S}(-)$ is indexed. The index $high_{P,\xi,\sigma,P',Q}$ indicates which weak transition in $\mathcal{S}(P, \xi, \sigma, P', Q)$ has being currently been picked by the bisimulation algorithm to match a transition $P \xrightarrow{\xi}{}^{\mathtt{nl}}_{\sigma} P'$. The converse applies for *low* indexes when analysing $Q \xrightarrow{\xi}{}^{\mathtt{nl}}_{(\sigma,\ )} Q'$ transitions. The *high* and *low* indexes are increased monotonically starting from 0, ensuring that all weak transitions in $\mathcal{S}(-)$ are attempted at most once.
- a set A of justification edges $((P, Q, \Delta), (\xi_i, \sigma), (P', Q', \Delta'))$ that have been invalidated because (P', Q', Δ') is in R.

The algorithm incrementally calculates a bisimulation graph with the aforementioned properties. The input of the procedure BISIM is a triple (P, Q, Δ), from which G is built. The state spaces of P and Q are explored using a depth-first strategy. Any triple (P, Q, Δ) is analyzed at most once. If a triple is rejected, it is moved from G to R, where the triple cannot be retrieved anymore. When the LTSs of P and Q have finite states and are image-finite, termination is guaranteed by the fact that all triples are evaluated once.

An implementation of the algorithm is available at the following URL:

`https://bitbucket.org/carlo_spaccasassi/communicating-transactions`

The bisimulation algorithm is implemented in Haskell, using the PatriciaTree graph library to represent the graph G. The implementation outputs the LTSs of the input processes and the bisimulation relation in the DOT format. Figure 4.5 shows the bisimulation automatically calculated by the implementation for the processes $P = [\![a.b.\mathtt{co} + b.a.\mathtt{co} \triangleright_{k_0} 0]\!]$ and $Q = [\![a.\mathtt{co} \triangleright_{k_0} 0]\!] \mid [\![b.\mathtt{co} \triangleright_{k_0} 0]\!]$, which is a version of the parallel expansion law of CCS in the transactional setting ($[\![a.b.\mathtt{co} + b.a.\mathtt{co} \triangleright_{k_0} 0]\!] \approx_{\mathsf{hl}} [\![a.\mathtt{co} \triangleright_{k_0} 0]\!] \mid [\![b.\mathtt{co} \triangleright_{k_0} 0]\!]$). The red dotted lines represent bisimilar states in the two LTSs. The label on such lines are dependency set Δ. The graphical rendering of the DOT file has been performed with GVEdit, from the Graphviz software.

Chapter 5

TransCML

A crucial aspect to asses the effectiveness of communicating transaction in a programming language is the invention of efficient runtime implementations. In this chapter we describe the challenges and the first experimental results in our investigation of pragmatic approaches to communicating transactions. We equip a simple concurrent functional language with communicating transactions, called TCML, and a novel rule for the sequential evaluation of transactions. TCML formalizes the informal programming language that we introduced in Chapter 1. We use TCML to discuss the challenges in making an efficient implementation of languages with communicating transactions (Section 5.1).

We also use this language to give a modular implementation of consensus scenarios such as the SNO example from Sec. 1.1. The simple operational semantics of this language allows for the communication of SNO processes with arbitrary other processes (such as the Babysitter process) without the need to add code for the SNO protocol in those processes. Moreover, the more efficient, partially aborting strategy discussed above is captured in this semantics.

Our semantics of this language is non-deterministic, allowing different runtime scheduling strategies of processes, some more efficient than others. To study their relative efficiency we have developed a skeleton implementation of the language which allows us to plug in and evaluate such runtime strategies (Section 5.2). We describe several such strategies (Section 5.3) and report the results of our evaluations (Section 5.4). We conclude by discussing the results and drawing some conclusions for efficient transaction scheduling (Section 5.5). This chapter's material is drawn from [Spaccasassi and Koutavas, 2013].

5.1 The TCML Language

We study TCML, a language combining a simply-typed λ-calculus with π-calculus and communicating transactions. For this language we use the abstract syntax shown in Fig. 5.1 and the usual abbreviations from the λ- and π-calculus. Values in TCML are either constants of base type (**unit**, **bool**, and **int**), pairs of values (of type $T \times T$), recursive functions ($T \to T$), and channels carrying values of type T (T **chan**). A simple type system (with appropriate progress and preservation theorems) can be found in the technical report from [Spaccasassi, 2013] and is omitted here.

Source TCML programs are expressions in the functional core of the language, ranged over by

$$
\begin{array}{rcl}
T & ::= & \mathbf{unit} \mid \mathbf{bool} \mid \mathbf{int} \mid T \times T \mid T \to T \mid T\,\mathbf{chan} \\
v & ::= & x \mid () \mid \mathbf{true} \mid \mathbf{false} \mid n \mid (v, v) \mid \mathbf{fun}\, f(x) = e \mid c \\
e & ::= & v \mid (e, e) \mid e\ e \mid op\ e \mid \mathbf{let}\, x = e\, \mathbf{in}\, e \mid \mathbf{if}\, e\, \mathbf{then}\, e\, \mathbf{else}\, e \\
 & \mid & \mathbf{send}\ e\ e \mid \mathbf{recv}\ e \mid \mathbf{newChan}_T \mid \mathbf{spawn}\, e \\
 & \mid & \mathbf{atomic}[\![e \rhd_k e]\!] \mid \mathbf{commit}\, k \\
P & ::= & e \mid P \parallel P \mid \nu c.P \mid [\![P \rhd_k P]\!] \mid \mathtt{co}k \\
op & ::= & \mathbf{fst} \mid \mathbf{snd} \mid \mathbf{add} \mid \mathbf{sub} \mid \mathbf{mul} \mid \mathbf{leq} \\
E & ::= & [] \mid (E, e) \mid (v, E) \mid E\ e \mid v\ E \mid op\ E \mid \mathbf{let}\, x = E\, \mathbf{in}\, e \\
 & \mid & \mathbf{if}\, E\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 \mid \mathbf{send}\ E\ e \mid \mathbf{send}\ v\ E \mid \mathbf{recv}\ E \mid \mathbf{spawn}\, E
\end{array}
$$

where $n \in \mathbb{N}$, $x \in \mathit{Var}$, $c \in \mathit{Chan}$, $k \in \mathcal{K}$

Fig. 5.1: TCML syntax.

e, whereas running programs are processes derived from the syntax of P. Besides standard lambda calculus expressions, the functional core contains the constructs **send** $c\ e$ and **recv** c to synchronously send and receive a value on channel c, respectively, and $\mathbf{newChan}_T$ to create a new channel of type $\mathbf{chan}\, T$. The constructs **spawn** and **atomic**, when executed, respectively spawn a new process and transaction; **commit** k commits transaction k—we will shortly describe these constructs in detail. As usual, the expression e_1; e_2 is syntactic sugar for $\mathtt{let}\, x = e_1\ \mathtt{in}\ e_2$ when x is not free in e_2.

A simple running process can be just an expression e. It can also be constructed by the parallel composition of P and Q ($P \parallel Q$). We treat free channels as in the π-calculus, considering them to be *global*. Thus if a channel c is free in both P and Q, it can be used for communication between these processes. The construct $\nu c.P$ encodes π-calculus restriction of the scope of c to process P. We use the Barendregt convention for bound variables and channels and identify terms up to alpha conversion. Moreover, we write $\mathrm{fc}(P)$ for the free channels in process P.

We write $[\![P_1 \rhd_k P_2]\!]$ for the process encoding a communicating transaction. This can be thought of as the process P_1, the *default* of the transaction, which runs until the transaction *commits*. If, however, the transaction *aborts* then P_1 is discarded and the entire transaction is replaced by its *alternative* process P_2. Intuitively, P_2 is the continuation of the transaction in the case of an abort. As we will explain, commits are asynchronous, requiring the addition of process $\mathtt{co}k$ to the language. The name k of the transaction is bound in P_1. Thus only the default of the transaction can potentially spawn a $\mathtt{co}k$. The meta-function $\mathit{ftn}(P)$ gives us the free transaction names in P.

Processes with no free variables can reduce using transitions of the form $P \longrightarrow Q$. These transitions for the functional part of the language are shown in Fig. 5.2 and are defined in terms of reductions $e \hookrightarrow e'$ (where e is a *redex*) and eager, left-to-right evaluation contexts E whose grammar is given in Fig. 5.1. Due to a unique decomposition lemma, an expression e can be decomposed to an evaluation context and a redex expression in only one way. Here we use $e[u/x]$ for the standard capture-avoiding substitution, and $\delta(op, v)$ for a meta-function returning the result of the operator op on v, when this is defined.

Rule Step lifts functional reductions to process reductions. The rest of the reduction rules of Fig. 5.2 deal with the concurrent and transactional side-effects of expressions. Rule Spawn reduces a $\mathbf{spawn}\, v$ expression at evaluation position to the unit value, creating a new process running the

IF-TT	$\mathbf{if}\,\mathbf{true}\,\mathbf{then}\,e_1\,\mathbf{else}\,e_2$	$\hookrightarrow$	e_1	
IF-FF	$\mathbf{if}\,\mathbf{false}\,\mathbf{then}\,e_1\,\mathbf{else}\,e_2$	$\hookrightarrow$	e_2	
LET	$\mathbf{let}\,x = v\,\mathbf{in}\,e$	$\hookrightarrow$	$e[v/x]$	
OP	$op\,v$	$\hookrightarrow$	$\delta(op, v)$	
APP	$(\mathbf{fun}\,f(x) = e)\;v_2$	$\hookrightarrow$	$e[\mathbf{fun}\,f(x) = e/f][v_2/x]$	
STEP	$E[e]$	$\longrightarrow$	$E[e']$	if $e \hookrightarrow e'$
SPAWN	$E[\mathbf{spawn}\,v]$	$\longrightarrow$	$v\;() \parallel E[()]$	
NEWCHAN	$E[\mathbf{newChan}_T]$	$\longrightarrow$	$\nu c.E[c]$	if $c \notin \mathrm{fc}(E[()])$
ATOMIC	$E[\mathbf{atomic}[\![e_1 \rhd_k e_2]\!]]$	$\longrightarrow$	$[\![E[e_1] \rhd_k E[e_2]]\!]$	
COMMIT	$E[\mathbf{commit}\,k]$	$\longrightarrow$	$\mathtt{co}k \parallel E[()]$	

Fig. 5.2: Sequential reductions

application $v\;()$. The type system of the language guarantees that value v here is a thunk. With this rule we can derive the reductions:

$$\begin{aligned}\mathbf{spawn}(\lambda().\,\mathbf{send}\;\;c\;1);\mathbf{recv}\;\;c &\longrightarrow (\lambda().\,\mathbf{send}\;\;c\;1)\;() \parallel \mathbf{recv}\;\;c\\ &\longrightarrow \mathbf{send}\;\;c\;1 \parallel \mathbf{recv}\;\;c\end{aligned}$$

The resulting processes of these reductions can then communicate on channel c. As we previously mentioned, the free channel c can also be used to communicate with any other parallel process. Rule NEWCHAN gives processes the ability to create new, locally scoped channels. Thus, the following expression will result in an input and an output process that can *only* communicate with each other:

$$\begin{aligned}&\mathbf{let}\,x = \mathbf{newChan}_{\mathbf{int}}\,\mathbf{in}\,(\mathbf{spawn}\,(\lambda().\,\mathbf{send}\;\;x\;1);\mathbf{recv}\;\;x)\\ &\longrightarrow \nu c.\,(\mathbf{spawn}\,(\lambda().\,\mathbf{send}\;\;c\;1);\mathbf{recv}\;\;c)\\ &\longrightarrow^{*} \nu c.\,(\mathbf{send}\;\;c\;1 \parallel \mathbf{recv}\;\;c)\end{aligned}$$

Rule ATOMIC starts a new transaction in the current (expression-only) process, engulfing the entire process in it, and storing the abort continuation in the alternative of the transaction. Rule COMMIT spawns an asynchronous commit. Transactions can be arbitrarily nested, thus we can write:

$$\begin{aligned}&\mathbf{atomic}[\![\mathbf{spawn}(\lambda().\,\mathbf{recv}\,c;\mathbf{commit}\,k) \rhd_k ()]\!];\\ &\mathbf{atomic}[\![\mathbf{recv}\,d;\mathbf{commit}\,l \rhd_l ()]\!]\\ &\longrightarrow [\![\mathbf{spawn}(\lambda().\,\mathbf{recv}\,c;\mathbf{commit}\,k);\;\;\mathbf{atomic}[\![\mathbf{recv}\,d;\mathbf{commit}\,l \rhd_l ()]\!]\\ &\qquad \rhd_k ();\;\;\mathbf{atomic}[\![\mathbf{recv}\,d;\mathbf{commit}\,l \rhd_l ()]\!]]\!]\\ &\longrightarrow^{*} [\![(\mathbf{recv}\,c;\mathbf{commit}\,k) \parallel [\![\mathbf{recv}\,d;\mathbf{commit}\,l \rhd_l ()]\!]\\ &\qquad \rhd_k ();\mathbf{atomic}[\![\mathbf{recv}\,d;\mathbf{commit}\,l \rhd_l ()]\!]]\!]\end{aligned}$$

This process will commit the k-transaction after an input on channel c and the inner l-transaction after an input on d. As we will see, if the k transaction aborts then the inner l-transaction will be discarded (even if it has performed the input on d) and the resulting process (the alternative of k) will restart l:

$$();\mathbf{atomic}[\![\mathbf{recv}\,d;\mathbf{commit}\,l \rhd_l ()]\!]$$

The effect of this abort will be the rollback of the communication on d reverting the program to a

$$\textsc{Sync}\quad \frac{}{E_1[\mathbf{recv}\ \ c] \parallel E_2[\mathbf{send}\ \ c\ v] \longrightarrow E_1[v] \parallel E_2[()]} \qquad \textsc{Eq}\quad \frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q}$$

$$\textsc{Par}\quad \frac{P_1 \longrightarrow P_1'}{P_1 \parallel P_2 \longrightarrow P_1' \parallel P_2} \qquad \textsc{Chan}\quad \frac{P \longrightarrow P'}{\nu c.P \longrightarrow \nu c.P'}$$

$$\textsc{Emb}\quad \frac{}{P_1 \parallel [\![P_2 \rhd_k P_3]\!] \longrightarrow [\![(P_1 \parallel P_2) \rhd_k (P_1 \parallel P_3)]\!]} \qquad \textsc{Step}\quad \frac{P \longrightarrow P'}{[\![P \rhd_k P_2]\!] \longrightarrow [\![P' \rhd_k P_2]\!]}$$

$$\textsc{Co}\quad \frac{P_1 \equiv \mathtt{co}k \parallel P_1'}{[\![P_1 \rhd_k P_2]\!] \longrightarrow P_1'/k} \qquad \textsc{Abort}\quad \frac{}{[\![P_1 \rhd_k P_2]\!] \longrightarrow P_2}$$

Fig. 5.3: Concurrent and Transactional reductions (omitting symmetric rules).

consistent state.

Process and transactional reductions are handled by the rules of Fig. 5.3. The first four rules (Sync, Eq, Par, and Chan) are direct adaptations of the reduction rules of the π-calculus, which allow parallel processes to communicate, and propagate reductions over parallel and restriction. These rules use an omitted structural equivalence ($\equiv$) to identify terms up to the reordering of parallel processes and the extrusion of the scope of restricted channels, in the spirit of the π-calculus semantics. Rule Step propagates reductions of default processes over their respective transactions. The remaining rules are taken from TransCCS [de Vries et al., 2010].

Rule Emb encodes the *embedding* of a process P_1 in a parallel transaction $[\![P_2 \rhd_k P_3]\!]$. This enables the communication of P_1 with P_2, the default of k. It also keeps the current continuation of P_1 in the alternative of k in case it aborts. To illustrate the mechanics of the embed rule, let us consider the above nested transaction running in parallel with the process $P = \mathbf{send}\ \ d\ (); \mathbf{send}\ \ c\ ()$:

$$\begin{aligned}&[\![(\mathbf{recv}\, c; \mathbf{commit}\, k) \parallel [\![\mathbf{recv}\, d; \mathbf{commit}\, l \rhd_l ()]\!]\\ &\rhd_k (); \mathbf{atomic}[\![\mathbf{recv}\, d; \mathbf{commit}\, l \rhd_l ()]\!]]\!] \qquad \parallel \quad P\end{aligned}$$

After two embedding transitions we will have

$$[\![(\mathbf{recv}\, c; \mathbf{commit}\, k) \quad \parallel \quad [\![P \parallel \mathbf{recv}\, d; \mathbf{commit}\, l \ \ \rhd_l \ \ P \parallel ()]\!] \ \ \rhd_k \quad P \parallel \ldots]\!]$$

Now P can communicate on d with the inner transaction:

$$[\![(\mathbf{recv}\, c; \mathbf{commit}\, k) \quad \parallel \quad [\![\mathbf{send}\ \ c\ () \parallel \mathbf{commit}\, l \ \ \rhd_l \ \ P \parallel ()]\!] \ \ \rhd_k \quad P \parallel \ldots]\!]$$

Next, there are (at least) two options: either $\mathbf{commit}\, l$ spawns a $\mathtt{co}l$ process which causes the commit of the l-transaction, or the input on d is embedded in the l-transaction. Let us assume that the latter

occurs:

$$
\begin{aligned}
&[\![\ [\![(\mathbf{recv}\, c; \mathbf{commit}\, k) \ \parallel\ \mathbf{send}\ c\ () \ \parallel\ \mathbf{commit}\, l \\
&\quad \rhd_l\ (\mathbf{recv}\, c; \mathbf{commit}\, k) \ \parallel\ P \ \parallel\ ()]\!] \\
&\rhd_k\ P \parallel \ldots]\!] \\
&\longrightarrow^* [\![[\![\mathtt{co}k \parallel \mathtt{co}l \rhd_l \ldots]\!] \rhd_k \ldots]\!]
\end{aligned}
$$

The transactions are now ready to commit from the inner-most to the outer-most using rule Commit. Inner-to-outer commits are necessary to guarantee that all transactions that have communicated have reached an agreement to commit.

This also has the important consequence of making the following three processes behaviourally indistinguishable:

$$[\![P_1 \rhd_k P_2]\!] \parallel [\![Q_1 \rhd_l Q_2]\!]$$

$$[\![P_1 \parallel [\![Q_1 \rhd_l Q_2]\!] \rhd_k P_2 \parallel [\![Q_1 \rhd_l Q_2]\!]]\!]$$

$$[\![[\![P_1 \rhd_k P_2]\!] \parallel Q_1 \rhd_l [\![P_1 \rhd_k P_2]\!] \parallel Q_2]\!]$$

Therefore, an implementation of TCML, when dealing with the first of the three processes can pick any of the alternative, non-deterministic mutual embeddings of the k and l transactions without affecting the observable outcomes of the program. In fact, when one of the transactions has no possibility of committing or when the two transactions never communicate, an implementation can decide *never* to embed the two transactions in each-other. This is crucial in creating implementations that will only embed processes (and other transactions) only when necessary for communication, and pick the most *efficient* of the available embeddings. The development of implementations with efficient embedding strategies is one of the main challenges for scaling communicating transactions to pragmatic programming languages.

Similarly, aborts are entirely non-deterministic (Abort) and are left to the discretion of the underlying implementation. Thus in the above example any transaction can abort at any stage, discarding part of the computation. In such examples there is usually a multitude of transactions that can be aborted, and in cases where a "forward" reduction is not possible (due to deadlock) aborts are necessary. Making the TCML programmer in charge of aborts (as we do with commits) is not desirable since the purpose of communicating transactions is to lift the burden of manual error prediction and handling. Minimizing aborts, and automatically picking the aborts that will undo the fewer computation steps while still rewinding the program back enough to reach a successful outcome is another major challenge.

The SNO scenario can be simply implemented in TCML using *restarting transactions*. A restarting transaction uses recursion to re-initiate an identical transaction in the case of an abort:

$$\mathtt{atomic}_k[\![e]\!] \ \stackrel{\text{def}}{=}\ \mathbf{fun}\, r() = \mathbf{atomic}[\![e \rhd_k r\ ()]\!]$$

A transactional implementation of the SNO participants from Chapter 1 simply wraps their code in restating transactions:

$\mathbf{let}\ \mathit{alice} \ = \mathtt{atomic}_k[\![\mathit{sync\ dinner};\ \mathit{sync\ movie};\ \mathbf{commit}\, k]\!]\ \mathbf{in}$

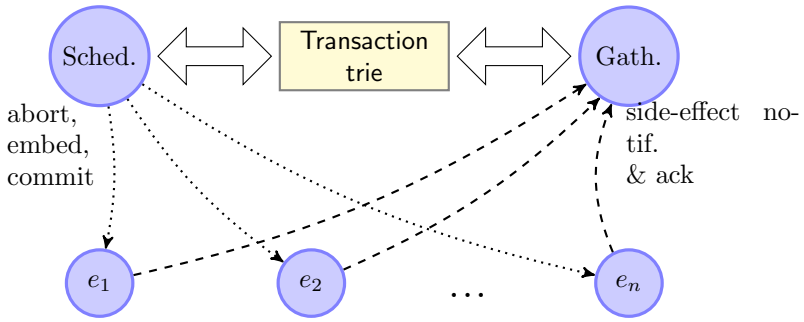


Fig. 5.4: TCML runtime architecture.

let *bob* $= \texttt{atomic}_k[\![$*sync dinner*; *sync dancing*; **commit** $k]\!]$ **in**
let *carol* $= \texttt{atomic}_k[\![$*sync dancing*; **commit** $k]\!]$ **in**
let *david* $= \texttt{atomic}_k[\![$*sync dancing*; *sync movie*; **commit** $k]\!]$ **in**
spawn *alice*; **spawn** *bob*; **spawn** *carol*; **spawn** *david*

Here *dinner*, *dancing*, and *movie* are implementations of CSP synchronization channels and *sync* a function to synchronize on these channels. Compared to a potential ad-hoc implementation of SNO in CML the simplicity of the above code is evident (the version of Bob communicating with the Babysitter is just as simple). However, as we discuss in Sec. 5.4, this simplicity comes with a severe performance penalty, at least for straightforward implementations of TCML. In essence, the above code asks from the underlying transactional implementation to solve an NP-complete satisfiability problem.

In the following sections we describe an implementation where these transactional scheduling decisions can be plugged in, and a number of heuristic transactional schedulers we have developed and evaluated. This work shows that although more advanced heuristics bring measurable performance benefits, the exponential number of runtime choices require the development of innovative compilation and execution techniques to make communicating transactions a realistic solution for programmers.

5.2 An Extensible Implementation Architecture

We have developed an interpreter for the reduction semantics of TCML using Concurrent Haskell [Jones et al., 1996, Marlow et al., 2001] to which we can plug-in different decisions about the non-deterministic transitions of our semantics. Here we briefly explain the runtime architecture of this interpreter, shown in Fig. 5.4.

The main Haskell threads are shown as round nodes in the figure. Each concurrent functional expression e_i is interpreted in its own thread according to the sequential reduction rules in Fig. 5.2 of the previous section. Side-effects in an expression will be generally handled by the interpreting thread, creating new channels, spawning new threads, and starting new transactions. Sequential and concurrent operations are mapped to their Haskell counterparts (e.g. TCML functions to Haskell functions). Synchronous channels are built on top of Haskell's `MVar`s, together with special code to preserve channel invariants in case their enclosing transactions are modified during communication.

Except for channel creation, the evaluation of all other side-effects in an expression will cause a *notification* (shown as dashed arrows in Fig. 5.2) to be sent to the *gatherer* process (Gath.). This

process is responsible for maintaining a global view of the state of the running program in a *Trie* data-structure. This data-structure essentially represents the transactional structure of the program; i.e., the logical nesting of transactions and processes inside running transactions:

```
data TTrie = TTrie {
  threads   :: Set ThreadID,
  children  :: Map TransactionID TTrie, ... }
```

A `TTrie` node represents a transaction, or the top-level of the program. The main information stored in such a node is the set of threads (`threads`) and transactions (`children`) running in that transactional level. Each child transaction has its own associated `TTrie` node. An invariant of the data-structure is that each thread and transaction identifier appears only once in it. For example the complex program we saw on page 85:

$$\begin{aligned}&[\![(\mathbf{recv}\, c; \mathbf{commit}\, k)^{\mathtt{tid}_1} \parallel [\![(\mathbf{recv}\, d; \mathbf{commit}\, l)^{\mathtt{tid}_2} \rhd_l ()]\!]\\ &\rhd_k (); [\![l \blacktriangleright \mathbf{recv}\, d; \mathbf{commit}\, l]\!]()]\!] \qquad\qquad \parallel P^{\mathtt{tid}_P}\end{aligned}$$

will have an associated trie:

```
TTrie{threads  = {tid_P},
      children = {k ↦ TTrie{threads  = {tid_1},
                            children = {l ↦ TTrie{threads  = {tid_2},
                                                  children = ∅}}}}}
```

The last ingredient of the runtime implementation is the *scheduler* thread (Sched. in Fig. 5.4). This makes decisions about the commit, embed and abort transitions to be performed by the expression threads, based on the information in the trie. Once such a decision is made by the scheduler, appropriate signals (implemented using Haskell asynchronous exceptions [Marlow et al., 2001]) are sent to the running threads, shown as dotted lines in Fig. 5.4. Our implementation is parametric to the precise algorithm that makes scheduler decisions, and in the following section we describe a number of such algorithms we have tried and evaluated.

A scheduler signal received by a thread will cause the update of the *local transactional state* of the thread, affecting the future execution of the thread. The local state of a thread is an object of the `TProcess` data-type:

```
data TProcess = TP {
  expr :: Expression,
  ctx  :: Context,
  tr   :: [Alternative] }
```

```
data Alternative = A {
  tname :: TransactionID,
  pr    :: TProcess }
```

The local state maintains the expression (`expr`) and evaluation context (`ctx`) currently interpreted by the thread and a list of *alternative* processes (represented by objects of the `Alternative` data-type). This list contains the continuations stored when the thread was embedded in transactions. The nesting of transactions in this list mirrors the transactional nesting in the global trie and is thus compatible with the transactional nesting of other expression threads. Let us go back to the example of page 85:

$$\begin{aligned}&[\![(\mathbf{recv}\, c; \mathbf{commit}\, k)^{\mathtt{tid}_1} \parallel [\![(\mathbf{recv}\, d; \mathbf{commit}\, l)^{\mathtt{tid}_2} \rhd_l ()]\!]\\ &\rhd_k (); [\![l \blacktriangleright \mathbf{recv}\, d; \mathbf{commit}\, l]\!]()]\!] \qquad\qquad \parallel P^{\mathtt{tid}_P}\end{aligned}$$

where $P = \texttt{send}d\ ();\texttt{send}c\ ()$. When P is embedded in both k and l, the thread evaluating P will have the local state object

$\texttt{TP\{expr = } P\texttt{, tr = [A\{tname = } l\texttt{, pr = } P\texttt{\}, A\{tname = } k\texttt{, pr = } P\texttt{\}]\}}$

recording the fact that the thread running P is part of the l-transaction, which in turn is inside the k-transaction. If either of these transactions aborts then the thread will rollback to P, and the list of alternatives will be appropriately updated (the aborted transaction will be removed).

Once a transactional reconfiguration is performed by a thread, an acknowledgment is sent back to the gatherer, who, as we discussed, is responsible for updating the global transactional structure in the trie. This closes a cycle of transactional reconfigurations initiated from the process (by starting a new transaction or thread) or the scheduler (by issuing a commit, embed, or abort).

What we described so far is a simple architecture for an interpreter of TCML. Various improvements are possible; for example, the gatherer is a message bottleneck, and together with the scheduler they are single points of failure. But such concerns are beyond the scope of this thesis. In the following section we discuss various policies for the scheduler which we then evaluate experimentally.

5.3 Transactional Scheduling Policies

We now turn our attention to investigate schedulers that make decisions on transactional reconfiguration based only on runtime heuristics. An important consideration when designing a scheduler is *adequacy* [Winskel, 1993, Chap. 13, Sec. 4]. For a given program, an adequate scheduler can produce all outcomes that the non-deterministic operational semantics can produce for that program. However, this does *not* mean that the scheduler should be able to produce all traces of the non-deterministic semantics. Many of these traces will simply abort and restart the same computations over and over again. Previous work on the behavioural theory of communicating transactions has shown that all program outcomes can be reached with traces that *never* restart a computation [de Vries et al., 2010]. Thus a goal for schedulers is to minimize re-computations by minimizing aborts.

Moreover, as we discussed at the end of Sec. 5.1, many of the exponential number of embeddings can be avoided without altering the observable behaviour of a program. This can be done by embedding a process inside a transaction only when this embedding is necessary to enable communication between the process and the transaction. We take advantage of this in a *communication-driven* scheduler we describe in this section.

Even after reducing the number of possible non-deterministic choices faced by the scheduler, in most cases we are still left with a multitude of alternative transactional reconfiguration options. Some of these are more likely to lead to efficient traces than other. However, to preserve adequacy we cannot exclude any of these options since the scheduler has no way to foresee their outcomes. In these cases we assign different, non-zero probabilities to the available choices, based on heuristics, which leads to measurable performance improvements without violating adequacy. Of course some program outcomes might be more likely than others. This approach trades measurable fairness for performance improvement.

However, the probabilistic approach is *theoretically fair*. Every finite trace leading to a program outcome has a non-zero probability. Diverging traces due to sequential reductions also have non-zero

probability to occur. The only traces with zero probability are those in the reduction semantics that have an infinite number of non-deterministic reductions. Intuitively, these are unfair traces that abort and restart transactions *ad infinitum*, even if other options are possible.

Random Scheduler (R). The first scheduler we consider is the random scheduler, whose policy is, at each point, to simply select one of all the non-deterministic choices with equal probability, without excluding any of them; any abort, embed, or commit actions are equally likely to happen. For example, the scheduler might decide at any time to embed Bob into Carol's transaction, or abort David. Although this naive scheduler is not particularly efficient, as one would expect, it is obviously adequate and fair according to the discussion above. If a reduction transition is available infinitely often, scheduler R will eventually select it.

There is much room for improvement. Suppose transaction k can commit:

$$[\![P \parallel \mathsf{co}k \triangleright_k Q]\!]$$

Since R makes no distinction between the choices of committing and aborting k, it will often unnecessarily abort k. All processes embedded in this transaction will have to roll back and re-execute; if k was a transaction that restarts, the transaction will also re-execute. This results to a considerable performance penalty. Similarly, scheduler R might preemptively abort a long-running transaction that could have committed, given enough time and embeddings.

Staged Scheduler (S). The staged scheduler partially addresses these issues by prioritizing its available choices. Whenever a transaction is ready to commit, scheduler S will always decide to send a commit signal to that transaction before aborting it or embedding another process in it. This does not violate adequacy; before continuing with the algorithm of S, let us examine the adequacy of prioritizing commits over other transactional actions with an example.

Consider the following program in which k is ready to commit.

$$[\![P \parallel \mathsf{co}k \triangleright_k Q]\!] \parallel R$$

If embedding R in k leads to a program outcome, then that outcome can also be reached after committing k from the residual $P \parallel R$.

Alternatively, a program outcome could be reachable by aborting k (from the process $Q \parallel R$). However, the co$k$ was spawned from one of the previous states of the program in the current trace. In that state, transaction k necessarily had the form: $[\![P' \parallel E[\mathbf{commit}\, k] \triangleright_k Q]\!]$, and the abort of k was enabled. Therefore, the staged interpreter indeed allows a trace leading to the program state $Q \parallel R$ from which the outcome in question is reachable.

If transaction T cannot commit, S prioritizes embeddings into T over aborting it. This decision is adequate because transactions that take an abort reduction before an embed step have an equivalent abort reduction after that step. When no commit nor embed options are available, the staged interpreter lets the transaction run with probability 0.95 to progress more in the current trace, and aborts it with probability 0.05 —these numbers have been fine-tuned experimentally.

This heuristic greatly improves performance by minimizing unnecessary aborts. Its drawback is that it does not abort transactions often, thus program outcomes reachable only from transactional alternatives are less likely to appear. Moreover, this scheduler does not avoid *unnecessary embeddings*.

Communication-Driven Scheduler (CD). To avoid spurious embeddings, scheduler CD improves over R by performing an embed transition only if it is *necessary* for an imminent communication. For example, at the very start of the SNO example the CD scheduler can only choose to embed Alice into Bob's transaction or viceversa, because they are the only processes ready to synchronize on *dinner*. Because of the equivalence

$$[\![P \triangleright_k Q]\!] \parallel R \equiv_{\text{cxt}} [\![P \parallel R \triangleright_k Q \parallel R]\!]$$

which we previously discussed, this scheduler is adequate.

For the implementation of this scheduler we augment the information in the trie data-structure (Sec. 5.2) with channels with a pending communication operation (if any). In Sec. 5.4 we show that this heuristic noticeably boosts performance because it greatly reduces the exponential number of embedding choices.

Delayed-Aborts Scheduler (DA). The final scheduler we report is DA, which adds a minor improvement upon scheduler CD. This scheduler keeps a timer for each running transaction k in the trie, and resets it whenever a non-sequential operation happens inside k. Transaction k can be aborted only when its timer expires. This strategy benefits transactions that perform multiple communications before committing. The CD scheduler is adequate because it only adds time delays.

5.4 Evaluation of the Interpreters

We now report the experimental evaluation of interpreters using the preceding Scheduling policies. The interpreters were compiled with GHC 7.0.3, and the experiments were performed on a Windows 7 machine with Intel® Core™i5-2520M (2.50 GHz) and 8Gb of RAM. We run several versions of two programs:

1. The three-way rendezvous (3WR) in which a number of processes compete to synchronize on a channel with *two* other processes, forming groups of three which then exchange values. This is a standard example of multi-party agreement [Reppy, 1999, Donnelly and Fluet, 2006, Lesani and Palsberg, 2011]. In the TCML implementation of this example each process nondeterministically chooses between being a *leader* or *follower* within a communicating transaction. If a leader and two followers communicate, they can all exchange values and commit; any other situation leads to deadlock and eventually to an abort of some of the transactions involved.

2. The SNO example of the introduction, as implemented in Section 5.1, with multiple instances of the Alice, Bob, Carol, and David processes.

To test scheduler scalability, we tested a number of versions of the above programs with differing numbers of competing parallel processes. Each test process continuously performs 3WR or SNO cycles and our interpreters are instrumented to measure the number of operations in a given time,

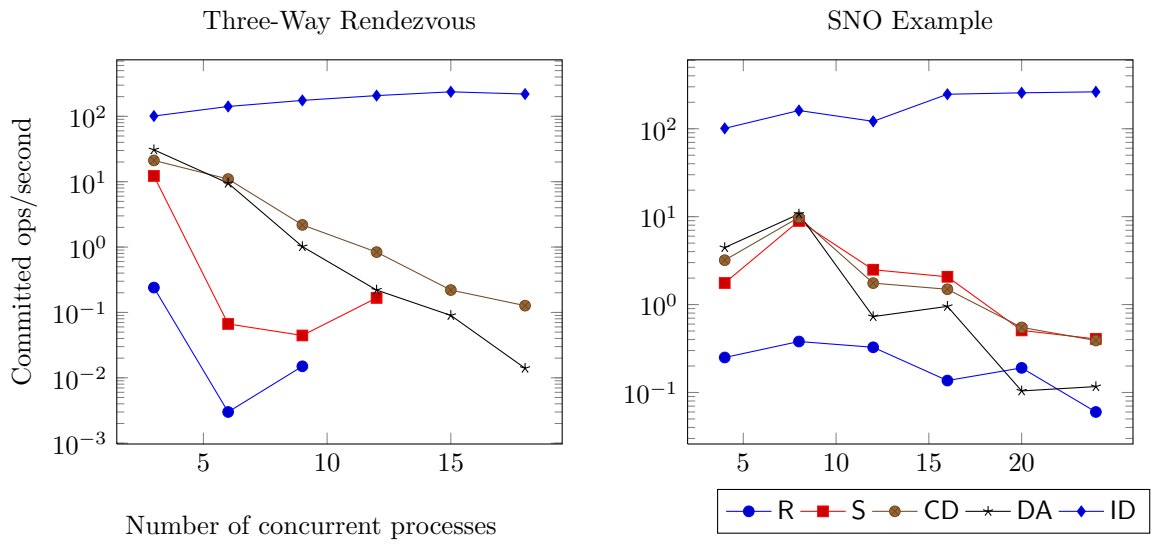


Fig. 5.5: Experimental Results.

from which we compute the *mean throughput* of successful operations. The results are shown in Fig. 5.5.

Each graph in the figure contains the mean throughput of operations (in logarithmic scale) as a function of the number of competing concurrent TCML processes. The graphs contain runs with each scheduler we discussed (random R, staged S, communication-driven, CD, and communication-driven with delayed aborts DA) as well as with an *ideal* non-transactional program (ID). The ideal program in the case of 3WR is similar to the TCML, non-abstract implementation [Reppy, 1999]. The ideal version of the SNO is running a simpler instance of the scenario, without any Carol processes—this instance has no deadlocks and therefore needs no error handling. Ideal programs give us a performance upper bound.

As predictable, the random scheduler (R)'s performance is the worst; in many cases R could not perform any operations in the window of measurements (30sec).

The other schedulers perform better than R by an order of magnitude. Even just prioritizing the transactional reconfiguration choices significantly cuts down the exponential number of inefficient traces. However, none of the schedulers scale to programs with more processes; their performance deteriorates exponentially. In fact, when we go from the communication-driven (CD) to the delayed aborts (DA) scheduler we see worst throughput in larger process pools. This is because with many competing processes there is more possibility to enter a path to deadlock; in these cases the results suggest that it is better to abort early.

The upper bound in the performance, as shown by the throughput of ID is one order of magnitude above that of the best interpreter, when there are few concurrent processes, and (within the range of our experiments) two orders when there are many concurrent processes. The performance of ID is increasing with more processes due to better utilization of the processor cores.

It is clear that in order to achieve a pragmatic implementation of TCML we need to address the exponential nature in consensus scenarios such as the ones we tested here. Our exploration of purely runtime heuristics shows that performance can be improved, but we need to turn to a different approach to close the gap between ideal ad-hoc implementations and abstract TCML implementations.

5.5 Conclusions

In this chapter we have presented TCML, a simple functional language with built-in support for consensus via communicating transactions. TCML has a simple operational semantics and can simplify the programming of advanced consensus scenarios; we introduced such an example (SNO and 3WR) which has a natural encoding in TCML. We also introduce the architecture and experimental results.

The usefulness of communicating transactions in real-world applications, however, depends on the invention of efficient implementations. We described the obstacles to overcome and our first experimental results. We gave a framework and a modular implementation to develop and evaluate current and future schedulers of communicating transactions, and used it to examine schedulers based solely on runtime heuristics. We have found that some of them improve upon the performance of a naive randomized implementation but do not scale to programs with significant contention, where exponential numbers of computation paths lead to necessary rollbacks. It is clear that purely dynamic strategies do not lead to sustainable performance improvements.

In order to obtain better performance, we intend to pursue a direction based on the extraction of information from the source code which will guide the language runtime. This information will include an abstract model of the communication behaviour of processes that can be used to predict with high probability their future communication pattern. This abstract model is session types, which will be the subject of the following two chapters. A promising approach to achieve this is the development of technology in type and effect systems and static analysis. Although the scheduling of communicating transactions is theoretically computationally expensive, realistic performance in many programming scenarios could be achievable by exploiting information from the static analysis.

Chapter 6

Session Types for ML

Naïve scheduling policies have been found to be insufficient to obtain a scalable implementation of communicating transactions in the presence of contention in Chapter 5. For example, in the case of the three-way rendezvous the scheduler will randomly abort transactions, until a successful configurations is found. This method is highly inefficient, because it is completely oblivious of the actual structure of the three-way rendezvous *protocol*.

We have already pointed out that, by analyzing the three-way rendezvous LTS in Fig. 1.4, it can be inferred that the only possible successful consensus group is one where a participant is the leader and two other are followers. If the scheduler had access to this information, its task could be simplified and limited to grouping together processes and assigning them proper roles, according to their protocols. It is interesting to note that the scheduler does not have to be aware of all the interactions happening between processes in a consensus group: once three TWR processes have been paired, they can complete the rendezvous without any intervention by the scheduler.

While the three-way rendezvous is a simple example, concurrent programming in general often requires processes to communicate according to intricate protocols. In mainstream programming languages these protocols are encoded implicitly in the program's control flow, and no support is available to extract them or for verifying their correctness. Therefore, before making a transaction scheduler protocol-aware, we first need mechanisms to extract protocols from programs in the first place. And in order to do this, we need a formalism to encode protocols in the first place.

[Honda, 1993] and then others ([Takeuchi et al., 1994, Honda et al., 1998, Gay and Hole, 1999]) suggested the use of *session types* to make communication protocols explicit and checkable in program typing. Since then, session type disciplines have been developed in a number of variations for process calculi and high-level programming languages (see [Hüttel et al.,] for an overview). In this chapter we provide a facility for static protocol checking in an ML-like programming languages equipped with session types, called $\mathrm{ML_S}$. We do not include communicating transactions in $\mathrm{ML_S}$ because of the complexity of the topic. We will discuss in Chapter 9 how we believe that session types and transactions can be combined for better scheduling performance.

The following section presents an overview of the current approaches to combine functional languages with session types, and the advantages contributed by our approach. Section 6.2 shows interesting examples which use session communication and we would like to type in $\mathrm{ML_S}$. Section 6.3

gives the syntax and operational semantics of the language. Section 6.4 presents the details of our typing system and the type soundness result. Section 6.5 discusses the extension of our type system to a form of recursive session types. Section 6.6 presents related work.

6.1 Overview

Many of the current efforts to developing high-level languages with session types use a single substructural type system which combines both expression and session typing (e.g., [Vasconcelos et al., 2006, Wadler, 2012, Ng et al., 2011]). This is a flexible approach, which enables the typing of programs mixing different language features, but poses significant challenges when used to extend an existing language such as ML: the extension needs to be conservative (i.e., type existing programs), scale to the full language, and enable type inference to considerably reduce the burden on providing type annotations. Although there are successes using this approach for new programming languages, to our knowledge there is yet to be an ML type system encompassing session types that addresses these challenges.

An alternative approach to extending a programming language with session types is through the use of *monads* [Toninho et al., 2013, Pucella and Tov, 2008]. With this approach session-typed communications are isolated from the rest of the language, providing a conservative-by-construction language extension. Pucella and Tov [Pucella and Tov, 2008] have showed that a level of type inference is possible in this setting, albeit it needs to be guided by the programmer with a number of type-level expressions. Perhaps the main challenge with using monads is the ability to combine language features. As explained in [Toninho et al., 2013], functional abstraction of common communication sequences, where the channel is given as an argument, is not possible. Moreover, the combination of a session communication monad with other effects such as mutable state and exceptions is not obvious.

In this chapter we put forward a new approach for adding session types to high-level programming languages, which we present in a core of ML. Rather than typing sessions directly on the source code, our approach is based on first extracting the *communication effect* of program expressions and then imposing a session type discipline on this effect. Our goal is to use this two-level approach to simplify the extension of ML with session types and achieve a sound and complete session type inference algorithm.

To extract the communication effect of ML programs we adapt and extend the work on type-and-effect systems developed by Amtoft, Nielson and Nielson [Amtoft et al., 1999, Nielson and Nielson, 1996]. Our extension provides a method for dealing with aliasing of session endpoints using regions, obviating the need for a substructural type system for ML. Furthermore, we develop a session type discipline for communication effects inspired by Castagna et al. [Castagna et al., 2009] and show that it guarantees a weak deadlock-freedom property. In principle, this discipline could be replaced with alternative ones from the literature, although care is needed to preserve inference and typing guarantees.

A simple extension and combination of the above techniques does not achieve our goal. We also develop in Chapter 7 a sound and complete inference algorithm for our session type discipline which includes delegation. To our knowledge this is the first such inference algorithm. Our two-level approach to extending ML with sessions offers the following benefits:

A sound and complete session type inference algorithm for ML: Our session type discipline admits a sound and complete inference algorithm which automatically discovers session types used in programs, and can handle session delegation. Our system also inherits a sound and complete inference algorithm of the type-and-effect system for ML [Amtoft et al., 1999] with modest modifications, which extends Milner's $\mathcal{W}$ algorithm [Milner, 1978a].

A conservative extension of the base language: any expression not using session communication is assigned a "pure" effect and is trivially typable in our system.

Feature combination and scalability: our approach already allows the combination of session communication with functional features and can type interesting programs. Moreover, the language of communication effects can be easily enriched with new linguistic features; in Section 6.5 we show an extension of our language with recursive session types.

Typing guarantees weak deadlock freedom: programs that do not diverge, where every request for opening a session is met, are guaranteed to run to completion and avoid type errors. We believe this to be a pragmatic compromise between session type systems that do not guarantee any deadlock freedom (e.g., [Vasconcelos et al., 2006]) and those that guarantee global deadlock freedom but reject programs with general recursion (e.g., [Wadler, 2012]). Our typing guarantee is similar to that in [Toninho et al., 2013] with the exception that we need to account for dynamic session creation.

Typing requires minimal programmer annotations: as we will show in Section 6.2, there is only one instance where programmers need to manually guide the type checker to type their code.

Here we focus on a pure core of ML. However, we believe that the techniques we use would work equally well in other high-level languages—we leave this for future work. The language employs Hindley-Milner polymorphism and primitives for sending pure monomorphic values, internal and external choices through choice labels L, and session delegation and resumption. The ability to send functions containing communication effects would be an implicit form of delegation and would unnecessarily complicate the exposition of our system.

For simplicity of presentation we develop our system for finite sessions and then show how it can be extended with recursive sessions. However, even with finite session types, we use a novel typing for a class of useful, non-pure recursive functions, which we call *self-contained functions*. For example consider the function of type $\mathsf{Unit} \to \mathsf{Unit}$:

$$\begin{aligned} &\texttt{rec}\, f(\_) \Rightarrow \texttt{let}\, z = \texttt{req-init}\, () \,\texttt{in} \\ &\qquad\qquad \texttt{case}\, z\, \{L_0 \Rightarrow f\,(),\quad L_1 \Rightarrow ()\} \end{aligned} \tag{$\star$}$$

When called, the function invokes `req-init` requesting to open a new session on a global channel `init`. When the request is accepted by a partner process, a *session endpoint* is bound to z through which private communication between the two processes is possible. The process presents its partner with choices L_0 and L_1. If L_0 is chosen, the function recurs; if L_1 is chosen the function terminates. In both cases no further communication on endpoint z will occur, and therefore z can be given the

finite session type $\Sigma\{?L_0.\mathsf{end},\ ?L_1.\mathsf{end}\}$. This behaviour is representative of systems that run a finite protocol for an arbitrary number of iterations.

6.2 Motivating Examples

Before presenting the details of the type system, we give and discuss two example implementations of a swap service, which symmetrically exchanges values between pairs of processes connecting to it. These are typical examples where process communication is used in programming.

6.2.1 A Swap Service

Our first example is a direct implementation of the swap service where a spawned coordinator process accepts two connections on a channel `swp`, opening two concurrent sessions with processes that want to exchange values. It then receives the two values from the processes, sends them back crosswise, and recurs.

$$\begin{aligned}
&\texttt{let}\ \mathit{coord} = \texttt{rec}\ f(_) \Rightarrow \texttt{let*}\ z_1 = \texttt{acc-swp}\ () \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad x_1 = \texttt{recv}\ z_1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad z_2 = \texttt{acc-swp}\ () \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad x_2 = \texttt{recv}\ z_2 \\
&\qquad\qquad\qquad\qquad\qquad\quad \texttt{in send}\ z_2\, x_1;\ \texttt{send}\ z_1\, x_2;\ f\,() \\
&\texttt{in spawn}\ \mathit{coord}; \ldots
\end{aligned}$$

It is easy to see that the two endpoints that the coordinator receives from the two calls to `acc-swp` will be used according to the session type:

$$?T.!T.\mathsf{end}$$

This says that, on each of the two endpoints, the coordinator will first read a value of some type T ($?T$) and then output a value of the same type ($!T$) and close the endpoint (end). From the code we see that the interleaving of sends and receives on the two endpoints achieves the desired swap effect.

To use this service we simply apply the following *swap* function to the value to be exchanged.

$$\begin{aligned}
&\texttt{let}\ \mathit{swap} = \texttt{fun}\ x \Rightarrow \texttt{let}\ z = \texttt{req-swp}\ () \\
&\qquad\qquad\qquad\qquad\qquad\quad \texttt{in send}\ z\, x;\ \texttt{recv}\ z \\
&\texttt{in}\ \ldots
\end{aligned}$$

When applied to x, *swap* requests a connection on `swp`, receiving a session endpoint z, then sends x on z and finally receives a value on z which is returned as the result of the function. The endpoint received by the call to `req-swp` will be used according to the session type:

$$!T_1.?T_2.\mathsf{end}$$

Where T_1 and T_2 are the argument and return types of the function, respectively. By comparing the two session types above we can see that the coordinator and the swap service can communicate

without type errors, and indeed are typable, when $T_1 = T_2 = T$.

Our type system is able to type this simple swap library, and our type inference algorithm can automatically deduce the two session types from the source code.

6.2.2 Delegation for Efficiency

The preceding simple swap service may become a bottleneck of the program since all data communications are sequentialized through the coordinator service. This can reduce performance, especially if values of significant size are swapped. Here we give a more efficient implementation in which the values exchanged do not go through the coordinator.

The new swap function is:

$$
\begin{array}{l}
\texttt{let}\ \mathit{swap} = \texttt{fun}\, x \Rightarrow \texttt{let}\, z = \texttt{req-swp}\, () \\
\qquad\qquad\qquad\qquad \texttt{in case}\, z\, \{\texttt{Fst} \Rightarrow \texttt{send}\, z\, x;\ \texttt{recv}\, z \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{Snd} \Rightarrow \texttt{let*}\, z' = \texttt{resume}\, z \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad y' = \texttt{recv}\, z' \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{in send}\, z'\, x;\ y' \quad \} \\
\texttt{in}\ \ldots
\end{array}
$$

It again connects to the coordinator over channel `swp`, but now offers two choices: `Fst` and `Snd`. If the coordinator selects the first one then the swap method behaves as before: it sends its value and receives another which it returns. If the coordinator selects the second choice then *swap* will resume (i.e., input) another endpoint by calling `resume` z, bind it to z', and then receive a value y' from z', send x on z' and finally return y'. Therefore, the session type of endpoint z is

$$\Sigma\{?\texttt{Fst}.!T_1.?T_2.\texttt{end},\quad ?\texttt{Snd}.?\eta.\texttt{end}\}$$

denoting the choice between the two options `Fst` and `Snd`, and the protocol followed in each one. Here η is the session type of endpoint z' which is $\eta = ?T_2.!T_1.\texttt{end}$. Again, T_1 and T_2 are the argument and return types of `swap`.

The new coordinator is:

$$
\begin{array}{l}
\texttt{let}\ \mathit{coord} = \texttt{rec}\, f(\_) \Rightarrow \texttt{let}\, z_1 = \texttt{acc-swp}\, () \\
\qquad\qquad\qquad\qquad\qquad\quad \texttt{in sel-Fst}\, z_1; \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{let}\, z_2 = \texttt{acc-swp}\, () \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{in sel-Snd}\, z_2; \texttt{deleg}\ z_2\, z_1; f\, () \\
\texttt{in spawn}\ \mathit{coord}; \ldots
\end{array}
$$

It accepts two sessions on `swp`, receiving two endpoints: z_1 and z_2. It selects `Fst` on z_1 (`sel-Fst` z_1) and `Snd` on z_2 (`sel-Snd` z_2). The coordinator then sends z_1 over z_2 and recurs.

The protocol followed by the coordinator over the two endpoints is now more intricate, and in fact different for each one. However, both endpoints must have the same session type because they are

Syntax

Expressions:	$e ::= v \mid x \mid (e,e) \mid e\,e \mid \texttt{let}\,x = e\,\texttt{in}\,e \mid \texttt{if}\,e\,\texttt{then}\,e\,\texttt{else}\,e$ $\mid \texttt{spawn}\,e \mid \texttt{case}\,e\,\{L_i \Rightarrow e_i\}_{i\in I}$
Systems:	$S ::= e \mid S \parallel S$
Values:	$v ::= k \in \mathsf{Const} \mid (v,v) \mid \texttt{fun}\,x \Rightarrow e \mid \texttt{rec}\,f(x) \Rightarrow e \mid p$ $\mid \texttt{req-}c \mid \texttt{acc-}c \mid \texttt{send} \mid \texttt{recv} \mid \texttt{sel-}L \mid \texttt{deleg} \mid \texttt{resume}$
Eval. Contexts:	$E ::= [\cdot] \mid (E,e) \mid (v,E) \mid E\,e \mid v\,E \mid \texttt{let}\,x = E\,\texttt{in}\,e \mid \texttt{if}\,E\,\texttt{then}\,e\,\texttt{else}\,e$ $\mid \texttt{spawn}\,E \mid \texttt{case}\,E\,\{L_i \Rightarrow e_i\}_{i\in I}$

Operational Semantics

$\texttt{if tt then}\,e_1\,\texttt{else}\,e_2$	$\hookrightarrow$	e_1			(RIft)
$\texttt{if ff then}\,e_1\,\texttt{else}\,e_2$	$\hookrightarrow$	e_2			(RIff)
$\texttt{let}\,x = v\,\texttt{in}\,e$	$\hookrightarrow$	$e[v/x]$			(RLet)
$(\texttt{fun}\,x \Rightarrow e)\,v$	$\hookrightarrow$	$e[v/x]$			(RApp)
$(\texttt{rec}\,f(x) \Rightarrow e)\,v$	$\hookrightarrow$	$e[\texttt{rec}\,f(x) \Rightarrow e/f][v/x]$			(RFix)
$E[e] \parallel S$	$\longrightarrow$	$E[e'] \parallel S$	if	$e \hookrightarrow e'$	(RBeta)
$E[\texttt{spawn}\,v] \parallel S$	$\longrightarrow$	$E[()] \parallel v\,() \parallel S$			(RSpn)
$E_1[\texttt{req-}c\;()] \parallel E_2[\texttt{acc-}c\;()] \parallel S$	$\longrightarrow$	$E_1[p] \parallel E_2[\overline{p}] \parallel S$	if	$p, \overline{p} \sharp E_1, E_2, S$	(RInit)
$E_1[\texttt{send}\;(p,v)] \parallel E_2[\texttt{recv}\,\overline{p}] \parallel S$	$\longrightarrow$	$E_1[()] \parallel E_2[v] \parallel S$			(RCom)
$E_1[\texttt{deleg}\;(p,p')] \parallel E_2[\texttt{resume}\,\overline{p}] \parallel S$	$\longrightarrow$	$E_1[()] \parallel E_2[p'] \parallel S$			(RDel)
$E_1[\texttt{sel-}L_j\,p] \parallel E_2[\texttt{case}\,\overline{p}\,\{L_i \Rightarrow e_i\}_{i\in I}] \parallel S$	$\longrightarrow$	$E_1[()] \parallel E_2[e_j] \parallel S$	if	$j \in I$	(RSel)

Fig. 6.1: $\mathrm{ML_S}$ syntax and operational semantics.

both generated by accepting a connection on `swp`. This can be encoded with an *internal choice*:

$$!\texttt{Fst}.\eta' \quad \oplus \quad !\texttt{Snd}.!\eta'.\texttt{end}$$

In the case where the coordinator chooses the first choice, the rest of the session η' over the endpoint is delegated, and therefore it can be any session—η' will be executed from the process that receives z_1 (running a *swap* function). If the coordinator selects the second choice then it simply delegates an endpoint with session type η'.

In our type system, if the coordinator is type-checked in isolation, then typing succeeds with any η'. However, if both coordinator and swap function are typed in the same program, typing succeeds only when $\eta' = \eta$ and $T_1 = T_2$. These equalities are necessary to guarantee that the two dual endpoints of `swp` have dual session types.

Our type inference algorithm is able to type this program and derive the above session types directly from the source code, with no programmer annotations.

6.3 The Language $\mathrm{ML_S}$

Here we introduce the untyped core of $\mathrm{ML_S}$. Its syntax and operational semantics are shown in Fig. 6.1. An $\mathrm{ML_S}$ expression can be one of the usual lambda expressions (value, variable, a pair constructor, let-binding, or conditional), or $\texttt{spawn}\,e$ which evaluates e to a function and asynchronously applies it to the unit value; it can also be $\texttt{case}\,e\,\{L_i \Rightarrow e_i\}_{i\in I}$ which, as we will see, implements finite *external choice*.

We use standard syntactic sugar for writing programs in $\mathrm{ML_S}$, such as $e_1; e_2$ instead of $\texttt{let}\,x =$

e_1 `in` e_2 when $x \notin \mathrm{fv}(e_2)$, infix operators, etc. A running *process* in $\mathrm{ML_S}$ is a closed expression and a running *system* S is a parallel composition of processes. We identify systems up to the reordering of parallel processes and the removal of terminated, unit-value processes. A single process with no active session endpoints is an $\mathrm{ML_S}$ *program*.

The values of $\mathrm{ML_S}$ contain the basic Unit, Bool, and Int constants and all standard integer and boolean operators. Values also include pairs (v, v'), and first class recursive ($\texttt{rec}\, f(x) \Rightarrow e$) and non-recursive ($\texttt{fun}\, x \Rightarrow e$) functions. Following the tradition of binary session types [Honda et al., 1998], communication between processes happens over dynamically generated entities called *sessions* which have exactly two *endpoints*. Thus, $\mathrm{ML_S}$ values contain a countably infinite set of endpoints, ranged over by p. We assume a total involution $(\overline{\cdot})$ over this set, with the property $\overline{p} \neq p$, which identifies *dual endpoints*. We write $o \sharp o'$ when the syntactic objects o and o' contain distinct endpoints.

The language is equipped with a small-step, call-by-value operational semantics. Fig. 6.1 shows the *redex* expressions that perform beta reductions $\hookrightarrow$. Systems take small-step transitions $\longrightarrow$ by decomposing a system into an evaluation context E with a beta redex in its hole (RBeta), or by the effectful transitions discussed below. Evaluation contexts include standard call-by-value contexts, but also the parallel system contexts $E \parallel S$ and $S \parallel E$. An RSpn reduction generates new processes containing a single application.

A process can request (or accept) to open a session by applying `req-`c (resp., `acc-`c) to the unit value, which returns the endpoint (resp., dual endpoint) of a new session. Here c ranges over an infinite set of global initialization names for sessions, called *channels*. To simplify presentation, global channels are not values; instead, $\mathrm{ML_S}$ has `req-`c and `acc-`c as values for each c.

Once two processes synchronize on a global channel and each receives a fresh, dual endpoint (RInit reduction), they can use the endpoints to exchange messages. Applying `send` to an endpoint and a value will send this value over the session, whereas applying `recv` to an endpoint will receive a value over the session. The synchronization of these two applications leads to a synchronous communication reduction (RCom).

A process can also offer a number of options to its dual with the construct $\texttt{case}\, e\, \{L_i \Rightarrow e_i\}_{i \in I}$, implementing, as mentioned earlier, a finite external choice. Here L ranges over a countably infinite set of choice labels, and I is a finite set of natural numbers. We assume a fixed enumeration of these labels, thus L_i denotes a unique label for each natural number i. When a process offers a choice of labels on session endpoint p, its dual can select one of those labels with the expression $\texttt{sel-}L_j\, \overline{p}$ (RSel).

The intuition of session types is that once a session is open, the processes controlling its endpoints are in charge of executing a predefined communication protocol. However, any of these processes may *delegate* this obligation to another process, by sending one endpoint p over another endpoint q, with the application $\texttt{deleg}\, q\, p$. A process with an endpoint $\overline{q}$ can receive p with the expression $\texttt{resume}\, q$, and continue executing the protocol over p (RDel). The example of the swap service in Example 6.2.2 used delegation to create a direct connection between processes that run the swap function.

Type Variables:	α
Behaviour Variables:	β
Region Variables:	ρ
Session Variables:	ψ
Type Schemas:	$TS ::= \forall(\vec{\alpha}\vec{\beta}\vec{\rho}\vec{\psi} : C).\, T$
Types:	$T ::= \mathsf{Unit} \mid \mathsf{Bool} \mid \mathsf{Int} \mid T \times T \mid T \xrightarrow{\beta} T \mid \mathsf{Ses}^{\rho} \mid \alpha$
Constraints:	$C ::= T \subseteq T \mid b \subseteq \beta \mid \rho \sim r \mid c \sim \eta \mid \overline{c} \sim \eta \mid C, C \mid \epsilon$
Type Envs:	$\Gamma ::= x : TS \mid \Gamma, \Gamma \mid \epsilon$
Regions:	$r ::= l \mid \rho$
Behaviours:	$b ::= \beta \mid \tau \mid b\,;b \mid b \oplus b \mid \mathtt{rec}_{\beta}\, b \mid \mathtt{spawn}\, b \mid \mathtt{push}(l : \eta)$ $\mid \mathtt{pop}\rho!T \mid \mathtt{pop}\rho?T \mid \mathtt{pop}\rho!\rho \mid \mathtt{pop}\rho?l \mid \mathtt{pop}\rho!L_i \mid \sum_{i \in I} \mathtt{pop}\rho?L_i\,;b_i$

Fig. 6.2: Syntax of types, behaviours, constraints, and session types.

6.4 Two-Level MLS Typing for Sessions

We give a type inference system for MLS organized in two levels. The first level is a type-and-effect system adding communication effects and endpoint regions to the Hindley-Milner type system. This is an extension of the type-and-effect system of Amtoft, Nielson and Nielson [Amtoft et al., 1999]. The second level imposes a session typing discipline to the communication effects of the first level.

6.4.1 First Level: Functional Types and Communication Effects

We use typing judgments of the form

$$C;\ \Gamma \vdash e : T \triangleright b$$

which assign to expression e the type T and behaviour b, under type environment Γ and *constraint environment* C. The constraint environment relates type-level variables to concrete terms and enables type inference. These components are defined in Fig. 6.2.

Static Endpoints. First we require that textual sources of session endpoints are annotated with *unique region labels* in a pre-processing step, updating MLS syntax as follows.

Values: $v ::= \ldots \mid p^{l} \mid \mathtt{req}\text{-}c^{l} \mid \mathtt{acc}\text{-}c^{l} \mid \mathtt{resume}^{l}$

With this extension, the system uses regions to statically approximate the endpoint that will be used at each communication at runtime, effectively creating one type Ses^{l} for each endpoint source.[1] The type-and-effect system uses *region variables* ρ to track the flow of labels at the type level. Dynamic endpoints generated at different source expressions are statically distinguished, but those generated from the same expression are identified, resulting to the rejection of some type safe programs (Sec. 6.4.2). This can be remedied with standard context-sensitive solutions such as k-CFA [Shivers, 1991].

Functional Types. An expression in MLS can have a base type Unit, Bool, or Int, a pair type $T \times T'$, a function type $T \xrightarrow{\beta} T'$, or a session endpoint type Ses^{ρ}. *Type variables* α are used for polymorphism (and type inference). Each function type is annotated with a *behaviour variable* β and each session

[1] Technically, endpoint types are annotated with session variables Ses^{ρ} which are related to endpoint labels through the constraint environment C.

endpoint type with a region variable ρ, respectively denoting the effect of the function body and the endpoint's textual source.

The types of session endpoints do not contain a session type because session types evolve during the execution of the program. In Example 6.2.1, the two uses of z in the body of *swap* refer to the same endpoint but at different states: at the first it can perform a send and then a receive, and at the second it can only perform the receive. Therefore, Ses^{ρ} only refers to the static identity of an endpoint through ρ, ignoring its session type.

Communication Effects. Inspired by Castagna et al. [Castagna et al., 2009], the behavioural effect of a $\mathrm{ML_S}$ expression can be thought of as describing operations on a *stack of session endpoints* Δ. This stack contains frames of the form $(l : \eta)$, where l is a static endpoint and η a session type (described in Sec. 6.4.2).

The expression can push a new frame on the stack ($\mathtt{push}(l : \eta)$), or reduce the top session type by performing an input ($\mathtt{pop}\rho?T$) or output ($\mathtt{pop}\rho!T$) of a value; a delegation ($\mathtt{pop}\rho!\rho$) or resumption ($\mathtt{pop}\rho?l$) of an endpoint; or an offer ($\sum_{i\in I} \mathtt{pop}\rho?L_i\,;b_i$) or selection ($\mathtt{pop}\rho!L_i$) of a choice. When the top session type of Δ is finished then it is popped from the stack. The application of $\mathtt{req}\text{-}c^{l}$ or $\mathtt{acc}\text{-}c^{l}$ has a $\mathtt{push}(l : \eta)$ effect; the application of `send`, `recv`, and `deleg`, has the corresponding effect with ρ calculated by the type of the first argument (Ses^{ρ}). Departing from [Amtoft et al., 1999], function `resume` is annotated with its own fresh label, instead of a variable which would be mapped through the constraint environment C to a single label. This allows the typing of programs where a `resume` statement can input endpoints with different labels.

Example 6.4.1. *Consider the following program P that spawns two clients, one proxy, and one server; P is typable in $\mathrm{ML_S}$, provided e_1 and e_2 are.*

$$
\begin{aligned}
\mathtt{let}\ cli_1 &= (\mathtt{fun}\ \_ \Rightarrow \mathtt{let}\ z_c = \mathtt{req}\text{-}\mathtt{c}^{l_1}\ ()\ \mathtt{in}\ e_1)\\
cli_2 &= (\mathtt{fun}\ \_ \Rightarrow \mathtt{let}\ z_c = \mathtt{req}\text{-}\mathtt{c}^{l_2}\ ()\ \mathtt{in}\ e_2)\\
prx &= (\mathtt{fun}\ \_ \Rightarrow \mathtt{let}\ z_c = \mathtt{acc}\text{-}\mathtt{c}^{l_c}\ ()\\
&\qquad\qquad\qquad\quad z_s = \mathtt{req}\text{-}\mathtt{s}^{l_s}\ ()\\
&\qquad\qquad\qquad \mathtt{in}\ \mathtt{deleg}\ z_s\ z_c)\\
srv &= (\mathtt{fun}\ \_ \Rightarrow \mathtt{let}\!*\ z_s = \mathtt{acc}\text{-}\mathtt{s}^{l_p}\ ()\\
&\qquad\qquad\qquad\quad x = \underline{\mathtt{resume}^{l}\ z_s}\\
&\qquad\qquad\qquad \mathtt{in}\ \mathtt{send}\ x\ \mathtt{1}; \mathtt{send}\ x\ \mathtt{tt})\\
\mathtt{in}\ \mathtt{spawn}\ &prx_1; \mathtt{spawn}\ prx_2; \mathtt{spawn}\ cli; \mathtt{spawn}\ srv;
\end{aligned}
$$

Both clients request a session on `c`*. The proxy accepts one of them and in turn requests a session with the server on* `s`*. Once the server accepts, the proxy delegates the client session over the server session. The server then sends two values to the connected client over the client session.*

In the absence of label l on the underlined resume construct of the server, the type system would calculate that both l_1 and l_2 endpoints can flow to x at runtime. Therefore, x will have type Ses^{ρ}, with ρ related to both l_1 and l_2 in the constraint environment, which violates the unique-label requirement mentioned earlier.

The rest of the behaviours follow the structure of the code: τ is the silent behaviour of pure computations; behaviour $b\,;b'$ allows sequencing, and $b \oplus b'$ internal choice. Behaviour $\mathtt{rec}_\beta\, b$ marks recursive behaviour. As we discussed $\mathrm{ML_S}$ does not have recursive session types but does allow recursive effectful functions, such as the coordinator in Example 6.2.1.

Constraints. Constraint sets C have *inclusion constraints* for types ($T \subseteq T'$) and behaviours ($b \subseteq \beta$), and equality constraints for regions ($\rho \sim r$). They also contain exactly two equality constraints ($c \sim \eta$ and $\overline{c} \sim \eta'$) per global channel c, one for the behaviour of the runtime endpoints returned by $\mathtt{acc}$-c and one for those returned by $\mathtt{req}$-c. These are the two *kinds of endpoints* of c.

The simple inclusion constraints for behaviours are sufficient for typing any functionally-typable program without the introduction of a sub-effecting relation [Talpin and Jouvelot, 1992]. Intuitively, $b \subseteq \beta$ means that β *may* behave as b. Although not strictly necessary here, type constraints are more general to enable principal typing in the presence of subtyping constraints $\mathsf{Int} \subseteq \mathtt{real}$ [Amtoft et al., 1999, §1.5.1], often appearing in the session-types literature. We write $C \vdash o \subseteq o'$ and $C \vdash o \sim o'$ for the reflexive and transitive closure of constraints in C; we say that these constraints are *derivable* from C. We write $C \vdash C'$ if all constraints of C' are derivable from C. We will work with well-formed constraints, satisfying the following conditions.

Definition 6.4.2 (Well-Formed Constraints)**.** *C is* well-formed *if:*

1. Type-Consistent: *for all type constructors tc_1, tc_2, if $(tc_1(\vec{t_1}) \subseteq tc_2(\vec{t_2})) \in C$, then $tc_1 = tc_2$, and for all $t_{1i} \in \vec{t_1}$ and $t_{2i} \in \vec{t_2}$, $(t_{1i} \subseteq t_{2i}) \in C$;*
2. Region-Consistent: *if $C \vdash l \sim l'$ then $l = l'$;*
3. Behaviour-Compact: *all cycles in behaviour constraints contain at least one $(\mathtt{rec}_\beta\, b \subseteq \beta) \in C$; also if $(\mathtt{rec}_\beta\, b \subseteq \beta') \in C$ then $\beta = \beta'$ and $\forall (b' \subseteq \beta) \in C$, $b' = \mathtt{rec}_\beta\, b$.*

The first condition disallows constraints such as $(\mathsf{Int} \subseteq T \times T')$ which lead to type errors, and deduces $(T_i \subseteq T_i')$ from $(T_1 \times T_2 \subseteq T_1' \times T_2')$. The second condition requires that only endpoints from a single source can flow in each ρ.

Example 6.4.3. *The following program requests two session endpoints, bound to x and y. It then binds one of these endpoints to z, depending on the value of e, and sends* `1` *over x and* `tt` *over z.*

$\mathtt{let*}\ (x, y) = (\mathtt{req\text{-}c}^{l_1}\ (), \mathtt{req\text{-}d}^{l_2}\ ())$

$\qquad z = \mathtt{if}\ e\ \mathtt{then}\ x\ \mathtt{else}\ y$

$\mathtt{in\ send}\ x\ \mathtt{1}; \mathtt{send}\ z\ \mathtt{tt}$

This program is not typable because communications on the `c-` *and* `d-`*endpoints depend on the value returned from e, which cannot be statically determined. In our framework, z will have type Ses^ρ and the constrain environment will contain $C \vdash \rho \sim l_1,\ \rho \sim l_2$. The program will be rejected by the second condition of Def. 6.4.2.*

In related work (e.g., [Gay and Hole, 2005]) such a program is rejected because of the use of substructural types for session endpoints.

The third condition of Def. 6.4.2 disallows recursive behaviours through the environment without the use of a $\mathtt{rec}_\beta\, b$ effect. The second part of the condition requires that there is at most one

recursive constraint in the environment using variable β. This condition is necessary to guarantee type preservation and the decidability of session typing.

Polymorphism. The type system extends Hindley-Milner polymorphism with type schemas TS of the form $\forall(\vec{\gamma} : C).\, T$, where γ ranges over any variable $\alpha, \beta, \rho, \psi$. Type environments Γ bind unique variable names to type schemas; we let $\forall(\emptyset).T = T$. Besides type ($\vec{\alpha}$), behaviour ($\vec{\beta}$), and region ($\vec{\rho}$) variables, type schemas also generalize *session variables* $\vec{\psi}$. A type schema contains a set C which imposes constraints on quantified variables. For TS to be *well-formed*, we must have $\mathrm{fv}(C) \subseteq \{\vec{\gamma}\}$. The polymorphic types of the constant $\mathrm{ML_S}$ functions are:

$$
\begin{array}{ll}
\texttt{req-}c^l & : \forall(\beta\rho\psi : \texttt{push}(l : \psi) \subseteq \beta, \rho \sim l, c \sim \psi).\, \mathsf{Unit} \xrightarrow{\beta} \mathsf{Ses}^{\rho} \\
\texttt{acc-}c^l & : \forall(\beta\rho\psi : \texttt{push}(l : \psi) \subseteq \beta, \rho \sim l, \overline{c} \sim \psi).\, \mathsf{Unit} \xrightarrow{\beta} \mathsf{Ses}^{\rho} \\
\texttt{send} & : \forall(\alpha\beta\rho : \texttt{pop}\rho!\alpha \subseteq \beta).\, \mathsf{Ses}^{\rho} \times \alpha \xrightarrow{\beta} \mathsf{Unit} \\
\texttt{recv} & : \forall(\alpha\beta\rho : \texttt{pop}\rho?\alpha \subseteq \beta).\, \mathsf{Ses}^{\rho} \xrightarrow{\beta} \alpha \\
\texttt{sel-}L & : \forall(\beta\rho : \texttt{pop}\rho?L \subseteq \beta).\, \mathsf{Ses}^{\rho} \xrightarrow{\beta} \mathsf{Unit} \\
\texttt{deleg} & : \forall(\beta\rho\rho' : \texttt{pop}\rho!\rho' \subseteq \beta).\, \mathsf{Ses}^{\rho} \times \mathsf{Ses}^{\rho'} \xrightarrow{\beta} \mathsf{Unit} \\
\texttt{resume}^l & : \forall(\beta\rho\rho' : \texttt{pop}\rho?\rho' \subseteq \beta,\ \rho' \sim l).\, \mathsf{Ses}^{\rho} \xrightarrow{\beta} \mathsf{Ses}^{\rho'}
\end{array}
$$

The effect of $\texttt{req-}c^l$ ($\texttt{acc-}c^l$) is to push a new static session endpoint l on the stack. The session type of the endpoint is a variable ψ, to be substituted with a concrete session type (or a fresh variable in the case of inference) at instantiation of the polymorphic type. This ψ has to be equal to the session type associated to the static endpoint c (resp., $\overline{c}$), expressed by the constraint $c \sim \psi$ (resp., $\overline{c} \sim \psi$). The return type of the function is Ses^{ρ}, where $\rho \sim l$. The types of the rest of the functions follow the same principles. The following definition allows the instantiation of a type schema under a global constraint environment C.

Definition 6.4.4 (Solvability)**.** $\forall(\vec{\gamma}{:}C_0).\, T$ *is* solvable *from* C *using substitution* σ *when* $dom(\sigma) \subseteq \{\vec{\gamma}\}$ *and* $C \vdash C_0\sigma$*. TS is solvable from* C *if it exists* σ *such that TS is solvable from* C *using* σ*.*

Typing Rules. The rules of our type-and-effect system are shown in Fig. 6.3. Most typing rules are standard—we discuss only those different from [Amtoft et al., 1999]. Rule TMatch types a case expression with an external-choice behaviour of the same number of branches. The choice labels L_i ($i \in I$) in the code determine those in the behaviour. The computation of each branch will return a value of the same type T but possibly have a different effect b_i. Rule TSub is for subtyping and sub-effecting. The latter can only replace behaviours with variables, avoiding a more complex relation.

Definition 6.4.5 (Functional Subtyping)**.** $C \vdash T <: T'$ *is the least reflexive, transitive, compatible relation on types with the axioms:*

$$
\frac{(T_1 \subseteq T_2) \in C}{C \vdash T_1 <: T_2} \qquad \frac{C \vdash \rho \sim \rho'}{C \vdash \mathsf{Ses}^{\rho} <: \mathsf{Ses}^{\rho'}}
$$

$$
\frac{C \vdash T_1' <: T_1 \quad C \vdash \beta \subseteq \beta' \quad C \vdash T_2 <: T_2'}{C \vdash T_1 \xrightarrow{\beta} T_2 <: T_1' \xrightarrow{\beta'} T_2'}
$$

TPair
$$\frac{C;\ \Gamma \vdash e_1 : T_1 \triangleright b_1 \quad C;\ \Gamma \vdash e_2 : T_2 \triangleright b_2}{C;\ \Gamma \vdash (e_1, e_2) : T_1 \times T_2 \triangleright b_1 ; b_2}$$

TLet
$$\frac{C;\ \Gamma \vdash e_1 : TS \triangleright b_1 \quad C;\ \Gamma, x : TS \vdash e_2 : T \triangleright b_2}{C;\ \Gamma \vdash \mathtt{let}\, x = e_1\ \mathtt{in}\ e_2 : T \triangleright b_1 ; b_2}$$

TApp
$$\frac{C;\ \Gamma \vdash e_1 : T' \xrightarrow{\beta} T \triangleright b_1 \quad C;\ \Gamma \vdash e_2 : T' \triangleright b_2}{C;\ \Gamma \vdash e_1\, e_2 : T \triangleright b_1 ; b_2 ; \beta}$$

TIf
$$\frac{C;\ \Gamma \vdash e_1 : \mathsf{Bool} \triangleright b_1 \quad C;\ \Gamma \vdash e_i : T \triangleright b_i \ \ (i \in \{1,2\})}{C;\ \Gamma \vdash \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3 : T \triangleright b_1 ; (b_2 \oplus b_3)}$$

TMatch
$$\frac{C;\ \Gamma \vdash e : \mathsf{Ses}^{\rho} \triangleright b \quad C;\ \Gamma \vdash e_i : T \triangleright b_i \ \ (i \in I)}{C;\ \Gamma \vdash \mathtt{case}\, e\, \{L_i \Rightarrow e_i\}_{i \in I} : T \triangleright b ; \sum_{i \in I} \mathsf{pop}\rho?L_i ; b_i}$$

TRec
$$\frac{C;\ \mathit{confined}_C(\Gamma), f : T \xrightarrow{\beta} T', x : T \vdash e : T' \triangleright b}{C;\ \Gamma \vdash \mathtt{rec}\, f(x) \Rightarrow e : T \xrightarrow{\beta} T' \triangleright \tau} \quad \begin{array}{l} C \vdash \mathit{confined}(T, T') \\ C \vdash \mathtt{rec}_{\beta}\, b \subseteq \beta \end{array}$$

TIns
$$\frac{C;\ \Gamma \vdash e : \forall(\vec{\gamma} : C_0).T \triangleright b}{C;\ \Gamma \vdash e : T\sigma \triangleright b} \quad \begin{array}{l} \mathit{dom}(\sigma) \subseteq \{\vec{\gamma}\} \\ \forall(\vec{\gamma} : C_0).\, T \text{ is solvable from } C \text{ by } \sigma \end{array}$$

TGen
$$\frac{C \cup C_0;\ \Gamma \vdash e : T \triangleright b}{C;\ \Gamma \vdash e : \forall(\vec{\gamma} : C_0).T \triangleright b} \quad \begin{array}{l} \{\vec{\gamma}\} \cap \mathrm{fv}(\Gamma, C, b) = \emptyset \\ \forall(\vec{\gamma} : C_0).T \text{ is WF, solvable from } C \end{array}$$

TVar
$$\frac{}{C;\ \Gamma \vdash x : \Gamma(x) \triangleright \tau}$$

TConst
$$\frac{}{C;\ \Gamma \vdash k : \mathit{typeof}(k) \triangleright \tau}$$

TFun
$$\frac{C;\ \Gamma, x : T \vdash e : T' \triangleright \beta}{C;\ \Gamma \vdash \mathtt{fun}\, x \Rightarrow e : T \xrightarrow{\beta} T' \triangleright \tau}$$

TEndp
$$\frac{}{C;\ \Gamma \vdash p^{l} : \mathsf{Ses}^{\rho} \triangleright \tau} \quad C \vdash \rho \sim l$$

TSub
$$\frac{C;\ \Gamma \vdash e : T \triangleright b}{C;\ \Gamma \vdash e : T' \triangleright \beta} \quad \begin{array}{l} C \vdash T <: T' \\ C \vdash b \subseteq \beta \end{array}$$

TSpawn
$$\frac{C;\ \mathit{confined}_C(\Gamma) \vdash e : \mathsf{Unit} \xrightarrow{\beta} \mathsf{Unit} \triangleright b}{C;\ \Gamma \vdash \mathtt{spawn}\, e : \mathsf{Unit} \triangleright b ; \mathtt{spawn}\, \beta}$$

Fig. 6.3: Type-and-Effect System for Expressions

An important rule in our system is that for recursive functions (TRec). Since we only consider finite session types, the communication effect of the body of a recursive function should be *confined*. This captures two interrelated conditions:

- The recursive function does not use previously opened endpoints or non-confined functions from the type environment, nor it returns any endpoint or non-confined function.
- The communication effect of applying the function is only on endpoints that the function opens internally, and the session type of these endpoints will be followed to completion (or delegated) before the function returns.

These two conditions guarantee that, no matter how many times the function recurs, it leaves the environment's sessions unaffected, even if the body of the function has non-trivial communication effects. The first condition is enforced in TRec by the side-condition $C \vdash \mathit{confined}(T, T')$, for the argument and return types of the function, and the typing of the function's body under $\mathit{confined}_C(\Gamma)$, the confined part of Γ.

Definition 6.4.6 (Confined Types)**.** *$C \vdash \mathit{confined}(o)$ is the least compatible relation on type schemas,*

types, and behaviours that admits the following axioms.

$$\frac{T \in \{\mathsf{Int}, \mathsf{Bool}, \mathsf{Unit}\}}{C \vdash \mathit{confined}(T)} \qquad \frac{b \in \{\tau,\ \mathtt{rec}_\beta\, b'\}}{C \vdash \mathit{confined}(b)}$$

$$\frac{\forall (T \subseteq \alpha) \in C.\ C \vdash \mathit{confined}(T) \quad \forall (\alpha \subseteq T) \in C.\ C \vdash \mathit{confined}(T)}{C \vdash \mathit{confined}(\alpha)}$$

$$\frac{C \vdash \mathit{confined}(T, \vec{T}_1, \vec{T}_2, \vec{b})}{C \vdash \mathit{confined}(\forall(\vec{\gamma} : (\vec{T}_1 \subseteq \vec{T}_2), (\vec{b} \subseteq \vec{\beta}), (\vec{o}_1 \sim \vec{o}_2)).T)}$$

$$\frac{C \vdash \mathit{confined}(T, T') \quad C \vdash \mathit{confined}(\beta)}{C \vdash \mathit{confined}(T \xrightarrow{\beta} T')}$$

$$\frac{\forall (b \subseteq \beta) \in C.\ C \vdash \mathit{confined}(b)}{C \vdash \mathit{confined}(\beta)}$$

Moreover, $\mathit{confined}_C(\Gamma)$ is the largest subset of Γ such that for all $(x : TS) \in \mathit{confined}_C(\Gamma)$ we have $C \vdash \mathit{confined}(TS)$.

The above definition does not enforce the second condition for confinement. This is done in the session typing discipline of the second level of our system (Sec. 6.4.2), and in fact only for those recursive functions that are applied and whose behaviour is part of the communication effect of the program. This separation between the two levels simplifies type inference.

Similarly to recursive functions, a spawned function must be confined. This is again enforced in part by the side-condition of Rule TSpawn and the rule for spawn in our session type discipline.

6.4.2 Second Level: Session Types

The type-and-effect system we presented so far is parametric to session type annotations in the constraint environment (constraints $c \sim \eta$ and $\bar{c} \sim \eta$). Indeed, our adaptation of the type inference algorithm for the type-and-effect system [Amtoft et al., 1999] will only produce *variables* for session type annotations ($c \sim \psi$ and $\bar{c} \sim \psi$). It is the job of the session typing in the second level of our system to check (and, in the case of inference, to deduce) session types for the endpoints used in the program.

As we discussed previously and showed in the examples of Sec. 6.2, $\mathrm{ML_S}$ session types are finite but allow useful recursive expressions with a communication effect. In Sec. 6.5 we show an extension of the type system with recursive session types. Session types are higher-order to allow for delegation of endpoints. Their syntax is:

$$\eta ::= \mathsf{end} \mid !T.\eta \mid ?T.\eta \mid !\eta.\eta \mid ?\eta.\eta \mid \bigoplus_{i \in I} !L_i.\eta_i \mid \sum_{i \in (I_1, I_2)} ?L_i.\eta_i \mid \psi$$

Session types describe the sequence of communications over a session endpoint. A session type is

finished (end) or it can describe further interactions: the input or output of a *confined* value (resp., $?T.\eta$ or $!T.\eta$), or the delegation or resumption of an endpoint of session type η' (resp., $!\eta'.\eta$ or $?\eta'.\eta$). It may also describe the ability of the process to choose a communication label L_i from a number of labels I ($\bigoplus_{i\in I}!L_i.\eta_i$), signifying to its partner that session type η_i is to be followed from that point on.

Moreover, a session type can describe the ability of the process to offer to its dual an external choice $\sum_{i\in(I_1,I_2)}?L_i.\eta_i$. Here I_1 contains the labels that the process *must* be able to accept and I_2 the labels that it *may* accept. We require that I_1 and I_2 are disjoint and I_1 is not empty. Hence, session types give a lower (I_1) and an upper ($I_1 \cup I_2$) bound of the labels in external choices. These two sets of labels are not necessary for typing external choice—we could use only the first set. However, the two sets of labels make type inference deterministic and independent of source code order, thus simpler to implement. It also makes typing more efficient, modular and intuitive.

Example 6.4.7. *Consider a program $P[e_1][e_2]$ containing the expressions:*

$$e_1 \stackrel{def}{=} \texttt{let}\, x = \texttt{acc-}c^{l_1}\, ()\, \texttt{in}\, \texttt{case}\, x\, \{L_1 \Rightarrow e,\ L_2 \Rightarrow e^\star\}$$
$$e_2 \stackrel{def}{=} \texttt{let}\, x = \texttt{req-}c^{l_2}\, ()\, \texttt{in}\, \texttt{sel-}L_2\, x$$

Suppose $e^\star$ contains a type error, possibly because of a mismatch in session types with another part of P. If a type inference algorithm run on $P[e_1][e_2]$ first examines e_1, it will explore both branches of the choice, tentatively constructing the session type $\Sigma\{L_1.\eta_1,\ L_2.\eta_2\}$, finding the error in $e^\star$. One strategy might then be to backtrack from typing $e^\star$ (and discard any information learned in the L_2 branch of this and possibly other choices in the code) and continue with the session type $\Sigma\{L_1.\eta_1\}$. However, once e_2 is encountered, the previous error in $e^\star$ should be reported. A programmer, after successfully type checking $P[e_1][()]$, will be surprised to discover a type error in e_1 after adding in e_2. The type-and-effect system here avoids such situations by typing all choice branches, even if they are inactive*, at the expense of rejecting some—rather contrived—programs. A similar approach is followed in the type-and-effect system of the previous section by requiring* all *branches to have the same type (Rule* TMatch *in Fig. 6.3).*

We express our session typing discipline as an *abstract interpretation semantics* for behaviours shown in Fig. 6.4, which conservatively approximates the communication effect of expressions at runtime. It describes transitions of the form $\Delta \vDash b \rightarrow_C \Delta' \vDash b'$, where b, b' are communication effects. The Δ and Δ' are *linear stacks* on which static endpoint labels together with their corresponding session types ($l : \eta$) can be pushed and popped. Inspired by Castagna et al. [Castagna et al., 2009], in the transition $\Delta \vDash b \rightarrow_C \Delta' \vDash b'$, the behaviour b can only use the top label in the stack to communicate, push another label on the stack, or pop the top label from the stack provided its session type is end. This stack principle gives us a weak deadlock freedom property (Theorem 6.4.16).

We will treat these stacks linearly, in the sense that a label can be pushed onto a stack only if it has not been previously pushed on (and possibly popped from) that stack. Therefore, every stack Δ contains an implicit set of the labels that have been pushed onto it, which is accessed by $\Delta.\mathsf{labels}$. This requirement guarantees that the following unsafe program is rejected.

End	$(l : \mathsf{end}) \cdot \Delta \vDash b \to_C \Delta \vDash b$	
Beta	$\Delta \vDash \beta \to_C \Delta \vDash b$	if $C \vdash b \subseteq \beta$
Plus	$\Delta \vDash b_1 \oplus b_2 \to_C \Delta \vDash b_i$	if $i \in \{1, 2\}$
Push	$\Delta \vDash \mathtt{push}(l : \eta) \to_C (l : \eta) \cdot \Delta \vDash \tau$	if $l \sharp \Delta.\mathsf{labels}$
Out	$(l : !T.\eta) \cdot \Delta \vDash \mathtt{pop}\rho!T' \to_C (l : \eta) \cdot \Delta \vDash \tau$	if $C \vdash \rho \sim l,\ \mathit{confined}_C(T'),\ T' <: T$
In	$(l : ?T.\eta) \cdot \Delta \vDash \mathtt{pop}\rho?T' \to_C (l : \eta) \cdot \Delta \vDash \tau$	if $C \vdash \rho \sim l,\ \mathit{confined}_C(T'),\ T <: T'$
Del	$(l : !\eta_d.\eta) \cdot (l_d : \eta'_d) \cdot \Delta \vDash \mathtt{pop}\rho!\rho_d \to_C (l : \eta) \cdot \Delta \vDash \tau$	if $C \vdash \rho \sim l,\ \rho_d \sim l_d,\ \eta'_d <: \eta_d$
Res	$(l : ?\eta_r.\eta) \vDash \mathtt{pop}\rho?l_r \to_C (l : \eta) \cdot (l_r : \eta_r) \vDash \tau$	if $(l \neq l_r),\ C \vdash \rho \sim l$
ICh	$(l : \bigoplus_{i \in I} !L_i.\eta_i) \cdot \Delta \vDash \mathtt{pop}\rho!L_j \to_C (l : \eta_j) \cdot \Delta \vDash \tau$	if $(j \in I),\ C \vdash \rho \sim l$
ECh	$(l : \sum_{i \in (I_1, I_2)} ?L_i.\eta_i) \cdot \Delta \vDash \sum_{j \in J} \mathtt{pop}\rho?L_j\,; b_j \to_C (l : \eta_k) \cdot \Delta \vDash b_k$	if $k \in J,\ C \vdash \rho \sim l,$ $I_1 \subseteq J \subseteq I_1 \cup I_2$
Rec	$\Delta \vDash \mathtt{rec}_\beta\, b \to_C \Delta \vDash \tau$	if $\epsilon \vDash b \Downarrow_{C'}$, $C' = (C \backslash (\mathtt{rec}_\beta\, b \subseteq \beta)) \cup (\tau \subseteq \beta)$
Spn	$\Delta \vDash \mathtt{spawn}\, b \to_C \Delta \vDash \tau$	if $\epsilon \vDash b \Downarrow_C$
Seq	$\Delta \vDash b_1; b_2 \to_C \Delta' \vDash b'_1; b_2$	if $\Delta \vDash b_1 \to_C \Delta' \vDash b'_1$
Tau	$\Delta \vDash \tau\,; b \to_C \Delta \vDash b$	

Fig. 6.4: Abstract Interpretation Semantics.

Example 6.4.8. *Consider the channel c whose session type is simply $\eta = !\mathsf{Int}.\mathsf{end}$ and the program:*

$$
\begin{aligned}
&\mathtt{let}\ f = \mathtt{fun}\ \_ \Rightarrow (\mathtt{req}\text{-}c^l\ ())\ \mathtt{in}\\
&\mathtt{let}\ x = f\ ()\ \mathtt{in}\\
&\mathtt{send}\ x\ \mathtt{1};\\
&\mathtt{let}\ y = f\ ()\ \mathtt{in}\\
&\mathtt{send}\ x\ \mathtt{1}
\end{aligned}
$$

The program does not obey session η, because it sends two integers on x, and none on y. According to the type-and-effect system of the previous section, x and y have type Ses^ρ, with $\rho \sim l$ in a constraint environment C. Moreover f has type $\mathsf{Unit} \xrightarrow{\beta} \mathsf{Unit}$ with

$$(\mathtt{push}(l : \eta); \mathtt{pop}\rho!\mathsf{Int}; \mathtt{push}(l : \eta); \mathtt{pop}\rho!\mathsf{Int}; \tau \subseteq \beta) \in C$$

Because region analysis identifies endpoints generated from the same source code location, the above behaviour does not differentiate between the labels in the second push and the final output. Therefore, the only reason to reject this behaviour is because it pushes the same location on the stack twice.

The linearity of the stacks means that the program in the above example is rejected, even if the second `send` x `1` is replaced with `send` y `1`, which would fix the problem in the code. We can remedy this with standard extensions from static analysis of [Shivers, 1991]. Note that the correct version of the above program is not typable under a more straightforward substructural session type discipline (e.g., [Vasconcelos et al., 2006]) because the function f would have a linear type and therefore it would not be possible to apply it twice.

Rule End from Fig. 6.4 simply removes a finished stack frame, and rule Beta looks up behaviour

variables in C; PLUS chooses one of the branches of non-deterministic behaviour. The PUSH rule extends the stack by adding one more frame to it, as long as the label has not been added before on the stack; this requirement will reject the program in Ex. 6.4.8. Rules OUT and IN reduce the top-level session type of the stack by an output and input, respectively. The requirement here is that the labels in the stack and the behaviour match, the usual subtyping [Gay and Hole, 2005] holds for the communicated types, and that the communicated types are *confined*. Note that sending confined (recursive) functions does not require delegation of endpoints because the typing rule TREC of Fig. 6.3 forbids open endpoints from the context to end up in these functions.

Transfer of endpoints is done by delegate and resume (rules DEL and RES). Delegate sends the second endpoint in the stack over the first; resume mimics this by adding a new endpoint label in the second position in the stack. Resume requires a one-frame stack to guarantee that the two endpoints of the same session do not end up in the same stack [Castagna et al., 2009], causing a deadlock. If we abandon the weak deadlock-freedom property guaranteed by our type system, then the conditions in RES can be relaxed and allow more than one frame.

A behaviour reduces an internal choice session type by selecting one of its labels (ICH). A behaviour offering an external choice is reduced non-deterministically to any of its branches. The behaviour must offer all *active* choices ($I_1 \subseteq J$) and all behaviour branches must be typable by the session type ($J \subseteq I_1 \cup I_2$).

Our session type discipline requires that behaviours follow to completion or delegate all ($l : \eta$) frames in a stack.

Definition 6.4.9. $\Delta \vDash b \Downarrow_C \vec{\Delta}'$ *when for all* b', Δ' *such that* $\Delta \vDash b \rightarrow_C^* \Delta' \vDash b' \not\rightarrow_C$ *we have* $b' = \tau$ *and* $\Delta' \in \{\vec{\Delta}'\}$. *We write* $\Delta \vDash b \Downarrow_C$ *when* $\Delta \vDash b \Downarrow_C \epsilon$, *where* ϵ *is the empty stack.*

Because of finite session stacks and session types, behaviour-compact constraint environments, and because of the rule for recursive behaviour explained below, there are no infinite sequences of reductions over this semantics. Therefore $\Delta \vDash b \Downarrow_C \vec{\Delta}'$ is decidable.

Note that according to the rules of Fig. 6.4, in order for $\Delta \vDash b \Downarrow_C$ to be true, session variables ψ can only appear as delegated types not used in the reductions. For typable programs, our session type inference algorithm can indeed infer session types that will satisfy this requirement (Sec. 6.4.3).

As we explained in the previous section, recursive functions in $\mathrm{ML_S}$ must be confined. In part this means that the communication effect of applying the function is only on endpoints that the function opens internally, and the session type of these endpoints will be followed to completion (or delegated) before the function returns. This is enforced in Rule REC, where $\mathtt{rec}_\beta\, b$ must have no net effect on the stack, guaranteed by $\epsilon \vDash b \Downarrow_{C'}$. Here $C' = (C \backslash (\mathtt{rec}_\beta\, b \subseteq \beta)) \cup (\tau \subseteq \beta)$ is the original C with constraint ($\mathtt{rec}_\beta\, b \subseteq \beta$) replaced by ($\tau \subseteq \beta$) (cf., Def. 6.4.2).

Similarly to recursive functions, a spawn will create a new, confined process. This is guaranteed by Rule TSPAWN in Fig. 6.3, and Rule SPN here which requires that the effect b of the spawned process satisfies $\Delta \vDash b \Downarrow_C$.

Type annotations The rule for recursive behaviour gives a method to safely bypass the linearity principle of the stacks, which requires that each label is pushed on the stack at most once, and type

more programs. Because of this restriction, the *swap* method of Ex. 6.2.1 can only be applied once in the body of the let.

To explain this let us consider the type schema of the *swap* function: $\forall(\alpha_1\alpha_2\beta : C_0)\alpha_1 \xrightarrow{\beta} \alpha_2$, where

$$C = (\mathtt{push}(l : \eta); \mathtt{pop}\rho!\alpha_1; \mathtt{pop}\rho?\alpha_2; \tau \subseteq \beta), (\rho \sim l), (\alpha_1 \subseteq \alpha_2)$$

The expression

$$\mathit{swap}\,(); \mathit{swap}\,()$$

which applies *swap* twice at type $\mathsf{Unit} \xrightarrow{\beta} \mathsf{Unit}$ will have a behaviour (omitting some τs):

$$\begin{aligned}&\mathtt{push}(l : \eta); \mathtt{pop}\rho!\mathsf{Unit}; \mathtt{pop}\rho?\mathsf{Unit}; \tau;\\ &\mathtt{push}(l : \eta); \mathtt{pop}\rho!\mathsf{Unit}; \mathtt{pop}\rho?\mathsf{Unit}; \tau\end{aligned}$$

This behaviour clearly violates stack linearity because it pushes l twice on the stack, and therefore the expression is not typable in our system.

However, the communication effect of *swap* is confined, and therefore we can define the function using the `rec` construct instead of `fun`. In this case, the type schema of *swap* will contain the constraint set

$$C = (\mathtt{rec}_\beta\,(\mathtt{push}(l : \eta); \mathtt{pop}\rho!\alpha_1; \mathtt{pop}\rho?\alpha_2; \tau) \subseteq \beta), (\rho \sim l)$$

and the above expression will have behaviour:

$$\begin{aligned}b = \mathtt{rec}_\beta\,(\mathtt{push}(l : \eta); \mathtt{pop}\rho!\mathsf{Unit}; \mathtt{pop}\rho?\mathsf{Unit}; \tau);\\ \mathtt{rec}_\beta\,(\mathtt{push}(l : \eta); \mathtt{pop}\rho!\mathsf{Unit}; \mathtt{pop}\rho?\mathsf{Unit}; \tau)\end{aligned}$$

It is easy to verify that this behaviour satisfies $\epsilon \vDash b \Downarrow_C$.

Therefore the `rec` construct can be used by programmers as an annotation to mark the functions that are confined and therefore it is safe to apply multiple times in programs.

Endpoint Duality So far our type discipline does not check session types for duality. The program

$$\begin{aligned}&\mathtt{spawn}\,(\mathtt{fun}\ \_ \Rightarrow \mathtt{let}\ x = \mathtt{req}\text{-}c\,()\ \mathtt{in}\ \mathtt{send}\ x\ \mathtt{tt});\\ &\mathtt{let}\ x = \mathtt{acc}\text{-}c\,()\ \mathtt{in}\ \mathtt{recv}\ x + 1\end{aligned}$$

should not be typable because its processes use dual endpoints at incompatible session types. Therefore we require that dual session endpoints ($c \sim \eta$ and $\overline{c} \sim \eta'$) have dual session types.

Definition 6.4.10 (Valid Constraint Environment)**.** *C is a* valid *constraint environment if there exists a substitution σ of variables ψ with closed session types, such that $C\sigma$ is well-formed and for all $\forall(c \sim \eta), (\overline{c} \sim \eta') \in C\sigma$ we have $C \vdash \eta \bowtie \eta'$.*

Definition 6.4.11 (Duality)**.** *$C \vdash \eta \bowtie \eta'$ if the following rules and their symmetric ones are satisfied.*

$$\frac{}{C \vdash \mathsf{end} \bowtie \mathsf{end}} \qquad \frac{C \vdash T <: T' \quad C \vdash \eta \bowtie \eta'}{C \vdash !T.\eta \bowtie ?T'.\eta'} \qquad \frac{C \vdash \eta_0 <: \eta_0' \quad C \vdash \eta \bowtie \eta'}{C \vdash !\eta_0.\eta \bowtie ?\eta_0'.\eta'} \qquad \frac{\forall i \in I_0.\ \ C \vdash \eta_i \bowtie \eta_i'}{C \vdash \bigoplus_{i \in I_0} !L_i.\eta_i \bowtie \sum_{i \in (I_0 I_1, I_2)} ?L_i.\eta_i')}$$

where $C \vdash \eta <: \eta'$ is [Gay and Hole, 2005] subtyping, with C needed for inner uses of $C \vdash T <: T'$, extended to our form of external choice, where $C \vdash \sum_{i \in (I_1, I_2)} ?L_i.\eta_i' <: \sum_{i \in (J_2, J_2)} ?L_i.\eta_i'$ when $I_1 \subseteq J_1$ and $J_1 \cup J_2 \subseteq I_1 \cup I_2$ and $\forall (i \in J_1 \cup J_2).\ C \vdash \eta_i <: \eta_i'$.

6.4.3 Combining the Two Levels

We are interested in typing source-level programs which contain a single process e with no open endpoints. However, to prove type soundness, we need to type running systems containing multiple running processes and open endpoints. For each running process we need to maintain a stack Δ, containing the session types of the endpoints opened by the process.

Definition 6.4.12 (Typing). *We write $C \Vdash \overrightarrow{(\Delta \vDash b,\ e)}$ if C is well-formed and valid, (C; $\emptyset \vdash \overrightarrow{e : T \triangleright b}$), and ($\overrightarrow{\Delta \vDash b} \Downarrow_C$), for some type T.*

Program e is well-typed if $C \Vdash (\epsilon \vDash b,\ e)$ for some C and b.

Type soundness in our system guarantees a weak deadlock freedom property of typed programs, which we discuss here. The key property is *well-stackedness*, the fact that in a running system, there is always a way to repeatedly remove dual endpoints with dual session types from the top of two stacks, until all stacks are empty. Note that this does not mean that programs are deterministic. Multiple pairs of dual endpoints can be at the top of a set of stacks at any time.

We let $\mathcal{S}$ range over $\widetilde{(\Delta \vDash b,\ e)}$, identify $\mathcal{S}$ up to reordering, and write S for $\widetilde{e}$ when $\mathcal{S} = \widetilde{(\Delta \vDash b,\ e)}$. In this section, in addition to labels, we also store the corresponding endpoints in Δ stacks (and trivially lift $\Downarrow_C$ to such stacks).

Definition 6.4.13 (Well-stackedness). *$C \Vdash_{ws} \mathcal{S}$ is the least relation satisfying the rules:*

$$\frac{}{C \Vdash_{ws} \epsilon}$$

$$\frac{C \Vdash_{ws} \mathcal{S},\ (\Delta \vDash b,\ e),\ (\Delta' \vDash b',\ e') \qquad C \vdash \eta \bowtie \eta' \qquad p, \overline{p} \sharp \Delta, \Delta', \mathcal{S}}{C \Vdash_{ws} \mathcal{S}, (p^l : \eta) \cdot \Delta \vDash b,\ e), ((\overline{p}^{l'} : \eta') \cdot \Delta' \vDash b',\ e')}$$

Theorem 6.4.14 (Type Preservation). *Suppose $C \Vdash \mathcal{S}$ and $C \Vdash_{ws} \mathcal{S}$. If $S \longrightarrow \widetilde{e'}$, then there exist $\widetilde{\Delta'}, \widetilde{b'}$ such that $\mathcal{S}' = \widetilde{(\Delta' \vDash b',\ e')}$ and $C \Vdash \mathcal{S}'$ and $C \Vdash_{ws} \mathcal{S}'$.*

Proof. See Appendix A. □

Type soundness is more technical. We divide system transitions to communication transitions between processes ($\longrightarrow_{\mathsf{c}}$) and internal transitions ($\longrightarrow_{\mathsf{i}}$). Let $\mathcal{S} \longrightarrow_{\mathsf{c}} \mathcal{S}'$ ($\mathcal{S} \longrightarrow_{\mathsf{i}} \mathcal{S}'$) when $S \longrightarrow S'$, derived by Rule RInit, RCom, RDel or RSel of Fig. 6.1 (resp., any other rule); $S \Longrightarrow_{\mathsf{c}} S'$ when $S \longrightarrow_{\mathsf{i}}^* \longrightarrow_{\mathsf{c}} \longrightarrow_{\mathsf{i}}^* S'$.

We also define dependencies between processes of a running system according to the following definition.

Definition 6.4.15 (Dependencies)**.** *Let* $P = \big(\Delta \vDash b,\ e\big)$ *and* $Q = \big(\Delta' \vDash b',\ e'\big)$ *be processes in* $\mathcal{S}$*.*

P **and** Q **are *ready*** $(P \leftharpoonup Q)$**:** *if* $\Delta = (p^l : \eta) \cdot \Delta_0$ *and* $\Delta' = (\overline{p}^{l'} : \eta') \cdot \Delta'_0$*;*

P **is *waiting* on** Q $(P \mapsto Q)$**:** *if* $\Delta = (p^l : \eta) \cdot \Delta_0$ *and* $\Delta' = \Delta'_1 \cdot (\overline{p}^{l'} : \eta') \cdot \Delta'_0$ *and* $\Delta'_1 \neq \epsilon$*;*

P ***depends*** **on** Q, R $(P \Mapsto (Q, R))$**:** *if* $P = Q \leftharpoonup R$*, or* $P \mapsto^+ Q \leftharpoonup R$*.*

The following type soundness theorem describes a system that cannot take any communication transitions.

Theorem 6.4.16 (Type Soundness)**.** *Let* $C \Vdash \mathcal{S}$ *and* $C \Vdash_{ws} \mathcal{S}$*. Then*

1. $\mathcal{S} \Longrightarrow_{\mathsf{c}} \mathcal{S}'$*, or*
2. $\mathcal{S} \longrightarrow_{\mathsf{i}}^* (\mathcal{F}, \mathcal{D}, \mathcal{W}, \mathcal{B})$ *such that:*

   Processes in $\mathcal{F}$ are finished: $\forall\big(\Delta \vDash b,\ e\big) \in \mathcal{F}.\ \Delta = \epsilon, b = \tau$ *and* $e = v$*.*

   Processes in $\mathcal{D}$ diverge: $\forall\big(\Delta \vDash b,\ e\big) \in \mathcal{D}.\ \big(\Delta \vDash b,\ e\big) \longrightarrow_{\mathsf{i}}^{\infty}$*.*

   Processes in $\mathcal{W}$ wait on channels: $\forall\big(\Delta \vDash b,\ e\big) \in \mathcal{W}.\ e = E[\texttt{req-}c^l]$ *or* $e = E[\texttt{acc-}c^l]$*.*

   Processes in $\mathcal{B}$ block on sessions: $\forall P = \big(\Delta \vDash b,\ e\big) \in \mathcal{B}.\ e = E[e_0]$ *and* e_0 *is* $\texttt{send}\ v$*,* $\texttt{recv}\ v$*,* $\texttt{deleg}\ v$*,* $\texttt{resume}\ v$*,* $\texttt{sel-}L\ v$*, or* $\texttt{case}\ v\ \{L_i \Rightarrow e_i\}_{i \in I}$ *and*

$$\exists Q \in (\mathcal{D}, \mathcal{W}).\ \exists R \in (\mathcal{D}, \mathcal{W}, \mathcal{B}).\ \mathcal{S} \vdash P \Mapsto (Q, R)$$

Proof. See Appendix A. □

The theorem needs to take into account several possibilities. $\mathrm{ML_S}$ has general recursion and therefore processes may diverge (set $\mathcal{D}$). Moreover, sessions open and close dynamically, thus processes may not find a communication partner when they are trying to open a new session (set $\mathcal{W}$). Processes may also block waiting to communicate on a session endpoint (set $\mathcal{B}$). The above theorem says that in systems where no more communication steps are possible, we will always find a diverging or waiting process if we follow the dependencies of a blocked processes.

The intuitive consequence of type soundness is that in the absence of divergence and in the presence of enough communication partners to start new sessions, there are no blocked processes, and when communications are no longer possible all processes have reduced to a value.

Corollary 6.4.17. *Let* $C \Vdash S$ *and* $C \Vdash_{ws} S$*. If* $\mathcal{S} \not\Longrightarrow_{\mathsf{c}}$ *and* $\mathcal{S} \longrightarrow_{\mathsf{i}}^* (\mathcal{F}, \emptyset, \emptyset, \mathcal{B})$*, then* $\mathcal{B} = \emptyset$*.*

6.5 Extension to Recursive Session Types

So far we developed our type system and inference algorithm considering only finite session types. Here we sketch the extension of our work to a form of recursive types. We first extend the syntax of expressions and behaviours with new recursive constructs, and add a form of recursive session types.

$$\begin{aligned} v &::= \ldots \mid \texttt{recses}\, f(x) \Rightarrow e \\ b &::= \ldots \mid \texttt{rec}^{\rho}_{\beta}\, b \\ \eta &::= \ldots \mid \mu X.\eta \mid X \end{aligned}$$

The new construct for recursive functions distinguishes these functions from confined recursive functions, which should be treated differently in the type system. The recursive behaviour $\mathtt{rec}^{\rho}_{\beta}\, b$ is a new behaviour construct which records the static identity ρ of an endpoint with a recursive type used in the body b of the behaviour. A recursive session type $\mu X.\eta$ is well-formed only when it is *guarded*; i.e., X appears in η under prefixes containing choice labels. Moreover, X cannot be delegated in η. These restrictions are to avoid complications with unfolding recursion.

A new rule in our type-and-effect system types the new recursive functions:

$$\frac{\begin{array}{c}\textsc{TRec2}\\ C;\ \mathit{confined}_C(\Gamma), f : T \xrightarrow{\beta} T', x : T, z : \mathsf{Ses}^{\rho} \vdash e : T' \triangleright b \\ C \vdash \mathit{confined}(T, T') \qquad C \vdash \mathtt{rec}^{\rho}_{\beta}\, b \subseteq \beta\end{array}}{C;\ \Gamma, z : \mathsf{Ses}^{\rho} \vdash \mathtt{recses}\, f(x) \Rightarrow e : T \xrightarrow{\beta} T' \triangleright \tau}$$

This rule is similar to TRec for recursive functions, with the exception that it requires that exactly one session endpoint from the environment to be used in the function. The static identity ρ of the endpoint is recorded in the behaviour of the function.

The additional rules in the second level of our type system is:

$$(l : \mu X.\eta) \cdot \Delta \vDash \mathtt{rec}^{\rho}_{\beta}\, b \rightarrow_C \Delta \vDash \tau$$

when $(l : \eta) \vDash b \Downarrow_{C'}$, $C \vdash \rho \sim l$ and

$$C' = (C \backslash (\mathtt{rec}^{\rho}_{\beta}\, b \subseteq \beta)) \cup (\tau^X \subseteq \beta)$$

Moreover,

$$(l : X) \vDash \tau^X \rightarrow_C \epsilon \vDash \tau$$

These two rules check that the behaviour proceeds according to the recursive session type. Note that these rules keep the length of reductions finite, and therefore $\Delta \vDash b \Downarrow_C$ is still decidable.

Because recursive session types are guarded, the subtype relation can be easily extended:

$$\frac{}{C \vdash X <: X} \qquad \frac{C \vdash \eta <: \eta'}{C \vdash \mu X.\eta <: \mu X.\eta'}$$

With the addition of recursive session types we can write programs that run potentially infinite protocols.

Example 6.5.1. *Consider the following server expression:*

$$\begin{array}{l}\mathtt{let}\, z = \mathtt{acc\text{-}init}\, () \\ \mathtt{in}\, (\mathtt{recses}\, f(x) \Rightarrow \mathtt{send}\, z\, x;\ \mathtt{case}\, z\, \{\ \mathtt{Inc} \Rightarrow f\,(x+1), \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{End} \Rightarrow ()\qquad\quad \})\ 0\end{array}$$

The server accepts a connection on `init` *creating the endpoint* z*. It then enters a loop which sends a number on* z *and offers two choices to its partner: to terminate* (`End`) *or to loop* (`Inc`)*. The session*

type of endpoint z here is

$$\mu X.!\mathsf{Int}.\Sigma\{?\mathtt{Inc}.X,\ ?\mathtt{End}.\mathsf{end}\}$$

6.6 Related Work

We presented a new approach for adding session types to high-level programming languages, and used it to give a conservative extension of a core of ML. In the extended language we can type interesting programs with only minimal annotations. For example, in order to send a recursive function as a value over some endpoint p, the function must be recursive function. We consider it an annotation when a function must be expressed as a recursive function in order to be sent. We showed that type soundness guarantees a weak deadlock-freedom property. A sound and complete type inference algorithm for our type system is developed in Chapter 7. To our knowledge this is the first such algorithm for session types that supports delegation.

Our approach is based on extracting the communication effect of program expressions and then imposing a session type discipline on this effect. To extract communication effects we extended the work of [Amtoft et al., 1999] for inferring the communication behaviour of CML programs. This extends foundational work, such as Milner's original polymorphic type inference ([Milner, 1978b]) and Tofte and Talpin's region analysis ([Tofte and Talpin, 1994]). This effect is a term in a restricted process algebra which lends itself well for session type checking. The session type discipline we use is inspired by [Castagna et al., 2009] where a stack principle is imposed on session types. This stack principle gives us weak deadlock freedom. The accuracy of regions for approximating the endpoints in a program can be improved using context-sensitive techniques from static analysis (e.g., k-CFA from [Shivers, 1991]).

Our session type discipline is presented as an abstract interpretation where $\Delta \vDash b \Downarrow_C$ means that all paths from b with stack Δ and environment C reduce to a terminal configuration $\epsilon \vDash \tau$. An equivalent definition for $\Delta \vDash b \Downarrow_C$ can be given using inference rules.

Another approach to checking session types in high-level languages include substructural type systems. As representative examples, [Vasconcelos et al., 2006] develop such a type system for a functional language with threads, and [Wadler, 2012] presents a linear functional language with effects. Type soundness in the former guarantees only that communications respect session types, whereas in the latter it also guarantees deadlock freedom. Our type soundness sits in between these two extremes: deadlock freedom is guaranteed only when processes do not diverge and their requests for opening new sessions are met. We believe that we could replace our session discipline with any of these (or indeed others) giving a more relaxed or more strict typing system. However it is unclear whether a sound and complete inference algorithm would still be possible with such a change.

[Toninho et al., 2013] add session-typed communication to a functional language using a monad. This approach cleanly separates language features and type soundness gives deadlock freedom guarantee similar to ours. [Pucella and Tov, 2008] have used an indexed monad to embed in Haskell session types, including a form of recursive sessions but without endpoint delegation. In this system session types are inferred by Haskell's type inference. However, the programmer needs to guide type inference using type-level operations with no runtime effect.

[Mezzina, 2008] developed a type inference algorithm for session types in a calculus of services. The type system does not have recursive session types but it can type replicated processes that only use finite session types, similar to our approach. This system also does not support endpoint delegation.

Chapter 7

Session Types Inference Algorithm

This chapter presents the inference algorithms to infer session types for any given expression e in $\mathrm{ML_S}$. Session type inference does not require the programmer to add any annotations, but only to indicate which recursive functions need to be self-confined, as explained in Remark 6.4.2. We will explain that our algorithms are sound and complete with reference to the two stages type system of Chapter 6.

Our approach consists of three inference algorithms, $\mathcal{W}$, $\mathcal{SI}$ and $\mathcal{D}$. Algorithm $\mathcal{W}$ calculates the types, behaviours and constraints to type e under the first stage of Sec. 6.4.1. Algorithm $\mathcal{SI}$ and $\mathcal{D}$ correspond to the second stage of type checking: the former calculate the session types of each channel c and $\bar{c}$ in e, while the latter checks that the inferred session types associated with c and $\bar{c}$ are indeed dual. The heart of session type inference is Algorithm $\mathcal{SI}$, which is the main focus of this chapter.

Algorithm $\mathcal{W}$ is a straightforward adaptation of the homonymous algorithm of [Amtoft et al., 1999]: given an expression e, $\mathcal{W}$ calculates its type t, behaviour b and constraints set C; no session information is calculated. Apart from simple adaptations on concurrency primitives from CML to $\mathrm{ML_S}$, the algorithm generates pairs of constraints $c \sim \psi$ and $\bar{c} \sim \psi'$ for any global channel occurring in the e, with ψ and ψ' unique *session variables*. These variables only occur in the behaviour b in the form of $\texttt{push}(l : \psi)$ operations. Soundness and completeness results of $\mathcal{W}$ follow from [Amtoft et al., 1999], as explained in Section 7.1.

Given the behaviour b of e and the constraint set C calculated by $\mathcal{W}$, Algorithm $\mathcal{MC}$ recursively explores all the transitions allowed by the abstract interpretation semantics of Fig. 6.4 with a depth-first strategy. During this process, the algorithm expands the ψ variables in b to concrete session types according to the operators in b. The output is a set of substitutions $[\psi \mapsto \eta]$ for all the ψ variables contained in b, and a refined set of constraints C'. Algorithm $\mathcal{SI}$ is described in Section 7.2. Its termination is described in Section 7.2.1, which hinges on the fact that C is behaviour-compact. Soundness and completeness results are discussed respectively in Section 7.2.2 and 7.2.3.

For each pair of constraints $c \sim \eta$ and $\bar{c} \sim \eta'$, Algorithm $\mathcal{D}$ checks that η and η' are dual. More constraints and substitutions might be discovered in this last phase, for example in case an endpoint from c is delegated, and the dual endpoint from $\bar{c}$ completes the session. Section 7.3 describes this last algorithm and briefly discusses its soundness and completeness.

7.1 Algorithm $\mathcal{W}$

Given an expression e, the first step to session type inference is inferring a type T, behaviour b and constraints C such that $C;\ \Gamma \vdash e : T \triangleright b$ holds, according to the typing rules of Fig. 6.3, if any such typings exists. As already mentioned, Algorithm $\mathcal{W}$ can be adapted straightforwardly from the homonymous Algorithm $\mathcal{W}$ of [Amtoft et al., 1999], and therefore its definition is not included in this thesis. Algorithm $\mathcal{W}$ is composed by three sub-algorithms: Algorithm $\mathcal{W}'$, $\mathcal{F}$ and $\mathcal{R}$.

Algorithm $\mathcal{W}'$ is the core inference algorithm. Given as inputs a typing environment Γ, an empty constraints set C and an expression e, Algorithm $\mathcal{W}'$ recursively traverses the abstract syntax tree of e. The leaves are either base values, such as integers, or variables. The former is assigned its base type and the empty behaviour (e.g. Int and τ); the latter is assigned a fresh type variable α and again the empty behaviour τ. Instead of using unification, as the original Algorithm W of [Milner, 1978b], the algorithm generates *type constraints*. For example, suppose that $e = \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3$ and that the following holds:

$$C;\ \Gamma \vdash e_1 : \mathsf{Bool} \triangleright b_1$$
$$C;\ \Gamma \vdash e_2 : T_2 \triangleright b_2$$
$$C;\ \Gamma \vdash e_3 : T_3 \triangleright b_3$$

Instead of finding a most general substitution σ such that $T_2\sigma = T_3\sigma$ through unification, $\mathcal{W}'$ generates a fresh variable α and two new constraints $C' = \{T_2 \subseteq \alpha, T_3 \subseteq \alpha\}$, and it returns the following typing:

$$C \cup C';\ \Gamma \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 : \alpha \triangleright b_1; b_2; b_3$$

As explained in Sec. 6.4.1, concurrency primitives are values with polymorphic types of the form $TS = \forall(\vec{\alpha}\vec{\beta}\vec{\rho}\vec{\psi} : C_0).\, T$. For example, `send` has type $\forall(\alpha\beta\rho : \texttt{pop}\rho!\alpha \subseteq \beta).\, \mathsf{Ses}^{\rho} \times \alpha \xrightarrow{\beta} \mathsf{Unit}$. Algorithm first $\mathcal{W}'$ creates a substitution σ that instantiates the variables $\vec{\alpha}\vec{\beta}\vec{\rho}\vec{\psi}$ in TS with fresh variables. The primitive is then assigned type $T\sigma$, and a new constraint set $C_0\sigma$ is added to C. For example, `send` can be assigned type $\mathsf{Ses}^{\rho_{fr}} \times \alpha_{fr} \xrightarrow{\beta_{fr}} \mathsf{Unit}$ by substitution $\sigma = [\alpha, \beta, \rho \mapsto \alpha_{fr}, \beta_{fr}, \rho_{fr}]$, and the constraint set $C_0\sigma = \{\texttt{pop}\rho_{fr}!\alpha_{fr} \subseteq \beta_{fr}\}$ is added to C.

The only modification that we introduce with regards to primitives, is that each occurrence of the construct $\texttt{req-}c^l$ is always assigned the same, unique fresh variable ψ in its constraints set. For example Algorithm $\mathcal{W}'$ generates for $\texttt{req-}c^{l_1}$ and $\texttt{req-}c^{l_2}$ the constraints $\texttt{push}(l_1 : \psi) \subseteq \beta$ and $\texttt{push}(l_2 : \psi) \subseteq \beta'$ for some fresh variables ψ, β_1 and β_2. The same happens with $\texttt{acc-}c$; $\texttt{acc-}c$ and $\texttt{req-}c$ are assigned two distinct variables ψ and ψ'.

Constraints are generated and accumulated by $\mathcal{W}'$ throughout the abstract syntax tree of e. Algorithm $\mathcal{F}$ and $\mathcal{R}$ are optimizations of the constraints set C calculated by $\mathcal{W}'$. These two algorithms are not necessary for the inference process itself, but significantly decrease the number of constraints in C. They are therefore very useful by making $\mathcal{W}$ more efficient and its output more readable.

Algorithm $\mathcal{F}$ iteratively breaks complex type constraints into simpler type, and substitutes variables for concrete types as much as possible. For example, the constraint $\alpha \subseteq \mathsf{Int}$ yields the substitution

$[\alpha \mapsto \mathsf{Int}]$; the constraint $T_1 \overset{\beta}{\to} T_2 \subseteq T_1' \overset{\beta'}{\to} T_2'$ is replaced by the constraints $T_1' \subseteq T_1, T_2 \subseteq T_2'$ and $\beta \subseteq \beta'$. The only modification necessary for session types is to introduce a new case for the type of end-points Ses^ρ, which replaces $\mathsf{Ses}^\rho \subseteq \mathsf{Ses}^{\rho'}$ with $\rho \sim \rho'$.

Algorithm $\mathcal{R}$ focuses on constraints that only contain a variable on both the left and right-hand side (e.g. constraints of the form $\alpha_1 \subseteq \alpha_2$). The algorithm explores the graph that such constraints compose by transitivity, and reduces it by collapsing cycles and removing redunant constraints that don't occur in the behaviour b anymore, for example.

The soundness of Algorithm $\mathcal{W}$ can be stated as follows:

Theorem 7.1.1 (Soundness of Algorithm $\mathcal{W}$)**.** *If $\mathcal{W}([], e) = (\sigma, t, b, C)$ then C; $[] \vdash_{\mathsf{n}} e : t \triangleright b$.*

Regarding completeness, let $jdg^\star$ be any valid typing judgement for an expression e. Completeness first show that Algorithm $\mathcal{W}$ always calculates a judgment jdg for e. Moreover it also shows that $jdg^\star$ is a *lazy instance* of jdg: in the sense that there exists a substitution σ' such that $jdg^\star$ can always be derived by further instantiating jdg with σ' and by subtyping. This second property points to the fact that $\mathcal{W}$ calculates *principal types* for e (see [Amtoft et al., 1999], Sec. 1.5.1, p.30). Completeness is stated as follows:

Theorem 7.1.2 (Completeness of Algorithm $\mathcal{W}$)**.** *If $C^\star$; $[] \vdash^{\mathsf{at}}_{\mathsf{n}} e : t^\star \triangleright b^\star$ with $C^\star$ atomic (i.e. all type constraints in $C^\star$ have the form $\alpha_1 \subseteq \alpha_2$), then $\mathcal{W}([], e) = (\sigma, b, t, C)$ and there exists a substitution $\sigma^\star$ such that:*

- $C^\star \vdash C\sigma^\star$
- $C^\star \vdash b\sigma^\star \subseteq b^\star$
- $C^\star \vdash t\sigma^\star <: t^\star$

7.2 Algorithm $\mathcal{SI}$

Algorithm $\mathcal{SI}$ infers a session type η for each for each channel c and $\bar{c}$ in the constraints C and behaviour b calculated from Algorithm $\mathcal{W}$. The pseudo-code for the complete algorithm, together with helper functions, is presented in Appendix B.

Algorithm $\mathcal{SI}$ operates on a slightly different syntax of session types, which is defined as follows:

$$\eta ::= \psi \mid \mathsf{end} \mid !T.\eta \mid ?T.\eta \mid !\eta.\eta \mid ?\eta.\eta \mid \psi_{\mathtt{in}} \mid \psi_{\mathtt{ex}}$$

Variable ψ is a session type variable, which are introduced by Algorithm $\mathcal{W}$. Internal and external choices are removed from sessions. In their place, we have two special variables $\psi_{\mathtt{in}}$ and $\psi_{\mathtt{ex}}$, which are bound by *choice constraints*, defined by the following new constraints on $\psi_{\mathtt{in}}$ and $\psi_{\mathtt{ex}}$:

$$c ::= \quad \ldots \mid \bigoplus_{i \in I} !L_i.\eta_i \sim \psi_{\mathtt{in}} \mid \sum_{i \in (I_1, I_2)} ?L_i.\eta_i \sim \psi_{\mathtt{ex}}$$

During session inference, the constraint set C might be refined to a new set C', containing more precise session types for $\psi_{\mathtt{in}}$ and $\psi_{\mathtt{ex}}$ (for example by adding new labels to an internal choice, or by moving

an active label to inactive in an external choice), or new constraints on types (because of a `pop` for example). Choice constraints in C are ordered according to subtyping:

$$\frac{C \vdash \psi_{\texttt{in}} \sim \eta \quad C \vdash \eta <: \eta'}{C \vdash \psi_{\texttt{in}} \sim \eta'} \textsc{Inf-IChoice}$$

$$\frac{C \vdash \psi_{\texttt{ex}} \sim \eta \quad C \vdash \eta <: \eta'}{C \vdash \psi_{\texttt{ex}} \sim \eta'} \textsc{Inf-EChoice}$$

Abstract interpretation transitions can be naturally extended to the sub-language just presented, except for the two cases when b pops a label L_k, and the top of the stack contains either $\psi_{\texttt{in}}$ or $\psi_{\texttt{ex}}$. In these two cases, if $C \vdash \psi_{\texttt{ex}} \sim \eta$ or $C \vdash \psi_{\texttt{ex}} \sim \eta$, η substitutes $\psi_{\texttt{ex}}$ or $\psi_{\texttt{in}}$ on the stack.

For all constraints $(c \sim \psi) \in C$, Algorithm $\mathcal{SI}$ infers a substitution σ and a refined set C' such that $\epsilon \vDash b\sigma \Downarrow_{C'} \epsilon$ holds. The core of this algorithm is the abstract interpreter $\mathcal{MC}$. The inputs to $\mathcal{MC}$ are a configuration $\Delta \vDash b$, a set of constraints C and an internal *continuation stack* K. The continuation stack K is defined by the grammar $K ::= \epsilon \mid b \cdot K$. We indicate with $K[b]$ the stack $b \cdot K$. Algorithm $\mathcal{SI}$ performs a call $\mathcal{MC}\big(\epsilon \vDash b,\ \epsilon,\ C\big)$, where b and C are derived by Algorithm $\mathcal{W}$.

Algorithm $\mathcal{MC}$ recursively explores all the transitions allowed by the abstract interpretation semantics in Fig. 6.4. Session variables from constraints $c \sim \psi$ in C are eventually pushed on the stack by a `push` operation in b. Session inference is guided by `pop` operations. For example, $\mathcal{MC}\big((l : \psi) \cdot \Delta \vDash \texttt{pop}\rho!T,\ K,\ C\big)$ produces the substitution $[\psi \mapsto !\alpha.\psi']$, with α and ψ' fresh, and adds the constraint $(T \subseteq \alpha)$ to C.

All substitutions are applied eagerly and composed iteratively. Sequential behaviour $b_1; b_2$ is decomposed into b_1 and a continuation $[]; b_2$, which is pushed on the continuation stack K. Inference is first carried out on b_1; if b_1 evaluates to τ, then a continuation $[]; b_2$ is popped from K, and $\mathcal{MC}$ evaluates the behaviour $[\tau]; b_2$. Whenever a branching behaviour is explored, such as $b_1 \oplus b_2$, each branch b_i is explored separately, and their resulting substitutions and constraints are composed.

Algorithm $\mathcal{MC}$ terminates when b is τ and the continuation stack is empty. The output is a substitution σ that instantiates the session variables ψ generated by $\mathcal{W}$ with the accrued session types η calculated by $\mathcal{MC}$; the constraint set C with the accumulated constraints C' from $\mathcal{MC}$ are returned as well. Inference fails when $\mathcal{MC}$ reaches a configuration stuck $\Delta \vDash b$ in which either Δ is not the empty stack or b is not τ. This corresponds to an error in the session type discipline.

7.2.1 Finiteness of Abstract Interpretation

This section proves that the abstract interpretation of a configuration $\Delta \vDash b$ in a well-formed environment C always generates a finite state-space. We first formalize the notion of *behaviour compact* from Definition 6.4.2. Then we define a translation from behaviours with β variables to *ground behaviours*, i.e. behaviours without βs. We show that this translation is fully abstract with reference to the abstract interpretation semantics. Finally, we show that a configuration $\Delta \vDash b$ and constraints C generate a finite state-space when b is ground and C is well-formed.

Formalization of behaviour compactness

By Definition 6.4.2, a set of constraints C is well-formed only if it is *behaviour compact*, i.e. all cycles in behaviour constraints contain at least one $(\mathtt{rec}_\beta\, b \subseteq \beta) \in C$, and recursive behaviour constraints $(\mathtt{rec}_\beta\, b \subseteq \beta')$ in C are unique, and no other constraint can bind another behaviour b to β' in C.

Let $\mathcal{B}$ denote the set of all behaviours b, and let $\beta \;\mathsf{in}\; b$ hold whenever there is an occurrence of β in b (when $b = \mathtt{rec}_{b'}\, \beta$, we stipulate that $\beta \;\mathsf{in}\; b$ only if $\beta \;\mathsf{in}\; b'$). In order to formalize this particular notion of cyclicity, we introduce a binary relation on behaviours, that allows variables β from constraints $\mathtt{rec}_\beta\, b \subseteq \beta$ to form cycles, but disallows all other β to do so. This relation is called the *dependency relation*, and it is defined as follows:

Definition 7.2.1 (Dependency relation)**.** *Let C be a well-formed constraint set. The dependency relation $\dashrightarrow_C \subseteq \mathcal{B} \times \mathcal{B}$ is the least relation such that:*

$$\frac{}{\beta \dashrightarrow_C b} \quad C \vdash b \subseteq \beta, b \neq \mathtt{rec}_\beta\, b' \qquad \frac{\beta \dashrightarrow_C b}{\beta \dashrightarrow_C \beta'} \quad \beta' \;\mathsf{in}\; b \qquad \frac{\beta_1 \dashrightarrow_C \beta_2 \qquad \beta_2 \dashrightarrow_C b_3}{\beta_1 \dashrightarrow_C b_3}$$

We say that a constraint set C is *behaviour compact* when $\dashrightarrow_C$ is a strict well-founded order. It is well-known that if a relation R is a strict order, then it denotes a direct acyclic graph. Therefore if the relation $\dashrightarrow_C$ is a strict order, $\dashrightarrow_C$ is acyclic except on recursive constraints $\mathtt{rec}_\beta\, b \subseteq \beta$.

Lemma 7.2.2. *If $C_1 = C \uplus \{\mathtt{rec}_\beta\, b \subseteq \beta\}$ is behaviour compact, then $C_2 = C \cup \{\tau \subseteq \beta\}$ is behaviour compact.*

Proof. Since τ is a ground term, there is no variable β' in τ such that $\tau \dashrightarrow_C \beta'$. There is also no variable β' in $\mathtt{rec}_b\, \beta$ such that $\mathtt{rec}_b\, \beta \dashrightarrow_C \beta'$, since this is directly forbidden by the definition of dependency relation. Because of these two facts, the relation denoted by $\dashrightarrow_{C_1}$ is the same relation denoted by $\dashrightarrow_{C_2}$. And since the former is a strict order, the latter is a strict order, which proves the lemma. □

Behaviour variables elimination

The occurrence of a variable β in a behaviour b creates an indirect link between b and the constraints C where β is defined. This hidden connections introduces cumbersome technical complications when proving properties of the abstract interpretation semantics. On the contrary, *ground behaviours*, i.e. behaviours that do not contain β variables, are easier to reason about. This section introduces a translation from any behaviour b to the ground behaviour $[\![b]\!]^{\mathtt{g}}_C$, and shows that it is fully abstract w.r.t the abstract operational semantics.

The abstract interpretation semantics treats β variables as place-holders: Rule ICh replaces a β with any behaviour b to which β is bound in C; Rule Rec effectively replaces β with a τ inside recursive behaviours $\mathtt{rec}_\beta\, b$. This observation suggests that a β variable can be substituted either with the internal choice of all the behaviour it binds in C, or with a τ inside recursive behaviours. Such a translation is defined as follows:

Definition 7.2.3 (Ground translation)**.** *Let C be a well-formed constraint set. The* ground translation

$[\![-]\!]^{\mathtt{g}}_C :: \mathcal{B} \to \mathcal{B}$ *is the total function defined by the following equations:*

$$
\begin{array}{lll}
[\![\beta]\!]^{\mathtt{g}}_C & = \bigoplus\{\, [\![b_i]\!]^{\mathtt{g}}_C \mid b_i \subseteq \beta \in C \,\} & [\![\mathtt{rec}_\beta\, b]\!]^{\mathtt{g}}_{C \uplus \{b' \subseteq \beta\}} = \mathtt{rec}_\beta\, [\![b]\!]^{\mathtt{g}}_{C \cup \{\tau \subseteq \beta\}} \\
[\![b_1; b_2]\!]^{\mathtt{g}}_C & = [\![b_1]\!]^{\mathtt{g}}_C; [\![b_2]\!]^{\mathtt{g}}_C & [\![\mathtt{spawn}\, b]\!]^{\mathtt{g}}_C = \mathtt{spawn}\, [\![b]\!]^{\mathtt{g}}_C \\
[\![b_1 \oplus b_2]\!]^{\mathtt{g}}_C & = [\![b_1]\!]^{\mathtt{g}}_C \oplus [\![b_2]\!]^{\mathtt{g}}_C & [\![\sum_{i \in I} \mathtt{pop}\rho?L_i\, ; b_i]\!]^{\mathtt{g}}_C = \sum_{i \in I} \mathtt{pop}\rho?L_i\, ; [\![b_i]\!]^{\mathtt{g}}_C \\
[\![b]\!]^{\mathtt{g}}_C & = b \text{ if } b \in G &
\end{array}
$$

where $G = \{\, b \mid \forall \beta.\beta \sharp b \,\}$ *is the set of ground terms.*

Let $\mathcal{C}$ be the set of all constraint sets. We introduce an ordering on behaviours and constraints:

Definition 7.2.4 (Structural relation)**.** *Let* C *be well-formed, and* b *be finite. The relation* $\succ_s \subseteq \mathcal{B} \times \mathcal{B}$ *is defined as follows:*

$$
\begin{array}{llll}
\mathtt{rec}_\beta\, b & \succ_s b & b_1; b_2 \succ_s b_i & \text{for } i \in \{1,2\} \\
\mathtt{spawn}\, b & \succ_s b & b_1 \oplus b_2 \succ_s b_i & \text{for } i \in \{1,2\} \\
\sum_{i \in I} \mathtt{pop}\rho?L_i\, ; b_i \succ_s b_i & \text{for } i \in I & &
\end{array}
$$

The structural relation $\succ_s$ is obviously well-founded, since we only consider finite behaviours b in C.

Lemma 7.2.5. *Let* C *be well-formed, and let* $\succ \;=\; \succ_s \cup \dashrightarrow_C$. *Relation* $\succ$ *is well-founded.*

Proof. The lemma is proved by showing that any non-empty subset Q of $\mathcal{B}$ has a minimal element m, i.e. an element such that for all behaviours b if $m \succ b$, then $b \notin Q$.

Let Q be a subset of $\mathcal{B}$. Since $\dashrightarrow_C$ is well-founded, there must be a minimal element b_1 which is minimal in Q according to $\dashrightarrow_C$. If b_1 is a variable β_1 then β_1 is minimal in Q according to $\succ_s$ as well, because by definition there is no b such that $\beta_1 \succ_s b$ holds. Therefore β_1 is minimal according to $\succ$ too and the lemma is proved.

If b_1 is not a variable, then let B_2 be the intersection between Q and the successors of b_1 according to $\succ_s$. We need to consider three cases: B_2 is empty, B_2 contains no singleton variables β, or B_2 contains at least one singleton variable β_2.

If B_2 is the empty set, then b_1 is minimal in Q according to $\dashrightarrow_C$, because by definition $\dashrightarrow_C$ only relates variables and b_1 is not a variable. Therefore the lemma is proved by b_1.

If B_2 contains no singleton variables, then there must be a minimal element of them according to $\succ_s$, and the lemma is proved as in the case when $B_2 = \emptyset$.

If B_2 contains a variable β_2, then let B_3 be the intersection between Q and all the successors of β_2 according to $\dashrightarrow_C$. If B_3 is empty, then the lemma is proved as in the case $b_1 = \beta_1$. If B_3 is not empty, then there must be a minimal element b_3 of them. If we take B_4 to be the intersection between Q and the successors of b_3 according to $\succ_s$, notice that B_4 cannot contain any singleton variables β_4, because otherwise $\beta_2 \dashrightarrow_C \beta_4$ would hold, and the hypothesis that β_2 is minimal would be contradicted. Therefore B_4 is either empty or it contains no singleton variables, and the lemma is proved as the respective cases for B_2.

□

We now show that, when a constraint set C is well-formed, the ground translation of a behaviour b in C does not expand β variables infinitely, but it constructs a *finite* ground behaviour, i.e. a behaviour with a finite syntax tree:

Lemma 7.2.6. *Let C be well-formed and b be a finite behaviour. For any behaviour b, $[\![b]\!]^{\mathtt{g}}_C$ is a finite ground term.*

Proof. By well-founded induction on $\succ = \dashrightarrow_C \cup \succ_s$.

The base case is when b is a ground term in $\{\tau, \mathtt{push}(l:\eta), \mathtt{pop}\rho!L \ldots\}$. These are all ground terms in G, and for these terms the translation $[\![b]\!]^{\mathtt{g}}_C = b$, which is finite and ground by hypothesis. If $b \in \{b_1; b_2, b_1 \oplus b_2, \mathtt{spawn}\, b_1, [\![\sum_{i\in I} \mathtt{pop}\rho?L_i\,; b_i]\!]^{\mathtt{g}}_C\}$, then the lemma is proved by the inductive hypothesis, since for example if $b = b_1; b_2$, then $[\![b_1]\!]^{\mathtt{g}}_C$ and $[\![b_2]\!]^{\mathtt{g}}_C$ are finite ground terms, and therefore $[\![b_1]\!]^{\mathtt{g}}_C; [\![b_2]\!]^{\mathtt{g}}_C$ is finite and ground too.

Because of well-formedness, there are two cases to consider when $b = \beta$: either β is bound to a unique constraint $b \neq \mathtt{rec}_\beta\, b'$ in C, or it is bound to multiple b_i which are not recursive behaviours. In the case that $b \subseteq \beta$ is the only constraint on β in C, and we can write C ad $C' \uplus \{b \subseteq \beta\}$. By definition of translation we have $[\![\beta]\!]^{\mathtt{g}}_C = [\![\mathtt{rec}_\beta\, b']\!]^{\mathtt{g}}_{C'\uplus\{b\subseteq\beta\}} = \mathtt{rec}_\beta\, [\![b']\!]^{\mathtt{g}}_{C'\uplus\{b\subseteq\tau\}}$. Since $\dashrightarrow_{C'\uplus\{b\subseteq\beta\}} = \dashrightarrow_{C'\uplus\{b\subseteq\tau\}}$ by Lem. 7.2.2, then $\succ = \dashrightarrow_{C'\uplus\{b\subseteq\tau\}} \cup \succ_s$. By inductive hypothesis $[\![b']\!]^{\mathtt{g}}_{C'\uplus\{b\subseteq\tau\}}$ is finite and ground, therefore $\mathtt{rec}_\beta\, [\![b']\!]^{\mathtt{g}}_{C'\uplus\{b\subseteq\tau\}}$ is finite and ground too, and the lemma is proved. In the latter case, when β is bound to multiple non-recursive behaviours b_i, the set of all such b_i is finite by well-formedness, and the lemma is proved by the inductive hypothesis as in the case $b_1; b_2$. □

Having proved that the ground translation of a behaviour b always exists for well-formed constraints C, we show some property of the translation w.r.t. the abstract semantics:

Lemma 7.2.7. *Let C be well-formed.*

1. *if $\Delta \vDash b \to_C \Delta \vDash b'$, then $\Delta \vDash [\![b]\!]^{\mathtt{g}}_C \to_C \Delta' \vDash [\![b']\!]^{\mathtt{g}}_C$*
2. *If $\Delta \vDash [\![b]\!]^{\mathtt{g}}_C \to_C \Delta' \vDash b''$, then there exists a b' such that $b'' = [\![b']\!]^{\mathtt{g}}_C$.*

Proof. By rule induction. □

Since the ground translation is always defined, and since transition between a behaviour b and its ground translation are interchangeable by the previous lemma, we will only consider finite ground terms from now on.

Finite state-space

We conclude this section by showing that, given a well-formed C, all configurations $\Delta \vDash b$ always generate a finite state-space, i.e. the set of reachable states from $\Delta \vDash b$ is finite. We prove this result by designing a function that assigns an integer, or *size*, to any configuration $\Delta \vDash b$, and then show that the size of a configuration always decreases after taking a step in the abstract interpretation semantics. Since configurations of size 0 cannot take steps, and since the size decreases after taking a step in the semantics, the number of states that a finite configuration $\Delta \vDash b$ can reach is finite.

We first introduce the size function on behaviours:

Definition 7.2.8 (Behaviour size)**.** *For any behaviour b, the behaviour size $size(-) :: \mathcal{B} \to \mathcal{N}$ is the total function defined by the following equations:*

$$
\begin{array}{llll}
size(\tau) & = 0 & size(\beta) & = 0 \\
size(\mathtt{push}(l:\eta)) & = 2 & size(\mathtt{pop}\rho?\rho') & = 2 \\
size(\mathtt{pop}\rho!T) & = 1 & size(\mathtt{pop}\rho?T) & = 1 \\
size(\mathtt{pop}\rho!\rho') & = 1 & size(\mathtt{pop}\rho!L_i) & = 1 \\
size(\sum_{i\in I} \mathtt{pop}\rho?L_i\,;b_i) & = 1 + |I| + \sum_{i\in I} size(b_i) & size(\mathtt{rec}_\beta\, b) & = 1 + size(b) \\
size(b_1; b_2) & = 1 + size(b_1) + size(b_2) & size(\mathtt{spawn}\, b) & = 2 + size(b) \\
size(b_1 \oplus b_2) & = 2 + size(b_1) + size(b_2) &&
\end{array}
$$

According to the definition, τ is the behaviour with the smallest size, zero. Most `pop` operations have size 1, except for the resume operation $\mathtt{pop}\rho?\rho'$, which has size 2. Notice that `push` has size 2 as well. The reason for this difference is that these operations introduce new frames on the stack in the abstract interpretation semantics, and therefore have to be counted twice in order for the abstract interpretation semantics to be always decreasing in size. The size of the other behaviours is defined inductively.

We now introduce the size of a stacks:

Definition 7.2.9 (Stack size)**.** *The size of a stack Δ, or $size(\Delta)$, is defined by the following equations:*

$$size(\epsilon) = 0 \qquad\qquad size((l:\eta)\cdot\Delta) = 1 + size(\Delta)$$

In short, the size of a stack is its length, or total number of frames. We finally specify the size of configurations:

Definition 7.2.10. *The size of a configuration $\Delta \vDash b$, or $size(\Delta \vDash b)$, is the sum $size(\Delta \vDash b) = 1 + size(\Delta) + size(b)$.*

Lemma 7.2.11. *Let C be well-formed, and let b be a ground finite behaviour. If $\Delta \vDash b \to_C \Delta' \vDash b'$, then $size(\Delta \vDash b) > size(\Delta' \vDash b')$.*

Proof. By rule induction.

If Rule End is applied, then $(l:\mathsf{end})\cdot\Delta \vDash b \to_C \Delta \vDash b$. By definition of size, $size((l:\mathsf{end})\cdot\Delta \vDash b) = size((l:\mathsf{end})\cdot\Delta) + size(b) = 1 + size(\Delta) + size(b) = 1 + size(\Delta \vDash b)$, which proves the lemma.

By hypothesis b is a ground term, therefore Rule Beta cannot be applied.

If Rule Push is applied, then $\Delta \vDash \mathtt{push}(l{:}\eta) \to_C (l:\eta)\cdot\Delta \vDash \tau$ holds. By definition, $size(\Delta \vDash \mathtt{push}(l:\eta)) = 2 + size(\Delta) = 1 + size((l:\eta)\cdot\Delta) = 1 + size((l:\eta)\cdot\Delta \vDash \tau)$, which proves the lemma. The case for Rule Res is proved similarly.

If Rule Out is applied, then $(l:!T.\eta)\cdot\Delta \vDash \mathtt{pop}\rho!T' \to_C (l:\eta)\cdot\Delta \vDash \tau$ holds. By definition, $size((l:!T.\eta)\cdot\Delta \vDash \mathtt{pop}\rho!T') = size((l:!T.\eta)\cdot\Delta) + 1 = 1 + size(\Delta) + 1 = size((l:\eta)\cdot\Delta) + 1 = size((l:\eta)\cdot\Delta \vDash \tau)$, which proves the lemma. The cases for Rule In and Del are proved similarly.

If Rule Ich is applied, then $\Delta \vDash b_1 \oplus b_2 \to_C \Delta \vDash b_i$ holds for $i \in \{1,2\}$. By definition of size, then $size(\Delta \vDash b_1 \oplus b_2) = size(\Delta) + 1 + size(b_1) + size(b_2) = 1 + size(\Delta \vDash b_i) + size(b_j)$ for $\{i,j\} = \{1,2\}$.

The lemma is proved by $1 + size(\Delta \vDash b_i) + size(b_j) > size(\Delta \vDash b_i)$ for $i \in \{1, 2\}$. The cases for Rule ECh, Rec, Spn and Tau are proved similarly.

If Rule Seq is applied, then $\Delta \vDash b_1; b_2 \to_C \Delta' \vDash b_1'; b_2$ only if $\Delta \vDash b_1 \to_C \Delta' \vDash b_1'$ holds. By rule induction $size(\Delta \vDash b_1) > size(\Delta' \vDash b_1')$. By definition of size, $size(\Delta \vDash b_1; b_2) = 1 + size(\Delta) + size(b_1) + size(b_2)$; by the previous inequality we have that $1 + size(\Delta) + size(b_1) + size(b_2) > 1 + size(\Delta') + size(b_1') + size(b_2) = size(\Delta' \vDash b_1'; b_2)$, which proves the lemma. □

The state space of a configuration is defined as follows

Definition 7.2.12 (Execution states)**.** *Let $\Delta \vDash b$ and C be well-formed. The set of execution states of $\Delta \vDash b$ under C, $[\![\Delta \vDash b]\!]_C$, is the least set S that satisfies the following conditions:*

1. $\Delta \vDash b \in S$
2. *if $\Delta_1 \vDash b_1 \in S$ and $\Delta_1 \vDash b_1 \to_C \Delta_2 \vDash b_2$, then $\Delta_2 \vDash b_2 \in S$*
3. *if $\Delta_1 \vDash K[\mathtt{spawn}\, b_1] \in S$, then $[\![\epsilon \vDash b_1]\!]_C \subseteq S$*
4. *if $\Delta_1 \vDash K[\mathtt{rec}_\beta\, b_1] \in S$, then $[\![\epsilon \vDash b_2]\!]_{C \setminus (\mathtt{rec}_\beta\, b \subseteq \beta) \cup (\tau \subseteq \beta)} \subseteq S$*

The execution size *of $\Delta \vDash b$ under C, or $|\Delta \vDash b|_C$, is the size of $[\![\Delta \vDash b]\!]_C$.*

We now prove configurations in well-formed C always generate finite state spaces.

Theorem 7.2.13 (Finite state-space)**.** *Let $\Delta \vDash b$ be a configuration such that b is a finite ground behaviour. For any well-formed C, $|\Delta \vDash b|_C \leq size(\Delta \vDash b)$.*

Proof. By mathematical induction on $size(\Delta \vDash b)$. By definition of $size(-)$, the base case is when $size(\Delta \vDash b) = 1$, which is only possible when $\Delta = \epsilon$ and $b = \tau$. In this case the execution states of $\epsilon \vDash \tau$ is $[\![\epsilon \vDash \tau]\!]_C = \{\epsilon \vDash \tau\}$; the size of this set is therefore 1, and the base case is proved.

The inductive case is when $size(\Delta \vDash b) > 1$. Configuration $\Delta \vDash b$ might or might not be able to take a transition step $\Delta \vDash b \to_C \Delta' \vDash b'$. If it cannot take a step, then $|\Delta \vDash b|_C = 1$, and the proposition is proved by the hypothesis that $size(\Delta \vDash b) > 1$.

Suppose that there exists $\Delta' \vDash b'$ such that $\Delta \vDash b \to_C \Delta' \vDash b'$. Let us proceed by rule induction. Most cases are trivial, except when Rule Spn or Rec is applied. If Rule Spn is applied, then the following holds:

$$\frac{}{\Delta \vDash \mathtt{spawn}\, b \to_C \Delta \vDash \tau}\ \epsilon \vDash b \Downarrow_C$$

By definition of size, $size(\Delta \vDash \mathtt{spawn}\, b) = 1 + size(\Delta) + size(\mathtt{spawn}\, b) = 1 + size(\Delta) + 2 + size(b) = size(\Delta \vDash \tau) + 1 + size(\epsilon \vDash b)$. By definition of execution states, $[\![\Delta \vDash \mathtt{spawn}\, b]\!]_C = \{\Delta \vDash \mathtt{spawn}\, b\} \cup [\![\Delta \vDash \tau]\!]_C \cup [\![\epsilon \vDash b]\!]_C$. By inductive hypothesis we have that $[\![\Delta \vDash \tau]\!]_C \leq size(\Delta \vDash \tau)$ and $[\![\epsilon \vDash b]\!]_C \leq size(\epsilon \vDash b)$. Therefore, since the set $\{\Delta \vDash \mathtt{spawn}\, b\}$ has size 1, $|\Delta \vDash \mathtt{spawn}\, b|_C = |[\![\Delta \vDash \mathtt{spawn}\, b]\!]_C| = |\{\Delta \vDash \mathtt{spawn}\, b\}| + |[\![\Delta \vDash \tau]\!]_C| + |[\![\epsilon \vDash b]\!]_C| = 1 + |[\![\Delta \vDash \tau]\!]_C| + |[\![\epsilon \vDash b]\!]_C| \leq 1 + size(\Delta \vDash \tau) + size(\epsilon \vDash b) = size(\Delta \vDash \mathtt{spawn}\, b)$, which proves the proposition. The case for Rule Rec is proved similarly.

□

The result of finiteness gives us a convenient induction principle for configurations:

Corollary 7.2.14. *If* $\Delta \vDash b \rightarrow_C \Delta' \vDash b'$*, then* $[\![\Delta \vDash b']\!]_C \supset [\![\Delta' \vDash b']\!]_C$*.*

Proof. By Prop. 7.2.13 and Lem. 7.2.11.

□

Using this principle, we can prove termination of $\mathcal{MC}$:

Proposition 7.2.15 (Termination of $\mathcal{SI}$)**.** *For any well-formed* C *and* $\Delta \vDash K[b]$*,* $\mathcal{MC}\big(\Delta \vDash b,\ K,\ C\big)$ *terminates.*

Proof. By induction on the execution size of $\Delta \vDash K[b]$. For any clause $\mathcal{MC}\big(\Delta \vDash b,\ K,\ C\big)$, any recursive sub-call to $\mathcal{MC}$ either contains as input either a configuration $\Delta' \vDash b'$ that is directly smaller than $\Delta \vDash b$ by execution size (e.g. after a `pop` is executed), or can be unfolded until its input is smaller (e.g. when the behaviour is an external choice at line 78). The proposition follows directly from the inductive hypothesis □

Termination of $\mathcal{SI}$ follows directly from this result:

Theorem 7.2.16 (Termination of $\mathcal{SI}$)**.** *For any well-formed* C *and* b*,* $\mathcal{SI}(b, C)$ *terminates.*

Proof. By Prop. 7.2.15. □

7.2.2 Soundness of Algorithm $\mathcal{SI}$

We now present a proof of soundness for Algorithm $\mathcal{SI}$, namely that if $\mathcal{SI}(b, C) = (\sigma_1, C_1)$, then $\epsilon \vDash b\sigma_1 \Downarrow_{C_1}$; more informally, the substitution σ_1 found by $\mathcal{SI}$ is always a valid session type. The result of soundness depends on the soundness of $\mathcal{MC}$, namely that if $\mathcal{MC}(\Delta \vDash b, C, K) = (\sigma_1, C_1)$, then $\Delta\sigma_1 \equiv \Delta'$ and $\Delta' \vDash K[b]\sigma_1 \Downarrow_{C_1}$. We assume that all the free variables in the input configuration $\Delta \vDash K[b]$ also occur in some constraint in C.

We begin by defining a simple notion of equivalence to remove terminated sessions from a stack Δ:

Definition 7.2.17 (Terminated session equivalence)**.**

$$\frac{}{\epsilon \equiv \epsilon} \quad \frac{\Delta \equiv \Delta'}{(l : \mathsf{end}) \cdot \Delta \equiv \Delta'} \quad \frac{\Delta \equiv \Delta'}{(l : \eta) \cdot \Delta \equiv (l : \eta) \cdot \Delta'} \ \eta \neq \mathsf{end}$$

Algorithm $\mathcal{MC}$ progressively refines the input set of constraint C by refining session variables, assigning a lower session type to external and internal choices, and by adding type constraints. When the algorithm terminates, the resulting set C' is a refinement of C, after applying the inferred substitution σ to it. We define more formally the notion of constraints refinement as follows:

Definition 7.2.18 (Constraints refinement)**.** *A constraint set* C *is a* refinement *of constraint set* C'*, written* $C \vdash C'$*, when:*

- *for all constraints* $(C \vdash \psi_{in} \sim \eta')$ *in* C'*, there exists a session* η *such that* $C \vdash \eta' <: \eta$ *and* $C \vdash \psi_{in} \sim \eta$ *hold.*
- *for all constraints* $(C \vdash \psi_{ex} \sim \eta')$ *in* C'*, there exists session* η *such that* $C \vdash \eta' <: \eta$ *and* $C \vdash \psi_{ex} \sim \eta$ *hold.*

- *if* $C' \vdash \mathtt{push}(l : \eta') \subseteq \beta$, *then* $C \vdash \mathtt{push}(l : \eta) \subseteq \beta$ *and* $C \vdash \eta <: \eta'$
- *for all other constraints* $(g \subseteq g')$ *in* C', $C \vdash g \subseteq g'$ *holds*

Following [Amtoft et al., 1999, Sec. 2.2.5, p.51], substitution is defined as follows:

Definition 7.2.19 (Session inference substitution)**.** *An* inference substitution σ *is a total function from session variables* ψ *to sessions* η. *The* domain *of an inference substitution* σ *is* $dom(\sigma) = \{\, \psi \,|\, \sigma(\psi) \neq \psi \,\}$ *and its range is* $rg(\sigma) = \bigcup\{\, FV(\sigma(\psi)) \,|\, \psi \in dom(\sigma) \,\}$.

Substitutions preserve constraints refinement:

Lemma 7.2.20 (Substitution invariance)**.** *Let* C *and* C' *be well-formed, and* σ *a session inference substitution. If* $C' \vdash C$, *then* $C'\sigma \vdash C\sigma$.

Proof. Invariance on ψ and ρ variables is proved by case analysis as Lemma 2.17, p. 68 of [Amtoft et al., 1999]. The cases for $\psi_{\mathtt{in}}, \psi_{\mathtt{ex}}$ and α can be proved similarly. □

We now prove that the algorithm $\mathcal{MC}$ produces a refinement of its input constraints set C. We first need to show prove that the sub-type checking sub-routing produces refinements:

Lemma 7.2.21 (Constraint refinement on sub-types)**.** *If* $\mathsf{sub}(\eta_1, \eta_2, C) = (\sigma_1, C_1)$, *then* $C_1 \vdash C\sigma_1$.

Proof. By structural induction on η_1.

Function sub terminates when either η_2 has the same shape as η_1 (i.e. if $\eta_1 = !T_1.\eta_1'$ then $\eta_2 = !T_2.\eta_2'$), or η_2 is a variable ψ_2. In the latter case $(\eta_1, \psi_2, C) = (\sigma, C\sigma)$ with $\sigma = [\psi_2 \mapsto \eta_1]$, and the lemma is trivially proved by $C\sigma \vdash C\sigma$.

Consider the former case, when η_1 and η_2 have the same shape. When $\eta_1 = \mathsf{end}$ the lemma holds trivially. If η_1 is $!T_1.\eta_1'$, then $\eta_2 = !T_2.\eta_2'$; by induction we have $C_1 \vdash C\sigma_1 \cup \{T_2\sigma_1 \subseteq T_1\sigma_1\}$, which implies $C_1 \vdash C\sigma_1$ by definition of constraint refinement. The case for $\eta_1 = ?T_1.\eta_1'$ is proved similarly.

The cases for delegate and resume hold directly by inductive hypothesis.

When η_1 is an internal choice, we need to show that $f(I_2, C) = (\sigma_1, C_1)$ implies $C_1 \vdash C\sigma_1$. This can be easily proved by induction of the size of I_2. The base case (line 21) is trivial. In the inductive case, if a label k from I_2 is missing in I_1 (lines 22-25), then $\bigoplus_{i \in I_1 \cup \{k\}} !L_i.\eta_{1i} \oplus \eta_{2_k}$ is a subtype of $\bigoplus_{i \in I_1} !L_i.\eta_{1i}$ by definition (because $I_1 \cup \{k\} \subset I_1$) and the lemma follows by inductive hypothesis. If k is in I_1, then the lemma follows directly by inductive hypothesis.

When η_1 is an external choice, we need to show that $f(J_1 \cup J_2, C) = (\sigma_1, C_1)$ implies $C_1 \vdash C\sigma_1$. We prove this by induction on the size of $J_1 \cup J_2$. Let $\eta_1 = \sum_{i \in (I_1, I_2)} ?L_i.\eta_{1i}$ and $\eta_2 = \sum_{i \in (J_1, J_2)} ?L_i.\eta_{2i}$ be such that $I_1 \subseteq J_1$. The base case (line 36) is trivial. In the inductive case, suppose that a label k is either in the inactive labels J_2, or it is in both the active labels I_1 and J_1. Then it is sufficient to show that η_{1k} is a subtype of η_{2k}, which holds by inductive hypothesis on sub. If k is in the active labels I_1 but it is in the inactive labels J_2, then removing k from I_1 and adding it to the inactive labels I_2 makes $I_1 \backslash \{k\}$ be a subset of J_1; under this condition the subsequent call $\mathsf{f}(I \cup \{k\}, C_1)$ is proved as in the previous case. If k is neither in I_1 nor I_2, then k is added to the inactive labels I_2 and the lemma is proved as in the previous case too.

□

We can now prove that $\mathcal{MC}$ produces refinements:

Lemma 7.2.22 (Constraint refinement)**.** *If* $\mathcal{MC}(\Delta \vDash b, C, K) = (\sigma_1, C_1)$*, then* $C_1 \vdash C\sigma_1$*.*

Proof. By induction on the execution size $|\Delta\sigma \vDash b\sigma|_{C_1}$.

Most cases follow directly from the inductive hypothesis, such as the case for `push`:

```
11 -- push a new frame on the stack
12 MC(Δ ⊨ push(l : η), C, K) = (σ2σ1, C2)
13   if   (σ1, Δ1) = closeFrame(l, Δ)
14   and (σ2, C2) = MC((l : ησ1) · Δ1 ⊨ τ, Cσ1, Kσ1)
```

At line 14 $\mathcal{MC}\big((l:\eta\sigma_1)\cdot\Delta_1 \vDash \tau,\ C\sigma_1,\ K\sigma_1\big) = (\sigma_2, C_2)$. By inductive hypothesis we obtain directly that $C_2 \vdash C\sigma_2\sigma_1$.

The lemma is also trivial when b is a `pop` operation that sends a type T:

```
16 -- send
17 MC((l : ψ) · Δ ⊨ popρ!T, C, K) = (σ2σ1, C2)
18   if   C ⊢ l ∼ ρ
19   and σ1 = [ψ ↦ !α.ψ′] where α, ψ′ fresh
20   and (σ2, C2) = MC((l : ψ′) · Δσ1 ⊨ τ, Cσ1 ∪ {T ⊆ α}, Kσ1)
```

At line 20 $\mathcal{MC}\big((l:\psi')\cdot\Delta\sigma_1 \vDash \tau,\ C\sigma_1 \cup \{T \subseteq \alpha\},\ K\sigma_1\big) = (\sigma_2, C_2)$. By inductive hypothesis we obtain directly that $C_2 \vdash C\sigma_2\sigma_1 \cup \{T\sigma_2 \subseteq \alpha\sigma_2\}$, which by definition constraints refinement implies that $C_2 \vdash C\sigma_2\sigma_1$, and the lemma is proved. The lemma is proved similarly when a type T is received (lines 25-32), and when a session is resumed (lines 50-61). When `pop` delegates a session (lines 34-48), the lemma is proved by Lem. 7.2.21.

The lemma holds directly by inductive hypothesis when `pop` selects a label and either a new constraint is added (lines 63-67), or the label is already contained in the internal choice on the stack (lines 69-71). When a new label is added to the internal choice on the stack, the lemma is a straightforward consequence of the definition of subtyping for internal choice (the new internal choice is larger, therefore it is a subtype of the original one) and of inductive hypothesis.

Suppose that `pop` chooses a label from an external choice. If the clause at line 89 is called, then the lemma is proved straightforwardly by inductive hypothesis. If the clause at line 78 is called, then a new constraint on ψ_{ex} is added to C. Subsequently the clause at line 89 is called, and the lemma is proved as in the first case. If the clause at line 84 is called, then $\sum_{j\in(I_1J_1,I_2J_2)} ?L_j.\eta_j$ is a super-type of $\sum_{i\in(I_1,I_2I_3J_1J_2)} ?L_i.\eta_i$ by definition of sub-typing, since $I_1 \subseteq I_1J_1$ and the indexes in the former session are all contained in the indexes of the latter session. By definition, the clause at line 89 is called next, and the lemma is proved as in the first case.

□

Because of sub-typing, the structure of inferred sessions may change during inference (for example when a new label is added to an internal choice). Therefore we need to define the notion of *stack sub-typing*:

Definition 7.2.23 (Stack sub-typing)**.** *Let C be well-formed. A stack Δ_1 is a subtype of stack Δ_2, or $C \vdash \Delta_1 <: \Delta_2$, when the following relations are satisfied:*

$$C \vdash \epsilon <: \epsilon$$

$$C \vdash (l : \eta_1) \cdot \Delta_1 <: (l : \eta_2) \cdot \Delta_2 \qquad \text{if } C \vdash \eta_1 <: \eta_2 \text{ and } C \vdash \Delta_1 <: \Delta_2$$

The following theorem, also known as Liskov's substitution principle in [Vasconcelos et al., 2006], states that if a configuration $\Delta \vDash b$ can take a transition, then for any Δ' sub-stack of Δ, $\Delta' \vDash b$ can also take a transition. Substitutions also propagate over transitions:

Theorem 7.2.24 (Liskov's substitution principle)**.** *Let $C_2 \vdash C_1\sigma$, $C_2 \vdash \Delta_2 <: \Delta_1\sigma$ and $C_2 \vdash$ confined(Δ_2). If $\Delta_1 \vDash b \rightarrow_{C_1} \Delta_1' \vDash b'$, then:*

1. $\Delta_2 \vDash b\sigma \rightarrow_{C_2} \Delta_2' \vDash b'\sigma_1$
2. $C_2 \vdash \Delta_2' <: \Delta_1'\sigma$
3. $C_2 \vdash$ *confined*(Δ_2')

Proof. By rule induction. The proof of 3 is a trivial consequence of the hypothesis $C_2 \vdash$ *confined*(Δ_2), since the continuation of a confined session is itself confined. We only prove 1 and 2:

Case End**:** Suppose that $(l : \mathsf{end}) \cdot \Delta_1 \vDash b \rightarrow_{C_1} \Delta_1 \vDash b$. By definition of substitution $((l : \mathsf{end}) \cdot \Delta_1)\sigma = (l : \mathsf{end}) \cdot \Delta_1\sigma$; by definition of stack sub-typing the hypothesis $C_2 \vdash \Delta_2 <: ((l : \mathsf{end}) \cdot \Delta_1)\sigma$ implies that $\Delta_2 = (l : \mathsf{end}) \cdot \Delta_2'$ for some Δ_2' such that $C_2 \vdash \Delta_2' <: \Delta_1\sigma$. Rule End yields $\Delta_2 \vDash b\sigma \rightarrow_{C_2} \Delta_2' \vDash b\sigma$, which proves the proposition together with $C_2 \vdash \Delta_2' <: \Delta_1\sigma$

Case Beta**:** Let $\Delta_1 \vDash \beta \rightarrow_{C_1} \Delta_1 \vDash b$, assuming that $b \vdash \beta \subseteq C_1$. By definition of refinement, $C_2 \vdash C_1\sigma$ implies that $C_2 \vdash b\sigma \subseteq \beta\sigma$. Since $\beta\sigma = \beta$ by definition of inference substitution, $C_2 \vdash b\sigma \subseteq \beta$ holds, and therefore the proposition is proved by applying Rule Beta on $\Delta_2 \vDash \beta$.

Case Plus**:** Let $\Delta_1 \vDash b_1 \oplus b_2 \rightarrow_{C_1} \Delta_1 \vDash b_i$ with $i \in \{1, 2\}$. By definition of substitution $(b_1 \oplus b_2)\sigma = b_1\sigma \oplus b_2\sigma$. The proposition is proved by straightforward application of Rule Plus, which yields $\Delta_2 \vDash b_1\sigma \oplus b_2\sigma \rightarrow_{C_2} \Delta_2 \vDash b_i\sigma$.

Case Push**:** Let $\Delta_1 \vDash \mathtt{push}(l : \eta) \rightarrow_{C_1} (l : \eta) \cdot \Delta_1 \vDash \tau$ with $l \sharp \Delta$. By application of Rule Push, $\Delta_2 \vDash \mathtt{push}(l : \eta)\sigma \rightarrow_{C_2} (l : \eta\sigma) \cdot \Delta_2 \vDash \tau$ holds because $\mathtt{push}(l : \eta)\sigma = \mathtt{push}(l : \eta\sigma)$. By definition of subtyping $C_2 \vdash \eta\sigma <: \eta\sigma$ holds by reflexivity, and therefore $C_2 \vdash (l : \eta\sigma) \cdot \Delta_2 <: (l : \eta\sigma) \cdot \Delta_1\sigma$ holds, because $C_2 \vdash \Delta_2 <: \Delta_1\sigma$ holds by hypothesis; therefore the proposition is proved.

Case Out**:** Let $(l : !T.\eta) \cdot \Delta_1 \vDash \mathtt{pop}l!T_0 \rightarrow_{C_1} (l : \eta) \cdot \Delta_1 \vDash \tau$, with $C_1 \vdash$ *confined*(T_0) and $C_1 \vdash T_0 <: T$. By definition of substitution $((l : !T.\eta) \cdot \Delta_1)\sigma = (l : !T\sigma.\eta\sigma) \cdot \Delta_1\sigma$ holds. By definition of stack sub-typing, $C_2 \vdash \Delta_2 <: (l : !T\sigma.\eta\sigma) \cdot \Delta_1\sigma$ implies that $\Delta_2 = (l : !T'.\eta') \cdot \Delta_2'$ such that $C_2 \vdash T\sigma <: T'$ and $C_2 \vdash \Delta_2' <: \Delta_1\sigma$.

Since $C_2 \vdash C_1\sigma$ holds by hypothesis, then $C_1 \vdash T_0 <: T$ implies $C_2 \vdash T_0\sigma <: T\sigma$. By transitivity $C_2 \vdash T_0\sigma <: T\sigma$ and $C_2 \vdash T\sigma <: T'$ imply $C_2 \vdash T_0 <: T'$. Since $\mathtt{pop}l!T_0\sigma = \mathtt{pop}l!(T_0\sigma)$, an application Rule Out yields $(l : !T'.\eta') \cdot \Delta_2' \vDash \mathtt{pop}l!T_0\sigma \rightarrow_{C_2} (l : \eta') \cdot \Delta_2' \vDash \tau$, which proves 1. By definition of session sub-typing, the hypothesis $C_2 \vdash (l : !T'.\eta') \cdot \Delta_2' <: (l : !T\sigma.\eta\sigma) \cdot \Delta_1\sigma$ implies $\eta' \vdash \eta\sigma <: C_2$. We have

already proved that $C_2 \vdash \Delta_2' <: \Delta_1\sigma$ holds, therefore we can infer $C_2 \vdash (l : \eta') \cdot \Delta_2' <: (l : \eta\sigma) \cdot \Delta_1\sigma$, which proves 2.

Cases In, Del, Res**:** proved as in case Out, using the fact that sub-typing is covariant for In, Res, and it is contravariant for Del.

Case ICh**:** Let $(l : \bigoplus_{i\in I} !L_i.\eta_i) \cdot \Delta_1 \vDash \mathtt{pop} l!L_k \rightarrow_{C_1} (l : \eta_k) \cdot \Delta_1 \vDash \tau$ with $k \in I$. By definition of internal choice variables, configuration $(l : \bigoplus_{i\in I} !L_i.\eta_i) \cdot \Delta_1 \vDash \mathtt{pop}\rho!L_k$ is equivalent to $(l : \psi_{\mathtt{in}}) \cdot \Delta_1 \vDash \mathtt{pop}\rho!L_k$ with $C_1 \vdash \psi_{\mathtt{in}} \sim \bigoplus_{i\in I} !L_i.\eta_i$ for some $\psi_{\mathtt{in}}$. Since $C_2 \vdash C_1\sigma$ holds by hypothesis, by definition of constraint refinement there exists a session $\eta' = \bigoplus_{j\in J} !L_j.\eta_j'$ such that $C_2 \vdash \psi_{\mathtt{in}} \sim \eta'$ and $C_2 \vdash \bigoplus_{j\in J} !L_j.\eta_j' <: \bigoplus_{i\in I} !L_i.\eta\sigma_i$ hold, with $J \subseteq I$.

Since $J \subseteq I$, $k \in I$ implies $k \in J$; and since $(\mathtt{pop}\rho!L_k)\sigma = \mathtt{pop}\rho!L_k$, an application of Rule ICh yields $(l : \bigoplus_{j\in J} !L_j.\eta_j') \cdot \Delta_2' \vDash (\mathtt{pop}\rho!L_k)\sigma \rightarrow_{C_2} (l : \eta_k') \cdot \Delta_2' \vDash \tau$, which proves 1. By definition of session sub-typing $C_2 \vdash \bigoplus_{j\in J} !L_j.\eta_j' <: \bigoplus_{i\in I} !L_i.\eta_i$ implies $C_2 \vdash \eta_k' <: \eta_k$; since $C_2 \vdash \Delta_2' <: \Delta_1\sigma$ holds by hypothesis, then $C_2 \vdash (l : \eta_k') \cdot \Delta_2' <: \Delta_1\sigma$ holds too, which proves 2.

Case ECh**:** similar to the case for ICh. □

Lemma 7.2.25 (Soundness of sub)**.** *If* $\mathsf{sub}(\eta_1, \eta_2, C) = (\sigma_1, C_1)$*, then* $C_1 \vdash \eta_1\sigma_1 <: \eta_2\sigma_1$*.*

Proof. By structural induction on η_1. The structure of the proof is similar to the structure of the proof for Lem. 7.2.21.

When η_1 and η_2 have mismatching shapes, sub substitutes η_1 for η_2 or viceversa, and the lemma is proved straightforwardly by reflexivity of subtyping. When $\eta_1 = !T_1.\eta_1'$ and $\eta_2 = !T_2.\eta_2'$, C is expanded with the constraint $\{T_2 \subseteq T_1\}$. By structural induction η_1' is a subtype of η_2'; since C_1 contains the constraint $\{T_2 \subseteq T_1\}$, η_1 is a subtype of η_2 by definition of subtyping. The case for receive is similar; the cases for delegation and resume follow straightforwardly by structural induction. When η_1 is either an internal or external choice, the lemma follows by Lem. 7.2.21. □

Soundness of session inference depends on the following central result:

Theorem 7.2.26 (Soundness of Algorithm $\mathcal{MC}$)**.** *Let* C *be well-formed in* $\Delta \vDash b$*. If* $\mathcal{MC}(\Delta \vDash b, C, K) = (\sigma_1, C_1)$*, then* $\Delta\sigma_1 \equiv \Delta'$ *and* $\Delta' \vDash K[b]\sigma_1 \Downarrow_{C_1}$*.*

Proof. In order to account for K, the lemma is proved by induction on the lexicographic order given by the execution size $|\Delta\sigma_1 \vDash K[b]\sigma_1|_{C_1}$ and the length of K. Let us proceed by case analysis on $\mathcal{MC}(\Delta \vDash b, C)$.

- *-- remove terminated frames*

2 $\mathcal{MC}\big((l : \mathsf{end}) \cdot \Delta \vDash b,\ C,\ K\big) = \mathcal{MC}\big(\Delta \vDash b,\ C,\ K\big)$

This case follows directly from the inductive hypothesis.

- *-- $\mathcal{MC}$ terminates with behaviour τ*

5 $\mathcal{MC}\big(\Delta \vDash \tau,\ C,\ \epsilon\big) = (\sigma,\ C\sigma)$

6 if $\sigma = \mathsf{finalize}\ \Delta$

By definition of $\mathsf{finalize}$, a $\Delta\sigma$ is equivalent to the empty stack ϵ, since $\mathsf{finalize}$ succeeds only if it replaces all variables ψ in Δ with end, and only end sessions are left in it. The lemma is therefore proved by taking $\Delta' = \epsilon$, because $\epsilon \vDash \tau\sigma \Downarrow_C$ holds trivially.

- *−− pop a sub−behaviour from the continuation stack*

$\mathcal{MC}\big(\Delta \vDash \tau,\ C,\ b \cdot K\big) = \mathcal{MC}\big(\Delta \vDash b,\ C,\ K\big)$

By inductive hypothesis.

- *−− push a new frame on the stack*

$\mathcal{MC}\big(\Delta \vDash \mathtt{push}(l:\eta),\ C,\ K\big) = (\sigma_2\sigma_1, C_2)$
  if $(\sigma_1, \Delta_1) = \mathsf{closeFrame}(l, \Delta)$
  and $(\sigma_2, C_2) = \mathcal{MC}\big((l:\eta\sigma_1) \cdot \Delta_1 \vDash \tau,\ C\sigma_1,\ K\sigma_1\big)$

If this clause succeeds, then also the inner calls to closeFrame does, which implies $\Delta\sigma_1 \equiv \Delta_1$. Notice that Δ_1 contain no frame of the form $(l:\eta)$ for any η, i.e. $l \sharp \Delta_1$. Because of this and by taking $\Delta' = \Delta_1$, the LTS allows the following transition:

$$\Delta' \vDash K[\mathtt{push}(l:\eta)] \rightarrow_{C_2} (l:\eta) \cdot \Delta' \vDash K[\tau]$$

Since the inner call $\mathcal{MC}\big((l:\eta\sigma_1) \cdot \Delta_1 \vDash \tau,\ C\sigma_1,\ K\sigma_1\big)$ succeeds too, the lemma follows by inductive hypothesis.

- *−− send*

$\mathcal{MC}\big((l:\psi) \cdot \Delta \vDash \mathtt{pop}\rho!T,\ C,\ K\big) = (\sigma_2\sigma_1, C_2)$
  if $C \vdash l \sim \rho$
  and $\sigma_1 = [\psi \mapsto\ !\alpha.\psi']$ where α, ψ' fresh
  and $(\sigma_2, C_2) = \mathcal{MC}\big((l:\psi') \cdot \Delta\sigma_1 \vDash \tau,\ C\sigma_1 \cup \{T \subseteq \alpha\},\ K\sigma_1\big)$

Let us assume that the above clause has been used. Because the domain of inference substitutions is session variables ψ, in this case we have:

$$\big((l:\psi) \cdot \Delta\big)\sigma_2\sigma_1 \vDash K[\mathtt{pop}\rho!T]\sigma_2\sigma_1 = (l:!\alpha.(\psi'\sigma_2)) \cdot \Delta\sigma_2\sigma_1 \vDash (K\sigma_2\sigma_1)[\mathtt{pop}\rho!T]$$

From the algorithm we have that $C \vdash l \sim \rho$ holds. The inner call to $\mathcal{MC}$ has $C_1 = C\sigma_1 \cup \{T \subseteq \alpha\}$ as input constraints set. By Lem. 7.2.22, $C_2 \vdash C_1\sigma_2$, and therefore $C_2 \vdash (T \subseteq \alpha)\sigma_2$, which implies $C_2 \vdash T \subseteq \alpha$ because σ_2 is an inference substitution. Since $C \vdash l \sim \rho$ and $C_2 \vdash T \subseteq \alpha$ both hold, the abstract interpretation semantics allows the following transition:

$$(l:!\alpha.(\psi'\sigma_2)) \cdot \Delta\sigma_2\sigma_1 \vDash (K\sigma_2\sigma_1)\mathtt{pop}\rho!T_c \rightarrow_{C_2} (l:\psi'\sigma_2) \cdot \Delta\sigma_2\sigma_1 \vDash (K\sigma_2\sigma_1)[\tau]$$

By inductive hypothesis, $(l:\psi'\sigma_2) \cdot \Delta\sigma_2\sigma_1 \vDash (K\sigma_2\sigma_1)[\tau] \Downarrow_{C_2}$ holds. Because of this, and since $(l:!\alpha.(\psi'\sigma_2)) \cdot \Delta\sigma_2\sigma_1 \vDash (K\sigma_2\sigma_1)\mathtt{pop}\rho!T_c \rightarrow_{C_2} (l:\psi'\sigma_2) \cdot \Delta\sigma_2\sigma_1 \vDash (K\sigma_2\sigma_1)[\tau]$, it follows that $\big((l:\psi) \cdot \Delta\big)\sigma_2\sigma_1 \vDash K[\mathtt{pop}\rho!T]\sigma_2\sigma_1 \Downarrow_{C_2}$ holds, which proves the lemma.

- The other cases for $\mathtt{pop}\rho!T$, $\mathtt{pop}\rho?T$, $\mathtt{pop}\rho!L_k$, $\mathtt{pop}\rho?L_k$, $\mathtt{pop}\rho!\rho_d$ and $\mathtt{pop}\rho?l_d$ are all proved similarly, except for this case:

$\mathcal{MC}\big((l:!\eta_d.\eta) \cdot (l_d:\eta'_d) \cdot \Delta \vDash \mathtt{pop}\rho!\rho_d,\ C,\ K\big) = (\sigma_2\sigma_1, C_2)$
  if $C \vdash l \sim \rho$ and $C \vdash l_d \sim \rho_d$

```
39    and (σ1, C1) = sub(η'_d, η_d, C)
40    and (σ2, C2) = MC((l : η) · Δσ ⊨ τ, C1, Kσ1)
```

The only difference with the previous case is that we have to prove that $C' \vdash \eta_{d'}\sigma <: \eta_d\sigma$, which follows by Lem. 7.2.25.

- Suppose that the following $\mathcal{MC}$ clause has been applied:

```
96  -- sequencing
97  MC(Δ ⊨ b1 ; b2, C, K) = MC(Δ ⊨ b1, C, b2 · K)
```

The lemma is proved directly by inductive hypothesis.

- Suppose that the following $\mathcal{MC}$ clause has been used:

```
99  -- internal choice in the behaviour
100 MC(Δ ⊨ b1 ⊕ b2, C, K) = (σ2σ1, C2)
101   if  (σ1, C1) = MC(Δ ⊨ b1, C, K)
102   and (σ2, C2) = MC(Δσ1 ⊨ b2σ1, C1, Kσ1)
```

This case is proved by Lem. 7.2.22, inductive hypothesis and Prop. 7.2.24.

□

Soundness of Algorithm $\mathcal{SI}$ can be stated as follows:

Theorem 7.2.27 (Soundness of Algorithm $\mathcal{SI}$)**.** *Let C be a well-formed constraints set. If $\mathcal{SI}(b, C) = (\sigma_1, C_1)$, then $\epsilon \vDash b\sigma_1 \Downarrow_{C_1}$.*

Proof. The definition of Algorithm $\mathcal{SI}$ is:

```
1  SI(b, C) = (σ2σ1, C2)
2    if  (σ1, C1) = MC(ϵ ⊨ b, C, ϵ)
3    and (σ2, C2) = choiceVarSubst C1
```

By definition of choiceVarSubst, $C_2 = C_1\sigma_2$, which implies $C_2 \vdash C_1\sigma_2$. The theorem follows directly by soundness of $\mathcal{MC}$, i.e. Prop. 7.2.26, and by Lem. 7.2.22, since $\epsilon \equiv \epsilon$ holds. □

7.2.3 Completeness of Algorithm $\mathcal{SI}$

Lemma 7.2.28. *If $C \vdash C'\sigma$ and $\forall\psi.C \vdash \sigma(\psi) <: \sigma'(\psi)$, then $C \vdash C'\sigma'$.*

Proof. By structural induction on constraints over β, c, $\psi_{\mathtt{in}}$ and $\psi_{\mathtt{ex}}$. □

Lemma 7.2.29 (Completeness of sub)**.** *Let η and η' be well-formed in C and $C_0\sigma$. If $C \vdash \eta_1\sigma <: \eta_2\sigma$ and $C \vdash C_0\sigma$, then $\mathsf{sub}(\eta_1, \eta_2, C_0) = (\sigma_1, C_1)$ terminates, and there exists a substitution σ' such that $\forall\psi \in dom(\sigma).C \vdash \sigma(\psi) <: \sigma'(\sigma_1(\psi))$ and $C \vdash C_1\sigma'$.*

Proof. By structural induction on η_1. Let $\eta'_1 = \eta_1\sigma$ and $\eta'_2 = \eta_2\sigma$ be such that $C \vdash \eta'_1 <: \eta'_2$.

If η_1 is a variable ψ_1, then sub terminates at line 52 with $\sigma_1 = [\psi_1 \mapsto \eta_2]$ and $C_1 = C_0\sigma_1$, regardless of the shape of η_2. Moreover σ can be decomposed into $\sigma = [\psi_1 \mapsto \eta'_1] \cdot \sigma''$ for some σ''. The lemma

is proved by taking $\sigma' = \sigma''$, because $C \vdash \sigma(\psi) <: \sigma'(\sigma_1(\psi))$ holds for all $\psi \in dom(\sigma)$: for $\psi = \psi_1$ we have $\sigma(\psi_1) = \eta_1'$, which is a subtype of $\sigma'(\sigma_1(\psi)) = \sigma'(\eta_2) = \eta_2'$ by hypothesis; for any other ψ, $\sigma(\psi)$ and $\sigma'(\psi)$ are equal by definition. Moreover $C \vdash C_1\sigma'$ follows by Lem. 7.2.28.

If η_1 is a session of the form $!T_1.\eta_1''$, then either $\eta_2 = \psi_2$, or $\eta_2 = !T_2.\eta_2''$ with $C \vdash T_2 <: T_1$ holding by definition of subtyping. The first case is proved similarly to the case for $\eta_1 = \psi_1$, but using the clause at line 53. The second case is proved by inductive hypothesis, since sub in this case only adds a new constraint $\{T_2 \subseteq T_1\}$ to C_0 in the clause at line 5. The proof for the case $\eta_1 = ?T_1.\eta''$ is proved similarly at line 8, considering that session sub-typing is covariant instead of contravariant. When $\eta_1 = !\eta_{1d}.\eta_1''$ and $\eta_2 \neq \psi_2$, the inner clauses at line 13 terminates by inductive hypothesis, and the lemma is proved similarly by transitivity of the session subtyping relation and of constraint refinement. Similarly, when $\eta_1 = ?\eta_{1r}.\eta_1''$ and $\eta_2 \neq \psi_2$, the clauses at line 16 terminate by inductive hypothesis and the proof is by inductive hypothesis and transitivity too. When $\eta_1 = \psi_{\texttt{in}1}$ and $\eta_2 \neq \psi_2$, the lemma is proved by a induction on the size of the internal choice indexes I_2, where $\eta_2 = \psi_{\texttt{in}2}$ and $\psi_{\texttt{in}2} \sim \bigoplus_{i \in I_2} !L_i.\eta_i$ holds. Similarly, when $\eta_1 = \psi_{\texttt{ex}1}$ and $\eta_2 \neq \psi_2$, then $\eta_2 = \psi_{\texttt{ex}2}$ and $\psi_{\texttt{ex}2} \sim \sum_{j \in (J_1, J_2)} ?L_j.\eta_j$ hold, and the proof follows by induction on the size of $J_1 \cup J_2$ over the inner call to function f. □

The completeness of Algorithm $\mathcal{SI}$ relies on the completeness of Algorithm $\mathcal{MC}$, stated as follows:

Theorem 7.2.30 (Completeness of Algorithm $\mathcal{MC}$)**.** *Let $\Delta \vDash K[b]$ be well-formed in C and in $C_0\sigma$. If $(\Delta \vDash K[b])\sigma \Downarrow_C$ and $C \vdash C_0\sigma$, then there exists σ' such that $\mathcal{MC}(\Delta \vDash b, C_0) = (\sigma_1, C_1)$ terminates, $\forall \psi \in dom(\sigma).C \vdash \sigma(\psi) <: \sigma'(\sigma_1(\psi))$ and $C \vdash C_1\sigma'$*

Proof. The lemma is proved by induction of the lexicographic order between the execution size $|\Delta\sigma_1 \vDash K[b]\sigma_1|_{C_1}$ and length of a behaviour b (i.e. structural induction).

Base case. The base case is when the execution size is equal to 1, that is, when $[\![\Delta \vDash K[b]\sigma]\!]_C = \{\Delta \vDash K[b]\sigma\}$, and $\Delta \vDash K[b]\sigma \not\to_C$. Since $\Delta \vDash K[b]\sigma$ is strongly normalizing by hypothesis, then $\Delta = \epsilon$ and $K[b] = \tau$, which implies that $K = \epsilon$ and $b = \tau$. In such a case, the clause $\mathcal{MC}(\epsilon \vDash \tau, C_0, \epsilon) = (\sigma_{id}, C_0)$ at line 5 terminates trivially. The lemma is proved by taking $\sigma' = \sigma$, because $\sigma'(\sigma_{id}(\psi)) = \sigma'(\psi) = \sigma(\psi)$ for any ψ and therefore $C \vdash \sigma(\psi) <: \sigma'(\sigma_1(\psi))$ holds by reflexivity, and because $C \vdash C_0\sigma_{id} = C_0$ holds by hypothesis.

Inductive case. In the inductive case, the execution size $|\Delta\sigma \vDash K[b]\sigma|_C$ is greater than 1, and therefore there exists a configuration $\Delta' \vDash b'$ such that $\Delta\sigma \vDash K[b]\sigma \to_C \Delta' \vDash b'$ holds. It is easy to show that $\Delta\sigma \vDash K[b]\sigma \to_C \Delta' \vDash b'$ holds if and only if b is a pop, push We proceed by rule induction:

End: Suppose that Rule End has been used:

$$(l : \mathsf{end}) \cdot \Delta\sigma \vDash K[b]\sigma \to_C \Delta\sigma \vDash K[b]\sigma$$

There are two cases to consider: either $\Delta = (l : \mathsf{end}) \cdot \Delta'$, or $\Delta = (l : \psi) \cdot \Delta'$ and $\psi\sigma = \mathsf{end}$. In the first case, the clause $\mathcal{MC}\big((l : \mathsf{end}) \cdot \Delta \vDash b,\ C_0,\ K\big) = \mathcal{MC}\big(\Delta \vDash b,\ C_0,\ K\big)$ and the lemma is proved by the inductive hypothesis. In the second case, we need to show that $\mathcal{MC}\big((l : \psi) \cdot \Delta \vDash b,\ C_0,\ K\big)$ terminates; this can be proved by structural induction on $K[b]$. If $b = \tau$ and $K = \epsilon$, then $\mathcal{MC}$ terminates by applying finalize on Δ, and the proposition is proved as in the base case. If $b = \tau$ and $K \neq \epsilon$, then the proposition is proved by inductive hypothesis on the clause at line 9.

If $b = \texttt{push}(l : \eta')$, then closeFrame closes ψ in the clause at line 12, because a well-formed stack cannot push the same label l twice. In all other cases either the proposition follows by induction, or $K[b]$ must have the form of a pop operation. In the latter case, l cannot be contained in region ρ of the pop operation by well-formedness of the stack, and therefore the only applicable clause in the last one at line 120, that calls closeTop.

Push: Suppose that the following transition is taken:

$$\Delta\sigma \vDash (K[\texttt{push}(l : \eta)])\sigma \rightarrow_C (l : \eta\sigma) \cdot \Delta\sigma \vDash K[\tau] \qquad \text{if } l \sharp \Delta$$

We first need to show that $\mathcal{MC}\big(\Delta \vDash \texttt{push}(l : \eta),\ C_0,\ K\big)$ terminates. Suppose that the frames with closed sessions end are removed from the top of the stack, as in the case for End. The first clause that matches $\mathcal{MC}\big(\Delta \vDash \texttt{push}(l : \eta),\ C_0,\ K\big)$ is the one at line 12. By hypothesis $l \sharp \Delta$ holds, therefore the inner call to closeFrame can only return the identity substitution. Therefore $\sigma_1 = \sigma_{id}$ and $\Delta_1 = \Delta$ hold, and there exists a substitution σ'' such that the starting substitution σ can be split in the composition of σ_1 and σ'', i.e. $\sigma = \sigma''\sigma_1 = \sigma''$. The lemma is then proved by inductive hypothesis on the inner call to $\mathcal{MC}$ with the smaller configuration $(l : (\eta\sigma_1)\sigma'') \cdot (\Delta_1\sigma_1)\sigma'' \vDash \tau = (l : \eta\sigma) \cdot \Delta\sigma \vDash \tau$.

Out: Let $(\Delta \vDash K[b])\sigma = (l : !T.\eta\sigma) \cdot \Delta'\sigma \vDash \texttt{pop}\rho!T'$, and suppose that the following transition is taken:

$$(l : !T.\eta\sigma) \cdot \Delta'\sigma \vDash K[\texttt{pop}\rho!T'] \rightarrow_C (l : \eta\sigma) \cdot \Delta'\sigma \vDash K[\tau] \qquad \text{if } C \vdash \rho \sim l,\ \mathit{pure}(T'),\ T' <: T$$

We must consider two cases: $\Delta = (l : \psi) \cdot \Delta'$ and $\sigma(\psi) = !T.\eta$, or $(l : !T.\eta) \cdot \Delta'$.

In the first case the call to $\mathcal{MC}\big((l : \psi) \cdot \Delta' \vDash \texttt{pop}\rho!T',\ C_0,\ K\big)$ is matched by the clause at line 17, and it produces the substitution $\sigma_1 = [\psi \mapsto !\alpha.\psi']$ and a new constraint $T' \subseteq \alpha$ by definition, where ψ' and α' are fresh variables. By hypothesis σ has the form $[\psi \mapsto !T.\eta']$, and it can be decomposed into $\sigma'' = [\psi \mapsto !T.\psi']$ and $\sigma''' = [\psi' \mapsto \eta']$. The lemma is proved by applying the inductive hypothesis on the inner call to $\mathcal{MC}$ and by taking $\sigma' = \sigma'''[\alpha \mapsto T]$, since σ'' can be further decomposed into σ_1 and $[\alpha \mapsto T]$ and therefore the substitution $[\alpha \mapsto T]$ guarantees that $\sigma'\sigma_1 = \sigma$; therefore $\forall\psi \in \mathit{dom}(\sigma).C \vdash \sigma(\psi) <: \sigma'(\sigma_1(\psi))$ follows by reflexivity. Notice that $C \vdash C_1\sigma'$ holds too by definition of constraints refinement, because the new constraint $T' \subseteq \alpha\sigma$ becomes $T' \subseteq T$ when $\sigma' = \sigma'''[\alpha \mapsto T]$ is applied to it, and because $C \vdash T' <: T$ holds by hypothesis from the side-conditions on Rule Out.

If $\Delta = (l : !T.\eta) \cdot \Delta'$, then the call $\mathcal{MC}\big(\Delta \vDash \texttt{pop}\rho!T',\ C_0,\ K\big)$ is matched by the clause at line 22. The lemma is proved straightforwardly by taking $\sigma' = \sigma$, since Algorithm $\mathcal{MC}$ returns the identity substitution in this case.

In: This case is proved similarly to [Out], using the clauses at lines 26 and 31, recalling that session sub-typing is covariant instead of contravariant for inputs.

Del: This case is proved similarly to [Out], using the clauses at lines 35 and 40, together with Lem. 7.2.29 in the latter case. Notice that $l_d \sim \rho_d$ holds by hypothesis, and therefore the clause at

line 45 is not applicable.

Res: This case is proved similarly to [Out], using the clauses at lines 51 and 56. Notice that the clause at line 59 cannot be called, because the labels in the behaviour $b = \mathtt{pop}\rho?l_r$ match the label in the stack Δ by hypothesis.

Ich: Let $(\Delta \vDash K[b])\sigma = (l : \psi_{\mathtt{in}}) \cdot \Delta'\sigma \vDash K\sigma[\mathtt{pop}\rho!L_j]$, and suppose that the following transition is taken:

$$(l : \bigoplus_{i\in I} !L_i.\eta_i) \cdot \Delta' \vDash K\sigma[\mathtt{pop}\rho!L_k] \to_C (l : \eta_k) \cdot \Delta' \vDash K\sigma[\tau] \qquad \text{if } (j \in I),\ C \vdash \rho \sim l$$

There are two cases to consider: either $\Delta = (l : \psi) \cdot \Delta'$, or $\Delta = (l : \psi_{\mathtt{in}}) \cdot \Delta'$.

If $\Delta = (l : \psi) \cdot \Delta'$, then the only clause that matches the call to $\mathcal{MC}\big(\Delta \vDash b,\ K,\ C_0\big)$ is the one at line 64. Since $\psi\sigma = \psi_{\mathtt{in}}$, substitution σ can be decomposed as $\sigma = \sigma'[\psi \mapsto \psi_{\mathtt{in}}]$. Moreover $C \vdash \psi_{\mathtt{in}} \sim \bigoplus_{i\in J} !L_i.\eta_i$ and $k \in J$ hold by the side conditions of Rule Ich. Algorithm $\mathcal{MC}$ first produces the substitution $\sigma_1 = [\psi \mapsto \psi_{\mathtt{in}}]$, and introduces a new constraint $\psi_{\mathtt{in}} \sim \bigoplus_{i\in\{k\}} !L_i.\psi_i$ in C_0, where ψ_k is a fresh variable. Let $\sigma'' = \sigma'[\psi_k \mapsto \eta_k]\sigma_1$. Since $(l : \psi) \cdot \Delta' \vDash K\sigma''[\mathtt{pop}\rho L_k] \Downarrow_C$ holds by hypothesis, and since $((l : \psi_k) \cdot \Delta' \vDash K[\tau]) = (l : \eta_k) \cdot \Delta'\sigma'\sigma_1 \vDash K\sigma'\sigma_1[\tau]$ holds because ψ_k is a fresh variable, then $(l : \eta_k) \cdot \Delta'\sigma'\sigma_1 \vDash K\sigma'\sigma_1[\tau] \Downarrow_C$ holds too. Moreover, since $C \vdash C_0\sigma$ holds by hypothesis and $C \vdash \psi_{\mathtt{in}} \sim \bigoplus_{i\in J} !L_i.\eta_i$ where J contains k both hold by the side condition of Rule Ich, then $C \vdash C_0\sigma'' \cup \{\psi_{\mathtt{in}} \subseteq \psi_k\sigma''\}$ holds too, because by definition of constraint refinement and of session sub-typing $C \vdash \bigoplus_{j\in J} !L_j.\eta_j <: \bigoplus_{i\in\{k\}} !L_i.\eta_i$. Therefore the inductive hypothesis applies to the inner call of $\mathcal{MC}\big((l : \psi_k) \cdot \Delta' \vDash \tau,\ K,\ C_0 \cup \{\psi_{\mathtt{in}} \subseteq \psi_k\}\big)$, and which also proves the theorem, since ψ_k is not in the domain of σ.

In the case that $\Delta = (l : \psi_{\mathtt{in}}) \cdot \Delta'$, then C_0 must contain a constraint $\psi_{\mathtt{in}} \sim \bigoplus_{i\in I} !L_i.\eta_i$ by definition of constraint well-formedness. There are two sub-cases to consider: either k is in I, or it is not. In the former case the theorem is proved straightforwardly by inductive hypothesis on the inner clause at line 69. If k is not in I, then the only applicable clause of Algorithm $\mathcal{MC}$ is the one at line 73, which adds k to $\psi_{\mathtt{in}}$, and the theorem follows by inductive hypothesis on the inner call to $\mathcal{MC}$ with the extended internal choice.

Ech: Suppose that the following transition is taken:

$$(l : \sum_{i\in(I_1,I_2)} ?L_i.\eta_i) \cdot \Delta \vDash \sum_{j\in J} \mathtt{pop}\rho?L_j\,;b_j \to_C (l : \eta_k) \cdot \Delta \vDash b_k \qquad \text{if } k \in J,\ C \vdash \rho \sim l,\ I_1 \subseteq J \subseteq I_1 \cup I_2$$

The proof of this case is similar to the proof for Rule Ich: if $\Delta = (l : \psi) \cdot \Delta'$, then Algorithm $\mathcal{MC}$ terminates by applying the clause at line 78, which first creates a new constraint on $\psi_{\mathtt{ex}}$, and then recursively calls the clause at line 89, which is proved by inductive hypothesis and transitivity of session sub-typing. If $\Delta = (l : \sum_{i\in(I_1,I_2)} ?L_i.\eta_i) \cdot \Delta'$, then either the clause at line 84 is called, in case either some labels in J from the behaviour are missing from I_1 or I_2, or in case the active labels in I_1 are more than the labels in J; or the clause at line 89 is called, in

case the session type designated by $\psi_{\mathtt{ex}}$ contains all the labels Jin the behaviour, and the active labels I_1 are included in J. In the former case the constraint in the session type are adjusted appropriately and the theorem is proved as in the first case, because the clause at line 89 is the only clause that matches the adjusted session type. In the latter case the proof is by inductive hypothesis and transitivity as in the first case.

84 89

Spn: Suppose that the following transition is taken:

$$\Delta \vDash K[\mathtt{spawn}\, b] \rightarrow_C \Delta \vDash K[\tau] \qquad \text{if } \epsilon \vDash b \Downarrow_C$$

Assume that the frames with closed sessions `end` are removed from the top of the stack, as in the case for End, resulting in a stack Δ'. The clause $\mathcal{MC}\big(\Delta' \vDash \mathtt{spawn}\, b,\ K,\ C_0\big)$ is matched at line 105. Since $\epsilon \vDash b$ and $\Delta' \vDash K[\tau]$ are smaller configurations than $\Delta' \vDash \mathtt{spawn}\, b$, the theorem follows by inductive hypothesis and transitivity of session sub-typing.

Rec: This case is proved similarly to the case for Rule Spn, with the exception that the clause at line 110 is called, and that the environment C_0 is properly manipulated to swap the recursive constraint $\mathtt{rec}_b\, \beta \subseteq \beta$ with $\tau \subseteq \beta$.

Plus: Suppose that the following transition is taken:

$$\Delta \vDash K[b_1 \oplus b_2] \rightarrow_C \Delta \vDash K[b_i] \qquad \text{if } i \in \{1, 2\}$$

Assume that the frames with closed sessions `end` are removed from the top of the stack, as in the case for End. The only applicable clause in this case is the one at line 100. The inductive hypothesis can be applied directly on the first inner call to $\mathcal{MC}$, which implies that σ can be refined into $\sigma'\sigma_1$. By inductive hypothesis, $C \vdash \sigma(\psi) <: \sigma'(\sigma_1(\psi))$ holds for all ψ in the domain of σ. Notice that, by construction of algorithms `sub` and $\mathcal{MC}$, $\sigma'\sigma_1$ produces super types only in the case of delegation, therefore it is easy to show that $(\Delta \vDash K[b_1 \oplus b_2)\sigma'\sigma_1 \Downarrow_C$ holds as well. Therefore $(\Delta \vDash K[b_2)\sigma'\sigma_1 \Downarrow_C$ holds too, and the theorem is proved by the inductive hypothesis.

Beta: Suppose that the following transition is taken:

$$\Delta\sigma \vDash \beta\sigma \rightarrow_C \Delta\sigma \vDash b \qquad \text{if } C \vdash b \subseteq \beta$$

Since $\Delta \vDash \beta\sigma$ is well-formed in both C and $C_0\sigma$, variable β is constrained not only in C (by assumption $C \vdash b \subseteq \beta$ holds), but it is also constrained to some b' in $C_0\sigma$, i.e. $C_0\sigma \vdash b' \subseteq \beta\sigma$. By definition of constraint refinement, $C \vdash C_0\sigma$ implies that $b' = b\sigma$, because there is no sub-typing relation defined for behaviours, therefore $b\sigma$ and b' must be equal. The lemma is then proved directly by inductive hypothesis.

Assume that the frames with closed sessions `end` are removed from the top of the stack, as in the case for End. If the top of the stack in Δ is not `end`, the only applicable clause is the one at line

116, whereby $\mathcal{MC}(\Delta \vDash \beta,\ C_0,\ K) = \mathcal{MC}(\Delta \vDash b_0,\ C_0,\ K)$ with $b_0 = \bigoplus\{b_i \mid \exists i.\ (b_i \subseteq \beta) \in C\}$. By definition of $\mathcal{MC}$, the only clause that matches the inner call $\mathcal{MC}(\Delta \vDash b_0,\ C_0,\ K)$ is the clause for internal choice at line 100. Since the execution size of $\Delta\sigma \vDash \beta\sigma$ is greater than the execution size of each $\Delta\sigma \vDash b_i\sigma$ component, the proof follows by inductive hypothesis, in the same way as for Rule Plus.

Seq: Suppose that the following transition is taken:

$$\Delta \vDash K[b_1; b_2] \rightarrow_C \Delta' \vDash K[b_1'; b_2] \qquad \text{if } \Delta \vDash b_1 \rightarrow_C \Delta' \vDash b_1'$$

Assume that all closed end sessions are removed from Δ, resulting in Δ', as for the case End. The clause $\mathcal{MC}(\Delta' \vDash b_1; b_2,\ K,\ C_0)$ becomes $\mathcal{MC}(\Delta' \vDash b_1,\ b_2 \cdot K,\ C_0)$ at line 97, and the lemma is proved by inductive hypothesis, because the execution size of $\Delta' \vDash K[b_1; b_2]$ is equal to the execution size of $\Delta' \vDash b_2 \cdot K[b_1]$, but the size of b_1 is smaller than the size of $b_1; b_2$.

Tau: Suppose that the following transition is taken:

$$\Delta \vDash K[\tau; b] \rightarrow_C \Delta \vDash K[b]$$

Assuming that all closed session are removed from Δ as in the case for End and that a stack Δ' is returned, the clause at line 97 is called first, whereby b_2 is pushed on the stack K. Then the clause at line 9 is called recursively, whereby the τ behaviour is discarded and b_2 is popped back from the stack. The lemma follows by inductive hypothesis on the smaller configuration $\Delta' \vDash K[b_2]$.

□

Completeness for session type inference can be stated as follows:

Theorem 7.2.31 (Session type inference completeness)**.** *Let $\epsilon \vDash b\sigma^\star$ and $C^\star$ be well-formed. If $\epsilon \vDash b\sigma^\star \Downarrow_{C^\star}$ and $C^\star \vdash C$, then $\mathcal{SI}(b, C) = (\sigma_1, C_1)$ and there exists σ such that $C^\star \vdash C_1\sigma$ and $\forall\psi \in dom(\sigma^\star).C \vdash \sigma^\star(\psi) \subseteq \sigma\sigma_1(\psi)$.*

Proof. The proof follows by applying Proposition 7.2.30 on the configuration $\epsilon \vDash b$ under C first. Algorithm $\mathcal{MC}$ returns σ_1 and C_1 such that there exists a σ' such that $C \vdash \sigma^\star(\psi) \subseteq \sigma'\sigma_1(\psi)$ for any ψ in the domain of $\sigma^\star$, and $C \vdash C_1\sigma'$. Since the call to choiceSubst simply substitutes $\psi_{\texttt{in}}$ and $\psi_{\texttt{ex}}$ variables with their (unique) relative internal and external choices in C_1, the new substitution σ_2 and C_2 that this function returns does not change the typing of sessions, and therefore $\forall\psi \in dom(\sigma).C \vdash \sigma(\psi) <: \sigma'(\sigma_2(\sigma_1(\psi)))$ holds, and $C \vdash C_2\sigma'$ follows by $C \vdash C_1\sigma'$, which proves the lemma. □

7.3 Algorithm $\mathcal{D}$

The duality check algorithm, which we call Algorithm $\mathcal{D}$, takes the constraints set C calculated by the second stage, and returns a larger constraints set C'. As in Nielson& Nielson, the algorithm simply halts when a duality check fails, rather than throwing an exception.

$$
\begin{array}{lcll}
(\sigma, C \uplus \{\mathsf{end} \bowtie \mathsf{end}\}) & \hookrightarrow & (\sigma, C) & \\
(\sigma, C \uplus \{!T_1.\eta_1 \bowtie ?T_2.\eta_2\}) & \hookrightarrow & (\sigma, C \cup \{T_1 \subseteq T_2, \eta_1 \bowtie \eta_2\}) & \\
(\sigma, C \uplus \{!\eta_1'.\eta_1 \bowtie ?\eta_2'.\eta_2\}) & \hookrightarrow & (\sigma'\sigma, C' \cup \{\eta_1\sigma' \bowtie \eta_2\sigma'\}) & \text{if } (\sigma', C') = \mathsf{sub}(C, \eta_1, \eta_2) \\
(\sigma, C \uplus \{\bigoplus_{i \in I_0} !L_i.\eta_{1i} \bowtie \sum_{i \in (I_1, I_2)} ?L_i.\eta_{2i}\}) & \hookrightarrow & (\sigma, C \cup \bigcup_{i \in I_0} \{\eta_{1i} \bowtie \eta_{2i}\}) & \text{if } I_0 \subseteq I_1 \\
(\sigma, C \cup \{\psi_1 \bowtie \eta_2\}) & \hookrightarrow & (\sigma'\sigma, C') & \text{if } (\sigma', C') = \mathsf{expand}(C, \psi_1 \bowtie \eta_2) \\
(\sigma, C \cup \{\psi_{\mathtt{in}} \bowtie \eta_2\}) & \hookrightarrow & (\sigma'\sigma, C') & \text{if } (\sigma', C') = \mathsf{expand}(C, \psi_{\mathtt{in}} \bowtie \eta_2) \\
(\sigma, C \cup \{\psi_{\mathtt{ex}} \bowtie \eta_2\}) & \hookrightarrow & (\sigma'\sigma, C') & \text{if } (\sigma', C') = \mathsf{expand}(C, \psi_{\mathtt{ex}} \bowtie \eta_2)
\end{array}
$$

Fig. 7.1: Transition rules for Algorithm $\mathcal{D}$.

Algorithm $\mathcal{D}$ manipulates a new kind of constraints, called *duality constraints*. A duality constraint has the form $(\eta_1 \bowtie \eta_2)$, where η_1 and η_2 are inference session types. At the beginning, Algorithm $\mathcal{DC}$ creates a duality constraint $\eta_1 \bowtie \eta_2$ for each channel c and $\bar{c}$ such that session types η_1 and η_2 have been derived, i.e. such that $\{\eta_1 \subseteq c, \eta_2 \subseteq \bar{c}\} \in C$. For any other channel c such that $\{\eta_1 \subseteq c\}$ is in C, but no constraint $\{\eta_2 \subseteq \bar{c}\}$ is in C, Algorithm $\mathcal{DC}$ introduces a constraint $\eta_1 \bowtie \psi_2$, where ψ_2 is a fresh variable.

After this initial setup, Algorithm $\mathcal{DC}$ non-deterministically applies one of the following rules to the configuration (σ_{id}, C), until no more rules can be applied: where the helper function sub is the same helper function from Algorithm $\mathcal{SI}$ (which returns a substitution and a set of constraints such that the two input sessions are in the sub-typing relation).

The duality constraints are increasingly simplified, until no more simplifications are possible. After Algorithm $\mathcal{DC}$ is finished, the input set C fails the duality check when there exists a duality constraint $C\eta_1 \bowtie \eta_2$ such that neither η_1 or η_2 are fresh session type variables ψ, i.e. $\eta_1 \neq \psi_1$ and $\eta_2 \neq \psi_2$.

Algorithm $\mathcal{D}$ collects all $c \sim \eta_1$ and $\bar{c} \sim \eta_2$ constraints in C', generates duality constraints $\eta_1 \bowtie \eta_2$ and iteratively tries to discharge these constraints by applying the rules of Def. 6.4.11, possibly substituting variables with concrete terms. It ultimately returns a substitution σ and the set of constraints C'' such that C'' is a valid type solution according to Def. 6.4.10. For example, if $\eta_1 = !T_1.\eta_1'$ and $\eta_2 = ?T_2.\eta_2'$, the empty substitution and the constraints $\{T_1 \subseteq T_2, \eta_1' \bowtie \eta_2'\}$ are generated. When comparing internal and external choices, $\mathcal{D}$ checks that all the branches in the internal choice are included in the set of active branches in the external choice. When one of the two sessions is a variable ψ, the algorithm collects all constraints $\psi \bowtie \eta_i$ and calculates the least super-type of the dual of all η_i sessions. Algorithm $\mathcal{D}$ succeeds when no more simplification step can be taken, and only duality constraints among session variables remain. Soundness and completeness of Algorithm $\mathcal{D}$ is straightforward.

Chapter 8

Literature review

This chapter present an overview of the latest approaches to non-isolated, communicating transactions in the literature. The existing work in the literature can be categorized along three main approaches. The first one is a foundational approach, wherein communicating transactions constructs are investigated through process algebras, such as CCS, the π or Join calculus, in order to give a formal foundation to Service composition. Another approach is the extension of concurrent programming languages, such as CML, and concurrency models, such as the Actor model or the Join calculus, with transactional constructs, usually with the goal of making concurrent programming easier. Last is the attempt to extend the Software Transactional Memory (STM) model. By dropping the isolation requirement in a controlled manner, communication between transactions is allowed. The goal of this approach is to make concurrent programming easier as well, while exploiting previous research and implementation on STM.

We will discuss encodings of the Saturday Night Out problem, described in Chapter 1, for some of languages investigated in this survey. After having reviewed these approaches, we will draw parallels and differences among them, and we will present recurring properties in the literature that seem to characterise communicating transactions.

8.1 Process calculi

8.1.1 Overview

Many approaches to communicating transactions have been proposed in the area of process algebra. We can identify two main areas that motivate the need for transactions: service composition and modeling physical systems, mainly biochemical ones.

Ever since the advent of the Internet, an ever common trend that has been witnessed is the offer of computational services over the web. Web Services technology aims to facilitate the interoperability of such services in a network, which is often composed of very diverse systems that are not designed to cooperate with each other. It is generally useful to define new web services on top of smaller web services. This aggregated web service will have to execute all of its sub-services in order to be considered complete; if any of the sub-services fails or is not available, the aggregated service is not

valid and must annul all the partial work it completed thus far. We can see that these kinds of services enjoy some form of atomicity.

Particularly important in this regard is the concept of *long running transactions* (LRT). First introduced in [Garcia-Molina and Salem, 1987] (albeit under a different name), a long running transaction is a transaction whose execution takes a substantial amount of time, possibly on the order of hours or days, and that involves collaboration with distributed components. The cause of the lengthy duration might be the need to access to many external components, lengthy computations, interaction with human users, and so on. The target of [Garcia-Molina and Salem, 1987] was database applications, and LRTs cannot be implemented efficiently with traditional transactions because of the excessive latency. Traditional transactions need to maintain exclusive access of shared resources in order to preserve the semantic invariant of atomicity. It is not acceptable for this to happen for an extended period of time, because it penalizes other shorter-termed transactions, which end up aborting more frequently; LRTs themselves might struggle to have all the required resources at once.

The proposed solution to handle LRTs is to weaken the notion of atomicity from traditional ACID transactions. Rather than relying on atomicity to preserve database consistency, all long running transactions must specify a *compensation*. A compensation is an additional piece of code, whose purpose is to reestablish any invariant that might be violated during the execution of its respective transaction prior to commit. The first solution to handle LRTs in such a manner is [Garcia-Molina and Salem, 1987], *sagas*. LRTs are divided in smaller transactions, each representing a unit of work and each equipped with its own compensation. A saga comprises a top-level transaction that wraps the smaller unit of work, which are transactions as well. Sagas guarantee either the complete execution of all their nested transactions, or the execution of their compensations in reverse commit order, in case of aborted partial executions. For example, suppose that we need to book five seats in a flight reservation system. We can write a transaction that books a single seat, and whose compensation frees its reservation in the database. Such a saga will be an outer transaction containing five nested transactions, each booking a single seat. When running this saga, the flight reservation system will be able to run the five transactions concurrently with other sagas, rather than sequentially (i.e. getting exclusive access to the records of the whole flight, book five seats and then release access).

Many industrial proposal for web services, known as web service composition languages (WSCL), support long running transactions with varying forms of compensation (not necessarily sagas), such as WSDL, WSCI, WS-BPEL, WSFL, BizTalk, XLANG and BPMN. Unfortunately these proposals lack a foundational formal theory, and starting from the seminal work of [Garcia-Molina and Salem, 1987], many formal languages have been proposed to reason about LRTs. One such example among the first proposed formalisms is the πt-calculus from [Bocchi et al., 2003, Bocchi, 2006]. The πt-calculus extends the asynchronous π calculus with a transactional construct to handle arbitrarily nested LRTs (unlike sagas). LRTs are expressed by the $trans(P, F, B, C)$ construct, that monitors the execution of process P. Parent transactions collect compensations C from nested transactions in the failure bag B. In case of failure, a parent transaction executes the failure bag B first, followed by the failure manager F.

An important difference between sagas and the πt-calculus is that transactions in the πt-calculus

are allowed to interact with each other, whereas no notion of communication is present in sagas. This is to be expected, since sagas were designed to maintain consistency in database updates, rather than the coordination of distributed components, as in web services. In the πt-calculus, transactions are allowed to freely communicate with any other process. In case of aborts, asynchronous messages generated within a transaction are not removed, but are visible to other processes. As in the πt-calculus, StAC from [Butler and Ferreira, 2004, Butler et al., 2005] extends Communicating Sequential Processes [Hoare, 1978a], with primitives to install, remove and activate compensations. StaC allows arbitrary nesting of transactions too, and synchronization among processes via events, as is common in CSP. Web$\pi\infty$ [Mazzara and Lucchi, 2004], has a transaction model similar to that of the πt-calculus, but transactions are named. Using transaction names, it is possible for any process to abort another one. Web$\pi\infty$ has been used to effectively model the BPEL language. $dc\pi$ [Vaz et al., 2009], allows for dynamically extensible compensations, which are built progressively depending on the actions executed by sub-transaction in an LRT. An interesting example of a language for web services is cJoin from [Bruni et al., 2015, Bruni et al., 2002], which we will discuss in greater detail in Section 8.1.2.

In addition to soundly aggregating business services, such as web services in Service-Oriented Architectures (SOA), communicating transactions have found another interesting application in meaningfully modeling biological systems. In the biological and chemical world, some systems are naturally *reversible*: if a biochemical system transitions from a configuration G to a configuration G', which we can represent as $G \to G'$, the system can spontaneously perform the *reverse* transition $G' \rightsquigarrow G$ and return back to its original state. For example, [Danos and Krivine, 2007] describes the principle of *protein elasticity* in DNA, in the absence of particular system configurations such as complexation or activation, proteins can fold together and unfold back to their original form freely. In the presence of complexation and activation, the principle of *plasticity* operates, according to which proteins will *not* be able to revert back to their original form. The principles of elasticity and plasticity have a parallel in communicating transactions by the principles of transaction reversibility and irreversibility. As a further example, a fundamental mechanism called *toehold mediated branch migration* [Cardelli and Laneve, 2011a], a special DNA strand called *toehold* can displace another toehold within a double DNA strand. The operation releases the affected toehold, which in turn can displace the initial toehold itself, thus reverting the DNA strand back to its original form. Reversibility has been found to be useful to model quantum computing and software testing too [Phillips and Ulidowski, 2007].

The first formal model proposed for this kind of reversible systems is Reversible Communicating Concurrent Systems [Danos et al., 2004, Danos and Krivine, 2005](RCCS), where each CCS process can undo any of the interaction it has had with other partners at any time. In a followup work, this initial study was complemented with a notion of irreversibility within reversible transactions. We will describe in more detail RCCS in Section 8.1.3, but we will provide a short introduction to it here in order to discuss the subsequent work in the area. In Reversible CCS, each standard CCS process is equipped with a *memory*, which is a stack that records which actions the process has taken, and with whom. A *forward* transition will perform a standard CCS action and store the relative information in the memory; a *backward* transition will pop information from the memory stacks of the involved processes, and rebuild their original processes as if they had not communicated at all. A key

contribution in this work is the realization that reversible actions need only be *causally consistent*: past actions need not be reverted deterministically, undoing each single operation as it has been performed in a time line, but they can be undone non-deterministically, thus swapping some past transitions.

Phillips and Ulidowski describe an alternative technique to obtain reversible computations in [Phillips and Ulidowski, 2007]. Rather than equipping processes with memories to store their original form, evaluation in [Phillips and Ulidowski, 2007] does not consume processes, but just moves an "execution pointer" across the syntactical definition of a process, in order to keep track of which part of it is currently being executed. To clarify what we mean, consider the following CCS process $P = a.b + c$. Under CCS's operational semantics, P can perform either one of these two transitions, either $P \xrightarrow{a} b = Q$ or $P \xrightarrow{c} 0 = Q'$. Notice that the plus operator "+" disappears during the transition, and thus it is impossible to reverse Q or Q' to the original process P. Rather than substituting a process during evaluation, we can mark which branch is currently being evaluated, and which actions have been executed. For example, if we mark with an underscore sign the actions that have been executed, we can get the following alternative evaluations: $P \xrightarrow{a} \underline{a}.b + c$ and $P \xrightarrow{c} a.b + \underline{c}$. We now have enough information to revert the evaluation of a process, by just removing the marks in reverse order. This approach is not enough though. Consider the process $P = a \mid a$ and $Q = a.a$. Both processes can perform two a actions, but in P both actions can be reversed in any order, whereas in Q the second action a must be reverted before the first one (because the second term needs to revert from $\underline{a}.\underline{a}$ to obtain $\underline{a}.a$). To avoid this problem, past CCS prefixes are decorated with unique identifiers to distinguish them from other prefixes with the same name.

A further alternative is presented in [Cardelli and Laneve, 2011b, Cardelli and Laneve, 2011a] with the presentation of *reversible structures*. Reversible structures is a simple algebra based on asynchronous message passing. The basic units of computation are gates, which define sequences of input messages to capture, followed by sequences of output messages to release. Prefixes are coupled with a similar mechanism as in [Phillips and Ulidowski, 2007] based on unique IDs. All reductions on gates (i.e. consummation or production of messages) are reversible. With the addition of a parallel and restriction operator, reversible structure can encode CCS's choice operator and the Join calculus Join definitions; an encoding of the asynchronous RCCS in reversible structures is provided, together with a standardization lemma over weakly coherent structures (that is, structures in which all message IDs are unique), which allows reasoning about these structures without considering converse reductions. Reversible structures are used to model actual instances of biological systems, such as the toehold mediated branch migration.

Starting from CCS, the idea of reversible computations has been ported to different languages and extensions of CCS. A probabilistic analysis of a slight generalization of RCCS is carried out in [Bacci et al., 2011], in which a lower bound on energy costs that guarantee convergence to a successful state in finite time is proved in a probabilistic setting. Recent work from Lanese at el. has focused on extending the idea of reversible computations to more expressive languages, such as the Higher Order π-calculus [Lanese et al., 2013a], the addition of a finer grained control over reversibility [Lanese et al., 2010] and an extension to the Oz language with reversible computations in [Lienhardt et al., 2012]. A compositional semantics for Reversible Higher Order π is presented

in [Cristescu et al., 2013]. Reversible computations have also been embedded in a language for distributed systems with CSP-style communications in [Brown and Sabry, 2015].

Finally, we mention TransCCS, a language that, unlike cJoin and the πt-calculus, models perfect rollback rather than compensations; and, rather than reversible computations as in RCCS, features checkpointing, that is, reversibility to previous states of a program. We will discuss TransCCS in 8.1.4.

8.1.2 cJoin

The cJoin calculus from [Bruni et al., 2015] and [Bruni et al., 2002] is based on the Join calculus of [Fournet and Gonthier, 1996], a process calculus equivalent to the asynchronous π-calculus but designed to enforce a locality principle on extruded names, with the addition of a few well-disciplined primitives for LRTs. As mentioned in the introduction, this calculus is born in the context of web services composition and in particular in Service Oriented Architectures (SOA), whereby new components and applications (or services) can be developed by assembling existing ones. It does not model a specific web service industry proposal, but some of its roots can be found in the BizTalk language.

Transactions in cJoin are intended to describe the transactional interaction of several partners, under the assumption that any partner executing a transaction may communicate only with other transactional partners. In such a case, transactions run by other parties are bound to achieve the same outcomes (i.e. either they all succeed or all fail). Hence, a distinguishing feature of cJoin is that ongoing transactions can be merged to complete their tasks and when this happens either all succeed or all abort. Additionally, cJoin is based on compensations, i.e., partial executions of transactions are recovered by executing user-defined programs instead of providing automatic roll-back.

In addition to standard Join processes, cJoin provide a new kind of term, generally of the form $[P : Q]$, involving a process P that is required to be executed until completion and the corresponding compensation Q, to be executed in case P cannot complete successfully. Upon reaching a special process *abort*, a transaction is canceled and its compensation Q is released.

Note that the transactional primitive in cJoin relieves programmers from coding protocols needed to agree on a common result for a distributed transaction, while leaving to the programmer the responsibility for defining suitable compensations to recover aborted transactions, as is common with the compensation paradigm. Automatic perfect roll-back of a process Q can be encoded as the recursive transaction $P = [Q : P]$.

Several examples demonstrating the expressiveness of cJoin and a prototype language implementation based on the JoCaml compiler are provided in [Bruni et al., 2015]. The Saturday Night Out problem can be encoded in cJoin as shown in Figure 8.1.In this encoding, we simulate TransCCS's nondeterministic aborts by the introduction of $ab()$ messages, which can be consumed either to produce the *abort* process, or to commit the transaction if all its tasks T_i are performed.

8.1.3 Reversible CCS

Reversible Communicating Concurrent Systems [Danos et al., 2004], abbreviated RCCS, extends Milner's Calculus of Communicating Systems, or CCS [Milner, 1982], with backtracking. The operational

```
def  T1(w)|T1(w') ▶ w()|w'()
     T2(w)|T2(w') ▶ w()|w'()
     T3(w)|T3(w') ▶ w()|w'()  in
def  A() ▷ [def w() ▷ 0
                w'() ▷ 0
                ab() ▷ abort
                ab() | w() | w'() ▷ 0
           in dinner(w) | movie(w') | ab() : A() ]
     B() ▷ [def w() ▷ 0
                w'() ▷ 0
                ab() ▷ abort
                ab() | w() | w'() ▷ 0
           in dinner(w) | salsa(w') | ab() : A() ]
     C() ▷ [def w() ▷ 0
                ab() ▷ abort
                ab() | w() ▷ 0
           in movie(w) | ab() : A() ]
     D() ▷ [def w() ▷ 0
                w'() ▷ 0
                ab() ▷ abort
                ab() | w() | w'() ▷ 0
           in movie(w) | salsa(w') | ab() : A() ]
in A() | B() | C() | D()
```

Fig. 8.1: A solution to the Saturday Night Out problem in cJoin.

semantics of CCS is non-deterministic: whenever processes have partners with which they can synchronize, the semantics simply allows processes to pick a random partner among the many, without preferring a particular one. After such a choice is made, processes resume their evaluation.

Decisions made at non-deterministic choices cannot be undone. It might be the case that the systems evolves into an undesired state, whereas a different non-deterministic choice would have brought the system to a desired state. For instance, suppose that it is important for the system to perform an action ω. Consider the following scenario:

$$P = \bar{a}.0 \qquad Q = a.0 + a.\omega \qquad Sys = P \mid Q$$

In Sys, the first process P can non-deterministically synchronize with either the left or right branch of the choice in Q. If P synchronizes with the right branch, the system evolves into the desired state where it can perform action ω. But if P happens to synchronize with the left branch, the option to perform action ω is eliminated, and the system is deadlocked in an undesired state.

We can call such computations *forward* computations, where a CCS process evaluates from an initial state to a final state (or diverges). In the previous example it would be useful to *undo* the last choice of synchronization, if this turns out to be the wrong one. For example, if P and Q synchronize with each other, we would like the resulting system to be able to go back to state Sys and try again. We can call such a backtracking action a *backward* computation.

Reversible CCS provides support for backtracking and backward computations to CCS processes. In RCCS, each CCS process is equipped with a memory m that stores past actions that a process has performed, together with alternative choices that were available at the time of a forward computation. Without delving into the syntax and formal semantics of RCCS, let us revisit our previous

example with the addition of RCCS memories. Let us equip Sys with the empty memory $\langle\rangle$, and try to synchronize P and Q again:

$$\begin{array}{rl} & \langle\rangle \rhd Sys \\ \equiv & \langle 1\rangle \rhd \bar{a}.0 \mid \langle 2\rangle \rhd a.0 + a.\omega \\ \xrightarrow{\langle 1\rangle,\langle 2\rangle:\tau} & \langle 2, \bar{a}\rangle\langle 1\rangle \rhd 0 \mid \langle 1, a, a.\omega\rangle\langle 2\rangle \rhd 0 \end{array}$$

In RCCS, memories are stacks. The memories $\langle 1\rangle$ and $\langle 2\rangle$ indicate that the process $\langle 1\rangle P$ results from forking the Sys process to the left, whereas $\langle 2\rangle Q$ results from forking Sys to the right. In the final state of the system, the first memory contains the extra element $\langle 2, a\rangle\langle 1\rangle$, that records a communication over channel a with a process on the right. Interestingly, the other memory $\langle 1, a, a.\omega\rangle\langle 1\rangle$ records that Q synchronized over channel a with a process on its left, and that it chose to discard process $a.\omega$ at the same time.

The memories have enough information to reconstruct terms P and Q before they chose to synchronize on the left branch of Q, and choose again:

$$\begin{array}{rl} & \langle 2, \bar{a}\rangle\langle 1\rangle \rhd 0 \mid \langle 1, a, a.\omega\rangle\langle 1\rangle \rhd 0 \\ \xrightarrow{\langle 2,\bar{a}\rangle,\langle 1,a,a.\omega\rangle:\tau_*} & \langle 1\rangle \rhd a.0 \mid \langle 2\rangle \rhd a.0 + a.\omega \\ \xrightarrow{\langle 1\rangle,\langle 2\rangle:\tau} & \langle 2, \bar{a}\rangle\langle 1\rangle \rhd 0 \mid \langle 1, a, a.0\rangle\langle 2\rangle \rhd \omega \end{array}$$

Thanks to memories, we have been able undo the wrong choice, recover the system to a previous state and is now able to perform ω. The authors prove in [Danos et al., 2004] that backtracking in RCCS does not augment the expressive power of the language, that is, backward computations can only evaluate to states reachable using forward computations only.

Even though the system has reached a successful state, the system might still revert back to a previous state. To prevent further backtracking, RCCS processes can perform *irreversible* or *commit* actions, which are usual actions marked by an underscore, such as $\underline{\tau}$, $\underline{a}$, $\underline{b}$. Upon performing an irreversible action, an RCCS process empties its corresponding memory, thus barring it from reverting to previous states. Moreover processes which had previously interacted with it cannot backtrack either, since their memories now lack their complements, which have just been erased. Memories lacking a partner memory are called *locked* memories.

Because of the commit behaviour of RCCS, there is no simple encoding of the SNO example. In fact, any RCCS process can perform an irreversible action, regardless of agreement with other processes. In such a case, it is not possible for the partners to recover to a consistent state, since their memories will become locked. To overcome this problem, it is necessary to explicitly design a concurrent consensus algorithm; we will not investigate further into this encoding.

A proof method to reason on transactional RCCS processes is provided in [Danos and Krivine, 2005], based on a weak bisimulation relation between RCCS and CCS processes. An LTS representing a specification and an RCCS process are weakly Φ-bisimilar if a *causal encoding* relation can be established between the two. A causal encoding has a definition similar to bisimulation, but only relates LTS actions in Φ with irreversible RCCS actions, and is only defined on traces in *causal form*. Without entering into details, we can say that these are a kind of traces with mostly forward transitions and

whose reversibility is limited, so that the RCCS process does not backtrack excessively, and the specification cannot match the RCCS process anymore. Deadlocks and partial choice may happen in causal form traces, but they are not observable; only traces leading to an irreversible action are eventually observable.

8.1.4 TransCCS

TransCCS proposes a calculus to describe cooperation in distributed system that rely on automatic recovery in case of error or fault. Rather than installing *compensations* to counteract the side-effects occurred during the execution of a transaction as in cJoin (see Sec. 8.1.2), processes in TransCCS rely on *coordinated checkpointing*. Before executing a transaction or interacting with another process, a TransCCS process can store enough local information to be able to revert back to one of its previous states in case of fault. After several interactions, partners involved in a transaction can agree on a point beyond which they do not need to roll-back anymore, after which their interactions become definitive. TransCCS extends CCS with *communicating transactions*, that is, transactions that can interact with each other, and a mechanism, called *embedding*, to capture coordinated checkpointing in a disciplined way.

On top of standard CCS processes, TransCCS features transactions of the form $[\![P \rhd_k Q]\!]$, where P is the processes running within transaction k, and Q is the state to rollback to in case of failure. P can make transaction k definitive, or *commit* it, using the **co** k primitive . When a transaction commits, the alternative process Q is removed and only P remains. In TransCCS, failure is modeled by a spontaneous event that can occur at any time. When a transaction aborts, the running process P is removed and Q is run in its place.

Processes running within a transaction cannot communicate with processes outside of it. In order to allow communication, processes must be *embedded* in a transaction. When a process R is embedded, a copy of it is saved in Q, and a copy of it is run in parallel with P, as described by the following rule from the operational semantics of TransCCS:

$$R \mid [\![P \rhd_k Q]\!] \rightarrow [\![R \mid P \rhd_k R \mid Q]\!]$$

Once R is embedded in k, R and P can freely communicate with each other. R can be a transaction too, thus embedding can lead to nested transactions.

There is no complementary action to embedding: after being embedded in k, process R must wait for the transaction to either commit or abort to escape from it. Under this point of view, we can view embedding as a syntactical means to introduce dependencies between transactions. When several transactions need to interact, they have to be embedded into each other and become interdependent. Thanks to this mechanism, dependencies across transactions are manifest and their representation does not require a separate mechanism, such as a dependency graph.

Two *testing* theories [de Vries et al., 2010, De Vries et al., 2010] have been developed to study the behaviour of Communicating transactions. Testing theories in [Hennessy, 1988] study the behaviour of a process by observing its reactions to tests that interact with it. One setting, called *may testing*, focuses on whether a process might satisfy a given test; the other, called *fair testing*, studies whether

$$[\![\tilde{a}.\tilde{b}.\mathbf{co}\ alice]\!]_{alice} \mid [\![\tilde{a}.\tilde{c}.\mathbf{co}\ bob]\!]_{bob} \mid [\![\tilde{a}.\mathbf{co}\ carol]\!]_{carol} \mid [\![\tilde{b}.\tilde{c}.\mathbf{co}\ david]\!]_{david}$$

Fig. 8.2: A solution to the Saturday Night Out problem in TransCCS.

a process can always satisfy a test, excluding the case of divergence. In [De Vries et al., 2010] may testing is linked to *safety*, if we test a system with *failure tests*: we study systems that cannot satisfy these tests, so that "nothing bad will happen". Fair testing is linked to *liveness*, where we consider whether the system will eventually satisfy success tests, meaning that "something good will eventually happen". Both theories can be characterized by the two different notions of clean traces, which are exactly the sequences of actions that lead a transaction to a commit point, to the exclusion of non-committing traces. Communicating transactions are also formally proved to compose well under may and fair testing: given two processes P and Q with the same safe or live behaviour, it is possible to employ either P or Q interchangeably in a system without any noticeable behavioural difference.

A solution to the Saturday Night Out example is presented in Figure 8.2, which is essentially the same solution to the SNO presented in Sec. 1.2. We introduce two shortcuts in this solution for the sake of clarity. The first shortcut is *restarting transactions*, that is, transactions that are restarted in case of an abort:

$$[\![P]\!]_k \triangleq \mu X.[\![P \rhd_k X]\!]$$

We also define *synchronous channels*, that is, channels over which processes can synchronize with either the send or receive operator:

$$\tilde{c}.P \triangleq c.P + \bar{c}.P$$

8.2 Programming Language Extensions

8.2.1 Overview

It is common wisdom today that concurrent programming is difficult [Oram and Wilson, 2007]. Traditional solutions based on locks and monitors are often unsatisfactory, because they are very hard to reason about; thus programming with these mechanisms is very error prone. Much effort has been invested in finding new technique to alleviate these problems. Particularly interesting is the research of new programming language abstractions to express concurrent algorithms more easily and in a more modular fashion. Another thorny subject is error handling in distributed systems, where failures are more frequent and hard to recover from, as opposed to a non-distributed systems.

In the pursuit of abstractions to facilitate concurrent programming, CML [Reppy, 1999] introduced *events*. As a motivating example, suppose that we had to model a request/reply protocol between a client and a server. Suppose that the server side had the following ML implementation:

```
fun serverLoop () = if serviceAvailable()
```

```
then let
  val request = accept reqCh
  in
    send (replyCh, doIt request);
    serverLoop ()
  end
else doSomethingElse()
```

In this example, the server receives a request for a service from the `reqCh` channel, and provides a response on the `replyChy` channel. In order for this protocol to work, the client must behave exactly as the server expects. For example, a malicious client might send a request and never wait for the answer on channel `replyChy`. Because communication is synchronous in CML, the server waits indefinitely on the `send` operation, waiting for the client to synchronize. A standard solution in functional programming is to regulate the use of the `reqCh` and `replyCh` channels by encapsulating them into a function abstraction:

```
fun clientCall x = (send(reqCh, x); accept replyCh)
```

By limiting the scope in which these channels can be accessed to the abstraction, clients are guaranteed to respect the protocol. But this solution hides too much from the client. Suppose the server is not available. If the client tries to use this abstraction, it will get stuck on the first `send` operation. If the client had access to the actual `reqCh` channel, it might ping the server for its availability, or, if allowed by the programming language, perform *selective communication*, that is, communicate on either one of two (or more) channels, depending on which is available first.

The tension between abstraction on one hand, and selective communication on the other hand, is the main motivation behind Concurrent ML. In order to get the best of both worlds, CML introduces *events*. Events are the equivalent of lambda abstractions for sequential evaluation in the concurrent world. An event is a parametric type that describes a communication pattern, for example receiving a number over either a channel c_1 or c_2. A value of type event has no side effect until a special operator called `sync` is applied to it. On one hand, events provide ways to abstract communications; for example, in our client/server example, we might provide clients with an event that encapsulates a send followed by a receive. On the other hand, events are compositional, because, for example, the client can combine the event it gets from the server with another event of his own, and perform either one of them through a choice operator. The separation of communication pattern and actual execution is thus very beneficial. There are several other benefits as well, such as a clear semantics and an efficient implementation.

There are some limits to CML's power of abstraction though. As stated in Theorem 6.1 [Reppy, 1999], it is not possible to create an $n+1$-rendezvous abstraction from any given n-way rendezvous primitive (e.g. send and receive over channels). Moreover, it has been proved that CML events almost form a monad in [Jeffrey, 1997], but fail to do so. In order to overcome these problems, Transactional Events [Donnelly and Fluet, 2006] extends CML events with an ulterior sequencing operator `thenEvt`, that chains two concurrent events together into a single, combined one, which succeeds only if both sub-events synchronize successfully with other partners; otherwise the combined event fails. This new

combinator effectively increases the expressive power of CML, but at the cost of NP-Hardness. In fact, synchronizing transactional events becomes an NP-Hard problem, because an event scheduler has to exhaustively explore all possible event synchronizations beforehand in order to find a matching set of events. We will describe CML's events first, and then TE in more detail in Section 8.2.2.

This line of research has sparked interest in several directions. The type discipline imposed on Transactional Events does not allow side effects inside an event. For example it is not possible to read or write to cell references within a transactional event. Transactional Events for ML [Effinger-Dean et al., 2008] lifts the type discipline imposed in TE, and allows operations on ML references to freely mix with transactional events. Unlike TE, nested transactions are allowed, but they behave like a single, unique transaction. One of its main contributions is to provide a low-level semantics based on search threads for event synchronizations and separate heap search threads for memory references. The implementation is based on a chunking mechanism similar to the one used in Transactional Memory, which we will discuss later in Section 8.3.

In [Kehrt et al., 2009] transactional events are explored in a more concrete setting to find useful communication patterns in the programming practice, especially in client/server applications. Particularly interesting is a subtle difference in behaviour of CML's `wrap` and TE's `thenEvt` operators, because of the transactional flavour of transactional events. Consider the following program:

```
sync (sendEvt c1 4); sync (sendEvt c2 5)
```

where a client wants to first send a number on channel `c1`, and then another one in `c2`. Notice that two separate events are used; moreover, the first must be synchronized before evaluating the second one. Consider now the following server written in CML:

```
sync (chooseEvt
  (wrap (recvEvt c1) (fun x -> (x, sync (recvEvt c2))))
  (wrap (recvEvt c2) (fun x -> (sync (recvEvt c1), x))))
```

and the following server written in TE:

```
sync (chooseEvt
  (thenEvt (recvEvt c1)
    (fun x -> thenEvt (recvEvt c2) (fun y -> alwaysEvt (x,y))))
  (thenEvt (recvEvt c2)
    (fun y -> thenEvt (recvEvt c1) (fun x -> alwaysEvt (x,y)))))
```

Both servers describe a choice of two sub events, but the CML server can satisfy the client's request, because `wrap` can synchronize on its first `send` event, and then release a function to satisfy the second one. The TE server cannot do so, because the `thenEvt` creates a transaction that must be fully satisfied before it can commit. Thus it either performs both `recv` events in the same `sync` evaluation, or none of them. The client can only perform a single `send` operation inside its first `sync` operation; thus client and server cannot communicate because of the type discipline of TE events.

Recent work has studied fairness in transactional events [Amsden and Fluet, 2012]. The authors provide an instrumented semantics for Transactional Events, that is proved to be fair according

to the notion of *I/O* and *synchronization* fairness. Informally, a fair IO thread scheduler ensures that every all threads make progress, whereas a fair *synchronization* scheduler ensures that every thread trying to synchronize on an event often enough, will eventually do so. The proof is based on *action trace bisimilarity*, which is an equivalence based on trace inclusion. The authors also discuss an implementation based on *decidable synchronization groups*, that is, sets of events for which it is decidable to check whether they can be synchronized together. The implementation is only guaranteed to be fair in the presence of decidable synchronization groups, an assumption similar to *obstruction freedom* in STM (see Sec. 6 in [Amsden and Fluet, 2012]): a transaction executed in isolation will commit in a finite number of steps.

Concurrent programming is made difficult not only for its conceptual complexity, but also for the actual unreliability of distributed systems [Elnozahy et al., 2002b]. Large distributed systems are very common nowadays, for example web services and cloud computing, as has been argued in Section 8.1.1. However resilient single components may be, not all the components of a large distributed system can be trusted to be reliable (for example, web services provided by third parties), but might fail at any moment. We can distinguish between *permanent* and *transient* failures. A permanent failure is a failure from which recovery is impossible, such hardware failures. It is sometimes assumed that components subject to this kind of faults have access to a stable storage system, which is not subject to failure.

Transient faults are faults that are not permanent in nature, and might not occur again under different circumstances. A common cause of transient faults is resources unavailability. For example, a timeout exception might be raised during a channel operation, if the load on the communication network is too high, or by losing a race condition. Semantic inconsistency is another cause too, for example if serializability is violated in software transactional memory.

Failure management is thus another important aspect of distributed system, and a complicated one. On the one hand, dealing with errors programmatically can be cumbersome, for example when relying on time-outs (cfr. Section 3 in [Ziarek and Jagannathan, 2010]). On the other hand, the unreliable nature of distributed components might arise because of exceptional conditions that cannot be foreseen and thus adequately programmed for. So much so that many techniques have been developed to handle failure gracefully. A common technique to handle failure is *coordinated checkpointing*, (see [Elnozahy et al., 2002a] for a survey).

In coordinated checkpointing, processes periodically save *checkpoints*, that store local information about a process' state. In case of failure, processes can load up and resume one of their previous states and recover their previous execution. When restoring a checkpoint, it is important that all processes restore a *globally* consistent state of the system. In particular, a globally consistent state must guarantee that any side effect that occurred after a process has restored a local checkpoint, does not affect any other process. For example, suppose that process A saves a checkpoint, communicates with process B and a transient failure occurs on A. In order to have a globally consistent state, process B will have to restore a previous checkpoint too, otherwise it would witness an effect which has not taken place yet (i.e. the communication from A).

Stabilizers [Ziarek et al., 2006, Ziarek and Jagannathan, 2010] extend Concurrent ML with primitives to deal with transient faults through coordinated checkpointing. In particular, this language

extension provides three primitives: `stable`, `stabilize`, `cut`. Function abstractions enclosed in a `stable` section are automatically monitored, and checkpoints are automatically created whenever a communication or spawn operation is performed. When any of these checkpoints are created, each process also records a dependency with the other thread it is interacting with. All processes together form a *dependency graph* The `stabilize` keyword can be invoked within the `stable` section to activate checkpoint recovery. The dependency graph is examined, the inter-dependencies among the involved processes are examined, and a globally consistent checkpoint is calculated and restored. Note that failure conditions must be manually recognized in the monitored code according to this model. In order to limit the scope of rollback, the language provides a `cut` operation, after which it is impossible to revert a portion of stabilized code to its origin.

This last construct is useful to avoid problems such as the domino effect [Elnozahy et al., 2002b], but its resulting commit behaviour makes it difficult to express examples such as the SNO problem. For example, in the scenario presented in [Spaccasassi and Koutavas, 2013], if Carol performs a `cut` after having found a partner for dinner, Alice and Bob will not be able to abort their own transactions and will be stuck with an unsatisfactory agreement. Coordinated checkpointing is shown to greatly simplify error management in concurrent code, in particular in dealing with programmable failures such as timeouts in the Swerve server (cfr. Section 3 in [Ziarek et al., 2006]).

Transactors [Field and Varela, 2005, Lesani et al., 2009] extends the Actor model with a mechanism similar to Stabilizers, but deals with both transient and permanent failures. We will discuss transactors from [Field and Varela, 2005] in more detail in Section 8.2.3. A final interesting example is Reagents [Turon, 2012, Turon, 2013], a novel programming language that encompasses communication pattern abstractions à la Join calculus (see Section 8.1.2), coordinated checkpointing through optimistic and isolation for shared state operations, as in Transactional Memory (see Section 8.3). We will discuss Reagents in greater detail in Section 8.2.4.

8.2.2 Transactional Events

Concurrent ML [Reppy, 1999], or CML for short, is an extension to Standard ML that adds support for concurrency in a modular fashion, so that selective communication can be easily integrated with function abstractions in standard Standard ML. CML's concurrency model features synchronous communication over typed channels and a novel abstraction called *events*. An event describes a communication pattern expressed either by simple primitives, such as channel send and receive primitives, or complex events defined by event combinators, such as the non-deterministic choice between two events. Events can only be evaluated by the `sync` operator. If another thread is evaluating a matching event, then the two threads can synchronize. Otherwise, if there are not matching events, it is blocked. Under this light, we can understand that events only *describe* communication patterns in a modular way; events can be freely composed and decomposed, because their actual evaluation is deferred to the evaluation of a `sync` expression.

The simplest primitives that CML offers are `send` and `receive`, to send and receive values over typed channels. The following is a very simple communication example, in which a thread sends the unit value `()` to another thread:

```
let c = channel
in
  spawn ( fun _ => sync ( send (c, ()) ) );
  sync  ( receive c )
```

The `alwaysEvt` operator can always be matched by any event, whereas the `neverEvt` can never be matched by another event. The `choice` operator allows a thread to synchronize on either one of two events. The `wrap` operator takes an event e and a function f in input. Whenever an event e is synchronized on, `wrap` applies the result of the synchronization to function f. The whole compound event is considered synchronized as soon as the e event has synchronized. Notice that function f cannot backtrack anymore at this point, in case the value it has received was not expected, or in case it tries to synchronize on a second event unsuccesfully.

As stated already stated in Sec. 8.2.1, the three-way rendezvous cannot be expressed in CML as an event. In order to overcome this limitation of CML and retain CML's modularity, Donnelly and Fluett introduced the notion of transactional events; they added a new operator `thenEvt`, that, given two events e_1 and e_2, succeeds if and only if e_1 is synchronized first, and then e_2. Thanks to this construct, the language acquires a transactional flavour, since transactional events either either synchronize all of their sub-events, or none of them.

Thanks to this new construct, Transactional Events can express n-way rendezvous straightforwardly, along with other useful programming patterns, such as guarded inputs. It can also be shown that `thenEvt`, together with the other event combinators, confers a monad-with-plus structure to Transactional Events. Transactional Events are shown to form a monad-with-plus, a mathematical structure that facilitates the composition of transactional events.

The implementation of the language is considerably complicated by the introduction of the new transactional operator. A refined semantics is presented, adding *suspended threads* and *search threads* to the operational semantics of TE. When a `sync` is performed, execution threads become suspended threads, and a new search thread is spawned. Search threads perform the communications specified in the code. At any source of non-determinism, such as the choice operator, two or more search threads are spawned, one for each non-deterministic branch. When the search threads finds a sequence of communications, such that all participating threads can all match their respective events, that particular interaction is replicated on the previously suspended threads. As can be expected, the exhaustive exploration of process states is a sound approach, but inefficient. Unfortunately this issue is inevitable, as the authors prove that the problem of finding matching transactional events is an NP-Hard problem, since it can encode the 3-boolean satisfiability problem (cfr Section 5.4 in [Donnelly and Fluet, 2006]).

8.2.3 Transactors

As mentioned in the introduction, a common concern in distributed systems is the maintenance of distributed state, i.e. maintaining state consistency across multiple distributed components in a network. Transactors address this concern by modeling component failures and providing primitives to manage state persistence at the semantics level.

In the Actor model processes, called *actors*, communicate with each other via asynchronous message passing. Communication is not based on channels: actors can reference each other directly by name. Each actor has a unique id for its name, and a *mailbox*, which is a essentially a queue holding all received messages. An Actor processes mailbox messages in succession. Consumed messages are pattern-matched and processed according to user-defined case expressions.

Transactors extend the Actors model with persistent state and a set of operations for distributed checkpointing. Each transactor is equipped with a state S, in which it can store a value v through the primitive `setstate(`v`)`, and retrieve it through the primitive `getstate`. When Transactors interact, each Actor records a dependency from receiver to the sender, together with the sender's dependencies. Dependencies between transactors fall into three categories: message dependencies, which we have just discussed, state dependencies and creation dependencies. Message dependencies are promoted to state dependencies whenever a transactor modifies its own state after having received a message. If the state is not modified, the transactor will ignore any message dependency in case of rollback or checkpointing. Creation dependencies dispose of transactors spawned after a checkpoint.

In addition to state and communication operations, a transactor has three checkpointing operations: `stabilize`, `checkpoint`, and `rollback`. After invoking `stabilize`, a transactor commits not to modify its internal state anymore: any subsequent invocation of `setstate` will not modify the actual state S. This immutability guarantee is instrumental in the calculation of global checkpoints together with other partners. Stabilization can be thought of as the first phase of a two-phase commitment protocol. After stabilizing, a transactor can attempt to checkpoint its current state by invoking the `checkpoint` operation. This operation is only successful if the transactor has invoked stabilize, and the transactors it depends on have either stabilized or created a checkpoint. If successful, a transactor will be able to save its current state. Checkpointing can be thought of as the second phase of a two-phase commitment protocol.

After checkpointing, a transactor will be able to restore its last checkpointed state by invoking the `rollback` operation, and become available to process new messages in its mailbox. If no checkpoints are available, the `rollback` operation will make the transactor disappear. Node and network failures are modeled in the semantics as spontaneous rollbacks, i.e. transitions in the operational semantics of the Transactors that can be taken at any time by any term, irrespective of the current redex being evaluated. Such kind of spontaneous rollbacks bear the same effect as a rollback. The only visible effect from a failure is the loss of messages being processed by transactors. Messages sent before checkpointing are invalidated. Note that the rule to fire a spontaneous failures is only valid when a transactor has not stabilized yet; the authors assume that stable transactors only need to save the "program counter" of its current evaluation to persistent storage, which is not modeled (see §6.5 in [Field and Varela, 2005]).

Two fundamental results are proved for Transactors. Evaluation is divided in normal reduction steps for transactors, and spontaneous failure reduction steps. The first result is that, for any configurations k and k', such that k evaluates to k' through either normal reduction steps or failure steps, there is another sequence of evaluation steps from k to k' that does not use any failure reduction step. Thus it is possible to describe an equivalence between traces, much like *clean traces* in TransCCS (see [de Vries et al., 2010]), with the exception that roll-back is almost perfect, because the only visible

effect of a rollback is a potential loss of messages. Transactors cannot find a global checkpoint in the general case. For example, it is possible to write a malicious transactor that creates dependencies by sending messages, but never stabilizes. The second result is that, under an assumption of fairness, it is possible to devise an algorithm to find a global checkpoint that is guaranteed to terminate.

8.2.4 Reagents

Reagents provide two main primitive operations: updates to references to memory cells, and synchronous communication over endpoints. Together with these operators, there are a number of other "low-level" primitives to manipulate state and continuations. Reagents can be composed by a choice operator, that succeeds only if one of two specified reagents can be matched; and a conjunction operator, that succeeds only if both its operands succeed. Reagents can be sequenced, as in Transactional Events, thus forming a monad. Reagents can also be composed into *catalysts*. A catalyst is a reagent that is not consumed after a reaction, but is always available for further reactions. Thus catalysts are reminiscent of the **def** expressions in the Join calculus of [Fournet and Gonthier, 1996], and of the chemical abstract machine of [Berry and Boudol, 1990]. Finally, it is possible to specify post-commit actions, that are performed after a reaction has taken place. Post-actions are not required to be executed atomically with the former part of a reagent. These combinators are affirmed to produce both an *arrow* and *monad* structure.

Dependency conflicts introduced by the joint use of communication and state manipulation are not considered in [Turon, 2012]. Consider for example the case in which two reagents communicate with a swap channel, but then write on the same state location. The two reagents are required to commit together, because of the communication interaction. At the same time, their reaction cannot be committed, because overwriting the same reference cell breaks memory isolation. The implementation does not deal with these cases, but just throws an error. The author refers to the approach taken in [Lesani and Palsberg, 2011], according to which the reaction would be aborted and tried again.

Failure to build reagent reactions is automatically handled by the implementation. A reagent can fail either *transiently*, if it loses a race to obtain a reactant with another reagent, or *permanently*, if it needs to block because the required reactant is missing. Reagents are tentatively evaluated according to the reactions they offer and the reactants they need. While tentatively evaluating reagents, the underlying implementation builds a continuation for each partial successful reaction, until a special `Commit` continuation is reached. At this point, the involved reagents race to rescind they offer they have used, and try to perform a kCAS (Compare-And-Set) operation to finalize the reaction. Commit is thus coordinated.

8.3 Transactional Memory Extensions

8.3.1 Overview

Software Transactional Memory was first invented by Shavit and Touitou in [Shavit and Touitou, 1995]. The main idea is to treat shared memory in a concurrent system as a database. This idea sparked

a lot of interest in the scientific community in the last twenty years. In the programming languages area, a chief example of STM is its Haskell variant, Haskell STM from [Harris et al., 2005]. Haskell STM offers a modular abstraction for transactions that significantly simplifies concurrent programming, as exemplified in [Peyton-Jones, 2007]. There are some limits to transactional memory though. Since transactions are isolated, constructions that require transactions to communicate are not possible. For example, the authors of Haskell STM cite swap channels as something which cannot be done in Haskell; three-way rendezvous, barriers and other kind of rendezvous abstractions cannot be implemented with STM alone as well [Smaragdakis et al., 2007, Lesani and Palsberg, 2011].

This line of work has transactional memory with isolation as a starting point, and extends it with primitives for inter-transaction communication. The attractiveness of this solution lies in taking advantage of the results of prior research in STM, where well-understood transaction theories (e.g. correctness and opacity in [Guerraoui and Kapalka, 2008]) and industrial implementations are available.

An issue that is currently being tackled in this area is how to combine STM's isolation mechanisms with communicating transactions, which are not isolated. Section 8.3.2 describes in greater details one such approach, Communicating Memory Transactions (CMT) [Lesani and Palsberg, 2011]. Reagents and Transactional Events for ML are related to this line of research too, to the extent that they try to combine either Join patterns or Transactional Events with stateful computations.

TIC [Smaragdakis et al., 2007] proposes to extend isolated transactions with a Wait primitive. Code sections within a Wait are allowed to break free from isolation and interact with other threads or modify memory state. Transaction Communicators and Synchronizers [Luchangco and Marathe, 2011, Luchangco and Marathe, 2005] are very similar in spirit to Communicating Memory Transactions, but they work on a different level of granularity, e.g. objects rather than memory cells. OCTM [Miculan et al., 2015] is a recent language inspired by both CMT and the transaction merging mechanism of [Koutavas et al., 2014]. The goal of the language is to provide non-isolated transactional memory for *loosely coupled* processes, i.e. processes for which it is not immediate to orchestrate successful consensus groups. The expressiveness of the language is demonstrated by an encoding of $TCCS^m$ in OCTM.

8.3.2 Communicating Memory Transactions

Communicating Memory Transactions describe Software Transactional Memory that, in addition to the usual STM semantics for shared memory, allow transactions to communicate with each other through asynchronous message passing.

CMT extends traditional STM semantics [Koskinen et al., 2010] with asynchronous communication. The semantics is extended with a dependency graph, that tracks interactions among transactions via message-passing. Send and receive operations have the standard semantics. Aborting a transaction implies aborting its dependent transactions in the dependency graph. Particular care has to be devoted to commits and transaction dependencies. A transaction can be committed only if it belongs to a *cluster*. A cluster is a set of transactions that have reached the end of their atomic blocks, and that depend either on committed transactions or transactions of the same cluster. The last condition

```
let alice(dinnerSc, dinnerRc, movieSc, movieRc)
  atomic
    swap( d, dinnerSc, dinnerRc, () );
    swap( m, movieSc, movieRc, () )

let bob(dinnerSc, dinnerRc, salsaSc, salsaRc)
  atomic
    swap( d, dinnerSc, dinnerRc, () );
    swap( s, salsaSc, salsaRc, () )

let carol(movieSc, movieRc)
  atomic
    swap( m, movieSc, movieRc, () )

let david(movieSc, movieRc, salsaSc, salsaRc)
  atomic
    swap( m, movieSc, movieRc, () );
    swap( s, salsaSc, salsaRc, () )

let system =
  dinnerSc := newChan(); dinnerRc := newChan();
  movieSc  := newChan(); movieRc  := newChan();
  salsaSc  := newChan(); salsaRc  := newChan();

// spawn Alice, Bob, Carol and David
Thread.create( alice(dinnerSc, dinnerRc, movieSc, movieRc) );
Thread.create( bob(dinnerSc, dinnerRc, salsaSc, salsaRc) );
Thread.create( carol(movieSc, movieRc) );
Thread.create( david(movieSc, movieRc, salsaSc, salsaRc) );
```

Fig. 8.3: A solution to the Saturday Night Out problem in CMT.

prevents transaction scheduling from deadlock, in case the dependency graph created by message passing among transactions contains cycles (in which case, all the transactions involved in the cycle must belong to the same cluster).

The semantics of CMT is shown to preserve *opacity*, a notion of correctness for software transactional memory. Communications in CTM are shown to be safe, meaning that it is not possible to receive messages from aborted transactions. CMT transactions are also shown to be strictly more expressive than CML through a series of examples, as in Transactional Events. For example, the authors provide encodings for three-way rendezvous, synchronous queues and barrier abstractions.

For the sake of efficiency, the proposed implementation of CMT does not guarantee Transactional Event's completeness property, namely that if a successful interleaving of transactional events exists, then the scheduler will find it. On the one hand, the authors claim that termination of concurrent algorithms should not depend on scheduler implementation. On the other hand, they state that the nature of transaction scheduling is computationally too expensive to find an exact solution too.

CMT is implemented on top of the Scala Transactors library (not to be confused with the Transactors model described in [Field and Varela, 2005]). It extends previous transactional memory systems (TL2 and DSTM2) with message passing. Whenever a transactor receives a message, the receiving transactor stores a dependency link to the sending transactor. These dependency links form a dependency graph, which is calculated whenever a transaction must be aborted or committed; the set

of terminated transactions whose dependent transactions belong to that same set, is called a *cluster search*. An abort will cause dependent transactions to be aborted as well, whereas a transaction can commit only if all other transactions in its cluster search are either committed or trying to commit. Moreover, the transactions in the cluster search must also verify that write and read operation on transactional memory by the involved transactions do not invalidate each other, but that the order of commits is consistent. The implementation is shown to have little overhead and yield up to a thousand times better execution performance than Transactional Events.

The SNO problem can be solved in CMT by the code listing in Figure 8.3. Our solution is quite similar to the Transactional Events' one. We have used CMT's implementation of `swap` channels in [Lesani and Palsberg, 2011] to simulate synchronous communication; .

8.4 Conclusions

We have reviewed three areas where the concept of communicating transactions is investigated: the foundational approach, the programming language approach and the STM approach. Four properties of communicating transactions seem to recur in these approaches, and each property is characterized by one of two extremes. We identify the following characteristics:

1. **atomicity**: strong versus weak
2. **fault model**: spontaneous versus induced
3. **commit behaviour**: coordinated versus uncoordinated
4. **reversibility**: transitions versus states

The first property we consider is *atomicity*. A transaction is *atomic* if either all the computations it comprises are evaluated up to completion, or none of them are; no partial execution of an atomic transaction is allowed. Constructs supporting this property significantly simplify concurrent programming in nearly all the areas we have investigated in this chapter. While atomicity has clear advantages from a programmer's point of view, it involves many difficulties in practice, the main challenge being how to deal with side-effects. In the domain of databases, traditional ACID transactions write data to shared memory as side-effects. When a transaction is aborted, the side-effects must be nullified too, because otherwise memory would contain side-effects from a transaction that in principle has not started yet, and thus atomicity would be compromised.

In traditional transactions most mechanisms such as logging and journaling support this kind of atomicity. In communicating transactions it is not always practical to guarantee atomicity, depending on the application domain. For example, interactions between transactional partners introduce dependencies among them. If network delays are significant, resources used within a transaction might be blocked for too long and lead to inefficiencies in case of aborts. This is the case for sagas, as explained in Sec. 8.1.1. Therefore, usually transactional systems guarantee one of two different kinds of atomicity: *strong* and *weak* atomicity.

In case of abortion, a system that enjoys strong atomicity will automatically revert all side-effects that occurred during a transaction. Interacting partners are automatically rolled back as

well. Any asynchronous message that has been sent from a transactional process is usually either deleted or marked as invalid. These systems usually provide stronger transactional abstraction to programmers, often at the cost of a slower runtime environment. Strongly atomic systems might construct an intricate dependency graph, which is expensive to query. We can abscribe to this trend Reversible CCS in [Danos et al., 2004], TransCCS in [de Vries et al., 2010], Transactional Events in [Donnelly and Fluet, 2006] and Communicating Memory Transactions in [Lesani and Palsberg, 2011].

According to weak atomicity, not all side-effects are automatically reversed, but must be countered manually, if at all. For example, compensations are custom pieces of code that restore system invariants that might be violated during a transaction (recall the flight preservation system example for sagas in Sec. 8.1.1). Alternatively, we can say that transaction aborts are observable in a weakly atomic system. Some languages belonging to this second group are cJoin in [Bruni et al., 2015] and Transactors in [Field and Varela, 2005]. Implementations usually perform better, in no small part because there are fewer dependencies and side-effects to track in this case, but programmers must assume more responsibilities to preserve system invariants (e.g. with compensations).

The second property in systems supporting communicating transactions is the *fault model*, which describes how faults and errors are raised. We can distinguish two kind of fault models: spontaneous and induced. According to the *spontaneus* fault model, faults can occur at any time for no foreseeable motive, such as hardware failure, violated invariants in an information system or a particular biochemical configuration. In a wider sense, deadlocks can be considered failures too, thus deadlocked transactions might be aborted spontaneously, in order to restore the system to a consistent state. If deadlocks are taken into account, then the design of such systems must take into account deadlock detection, which is an NP-hard problem. This is the case for Transactional Events for example in Sec. 8.2.2.

There is usually no explicit construct to abort transactions for systems with the spontaneous fault model. Rather, the operational semantics of such languages provides non-deterministic means for transactions to either perform a computaion (e.g. internally or by interacting with other processes) or abort at any time. Example of languages with spontaneous aborts are Transactors from [Field and Varela, 2005], RCCS from [Danos et al., 2004], TransCCS from [de Vries et al., 2010] and CMT in [Lesani and Palsberg, 2011]. Transactional Events from [Donnelly and Fluet, 2006] can be abscribed to this paradigm too, although technically a transactional event nevers aborts in the first place, because it only interacts with perfectly matching partners.

In languages with the *induced* fault model assumption, unpredictable failure is not of concern. The erroneous states that the system reaches are under the control of the programmer, who can react to their occurrences and deal with them programmatically. For example, price disagreement for a service might lead a business transaction to abort in cJoin. In this case the language provides a primitive to explicitly abort a transaction. This construct is also the only means of aborting a transaction; thus deadlocked transactions cannot be unblocked. Theoretically, spontaneous failure can be simulated in this kind of languages as well, as we have shown for the SNO example in Sec. 8.1.2, although it is unclear what their actual performance in practice will be. Stabilizers in [Ziarek et al., 2006] and cJoin are examples of this fault model.

The third property is *commit behaviour*. Nearly all the languages we have reviewed agree that interactions between communicating transactions introduces dependencies among partners, so that if one transaction aborts, all the other transactions must abort too. While there is homogeneity in the case of aborts, there is a slight diversity in regards to committing a set of interdependent transactions.

On the one hand, commitment of transactions is interpreted as an agreement between partners, and thus the set of transactions is committed if and only if each transaction is ready to commit. This is the case for Cjoin, TransCCS, CMT and Transactional Events. On the other hand, commitment is interpreted as the safety point after which a process is not willing to roll back anymore, regardless of the state of its partners. In this case the language provides primitives to effectively cut dependencies among transactions, and allows a single partner to impose its own decision on other interacting partners. For example, in Stabilizers a thread can use the `cut` operator to actively ignore dependencies and refuse to backtrack to previous checkpoints. Similarly, a process performing an irreversible action in RCCS can generate locked memories, thus effectively disallowing another process to revert after that point.

Finally, the fourth property is *reversibility*. By reversibility we mean the method by which a language allows a system to undo side-effects and revert to previous configurations. Again, this property seems to depend on the application domain. One approach is *transition* reversibility, which makes program evaluation reversible. For example, the operational semantics of reversible structures in [Cardelli and Laneve, 2011b] contains rules for both forward and backward computations. Transition reversibility is a fine-grained kind of reversibility, since each single transaction can be back-tracked. Transaction reversibility is a pervasive feature of RCCS and reversible systems, and seems more appropriate to model inherently reversible systems such as the biochemical ones.

Another approach is *state* reversibility, which seems more grounded in the software engineering practice, such as web server design in [Ziarek et al., 2006]. A language with state reversibility provides specific primitives to capture a state of the system, along with other constructs to restore the system to such previously saved states. The reversibility of such languages seems in a sense superimposed by the addition of such primitives. State reversibility is a coarse-grained kind of reversibility, similar to the chuncking technique in Transactional Events for ML from [Effinger-Dean et al., 2008] and logging in STM [Harris et al., 2005]. It is less pervasive than transaction reversibility on the operational semantics, since all transactions are definitive. The system itself might be taken back to one of its previous states by peculiar constructs. Embedding in TransCCS and check-pointing in Stabilizers can be viewed as an example of this property.

Chapter 9

Conclusions

In this chapter we summarize the work presented in this thesis, and discuss in Section 9.1 some remaining open questions that would be interesting to address in future work.

Consensus is an often occurring problem in concurrent and distributed programming. The need for developing verification techniques and programming language support for consensus has already been identified in previous work on transactional events [Donnelly and Fluet, 2006], communicating memory transactions (CMT) [Lesani and Palsberg, 2011], transactors [Field and Varela, 2005] and *cJoin* [Bruni et al., 2015], as it has been observed in Chapter 8.

In Chapter 2 we have presented history bisimulations [Koutavas et al., 2014], a behavioural theory for communicating transactions based on bisimulation equivalence. However histories can grow indeterminately by design, and therefore they are not amenable to be computed algorithmically. In Chapter 3 we have shown how history bisimulations can be reformulated as *historyless* bisimulations. Instead of configurations with histories, historyless bisimulation relates $TCCS^m$ processes with the aid of a *dependency set* Δ, a binary relation between transaction names of the processes being compared. The dependency set ensures that not only actions are matched, but also the consensus groups are matched throughout a bisimulation game. Historyless bisimulation is proved to be equivalent to history bisimulation.

After replacing histories with the dependency set, we have provided a discipline for generating fresh transaction names in a deterministic fashion with the Nameless LTS in Chapter 4. Based on the syntactic restriction for serial CCS processes in [Milner, 1982], we have proved that the same syntactic restrictions identify a class of $TCCS^m$ processes which have finite-state LTSs by construction. We have finally provided an algorithm to calculate historyless bisimulation equivalences for finite-state space processes, based on the bisimulation algorithm of Sokolsky and Cleaveland in [Bergstra, 2001].

In Chapter 5 we have defined the operational semantics of TCML, a functional concurrent programming language inspired by CML and equipped with communicating transactions. We have showed how some of the examples from Chapter 1, namely the three-way rendezvous and the Saturday Night Out problem, can be implemented easily and modularly in TCML. We have described the architecture of a prototype implementation that allows transaction scheduling policies to be defined and modularly plugged into the prototype. We have explored several naïve scheduling policies, and we have demonstrated by experimental results that such policies do not scale well with the number of participants

in consensus groups, resulting in exponentially decreasing numbers of committing groups.

The experimental study of transaction scheduling policies motivated us to develop more sophisticated tools of static analysis of the underlying protocols that concurrent systems implement. We have introduced in Chapter 6 a new type and effect system for a concurrent functional language equipped with session types, called $\mathrm{ML_S}$ and inspired by [Amtoft et al., 1999]. The type system of $\mathrm{ML_S}$ is divided in two stages, one that describes the (concurrent) behaviour of a program, and one that verifies that such behaviour respects not only session types, but a stack-based discipline of sessions inspired by [Castagna et al., 2009] that guarantees a weak form of deadlock freedom, under some specific circumstances. The language and type system of $\mathrm{ML_S}$ allow for the definition of session type inference algorithm based, which is described in Chapter 7, which is proved to be sound and complete.

9.1 Future directions

On the theoretical side, it would be interesting to extend the bisimulation theory from flat transactions to nested transactions. This entails extending the transaction renaming mechanism to allow communication between inner and outer transactions, which can be allowed freely to communicate as in $\mathrm{TCCS^m}$. However such freedom introduces inter-dependencies across transactions that have an unclear semantics.

Suppose that the following transition is allowed:

$$[\![P_1 \mid [\![a.P_2 \rhd_{k_2} Q_2]\!] \rhd_{k_1} Q_1]\!] \mid [\![\bar{a}.P_3 \rhd_{k_3} Q_3]\!] \xrightarrow{l(\tau)}_{[k_2,k_3 \mapsto l]} [\![P_1 \mid [\![P_2 \rhd_l Q_2]\!] \rhd_{k_1} Q_1]\!] \mid [\![P_3 \rhd_l Q_3]\!]$$

Ideally, l can commit only if k_1 can commit, because otherwise inconsistencies can be introduced. For example, if l is committed and then k_1 is aborted, process P_3 and its previous communication over process a cannot be rolled back anymore. This behaviour is not intended to occur in communicating transactions. Similarly, aborting k_1 should trigger the abortion of l as well.

After a careful LTS has been established, the notions of bisimulation presented in Chapter 2 form a solid starting point to formulate proof methods for nested transactions. An interesting research question is then whether nested transactions are more expressive than flat transactions, or whether an encoding exists from nested to flat transactions (the opposite direction is trivial).

On the practical level, it would be interesting to rethink TCML as a concurrent functional language that includes both transaction renamings instead of embeddings, and session types. Transaction renamings would allow the transaction scheduler in Chapter 5 not to record a TTrie data structure. This has the advantage of not having to deal with a forest of alternatives, from which the best configuration has to be picked. This is a big penalty in terms of computation time. Instead of a TTrie, a simpler structure such as a list of active transactions would suffice.

At compilation time, the type and effect system presented in Chapter 6 provide both the behaviour of a program, and the protocol with which each process it spawns can communicate. A simple extension to this system is to add a communicating transactions *behaviour* whenever a transaction is encountered in the source code, together with a commit behaviour `co`.

Having an abstract view of transactions on the behaviour level, we can statically determine which

combination of transactions can reach a commit, according to a semantics similar to TCCS^{m} . The successful consensus groups can be calculated at compilation time, and encoded in a suitable data structure for the transaction scheduler to consume. At run time the scheduler will use this information as a guide to group transactions together. In the SNO example, the scheduler would know that the only successful grouping is *Alice* | *Bob* | *David*. The efficacy of this approach will then have to be tested with a new experimental study, and then compared with the results presented in Chapter 5. Another interesting venue of research is to derive consensus groups *heuristics* based on the behaviour. This approach might be more attractive when the consensus groups cannot be determined statically, such as when the number of participants is too high and the computation of the consensus groups is too expensive.

Further extensions to the session type discipline presented in Chapter 6 is also possible. More powerful forms of recursion are worth exploring; a mechanism to generate channel names dynamically, with constructs such as newChan from TCML; more powerful analysis of static endpoints, such as k-CFA, that allow the typing of more programs.

Appendix A

Proof of Type Preservation and Soundness for $\mathrm{ML_S}$

A.1 Preliminaries

To prove type soundness (progress and preservation) we first define the typed reductions of $\mathcal{S}$ configurations shown in Fig. A.1. Recall that we let $\mathcal{S}$ range over $\widetilde{(\Delta \vDash b,\ e)}$, write S for $\widetilde{e}$ when $\mathcal{S} = \widetilde{(\Delta \vDash b,\ e)}$, and identify $\mathcal{S}$ and S up to reordering. Here we write $\vec{b}$ for an arbitrary sequential composition of behaviours, which may be empty (ϵ). We also superscripts in sequences of terms to identify them (e.g., $\vec{\tau}^1$ may be different than $\vec{\tau}^2$), and we identify sequential compositions up to associativity and the axiom $\epsilon\,;b = b\,;\epsilon = b$.

In Sec. 6.4.1 we assumed that programs are annotated by unique region labels in a pre-processing step. This is necessary to achieve the maximum accuracy of our system (and reject fewer programs). Because beta reductions can duplicate annotations here we drop the well-annotated property. Type soundness for general annotated programs implies type soundness for uniquely annotated programs.

Lemma A.1.1 (Weakening)**.** *Suppose* $C;\ \Gamma \vdash e : TS \triangleright b$*. Then* $C;\ \Gamma \uplus \Gamma' \vdash e : TS \triangleright b$*.*

Proof. By induction on $C;\ \Gamma \vdash e : TS \triangleright b$. □

Lemma A.1.2 (Type Decomposition)**.** *Suppose* $C;\ \Gamma \vdash E[e] : TS \triangleright b$*. Then there exist* TS'*,* b'*,* $\vec{b}_{\mathsf{nxt}}$ *and fresh* x *such that* $b = \vec{\tau}\,;b'\,;\vec{b}_{\mathsf{nxt}}$ *and* $C;\ \Gamma \vdash e : TS' \triangleright b'$ *and* $C;\ \Gamma, x{:}TS' \vdash E[x] : TS \triangleright \vec{\tau}\,;\tau\,;b_{\mathsf{nxt}}$*.*

Proof. By structural induction on E. □

Lemma A.1.3 (Type Composition)**.** *Suppose* $C;\ \Gamma, x{:}TS' \vdash E[x] : TS \triangleright b$ *and* $C;\ \Gamma \vdash e : TS' \triangleright b'$*. Then there exists* $\vec{b}_{\mathsf{nxt}}$ *such that* $b = \vec{\tau}\,;\tau\,;\vec{b}_{\mathsf{nxt}}$ *and* $C;\ \Gamma \vdash E[e] : TS \triangleright \vec{\tau}\,;b'\,;\vec{b}_{\mathsf{nxt}}$*.*

Proof. By structural induction on E using Lem. A.1.1. □

Lemma A.1.4. *Suppose* $C \Vdash_{ws} \mathcal{S}, \big(\Delta \cdot (p^l : \eta) \cdot \Delta' \vDash b,\ e\big)$ *and* $C \vdash \eta <: \eta'$*. Then* $C \Vdash_{ws} \mathcal{S}, \big(\Delta \cdot (p^l : \eta') \cdot \Delta' \vDash b,\ e\big)$*.*

A.2 Preservation

Preservation relies on two lemmas, the first is that typed reductions of Fig. A.1 preserve well-typedness, well-stackedness and well-annotatedness; the other is that untyped reductions can be simulated by the typed reductions.

Lemma A.2.1. *Suppose $C \Vdash \mathcal{S}$ and $C \Vdash_{ws} \mathcal{S}$ and $\mathcal{S} \to^C \mathcal{S}'$. Then $C \Vdash \mathcal{S}'$ and $C \Vdash_{ws} \mathcal{S}'$.*

Proof. By induction on $\mathcal{S} \to^C \mathcal{S}'$ using the type composition and decomposition (Lem.(s) A.1.3 and A.1.2).

The most interesting case is that of delegation (TRDEL). In this case we have

$$\begin{aligned}\mathcal{S} = {} & \big((p^{l_1} : !\eta_d.\eta_1) \cdot (p'^{l'_1} : \eta'_d) \cdot \Delta_1 \vDash \vec{\tau}^1 ; \tau ; \tau ; \tau ; \mathtt{pop}\rho_1!\rho_d ; \vec{b}_{\mathsf{nxt1}},\ E_1[\mathtt{deleg}\ (p^{l_1}, p'^{l'_1})]\big),\\ & \big((\overline{p}^{l_2} : ?\eta_r.\eta_2) \vDash \vec{\tau}^2 ; \tau ; \tau ; \mathtt{pop}\rho_2?\rho_r ; \vec{b}_{\mathsf{nxt2}},\ E_2[\mathtt{resume}^{l_r}\, \overline{p}^{l_2}]\big),\ \mathcal{S}_0\\ & \to^C \big((p^{l_1} : \eta_1) \cdot \Delta_1 \vDash \vec{\tau}^1 ; \tau ; \vec{b}_{\mathsf{nxt1}},\ E_1[()]\big),\ \big((\overline{p}^{l_2} : \eta_2) \cdot (p'^{l_r} : \eta_r) \vDash \vec{\tau}^2 ; \tau ; \vec{b}_{\mathsf{nxt2}},\ E_2[p'^{l_r}]\big),\ \mathcal{S}_0 = \mathcal{S}'\end{aligned}$$

and by $C \Vdash \mathcal{S}$:

$$\begin{gathered}C;\ \emptyset \vdash E_1[\mathtt{deleg}\ (p^{l_1}, p'^{l'_1})] : T_1 \triangleright \vec{\tau}^1 ; \tau ; \tau ; \tau ; \mathtt{pop}\rho_1!\rho_d ; \vec{b}_{\mathsf{nxt1}}\\ C;\ \emptyset \vdash E_2[\mathtt{resume}^{l_r}\, \overline{p}^{l_2}] : T_2 \triangleright \vec{\tau}^2 ; \tau ; \tau ; \mathtt{pop}\rho_2?\rho_r ; \vec{b}_{\mathsf{nxt2}}\\ (p^{l_1} : !\eta_d.\eta_1) \cdot (p'^{l'_1} : \eta'_d) \cdot \Delta_1 \vDash \vec{\tau}^1 ; \tau ; \tau ; \tau ; \mathtt{pop}\rho_1!\rho_d ; \vec{b}_{\mathsf{nxt1}} \Downarrow_C\\ (\overline{p}^{l_2} : ?\eta_r.\eta_2) \vDash \vec{\tau}^2 ; \tau ; \tau ; \mathtt{pop}\rho_2?\rho_r ; \vec{b}_{\mathsf{nxt2}} \Downarrow_C\end{gathered}$$

By type decomposition Lem. A.1.2 and inversion on the rules of Fig. 6.3 and Fig. 6.4:

$$\begin{array}{ll}C;\ \emptyset \vdash \mathtt{deleg}\ (p^{l_1}, p'^{l'_1}) : \mathsf{Unit} \triangleright \tau ; \tau ; \tau ; \mathtt{pop}\rho_1!\rho_d & C;\ x{:}\mathsf{Unit} \vdash E_1[x] : T_1 \triangleright \vec{\tau}^1 ; \tau ; \vec{b}_{\mathsf{nxt1}}\\ C;\ \emptyset \vdash \mathtt{resume}\, \overline{p}^{l_2} : \mathsf{Ses}^{\rho_r} \triangleright \tau ; \tau ; \mathtt{pop}\rho_2?\rho_r & C;\ x{:}\mathsf{Ses}^{\rho_r} \vdash E_2[x] : T_2 \triangleright \vec{\tau}^2 ; \tau ; \vec{b}_{\mathsf{nxt2}}\\ (p^{l_1} : \eta_1) \cdot \Delta_1 \vDash \tau ; \vec{b}_{\mathsf{nxt1}} \Downarrow_C & C \vdash \rho_1 \sim l_1,\ \rho_d \sim l'_1,\ \eta'_d <: \eta_d\\ (\overline{p}^{l_2} : \eta_2) \cdot (p'^{l_r} : \eta_r) \vDash \tau ; \vec{b}_{\mathsf{nxt2}} \Downarrow_C & C \vdash \rho_2 \sim l_2,\ \rho_r \sim l_r\end{array}$$

Note that the transition rules considered in $\Downarrow_C$ do not take into account the concrete endpoints p, p' and $\overline{p}$—they are existentially quantified in these rules. By Lem. A.1.3 and the rules of Fig. 6.4:

$$\begin{array}{ll}C;\ \emptyset \vdash E_1[()] : T_1 \triangleright \vec{\tau}^1 ; \tau ; \vec{b}_{\mathsf{nxt1}} & (p^{l_1} : \eta_1) \cdot \Delta_1 \vDash \vec{\tau}^1 ; \tau ; \vec{b}_{\mathsf{nxt1}} \Downarrow_C\\ C;\ \emptyset \vdash E_2[p'^{l_r}] : T_2 \triangleright \vec{\tau}^2 ; \tau ; \vec{b}_{\mathsf{nxt2}} & (\overline{p}^{l_2} : \eta_2) \cdot (p'^{l_r} : \eta_r) \vDash \vec{\tau}^2 ; \tau ; \vec{b}_{\mathsf{nxt2}} \Downarrow_C\end{array}$$

Therefore $C \Vdash \mathcal{S}'$.

From $C \Vdash_{\mathtt{ws}} \mathcal{S}$ we deduce:

$$\begin{gathered}C \Vdash_{\mathtt{ws}} \mathcal{S}_0,\ \big((p^{l_1} : !\eta_d.\eta_1) \cdot (p'^{l'_1} : \eta'_d) \cdot \Delta_1 \vDash b_1,\ e_1\big),\ \big((\overline{p}^{l_2} : ?\eta_r.\eta_2) \vDash b_2,\ e_2\big)\\ C \Vdash_{\mathtt{ws}} \mathcal{S}_0,\ \big((p'^{l'_1} : \eta'_d) \cdot \Delta_1 \vDash b_1,\ e_1\big),\ \big(\epsilon \vDash b_2,\ e_2\big)\\ C \vdash !\eta_d.\eta_1 \bowtie ?\eta_r.\eta_2,\ \eta_1 \bowtie \eta_2 \qquad p, \overline{p} \sharp p', \Delta_1, \mathcal{S}_0 \qquad p' \sharp \Delta_1, \mathcal{S}_0\end{gathered}$$

where b_1, e_1, b_2 and e_2 are the appropriate behaviours and expressions shown above. Therefore, $C \vdash \eta'_d <: \eta_r$ and from Lem. A.1.4 we deduce

$$C \Vdash_{\mathtt{ws}} \mathcal{S}_0,\ \big(\Delta_1 \vDash b_1,\ e_1\big),\ \big((p'^{l'_1} : \eta'_d) \cdot \epsilon \vDash b_2,\ e_2\big)$$
$$\text{and}\ \ C \Vdash_{\mathtt{ws}} \mathcal{S}_0,\ \big((p^{l_1} : \eta_1) \cdot \Delta_1 \vDash b_1,\ e_1\big),\ \big((\overline{p}^{l_2} : \eta_2) \cdot (p'^{l_r} : \eta_r) \vDash b_2,\ e_2\big)$$

which completes the proof for this case. The rest of the cases are similarly proved. □

Lemma A.2.2. *Suppose $C \Vdash \mathcal{S}$ and $S \longrightarrow \widetilde{e'}$. There exists $\mathcal{S}'$ such that $\mathcal{S} \rightarrow^C \mathcal{S}'$ and $S' = \widetilde{e}$.*

Proof. By the definitions of the reduction relations ($\rightarrow^C$) and ($\longrightarrow$) in Fig. 6.1 and Fig. fig:typed-reductions. In this proof the structure of behaviours needed for establishing the ($\rightarrow^C$) reductions are deduced using Lem. A.1.2 and inversion on the typing rules; the necessary structure of the stacks is deduced by inversion on the rules of the $\Downarrow_C$ relation (Fig. 6.4). □

The proof of preservation (Fig. 6.4.14) is a direct consequence of the preceding two lemmas.

A.3 Type Soundness

We first extend the notion of internal and communication steps to the reductions of Fig. A.1.

Lemma A.3.1. *Let $\mathcal{S} \rightarrow^C \mathcal{S}'$.*

- *$\mathcal{S} \rightarrow^C_{\mathsf{c}} \mathcal{S}'$ if the transition is derived with use of the rules* TRInit, TRCom, TRDel, TRSel*;*
- *$\mathcal{S} \rightarrow^C_{\mathsf{i}} \mathcal{S}'$ otherwise.*

A diverging process is one that has an infinite sequence of internal transitions ($\longrightarrow_{\mathsf{i}}$).

Definition A.3.2 (Divergence)**.** *A process P diverges if $P \rightarrow^C_{\mathsf{i}} \mathcal{S}_1 \rightarrow^C_{\mathsf{i}} \mathcal{S}_2 \rightarrow^C_{\mathsf{i}} \mathcal{S}_3 \rightarrow^C_{\mathsf{i}} \ldots$. A system $\mathcal{S}$ diverges if for any $P \in \mathcal{S}$, P diverges.*

These transitions may spawn new processes, and divergence can result from infinite spawn chains.

Example A.3.3. *Let $P \stackrel{def}{=} \big(\epsilon \vDash b,\ (\mathtt{rec}\, f(x) \Rightarrow \mathtt{spawn}\, f)\,()\big)$ where $b \stackrel{def}{=} \tau\,;\tau\,;\mathtt{rec}_\beta\,(\tau\,;\mathtt{spawn}\,\beta)$; P is a diverging process.*

To prove progress we first divide a system into its diverging and non-diverging parts. The non-diverging part of the system can only take a finite number of internal transitions.

Lemma A.3.4. *Let $C \Vdash \mathcal{S}$ and $C \Vdash_{ws} \mathcal{S}$. Then $\mathcal{S} = \mathcal{D}, \mathcal{ND}$ and $\mathcal{D}$ diverges and for some $\mathcal{ND}'$, $\mathcal{ND} \rightarrow^{C\,*}_{\mathsf{i}} \mathcal{ND}' \not\rightarrow^C_{\mathsf{i}}$.*

Proof. By definition of diverging system (and its negation). □

The non-diverging part of the system that cannot take any more internal steps consists of processes that are values, or stuck on global channels or session primitives.

Lemma A.3.5. *Let $C \Vdash \mathcal{ND}$ and $C \Vdash_{ws} \mathcal{ND}$ and $\mathcal{ND} \not\rightarrow^C_{\mathsf{i}}$. Then $\mathcal{ND} = \mathcal{F}, \mathcal{W}, \mathcal{B}$ and:*

Processes in $\mathcal{F}$ are finished: $\forall\big(\Delta \vDash b,\ e\big) \in \mathcal{F}.\ \Delta = \epsilon, b = \tau, e = v.$

Processes in $\mathcal{W}$ wait on global channels: $\forall\big(\Delta \vDash b,\ e\big) \in \mathcal{W}.\ e = E[\texttt{req-}c^l]$ *or* $e = E[\texttt{acc-}c^l]$.

Processes in $\mathcal{B}$ block on a session primitive: $\forall P = \big(\Delta \vDash b,\ e\big) \in \mathcal{B}.\ e = E[e_0]$ *and* $e_0 \in \{\texttt{send}\ v, \texttt{recv}\ v, \texttt{deleg}\ v, \texttt{resume}\ v, \texttt{sel-}L\ v, \texttt{case}\ v\ \{L_i \Rightarrow e_i\}_{i\in I}\}$.

Proof. If a process in $\mathcal{ND}$ is not in one of the three categories then it would be able to take an internal step. Moreover, the structure of the stack Δ and behaviour b of finished processes follows from $C \Vdash \mathcal{ND}$ and $C \Vdash_{\texttt{ws}} \mathcal{ND}$. □

From the preceding two lemmas we can easily derive the most part of progress.

Corollary A.3.6. *Let $C \Vdash \mathcal{S}$ and $C \Vdash_{ws} \mathcal{S}$. Then $\mathcal{S} \longrightarrow_{\mathsf{i}}^* (\mathcal{F}, \mathcal{D}, \mathcal{W}, \mathcal{B})$ such that:*

Processes in $\mathcal{F}$ are finished: $\forall\big(\Delta \vDash b,\ e\big) \in \mathcal{F}.\ \Delta = \epsilon, b = \tau, e = v.$

Processes in $\mathcal{D}$ diverge: $\forall\big(\Delta \vDash b,\ e\big) \in \mathcal{D}.\ \big(\Delta \vDash b,\ e\big) \longrightarrow_{\mathsf{i}}^{\infty}$.

Processes in $\mathcal{W}$ wait on global channels: $\forall\big(\Delta \vDash b,\ e\big) \in \mathcal{W}.\ e = E[\texttt{req-}c^l]$ *or* $e = E[\texttt{acc-}c^l]$.

Processes in $\mathcal{B}$ block on a session primitive: $\forall P = \big(\Delta \vDash b,\ e\big) \in \mathcal{B}.\ e = E[e_0]$ *and* $e_0 \in \{\texttt{send}\ v, \texttt{recv}\ v, \texttt{deleg}\ v, \texttt{resume}\ v, \texttt{sel-}L\ v, \texttt{case}\ v\ \{L_i \Rightarrow e_i\}_{i\in I}\}$.

What is missing is that when $\mathcal{S}$ cannot take any more communication steps ($\Rightarrow_{\mathsf{c}}$) then all processes in $\mathcal{B}$ depend on processes in $\mathcal{D}$ and $\mathcal{W}$. This follows by well-stackedness of (well-typed) systems. We write $(\mathcal{F}, \mathcal{D}, \mathcal{W}, \mathcal{B})$ for a system whose finished processes are in $\mathcal{F}$, diverging processes are in $\mathcal{D}$, waiting processes are in $\mathcal{W}$ and blocked processes are in $\mathcal{B}$.

Lemma A.3.7. *Let $C \Vdash \mathcal{S}$ and $C \Vdash_{ws} \mathcal{S}$ and $\mathcal{S} = (\mathcal{F}, \mathcal{D}, \mathcal{W}, \mathcal{B})$ and $\mathcal{W}, \mathcal{B} \not\longrightarrow_{\mathsf{i}}$. Then*

1. *If $P, Q \in \mathcal{W}$ and $P \leftrightharpoons Q$ then $P, Q \longrightarrow_{\mathsf{c}} \mathcal{S}'$, for some $\mathcal{S}'$.*
2. *The $(\mapsto)$ dependencies in $\mathcal{S}$ create a directed acyclic graph.*
3. *If $P \in \mathcal{B}$ then there exist $Q, R \in \mathcal{D}, \mathcal{W}, \mathcal{B}$ such that $P \Mapsto (Q, R)$.*

Proof. Prop. 1 follows from $C \Vdash \mathcal{S}$ and $C \Vdash_{\texttt{ws}} \mathcal{S}$. Prop. 2 is proved by induction on $C \Vdash_{\texttt{ws}} \mathcal{S}$.

Prop. 3: Because $P \in \mathcal{B}$ and P is well-typed, the stack of P is non-empty. Thus, by Prop. 1, there exists $Q \in \mathcal{S}$ such that $P \mapsto^+ Q$ is the longest sequence of dependencies *without repetitions* (this is possible because of Prop. 2). We examine two cases:

- $P \mapsto^* P' \mapsto Q$ and the top-level frame in the stack of P' has an endpoint p and $\overline{p}$ appears in the top frame of Q; then $P \leftrightharpoons Q$; therefore $P \Mapsto (P', Q)$.
- $P = P_1 \mapsto \ldots \mapsto P_n \mapsto Q$ and the top-level frame in the stack of P' has an endpoint p and $\overline{p}$ appears in a frame other than the top one in Q; then there exists R such that $Q \mapsto R$ (by $C \Vdash_{\texttt{ws}} \mathcal{S}$); R cannot be one of the processes $P_1, \ldots, P_n$ because of Prop. 2. Moreover R cannot be a process in $\mathcal{F}$ (because processes in $\mathcal{F}$ have empty stacks due to typing), and R cannot be any other processes in $\mathcal{D}, \mathcal{W}, \mathcal{B}$ because the sequence of dependencies is the longest. Thus this case is not possible. □

Type Soundness is a direct consequence of Cor. A.3.6 and Lem. A.3.7.

$$\textsc{TREnd}\quad \frac{\big(\Delta \vDash b,\ e\big),\ \mathcal{S} \to^{C} \mathcal{S}'}{\big((l : \mathsf{end}) \cdot \Delta \vDash b,\ e\big),\ \mathcal{S} \to^{C} \mathcal{S}'} \qquad \textsc{TRBeta}\quad \frac{C \vdash b' \subseteq \beta \qquad \big(\Delta \vDash b[b'/\beta],\ e\big),\ \mathcal{S} \to^{C} \mathcal{S}'}{\big(\Delta \vDash b,\ e\big),\ \mathcal{S} \to^{C} \mathcal{S}'}$$

$$\textsc{TRIft}\quad \frac{}{\big(\Delta \vDash \vec{\tau}\,;\tau\,;(b_1 \oplus b_2)\,;\vec{b}_{\mathsf{nxt}},\ E[\mathtt{if\ tt\ then}\ e_1\ \mathtt{else}\ e_2]\big) \to^{C} \big(\Delta \vDash \vec{\tau}\,;b_1\,;\vec{b}_{\mathsf{nxt}},\ E[e_1]\big)}$$

$$\textsc{TRIff}\quad \frac{}{\big(\Delta \vDash \vec{\tau}\,;\tau\,;(b_1 \oplus b_2)\,;\vec{b}_{\mathsf{nxt}},\ E[\mathtt{if\ ff\ then}\ e_1\ \mathtt{else}\ e_2]\big) \to^{C} \big(\Delta \vDash \vec{\tau}\,;b_2\,;\vec{b}_{\mathsf{nxt}},\ E[e_2]\big)}$$

$$\textsc{TRLet}\quad \frac{}{\big(\Delta \vDash \vec{\tau}\,;\tau\,;\vec{b}_{\mathsf{nxt}},\ E[\mathtt{let}\ x = v\ \mathtt{in}\ e]\big) \to^{C} \big(\Delta \vDash \vec{\tau}\,;\vec{b}_{\mathsf{nxt}},\ E[e[v/x]]\big)}$$

$$\textsc{TRApp}\quad \frac{}{\big(\Delta \vDash \vec{\tau}\,;\tau\,;\tau\,;\vec{b}_{\mathsf{nxt}},\ E[(\mathtt{fun}\,x \Rightarrow e)\,v]\big) \to^{C} \big(\Delta \vDash \vec{\tau}\,;\vec{b}_{\mathsf{nxt}},\ E[e[v/x]]\big)}$$

$$\textsc{TRRec}\quad \frac{}{\begin{array}{l}\big(\Delta \vDash \vec{\tau}\,;\tau\,;\tau\,;\mathtt{rec}_{\beta}\,b\,;\vec{b}_{\mathsf{nxt}},\ E[(\mathtt{rec}\,f(x) \Rightarrow e)\,v]\big)\\ \to^{C} \big(\Delta \vDash \vec{\tau}\,;b[\mathtt{rec}_{\beta}\,b/\beta]\,;\vec{b}_{\mathsf{nxt}},\ E[e[\mathtt{rec}\,f(x) \Rightarrow e/f][v/x]]\big)\end{array}}$$

$$\textsc{TRSpn}\quad \frac{}{\big(\Delta \vDash \vec{\tau}\,;\tau\,;(\mathtt{spawn}\,b)\,;\vec{b}_{\mathsf{nxt}},\ E[\mathtt{spawn}\,v]\big),\ \mathcal{S} \to^{C} \big(\Delta \vDash \vec{\tau}\,;\tau\,;\vec{b}_{\mathsf{nxt}},\ E[()]\big),\ \big(\epsilon \vDash \tau\,;\tau\,;b,\ v\,()\big),\ \mathcal{S}}$$

$$\textsc{TRInit}\quad \frac{p, \overline{p} \sharp E_1, E_2, \mathcal{S}, \Delta_1, \Delta_2}{\begin{array}{l}\big(\Delta_1 \vDash \vec{\tau}^{1}\,;\tau\,;\tau\,;\mathtt{push}(l_1 : \eta_1)\,;\vec{b}_{\mathsf{nxt1}},\ E_1[\mathtt{req}\text{-}c^{l_1}\,()]\big),\\ \big(\Delta_2 \vDash \vec{\tau}^{2}\,;\tau\,;\tau\,;\mathtt{push}(l_2 : \eta_2)\,;\vec{b}_{\mathsf{nxt2}},\ E_2[\mathtt{acc}\text{-}c^{l_2}\,()]\big),\ \mathcal{S}\\ \to^{C} \big((p^{l_1} : \eta_1) \cdot \Delta_1 \vDash \vec{\tau}^{1}\,;\tau\,;\vec{b}_{\mathsf{nxt1}},\ E_1[p^{l_1}]\big),\ \big((\overline{p}^{l_2} : \eta_2) \cdot \Delta_2 \vDash \vec{\tau}^{2}\,;\tau\,;\vec{b}_{\mathsf{nxt2}},\ E_2[\overline{p}^{l_2}]\big),\ \mathcal{S}\end{array}}$$

$$\textsc{TRCom}\quad \frac{}{\begin{array}{l}\big((p^{l_1} : !T_1.\eta_1) \cdot \Delta_1 \vDash \vec{\tau}^{1}\,;\tau\,;\tau\,;\tau\,;\mathtt{pop}\rho_1!T_1'\,;\vec{b}_{\mathsf{nxt1}},\ E_1[(\mathtt{send}\ (p^{l_1}, v))]\big),\\ \big((\overline{p}^{l_2} : ?T_2.\eta_2) \cdot \Delta_2 \vDash \vec{\tau}^{2}\,;\tau\,;\tau\,;\mathtt{pop}\rho_2?T_2'\,;\vec{b}_{\mathsf{nxt2}},\ E_2[\mathtt{recv}\,\overline{p}^{l_2}]\big),\ \mathcal{S}\\ \to^{C} \big((p^{l_1} : \eta_1) \cdot \Delta_1 \vDash \vec{\tau}^{1}\,;\tau\,;\vec{b}_{\mathsf{nxt1}},\ E_1[()]\big),\ \big((\overline{p}^{l_2} : \eta_2) \cdot \Delta_2 \vDash \vec{\tau}^{2}\,;\tau\,;\vec{b}_{\mathsf{nxt2}},\ E_2[v]\big),\ \mathcal{S}\end{array}}$$

$$\textsc{TRDel}\quad \frac{}{\begin{array}{l}\big((p^{l_1} : !\eta_d.\eta_1) \cdot (p'^{l_1'} : \eta_d') \cdot \Delta_1 \vDash \vec{\tau}^{1}\,;\tau\,;\tau\,;\tau\,;\mathtt{pop}\rho_1!\rho_d\,;\vec{b}_{\mathsf{nxt1}},\ E_1[\mathtt{deleg}\ (p^{l_1}, p'^{l_1'})]\big),\\ \big((\overline{p}^{l_2} : ?\eta_r.\eta_2) \vDash \vec{\tau}^{2}\,;\tau\,;\tau\,;\mathtt{pop}\rho_2?\rho_r\,;\vec{b}_{\mathsf{nxt2}},\ E_2[\mathtt{resume}^{l_r}\,\overline{p}^{l_2}]\big),\ \mathcal{S}\\ \to^{C} \big((p^{l_1} : \eta_1) \cdot \Delta_1 \vDash \vec{\tau}^{1}\,;\tau\,;\vec{b}_{\mathsf{nxt1}},\ E_1[()]\big),\ \big((\overline{p}^{l_2} : \eta_2) \cdot (p'^{l_r} : \eta_r) \vDash \vec{\tau}^{2}\,;\tau\,;\vec{b}_{\mathsf{nxt2}},\ E_2[p'^{l_r}]\big),\ \mathcal{S}\end{array}}$$

$$\textsc{TRSel}\quad \frac{k \in I}{\begin{array}{l}\Big(\Big(p^{l_1} : \bigoplus_{i \in I} !L_i.\eta_i\Big) \cdot \Delta_1 \vDash \vec{\tau}^{1}\,;\tau\,;\tau\,;\mathtt{pop}\rho_1!L_k\,;\vec{b}_{\mathsf{nxt1}},\ E_1[\mathtt{sel}\text{-}L_k\,p^{l_1}]\Big),\\ \Big(\Big(\overline{p}^{l_2} : \sum_{i \in (I_1, I_2)} ?L_i.\eta_i'\Big) \cdot \Delta_2 \vDash \vec{\tau}^{2}\,;\tau\,;\Big(\sum_{j \in J} \mathtt{pop}\rho_2?L_j\,;b_j\Big)\,;\vec{b}_{\mathsf{nxt2}},\ E_2[\mathtt{case}\,\overline{p}^{l_2}\,\{L_j \Rightarrow e_j\}_{j \in J}]\Big),\ \mathcal{S}\\ \to^{C} \big((p^{l_1} : \eta_k) \cdot \Delta_1 \vDash \vec{\tau}^{1}\,;\tau\,;\vec{b}_{\mathsf{nxt1}},\ E_1[()]\big),\ \big((\overline{p}^{l_2} : \eta_k') \cdot \Delta_2 \vDash \vec{\tau}^{2}\,;b_k\,;\vec{b}_{\mathsf{nxt2}},\ E_2[e_k]\big),\ \mathcal{S}\end{array}}$$

Fig. A.1: Typed Reductions.

Appendix B

Inference Algorithms

This appendix contains the full length version of Algorithm $\mathcal{SI}$ in Section B.1 and the helper function expand for Algorithm $\mathcal{D}$ in Section B.2.

B.1 Algorithm $\mathcal{SI}$

$$\mathcal{SI}(b,\ C) = (\sigma_2\sigma_1,\ C_2)$$
$$\quad \text{if} \quad (\sigma_1, C_1) = \mathcal{MC}\big(\epsilon \vDash b,\ C,\ \epsilon\big)$$
$$\quad \text{and } (\sigma_2, C_2) = \mathsf{choiceVarSubst}\ C_1$$

$$\mathsf{choiceVarSubst}\ (C \uplus \overrightarrow{\{\eta_i \subseteq \psi_{\mathtt{in}_i}\}} \uplus \overrightarrow{\{\eta_j \subseteq \psi_{\mathtt{ex}_j}\}}) = (\sigma, C\sigma)$$
$$\quad \text{if} \quad \sigma = \overrightarrow{[\psi_{\mathtt{in}_i} \mapsto \eta_i]}\overrightarrow{[\psi_{\mathtt{ex}_j} \mapsto \eta_j]}$$
$$\quad \text{and} \quad \psi_{\mathtt{in}}, \psi_{\mathtt{ex}} \sharp RHS(C) \text{ for any } \psi_{\mathtt{in}}, \psi_{\mathtt{ex}}$$

B.1.1 Algorithm $\mathcal{MC}$

-- remove terminated frames

$$\mathcal{MC}\big((l : \mathsf{end}) \cdot \Delta \vDash b,\ C,\ K\big) = \mathcal{MC}\big(\Delta \vDash b,\ C,\ K\big)$$

-- $\mathcal{MC}$ terminates with behaviour τ

$$\mathcal{MC}\big(\Delta \vDash \tau,\ C,\ \epsilon\big) = (\sigma,\ C\sigma)$$
$$\quad \text{if} \ \ \sigma = \mathsf{finalize}\ \Delta$$

-- pop a sub-behaviour from the continuation stack

$$\mathcal{MC}\big(\Delta \vDash \tau,\ C,\ b \cdot K\big) = \mathcal{MC}\big(\Delta \vDash b,\ C,\ K\big)$$

-- push a new frame on the stack

$$\mathcal{MC}\big(\Delta \vDash \mathtt{push}(l : \eta),\ C,\ K\big) = (\sigma_2\sigma_1, C_2)$$
$$\quad \text{if} \quad (\sigma_1, \Delta_1) = \mathsf{closeFrame}(l, \Delta)$$
$$\quad \text{and } (\sigma_2, C_2) = \mathcal{MC}\big((l : \eta\sigma_1) \cdot \Delta_1 \vDash \tau,\ C\sigma_1,\ K\sigma_1\big)$$

-- send

$$\mathcal{MC}\big((l : \psi) \cdot \Delta \vDash \mathsf{pop}\rho!T,\ C,\ K\big) = (\sigma_2\sigma_1, C_2)$$

$$\quad \text{if} \quad C \vdash l \sim \rho$$
$$\text{and } \sigma_1 = [\psi \mapsto !\alpha.\psi'] \text{ where } \alpha, \psi' \text{ fresh}$$
$$\text{and } (\sigma_2, C_2) = \mathcal{MC}\big((l : \psi') \cdot \Delta\sigma_1 \vDash \tau,\ C\sigma_1 \cup \{T \subseteq \alpha\},\ K\sigma_1\big)$$

$$\mathcal{MC}\big((l : !T'.\eta) \cdot \Delta \vDash \mathsf{pop}\rho!T,\ C,\ K\big) = \mathcal{MC}\big((l : \eta) \cdot \Delta \vDash \tau,\ C \cup \{T \subseteq T'\},\ K\big)$$
$$\quad \text{if} \quad C \vdash l \sim \rho$$

$$-\!- \textit{ recv}$$
$$\mathcal{MC}\big((l : \psi) \cdot \Delta \vDash \mathsf{pop}\rho?T,\ C,\ K\big) = (\sigma_2\sigma_1,\ C_2)$$
$$\quad \text{if} \quad C \vdash l \sim \rho$$
$$\text{and } \sigma_1 = [\psi \mapsto ?\alpha.\psi'] \text{ where } \alpha, \psi' \text{ fresh}$$
$$\text{and } (\sigma_2, C_2) = \mathcal{MC}\big((l : \psi') \cdot \Delta\sigma_1 \vDash \tau,\ C\sigma_1 \cup \{\alpha \subseteq T\},\ K\sigma_1\big)$$

$$\mathcal{MC}\big((l : ?T'.\eta) \cdot \Delta \vDash \mathsf{pop}\rho?T,\ C,\ K\big) = \mathcal{MC}\big((l : \eta) \cdot \Delta \vDash \tau,\ C \cup \{T' \subseteq T\},\ K\big)$$
$$\quad \text{if} \quad C \vdash l \sim \rho$$

$$-\!- \textit{ delegation}$$
$$\mathcal{MC}\big((l : \psi) \cdot (l_d : \eta_d) \cdot \Delta \vDash \mathsf{pop}\rho!\rho_d,\ C,\ K\big) = (\sigma_2\sigma_1, C_2)$$
$$\quad \text{if} \quad C \vdash l \sim \rho \text{ and } C \vdash l_d \sim \rho_d$$
$$\text{and } \sigma_1 = [\psi \mapsto !\eta_d.\psi'] \text{ where } \psi' \text{ fresh}$$
$$\text{and } (\sigma_2, C_2) = \mathcal{MC}\big((l : \psi') \cdot \Delta\sigma \vDash \tau,\ C\sigma_1,\ K\sigma_1\big)$$

$$\mathcal{MC}\big((l : !\eta_d.\eta) \cdot (l_d : \eta'_d) \cdot \Delta \vDash \mathsf{pop}\rho!\rho_d,\ C,\ K\big) = (\sigma_2\sigma_1, C_2)$$
$$\quad \text{if} \quad C \vdash l \sim \rho \text{ and } C \vdash l_d \sim \rho_d$$
$$\text{and } (\sigma_1, C_1) = \ \mathsf{sub}(\eta'_d, \eta_d, C)$$
$$\text{and } (\sigma_2, C_2) = \mathcal{MC}\big((l : \eta) \cdot \Delta\sigma \vDash \tau,\ C_1,\ K\sigma_1\big)$$

$$\mathcal{MC}\big((l : \eta) \cdot (l_d : \eta'_d) \cdot \Delta \vDash \mathsf{pop}\rho!\rho_d,\ C,\ K\big) = (\sigma_2\sigma_1, C_2)$$
$$\quad \text{if} \quad C \vdash l_d \not\sim \rho_d$$
$$\text{and } (\sigma_1, \Delta_1) = \mathsf{closeFrame}(l_d, (l : \eta) \cdot (l_d : \eta'_d) \cdot \Delta)$$
$$\text{and } (\sigma_2, C_2) = \mathcal{MC}\big(\Delta_1 \vDash \mathsf{pop}\rho!\rho_d,\ C\sigma_1,\ K\sigma_1\big)$$

$$-\!- \textit{ resume}$$
$$\mathcal{MC}\big((l : \psi) \cdot \epsilon \vDash \mathsf{pop}\rho?l_r,\ C,\ K\big) = (\sigma_2\sigma_1, C_2)$$
$$\quad \text{if} \quad C \vdash l \sim \rho \text{ and } l \neq l_r$$
$$\text{and } \sigma_1 = [\psi \mapsto ?\psi'_r.\psi'] \text{ where } \psi', \psi'_r \text{ fresh}$$
$$\text{and } (\sigma_2, C_2) = \mathcal{MC}\big((l : \psi') \cdot (l_r : \psi'_r) \cdot \epsilon \vDash \tau,\ C\sigma_1,\ K\sigma_1\big)$$

$$\mathcal{MC}\big((l : ?\eta_r.\eta) \cdot \epsilon \vDash \mathsf{pop}\rho?l_r,\ C,\ K\big) = \mathcal{MC}\big((l : \eta) \cdot (l_r : \eta_r) \cdot \epsilon \vDash \tau,\ C,\ K\big)$$
$$\quad \text{if} \quad C \vdash l \sim \rho \text{ and } l \neq l_r$$

$$\mathcal{MC}\big((l : ?\eta) \cdot (l_d : \eta_d) \cdot \Delta \vDash \mathsf{pop}\rho?l_r,\ C,\ K\big) = (\sigma_2\sigma_1, C_2)$$
$$\quad \text{if} \quad (\sigma_1, \Delta_1) = \mathsf{closeFrame}(l_d, (l : ?\eta) \cdot (l_d : \eta_d) \cdot \Delta)$$
$$\text{and } (\sigma_2, C_2) = \mathcal{MC}\big(\Delta_1 \vDash \mathsf{pop}\rho?l_r,\ C\sigma_1,\ K\sigma_1\big)$$

$$\mathcal{MC}\big((l:\psi)\cdot\Delta \vDash \mathsf{pop}\rho!L_k,\ C,\ K\big) = (\sigma_2\sigma_1, C_2)$$
if $C \vdash l \sim \rho$
and $\sigma_1 = [\psi \mapsto \psi_{\mathtt{in}}]$ where $\psi_k, \psi_{\mathtt{in}}$ fresh
and $(\sigma_2, C_2) = \mathcal{MC}\big((l:\psi_k)\cdot\Delta\sigma \vDash \tau,\ C\sigma_1 \cup \{\psi_{\mathtt{in}} \sim \bigoplus_{j\in\{k\}} !L_j.\psi_j\},\ K\sigma_1\big)$

$$\mathcal{MC}\big((l:\psi_{\mathtt{in}})\cdot\Delta \vDash \mathsf{pop}\rho!L_i,\ C,\ K\big) = \mathcal{MC}\big((l:\eta_i)\cdot\Delta \vDash \tau,\ C,\ K\big)$$
if $C \vdash l \sim \rho$
and $i \in J$ and $(\psi_{\mathtt{in}} \sim \bigoplus_{j\in J} !L_j.\eta_j) \in C$

$$\mathcal{MC}\big((l:\psi_{\mathtt{in}})\cdot\Delta \vDash \mathsf{pop}\rho!L_i,\ C,\ K\big) = \mathcal{MC}\big((l:\psi_i)\cdot\Delta \vDash \tau,\ C' \uplus \{\psi_{\mathtt{in}} \sim \bigoplus_{j\in J,i} !L_j.\eta_j\},\ K\big)$$
if $C \vdash l \sim \rho$
and $i \notin J$ and $C = C' \uplus \{\psi_{\mathtt{in}} \sim \bigoplus_{j\in J} !L_j.\eta_j\}$ where $\eta_i = \psi_i$ fresh

-- *ex. choice*

$$\mathcal{MC}\big((l:\psi)\cdot\Delta \vDash \sum_{i\in I} \mathsf{pop}\rho?L_i\,;b_i,\ C,\ K\big) = (\sigma_2\sigma_1, C_2)$$
if $C \vdash l \sim \rho$
and $\sigma_1 = [\psi \mapsto \psi_{\mathtt{ex}}]$ where $\psi_{\mathtt{ex}}$ fresh
and $C_1 = \{\psi_{\mathtt{ex}} \sim \sum_{i\in(I,\emptyset)} ?L_i.\psi_i\}$ where $\overrightarrow{\psi_i}$ fresh
and $(\sigma_2, C_2) = \mathcal{MC}\big((l:\psi_{\mathtt{ex}})\cdot\Delta\sigma_1 \vDash \big(\sum_{i\in I} \mathsf{pop}\rho?L_i\,;b_i\big)\sigma_1,\ C\sigma_1 \cup C_1,\ K\sigma_1\big)$

$$\mathcal{MC}\big((l:\psi_{\mathtt{ex}})\cdot\Delta \vDash \sum_{j\in I_1\uplus I_2\uplus I_3} \mathsf{pop}\rho?L_j\,;b_j,\ C,\ K\big) = \mathcal{MC}\big((l:\psi_{\mathtt{ex}})\cdot\Delta \vDash \sum_{j\in I_1\uplus I_2\uplus I_3} \mathsf{pop}\rho?L_j\,;b_j,\ C_1,\ K\big)$$
if $C \vdash l \sim \rho$ and $I_3 \sharp J_1 \uplus J_2$ and $(I_3 \neq \emptyset$ and $J_1 \neq \emptyset$
and $C = C' \uplus \{\psi_{\mathtt{ex}} \sim \sum_{j\in(I_1J_1,I_2J_2)} ?L_j.\eta_j\}$
and $C_1 = C' \cup \{\psi_{\mathtt{ex}} \sim \sum_{i\in(I_1,I_2I_3J_1J_2)} ?L_i.\eta_i\}$

$$\mathcal{MC}\big((l:\psi_{\mathtt{ex}})\cdot\Delta \vDash \sum_{i\in I} \mathsf{pop}\rho?L_i\,;b_i,\ C,\ K\big) = (\sigma_n\ldots\sigma_0, C_n)$$
if $C \vdash l \sim \rho$ and $J_1 \subseteq I$ and $I \subseteq J_1 \uplus J_2$
and $C = C' \uplus \{\psi_{\mathtt{ex}} \sim \sum_{j\in(J_1,J_2)} ?L_j.\eta_j\}$
and for each $k \in [1\ldots n]$ where $I = \{L_{j_1},\ldots,L_{j_n}\}$ and $n \geq 0$
$(\sigma_k, C_k) = \mathcal{MC}\big(((l:\eta_{j_k})\cdot\Delta \vDash b_{j_k})\sigma_{k-1}\ldots\sigma_0,\ C_{k-1},\ K\sigma_{k-1}\ldots\sigma_0\big)$
where $\sigma_0 = \sigma_{id}$ and $C_0 = C$

-- *sequencing*

$$\mathcal{MC}\big(\Delta \vDash b_1\,;b_2,\ C,\ K\big) = \mathcal{MC}\big(\Delta \vDash b_1,\ C,\ b_2\cdot K\big)$$

-- *internal choice in the behaviour*

$$\mathcal{MC}\big(\Delta \vDash b_1 \oplus b_2,\ C,\ K\big) = (\sigma_2\sigma_1, C_2)$$
if $(\sigma_1, C_1) = \mathcal{MC}\big(\Delta \vDash b_1,\ C,\ K\big)$

```
  and (σ2, C2) = MC(Δσ1 ⊨ b2σ1, C1, Kσ1)

-- spawn
MC(Δ ⊨ spawn b, C, K) = (σ2σ1, C2)
  if  (σ1, C1) = MC(ϵ ⊨ b, C, ϵ)
  and (σ2, C2) = MC(Δσ1 ⊨ τ, C1, Kσ1)

-- rec
MC(Δ ⊨ rec_β b, C) = (σ2σ1, C2)
  if  C = C' ⊎ {b' ⊆ β}
  and (σ1, C1) = MC(ϵ ⊨ b, C' ∪ {τ ⊆ β}, ϵ)
  and (σ2, C2) = MC(Δσ1 ⊨ τ, C1 ∪ ({b' ⊆ β})σ1, Kσ1)

-- behaviour variable
MC(Δ ⊨ β, C, K) = MC(Δ ⊨ b, C, K)
  where  b = ⨁{b_i | ∃i. (b_i ⊆ β) ∈ C}

-- try to close the top session if all previous clauses fail
MC(Δ ⊨ b, C, K) = (σ2σ1, C2)
  if  (σ1, Δ1) = closeTop(Δ)
  and (σ2, C2) = MC(Δ1 ⊨ bσ1, Cσ1, Kσ1)
```

B.1.2 Helper functions

```
-- closeFrame forces frame l in the input stack to be the closed session
closeFrame(l, ϵ) = (σ_id, ϵ)
closeFrame(l, (l : end) · Δ) = (σ_id, Δ)
closeFrame(l, (l : ψ) · Δ) = ([ψ ↦ end], Δ[ψ ↦ end])
closeFrame(l, (l' : η) · Δ) = (σ, (l' : ησ) · Δ')
  if  l' ≠ l and (σ, Δ') = closeFrame(l, Δ)

-- closeTop forces the top of the stack to be end
closeTop((l : η) · Δ) = closeFrame(l, (l : η) · Δ)

-- finalize matches the input stack with the empty stack
finalize  ϵ = σ_id
finalize  (l : end) · Δ = finalize Δ
finalize  (l : ψ) · Δ = σ2σ1
  if  σ1 = [ψ ↦ end]
  and σ2 = finalize(Δ)
```

B.1.3 Subtype checking

```
-- end
sub(end, end, C) = (σ_id, C)

```

$$
\begin{array}{l}
\textit{-- send/recv} \\
\mathsf{sub}(!T_1.\eta_1, !T_2.\eta_2, C) = (\sigma_1, C_1 \cup \{T_2 \subseteq T_1\}) \\
\quad \text{if } \ (\sigma_1, C_1) = \mathsf{sub}(\eta_1, \eta_2, C) \\
\\
\mathsf{sub}(?T_1.\eta_1, ?T_2.\eta_2, C) = (\sigma_1, C_1 \cup \{T_1 \subseteq T_2\}) \\
\quad \text{if } \ (\sigma_1, C_1) = \mathsf{sub}(\eta_1, \eta_2, C) \\
\\
\textit{-- deleg/resume} \\
\mathsf{sub}(!\eta_{d1}.\eta_1, !\eta_{d2}.\eta_2, C) = (\sigma_2\sigma_1, C_2) \\
\quad \text{if } \ (\sigma_1, C_1) = \mathsf{sub}(\eta_{d2}, \eta_{d1}, C) \text{ and } (\sigma_2, C_2) = \mathsf{sub}(\eta_1\sigma_1, \eta_2\sigma_1, C_1) \\
\\
\mathsf{sub}(?\eta_{r1}.\eta_1, ?\eta_{r2}.\eta_2, C) = (\sigma_2\sigma_1, C_2) \\
\quad \text{if } \ (\sigma_1, C_1) = \mathsf{sub}(\eta_{r1}, \eta_{r2}, C) \text{ and } (\sigma_2, C_2) = \mathsf{sub}(\eta_1\sigma_1, \eta_2\sigma_1, C_1) \\
\\
\textit{-- in. choice} \\
\mathsf{sub}(\psi_{\mathtt{in}_1}, \psi_{\mathtt{in}_2}, C) = \mathsf{f}\ (I_2, C) \\
\quad \text{if } \ (\psi_{\mathtt{in}_2} \sim \bigoplus_{i \in I_2} !L_i.\eta_{2i}) \in C \\
\quad \text{and } \mathsf{f}\ (\emptyset, C) = (\sigma_{id}, C) \\
\quad \text{and } \mathsf{f}\ (I \uplus \{k\}, C) = \mathsf{f}\ (I \uplus \{k\}, C_1) \\
\qquad \text{if } \ C = C' \uplus (\psi_{\mathtt{in}_1} \sim \bigoplus_{i \in I_1} !L_i.\eta_{1i}) \uplus (\psi_{\mathtt{in}_2} \sim \bigoplus_{i \in I_2} !L_i.\eta_{2i}) \\
\qquad \text{and } k \notin I_1 \text{ and } k \in I_2 \\
\qquad \text{and } C_1 = C' \uplus (\psi_{\mathtt{in}_1} \sim \bigoplus_{i \in I_1 \cup \{k\}} !L_i.\eta_{1i} \oplus \eta_{2_k}) \uplus (\psi_{\mathtt{in}_2} \sim \bigoplus_{i \in I_2} !L_i.\eta_{2i}) \\
\quad \text{and } \mathsf{f}\ (I \uplus \{k\}, C) = (\sigma_2\sigma_1, C_2) \\
\qquad \text{if } \ (\sigma_1, C_1) = \mathsf{f}\ (I, C) \\
\qquad \text{and } (\psi_{\mathtt{in}_1} \sim \bigoplus_{i \in I_1} !L_i.\eta_{1i}), (\psi_{\mathtt{in}_2} \sim \bigoplus_{i \in I_2} !L_i.\eta_{2i}) \in C_1 \\
\qquad \text{and } k \in I_1 \text{ and } k \in I_2 \\
\qquad \text{and } (\sigma_2, C_2) = \mathsf{sub}\ (\eta_{1k}\sigma_1, \eta_{2k}\sigma_1, C_1) \\
\\
\textit{-- ex. choice} \\
\mathsf{sub}(\psi_{\mathtt{ex}_1}, \psi_{\mathtt{ex}_2}, C) = \mathsf{f}\ (J_1 \cup J_2, C) \\
\quad \text{if } \ (\psi_{\mathtt{ex}_1} \sim \sum_{i \in (I_1, I_2)} ?L_i.\eta_{1i}), (\psi_{\mathtt{ex}_2} \sim \sum_{i \in (J_1, J_2)} ?L_i.\eta_{2i}) \in C \\
\quad \text{and } I_1 \subseteq J_1 \\
\quad \text{and } \mathsf{f}\ (\emptyset, C) = (\sigma_{id}, C) \\
\quad \text{and } \mathsf{f}\ (I \cup \{k\}, C) = (\sigma_2\sigma_1, C_2) \\
\qquad \text{if } \ (\psi_{\mathtt{ex}_1} \sim \sum_{i \in (I_1, I_2)} ?L_i.\eta_{1i}), (\psi_{\mathtt{ex}_2} \sim \sum_{i \in (J_1, J_2)} ?L_i.\eta_{2i}) \in C \\
\qquad \text{and } (k \in I_2 \text{ or } (k \in I_1 \text{ and } k \in J_1) \\
\qquad \text{and } (\sigma_1, C_1) = \mathsf{f}\ (I, C) \\
\qquad \text{and } (\sigma_2, C_2) = \mathsf{sub}\ (\eta_{1k}\sigma_1, \eta_{2k}\sigma_1, C_1) \\
\quad \text{and } \mathsf{f}\ (I \cup \{k\}, C) = \mathsf{f}\ (I \cup \{k\}, C_1) \\
\qquad \text{if } \ C = C' \uplus (\psi_{\mathtt{ex}_1} \sim \sum_{i \in (I_1, I_2)} ?L_i.\eta_{1i}) \uplus (\psi_{\mathtt{ex}_2} \sim \sum_{i \in (J_1, J_2)} ?L_i.\eta_{2i}) \\
\qquad \text{and } k \in I_1 \text{ and } k \in J_2
\end{array}
$$

45 and $C_1 = C' \uplus (\psi_{\mathtt{ex}_1} \sim \sum_{i\in(I_1\setminus\{k\},I_2\cup\{k\})} ?L_i.\eta_{1i}) \uplus (\psi_{\mathtt{ex}_2} \sim \sum_{i\in(J_1,J_2)} ?L_i.\eta_{2i})$

46 $\mathsf{f}\ (I \cup \{k\}, C) = \mathsf{f}\ (I \cup \{k\}, C_1)$

47 if $C = C' \uplus (\psi_{\mathtt{ex}_1} \sim \sum_{i\in(I_1,I_2)} ?L_i.\eta_{1i}) \uplus (\psi_{\mathtt{ex}_2} \sim \sum_{i\in(J_1,J_2)} ?L_i.\eta_{2i})$

48 and $k \notin I_1$ and $k \notin I_2$

49 and $C_1 = C' \uplus (\psi_{\mathtt{ex}_1} \sim \sum_{i\in(I_1,I_2\cup\{k\})} ?L_i.\eta_{1i} + ?L_k.\eta_{2k}) \uplus (\psi_{\mathtt{ex}_2} \sim \sum_{i\in(J_1,J_2)} ?L_i.\eta_{2i})$

50

51 *-- session inference*

52 $\mathsf{sub}(\psi, \eta, C) = (\sigma, C\sigma)$ if $\sigma = [\psi \mapsto \eta]$

53 $\mathsf{sub}(\eta, \psi, C) = (\sigma, C\sigma)$ if $\sigma = [\psi \mapsto \eta]$

B.2 Function expand for Algorithm $\mathcal{D}$

The helper function expand is defined as follows:

$$
\begin{array}{lll}
\mathsf{expand}(C, \psi_1 \bowtie \mathsf{end}) & = (\sigma, C\sigma) & \text{if } \sigma = [\psi_1 \mapsto \mathsf{end}] \\
\mathsf{expand}(C, \psi_1 \bowtie ?T.\eta_2) & = (\sigma, C\sigma) & \text{if } \sigma = [\psi_1 \mapsto !\alpha.\eta_2] \\
\mathsf{expand}(C, \psi_1 \bowtie !T.\eta_2) & = (\sigma, C\sigma) & \text{if } \sigma = [\psi_1 \mapsto ?\alpha.\eta_2] \\
\mathsf{expand}(C, \psi_1 \bowtie !\eta_2'.\eta_2) & = (\sigma, C\sigma) & \text{if } \sigma = [\psi_1 \mapsto !\psi_1'.\psi_1''] \\
\mathsf{expand}(C, \psi_1 \bowtie ?\eta_2'.\eta_2) & = (\sigma, C\sigma) & \text{if } \sigma = [\psi_1 \mapsto ?\psi_1'.\psi_1''] \\
\mathsf{expand}(C, \psi_1 \bowtie \bigoplus_{i\in I} !L_i.\eta_{2i}) & = (\sigma, C\sigma \cup C') & \text{if } \sigma = [\psi_1 \mapsto \psi_{\mathtt{ex}}] \text{ and } C' = \{\psi_{\mathtt{ex}} \sim \sum_{i\in(I,\emptyset)} ?L_i.\psi_{2i}\} \\
\mathsf{expand}(C, \psi_1 \bowtie \sum_{i\in(I_1,I_2)} ?L_i.\eta_{2i}) & = (\sigma, C\sigma \cup C') & \text{if } \sigma = [\psi_1 \mapsto \psi_{\mathtt{in}}] \text{ and } C' = \{\psi_{\mathtt{in}} \sim \bigoplus_{i\in I_1} !L_i.\psi_{2i}\} \\
\mathsf{expand}(C, \psi_{\mathtt{ex}} \bowtie \bigoplus_{i\in I_0} !L_i.\eta_{2i}) & = (\sigma_{id}, C') & \text{if } C = C'' \cup \{\psi_{\mathtt{ex}} \sim \sum_{i\in(I_1,I_2)} ?L_i.\eta_{1i}\} \\
 & & \text{and } C' = C'' \cup \{\psi_{\mathtt{ex}} \sim \sum_{i\in(I_1\cup I_0,I_2\setminus I_0)} ?L_i.\eta_{1i}\} \cup \bigcup_{i\in I_0} \{\eta_{1i} \bowtie \eta_{2i}\} \\
\mathsf{expand}(C, \psi_{\mathtt{in}} \bowtie \sum_{i\in(I_1,I_2)} ?L_i.\eta_{2i}) & = (\sigma_{id}, C') & \text{if } C = C'' \cup \{\psi_{\mathtt{in}} \sim \bigoplus_{i\in I_0} !L_i.\eta_{1i}\} \\
 & & \text{and } C' = C'' \cup \{\psi_{\mathtt{ex}} \sim \bigoplus_{i\in(I_0\setminus I_2)\cap I_1} !L_i.\eta_{1i}\} \cup \bigcup_{i\in I_0\cap I_1} \{\eta_{1i} \bowtie \eta_{2i}\} \\
\mathsf{expand}(C, \psi_{\mathtt{in}} \bowtie \psi_{\mathtt{ex}}) & = (\sigma_{id}, C') & \text{if } C = C'' \cup \{\bigoplus_{i\in I_0} !L_i.\eta_{1i} \subseteq \psi_{\mathtt{in}}\} \cup \{\sum_{i\in(I_1,I_2)} ?L_i.\eta_{1i} \subseteq \psi_{\mathtt{ex}}\} \\
 & & \text{and } C' = C'' \cup \{\psi_{\mathtt{in}} \sim \bigoplus_{i\in(I_0\setminus I_2)\cup I_1} !L_i.\eta_{1i}\} \cup \{\psi_{\mathtt{ex}} \sim \sum_{i\in(I_1\cap(I_0\setminus I_2),I_2\cup(I_0\setminus I_1))} ?L_i.\eta_{1i}\} \\
 & & \qquad \cup \bigcup_{i\in(I_0\setminus I_2)\cap I_1} \{\eta_{1i} \bowtie \eta_{2i}\}
\end{array}
$$

where all $\alpha, \psi, \psi_{\mathtt{in}}, \psi_{\mathtt{ex}}$ variables on the right-hand side are fresh, and where fresh variables ψ_{1i} and ψ_{2i} are generated, in case index i is not defined in the starting internal or external choice. Symmetric rules are omitted.

Bibliography

[Amsden and Fluet, 2012] Amsden, E. and Fluet, M. (2012). Fairness for transactional events. In *Proceedings of the 23rd international conference on Implementation and Application of Functional Languages*, IFL'11, pages 17–34, Berlin, Heidelberg. Springer-Verlag.

[Amtoft et al., 1999] Amtoft, T., Nielson, H. R., and Nielson, F. (1999). *Type and effect systems - behaviours for concurrency*. Imperial College Press.

[Bacci et al., 2011] Bacci, G., Danos, V., and Kammar, O. (2011). On the statistical thermodynamics of reversible communicating processes. In *Proceedings of the 4th international conference on Algebra and coalgebra in computer science*, CALCO'11, pages 1–18, Berlin, Heidelberg. Springer-Verlag.

[Bergstra, 2001] Bergstra, J. A. (2001). *Handbook of Process Algebra*. Elsevier Science Inc., New York, NY, USA.

[Berry and Boudol, 1990] Berry, G. and Boudol, G. (1990). The chemical abstract machine. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, pages 81–94.

[Bocchi, 2006] Bocchi, L. (2006). *Transactional Aspects in Coordination and Composition of Web Services*. PhD thesis, University of Bologna.

[Bocchi et al., 2003] Bocchi, L., Laneve, C., and Zavattaro, G. (2003). A calculus for long-running transactions. In *FMOODS*, pages 124–138.

[Brown and Sabry, 2015] Brown, G. and Sabry, A. (2015). Reversible communicating processes. *PLACES*, page 85.

[Bruni et al., 2002] Bruni, R., Laneve, C., and Montanari, U. (2002). Orchestrating transactions in join calculus. In *Proceedings of the 13th International Conference on Concurrency Theory*, CONCUR '02, pages 321–337, London, UK, UK. Springer-Verlag.

[Bruni et al., 2015] Bruni, R., Melgratti, H. C., and Montanari, U. (2015). cjoin: Join with communicating transactions. *Mathematical Structures in Computer Science*, 25(3):566–618.

[Butler and Ferreira, 2004] Butler, M. and Ferreira, C. (2004). An operational semantics for stac, a language for modelling long-running business transactions. In *In Coordination 2004, volume 2949 of LNCS*, pages 87–104. Springer-Verlag.

[Butler et al., 2005] Butler, M., Hoare, T., and Ferreira, C. (2005). A trace semantics for long-running transactions. In *Proceedings of the 2004 international conference on Communicating Sequential Processes: the First 25 Years*, CSP'04, pages 133–150, Berlin, Heidelberg. Springer-Verlag.

[Cardelli and Laneve, 2011a] Cardelli, L. and Laneve, C. (2011a). Reversibility in massive concurrent systems. *CoRR*, abs/1108.3419.

[Cardelli and Laneve, 2011b] Cardelli, L. and Laneve, C. (2011b). Reversible structures. In *Proceedings of the 9th International Conference on Computational Methods in Systems Biology*, CMSB '11, pages 131–140, New York, NY, USA. ACM.

[Castagna et al., 2009] Castagna, G., Dezani-Ciancaglini, M., Giachino, E., and Padovani, L. (2009). Foundations of session types. In *PPDP*, pages 219–230.

[Chandra et al., 2007] Chandra, T., Griesemer, R., and Redstone, J. (2007). Paxos made live: an engineering perspective. In *In Proc. of PODC*, pages 398–407. ACM Press.

[Cristescu et al., 2013] Cristescu, I., Krivine, J., and Varacca, D. (2013). A compositional semantics for the reversible p-calculus. In *LICS*, pages 388–397.

[Danos and Krivine, 2005] Danos, V. and Krivine, J. (2005). Concur 2005 - concurrency theory. chapter Transactions in RCCS, pages 398–412. Springer-Verlag, London, UK, UK.

[Danos and Krivine, 2007] Danos, V. and Krivine, J. (2007). Formal molecular biology done in ccs-r. *Electron. Notes Theor. Comput. Sci.*, 180(3):31–49.

[Danos et al., 2004] Danos, V., Krivine, J., Paris, U., and Rocquencourt, I. (2004). Reversible communicating systems. In *in: CONCUR04, LNCS 3170 (2004*, pages 292–307. Springer.

[de Vries et al., 2010] de Vries, E., Koutavas, V., and Hennessy, M. (2010). Communicating transactions. In *Proceedings of the 21st international conference on Concurrency theory*, CONCUR'10, pages 569–583, Berlin, Heidelberg. Springer-Verlag.

[De Vries et al., 2010] De Vries, E., Koutavas, V., and Hennessy, M. (2010). Liveness of communicating transactions. In *Proceedings of the 8th Asian conference on Programming languages and systems*, APLAS'10, pages 392–407, Berlin, Heidelberg. Springer-Verlag.

[Dijkstra, 1975] Dijkstra, E. W. (1975). Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457.

[Donnelly and Fluet, 2006] Donnelly, K. and Fluet, M. (2006). Transactional events. *SIGPLAN Not.*, 41(9):124–135.

[Effinger-Dean et al., 2008] Effinger-Dean, L., Kehrt, M., and Grossman, D. (2008). Transactional events for ml. *SIGPLAN Not.*, 43(9):103–114.

[Elnozahy et al., 2002a] Elnozahy, E. N., Alvisi, L., Wang, Y., and Johnson, D. B. (2002a). A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408.

[Elnozahy et al., 2002b] Elnozahy, E. N. M., Alvisi, L., Wang, Y.-M., and Johnson, D. B. (2002b). A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408.

[Field and Varela, 2005] Field, J. and Varela, C. A. (2005). Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. *SIGPLAN Not.*, 40(1):195–208.

[Fournet and Gonthier, 1996] Fournet, C. and Gonthier, G. (1996). The reflexive cham and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 372–385, New York, NY, USA. ACM.

[Garcia-Molina and Salem, 1987] Garcia-Molina, H. and Salem, K. (1987). Sagas. *SIGMOD Rec.*, 16(3):249–259.

[Gay and Hole, 1999] Gay, S. and Hole, M. (1999). Types and subtypes for client-server interactions. In Swierstra, S. D., editor, *PLS (ESOP)*, volume 1576 of *LNCS*, pages 74–90. Springer.

[Gay and Hole, 2005] Gay, S. and Hole, M. (2005). Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225.

[Grossman, 2007] Grossman, D. (2007). The transactional memory / garbage collection analogy. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 695–706, New York, NY, USA. ACM.

[Guerraoui and Kapalka, 2008] Guerraoui, R. and Kapalka, M. (2008). On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 175–184, New York, NY, USA. ACM.

[Harris et al., 2005] Harris, T., Marlow, S., Peyton-Jones, S., and Herlihy, M. (2005). Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 48–60, New York, NY, USA. ACM.

[Hennessy, 1988] Hennessy, M. (1988). *Algebraic theory of processes.* MIT Press series in the foundations of computing. MIT Press.

[Herlihy and Shavit, 2008] Herlihy, M. and Shavit, N. (2008). *The art of multiprocessor programming.* Kaufmann.

[Hoare, 1978a] Hoare, C. A. R. (1978a). Communicating sequential processes. *Commun. ACM*, 21(8):666–677.

[Hoare, 1978b] Hoare, C. A. R. (1978b). Communicating sequential processes. *Commun. ACM*, 21(8):666–677.

[Honda, 1993] Honda, K. (1993). Types for dyadic interaction. In Best, E., editor, *CONCUR*, volume 715 of *LNCS*, pages 509–523. Springer.

[Honda et al., 1998] Honda, K., Vasconcelos, V. T., and Kubo, M. (1998). Language primitives and type discipline for structured communication-based programming. In Hankin, C., editor, *PLS (ESOP)*, volume 1381 of *LNCS*, pages 122–138. Springer.

[Honda and Yoshida, 1995] Honda, K. and Yoshida, N. (1995). On reduction-based process semantics. *Theor. Comput. Sci.*, 151(2):437–486.

[Hüttel et al.,] Hüttel, H., Lanese, I., Vasconcelos, V. T., Caires, L., Carbone, M., Deniélou, P.-M., Padovani, L., Ravara, A., Tuosto, E., Vieira, H. T., and Zavattaro, G. Foundations of behavioural types. `http://www.behavioural-types.eu/publications/WG1-State-of-the-Art.pdf`. Accessed 20 April 2015.

[Jeffrey, 1997] Jeffrey, A. (1997). Semantics for core concurrent ml using computation types. In *Higher Order Operational Techniques in Semantics, Proceedings*. Cambridge University Press.

[Jones et al., 1996] Jones, S. P. L., Gordon, A. D., and Finne, S. (1996). Concurrent Haskell. In *POPL*, pages 295–308, NY. ACM.

[Kehrt et al., 2009] Kehrt, M., Effinger-Dean, L., Schmitz, M., and Grossman, D. (2009). Programming idioms for transactional events. In *PLACES*, pages 43–48.

[Koskinen et al., 2010] Koskinen, E., Parkinson, M., and Herlihy, M. (2010). Coarse-grained transactions. *SIGPLAN Not.*, 45(1):19–30.

[Koutavas et al., 2014] Koutavas, V., Spaccasassi, C., and Hennessy, M. (2014). Bisimulations for communicating transactions - (extended abstract). In *Foundations of Software Science and Computation Structures - 17th International Conference, FOSSACS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, pages 320–334.

[Kshemkalyani and Singhal, 2008] Kshemkalyani, A. D. and Singhal, M. (2008). *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, New York, NY, USA, 1 edition.

[Lamport, 2002] Lamport, L. (2002). Paxos made simple, fast, and byzantine. In *OPODIS*, pages 7–9.

[Lanese et al., 2013a] Lanese, I., Lienhardt, M., Mezzina, C. A., Schmitt, A., and Stefani, J.-B. (2013a). Concurrent flexible reversibility. In *Proceedings of the 22nd European conference on Programming Languages and Systems*, ESOP'13, pages 370–390, Berlin, Heidelberg. Springer-Verlag.

[Lanese et al., 2013b] Lanese, I., Lienhardt, M., Mezzina, C. A., Schmitt, A., and Stefani, J.-B. (2013b). Concurrent flexible reversibility. In *Proceedings of the 22nd European conference on Programming Languages and Systems*, ESOP'13, pages 370–390, Berlin, Heidelberg. Springer-Verlag.

[Lanese et al., 2010] Lanese, I., Mezzina, C. A., and Stefani, J.-B. (2010). Reversing higher-order pi. In *Proceedings of the 21st international conference on Concurrency theory*, CONCUR'10, pages 478–493, Berlin, Heidelberg. Springer-Verlag.

[Lesani et al., 2009] Lesani, M., Odersky, M., and Guerraoui, R. (2009). Transactors: Unifying Transactions and Actors. Technical report.

[Lesani and Palsberg, 2011] Lesani, M. and Palsberg, J. (2011). Communicating memory transactions. *SIGPLAN Not.*, 46(8):157–168.

[Lienhardt et al., 2012] Lienhardt, M., Lanese, I., Mezzina, C. A., and Stefani, J.-B. (2012). A reversible abstract machine and its space overhead. In *Proceedings of the 14th joint IFIP WG 6.1 international conference and Proceedings of the 32nd IFIP WG 6.1 international conference on Formal Techniques for Distributed Systems*, FMOODS'12/FORTE'12, pages 1–17, Berlin, Heidelberg. Springer-Verlag.

[Luchangco and Marathe, 2005] Luchangco, V. and Marathe, V. J. (2005). Transaction synchronizers. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*.

[Luchangco and Marathe, 2011] Luchangco, V. and Marathe, V. J. (2011). Transaction communicators: enabling cooperation among concurrent transactions. *SIGPLAN Not.*, 46(8):169–178.

[Marlow et al., 2001] Marlow, S., Jones, S. P. L., Moran, A., and Reppy, J. H. (2001). Asynchronous exceptions in Haskell. In *PLDI 2001*, pages 274–285, NY. ACM.

[Mazzara and Lucchi, 2004] Mazzara, M. and Lucchi, R. (2004). A framework for generic error handling in business processes. *Electron. Notes Theor. Comput. Sci.*, 105:133–145.

[Mezzina, 2008] Mezzina, L. G. (2008). How to infer finite session types in a calculus of services and sessions. In Lea, D. and Zavattaro, G., editors, *Coordination Models and Languages*, volume 5052 of *LNCS*, pages 216–231. Springer.

[Miculan et al., 2015] Miculan, M., Peressotti, M., and Toneguzzo, A. (2015). Open transactions on shared memory. In *Coordination Models and Languages - 17th IFIP WG 6.1 International Conference, COORDINATION 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings*, pages 213–229.

[Milner, 1978a] Milner, R. (1978a). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348 – 375.

[Milner, 1978b] Milner, R. (1978b). A theory of type polymorphism in programming. *J. CSS*, 17(3):348 – 375.

[Milner, 1982] Milner, R. (1982). *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Ng et al., 2011] Ng, N., Yoshida, N., Pernet, O., Hu, R., and Kryftis, Y. (2011). Safe parallel programming with session java. In De Meuter, W. and Roman, G.-C., editors, *Coordination Models and Languages*, volume 6721 of *LNCS*, pages 110–126. Springer.

[Nielson and Nielson, 1996] Nielson, F. and Nielson, H. R. (1996). From CML to its process algebra. *Theoretical Computer Science*, 155(1):179 – 219.

[Oram and Wilson, 2007] Oram, A. and Wilson, G. (2007). *Beautiful code*. O'Reilly, first edition.

[Paige and Tarjan, 1987] Paige, R. and Tarjan, R. E. (1987). Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989.

[Peyton-Jones, 2007] Peyton-Jones, S. (2007). Beautiful concurrency. In Oram, A. and Wilson, G., editors, *Beautiful Code: Leading Programmers Explain How They Think*. O'Reilly.

[Phillips and Ulidowski, 2007] Phillips, I. and Ulidowski, I. (2007). Reversing algebraic process calculi. *The Journal of Logic and Algebraic Programming*, 73(12):70 – 96. ¡ce:title¿Foundations of Software Science and Computation Structures 2006 (FOSSACS 2006)¡/ce:title¿.

[Pierce, 2002] Pierce, B. C. (2002). *Types and programming languages*. MIT Press.

[Pucella and Tov, 2008] Pucella, R. and Tov, J. A. (2008). Haskell session types with (almost) no class. In *Haskell Symposium*, pages 25–36.

[Reppy, 1999] Reppy, J. H. (1999). *Concurrent programming in ML*. Cambridge University Press, New York, NY, USA.

[Sangiorgi and Walker, 2001] Sangiorgi, D. and Walker, D. (2001). *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press.

[Shavit and Touitou, 1995] Shavit, N. and Touitou, D. (1995). Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA. ACM.

[Shivers, 1991] Shivers, O. (1991). *Control-flow analysis of higher-order languages*. PhD thesis, CMU.

[Smaragdakis et al., 2007] Smaragdakis, Y., Kay, A., Behrends, R., and Young, M. (2007). Transactions with isolation and cooperation. *SIGPLAN Not.*, 42(10):191–210.

[Spaccasassi, 2013] Spaccasassi, C. (2013). Transactional concurrent ml. Technical Report TCD-CS-2013-04, Trinity College Dublin.

[Spaccasassi and Koutavas, 2013] Spaccasassi, C. and Koutavas, V. (2013). Towards efficient abstractions for concurrent consensus. *CoRR*, abs/1304.1913.

[Stirling, 1998] Stirling, C. (1998). Playing games and proving properties of concurrent systems. *J. Comput. Sci. Technol.*, 13(6):482.

[Takeuchi et al., 1994] Takeuchi, K., Honda, K., and Kubo, M. (1994). An interaction-based language and its typing system. In Halatsis, C., Maritsas, D., Philokyprou, G., and Theodoridis, S., editors, *PARLE*, volume 817 of *LNCS*, pages 398–413. Springer.

[Talpin and Jouvelot, 1992] Talpin, J.-P. and Jouvelot, P. (1992). Polymorphic type, region and effect inference. *JFP*, 2:245–271.

[Tofte and Talpin, 1994] Tofte, M. and Talpin, J. (1994). Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *POPL*, pages 188–201.

[Toninho et al., 2013] Toninho, B., Caires, L., and Pfenning, F. (2013). Higher-order processes, functions, and sessions: A monadic integration. In Felleisen, M. and Gardner, P., editors, *Programming Languages and Systems*, volume 7792 of *LNCS*, pages 350–369. Springer.

[Turon, 2012] Turon, A. (2012). Reagents: expressing and composing fine-grained concurrency. *SIGPLAN Not.*, 47(6):157–168.

[Turon, 2013] Turon, A. (2013). *Understanding and expressing scalable concurrency*. PhD thesis, Northeastern University.

[Vasconcelos et al., 2006] Vasconcelos, V. T., Gay, S. J., and Ravara, A. (2006). Type checking a multithreaded functional language with session types. *Theoretical Computer Science*, 368(1-2):64–87.

[Vaz et al., 2009] Vaz, C., Ferreira, C., and Ravara, A. (2009). Dynamic recovering of long running transactions. In Kaklamanis, C. and Nielson, F., editors, *Trustworthy Global Computing*, chapter Dynamic Recovering of Long Running Transactions, pages 201–215. Springer-Verlag, Berlin, Heidelberg.

[Wadler, 2012] Wadler, P. (2012). Propositions as sessions. In *ICFP '12: Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, page 273. ACM.

[Winskel, 1993] Winskel, G. (1993). *The Formal Semantics of Programming Languages: An Introduction*. Foundations of computing. MIT Press.

[Ziarek and Jagannathan, 2010] Ziarek, L. and Jagannathan, S. (2010). Lightweight checkpointing for concurrent ml. *J. Funct. Program.*, 20(2):137–173.

[Ziarek et al., 2006] Ziarek, L., Schatz, P., and Jagannathan, S. (2006). Stabilizers: a modular checkpointing abstraction for concurrent functional programs. *SIGPLAN Not.*, 41(9):136–147.