TRINITY COLLEGE DUBLIN
COLÁISTE NA TRÍONÓIDE, BAILE ÁTHA CLIATH

Transactional Concurrent ML

Carlo Spaccasassi

Computer Science Department Technical Report **TCD-CS-2013-01**
Foundations and Methods Research Group **February 15, 2013**

Transactional Concurrent ML

Carlo Spaccasassi

February 15, 2013

email address: *spaccasc@scss.tcd.ie*

Abstract

Transactional Memory is a now popular abstraction to implement mutual exclusion in concurrent shared-memory systems. TM achieves this by effectively isolating processes from each other, but for this reason it is not suitable for implementing consensus, where processes need to communicate. Recently proposed abstractions for streamlining consensus in concurrent programming also take a transactional form, but these transactions are allowed to communicate with their environment rather than being isolated from it.

We present TransCML, a functional programming language that adds Communicating Transactions to Concurrent ML. Communicating Transactions are an abstraction developed within TransCCS, a calculus that extends Milners CCS and that has a simple behavioural theory capturing the notions of safety and liveness. The ultimate goal of this work is to develop a language with an intuitive semantics, simple reasoning techniques and an efficient runtime system, and in which programmers can simply specify the local conditions for consensus, without spelling out how it can be achieved.

Contents

1 Introduction

Transactional memory is a concurrent programming technique that has been recently praised for significantly reducing at the same time the implementation and verification effort to build concurrent system. In a recently drawn analogy [8], transactional memory for shared-memory concurrency has been compared to garbage collection for memory management. Just as garbage collection eases memory management in a program by automating memory reclamation, transactional memory greatly simplifies concurrent programming by automatically managing conflicting operations on a region of memory shared by cooperating processes.

In distributed systems, consensus [12] is a class of problems that is missing programming language support comparable to garbage collection. In general, solutions for consensus problems are well known, such the Paxos algorithm [13]. On the one hand, implementations in low-level languages can be quite complicated to program, scaling up to several thousands of lines of C++ code [3]. On the other hand, high-level implementations tend to be ad-hoc solutions that do not integrate well with the rest of the language. For example, it is not possible to write a three-way rendezvous in CML [16], and Haskell's STM cannot implement a composable swap operation inside an STM transaction [9]. These short-comings point to the need of programming language support to address consensus problems.

On the wave of the success of transactional memory, the traditional concept of transactions has attracted attention in distributed systems too. Many constructs drawing from this concept have been proposed recently, such as cJoin[1], communicating memory transactions [14], transactors [18], stabilizer [7]. This report focuses on *communicating transactions*, a construct proposed in [4] that drops the isolation requirement from traditional transactions to model automatic system recovery in distributed systems. We present TransCML, or TCML, a functional language that draws inspiration for its concurrency model from a core version of Concurrent ML [11] and its transactional model from TransCCS in [4]. TransCML serves as the basis to investigate communicating transactions in the more concrete setting of a concurrent functional language.

To draw a further analogy, transactions in TCML are similar to exception-handling blocks, in which exceptions are thrown whenever processes in a transaction are deadlocked. Whenever such an exception is thrown, all side effect that took place during the execution of a transactional block are automatically annulled. Exceptions are automatically raised by the system, programmers do not need to specify aborting points to throw them. Consider for example the following scenario, in which some friends, among which Alice, Bob and Carol, want to find a partner for all activities that each of them has planned for the night. Let us assume that Alice wants to find a dinner partner and a salsa dancing partner, Bob a dinner and cinema partner, and Carol a salsa dancing partner. If Alice, Bob and Carol were to go out together, they would not be able to reach an agreement, because Bob has no movie partner. In an ad-hoc solution, a programmer would have to encode a protocol to find an agreement and also to handle the failure cases, where an agreement cannot be met. Such a solution would be burdersome to implement and hard to prove correct. In TCML this scenario can be implemented by the following code:

`// Alice`
$\textbf{atomic}_{rec\ alice}\,[\![\,\mathit{dinner}\ ();\mathit{salsa}\ ();\textbf{commit}\ \mathit{alice}\,]\!]$

`// Bob`
$\textbf{atomic}_{rec\ bob}\,[\![\,\mathit{dinner}\ ();\mathit{cinema}\ ();\textbf{commit}\ \mathit{bob}\,]\!]$

`// Carol`
$\textbf{atomic}_{rec\ carol}\,[\![\,\mathit{salsa}\ ();\textbf{commit}\ \mathit{carol}\,]\!]$

where *dinner*, *salsa* and *cinema* are protocols that involve some communication among two partners to agree on where to go for dinner, salsa dancing or for a movie. In TransCML failure is handled automatically, and the programmer needs not to worry about recovering from deadlocks for example; for the sake of efficiency, partial agreement can be preserved (for example an eventual partial agreement between Alice and Bob for dinner), instead of aborting the whole agreement, in case consensus can be reached by a different set of participants (for example by replacing Carol with a friend that matches Alice and Bob's activities).

In this report we show the reduction semantics for TransCML, a type system along with a proof of soundness. We discuss the design choices that led to the formulation of the language as it is. We provide examples of common interaction patterns such as a single consumer/multiple consumers scenario, and show the expressiveness of the language by implementing a three-way rendezvous operation and guarded commands. The example of Alice, Bob and Carol is an example of three-way rendezvous. We also show a parallel between Prolog's backtracking capabilities and TransCML restarting transactions, through the standard example of graph search, demonstrating how TransCML can be used for speculative computing [2, 1]. Finally, we provide a draft Labelled Transition System, to better study the dynamic behaviour of individual processes and to guide a future implementation of the language.

2 Language definition

We will now present the syntax, type system and reduction semantics for TCML. We will prove subject reduction, among with other lemmas such as the substitution lemma. Our work is inspired by Reppy's [16] Concurrent ML and Jeffrey's core μCML [11], and by TransCCS in [4] and [5]. Notably, TCML does not feature CML's *Events* but *communicating transactions*, or more simply transactions, in their place.

2.1 Syntax

Table 1 summarizes TCML's syntax. We assume a countable set of variables *Var*, channel names *Chan* and transaction names $\mathcal{K}$. The set $\mathbb{N}$ describes the set of natural numbers.

We can divide TCML's syntax in two main categories: a functional core and processes. On the one hand, the functional core comprises common expressions, values and operations in programming languages, such as **if - then - else** expressions and natural numbers. On the other hand, an expression is also a process, which can be run in parallel with other processes and eventually communicate with them over channels. Processes can also run within a transaction, which uses the syntax of $[\![\, P \;\rhd_k\; P \,]\!]$.

$$\begin{array}{lll}
\mathrm{v} & ::= & () \mid \textbf{true} \mid \textbf{false} \mid n \mid x \mid c \\
 & \mid & (v,v) \mid \textbf{fun}\ f(x) = e \\
 & & \\
\mathrm{e} & ::= & v \mid (e,e) \mid e\ e \mid op\ e \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \\
 & \mid & \textbf{send}\ e\ e \mid \textbf{recv}\ e \mid \textbf{newChan}_A \mid \textbf{spawn}\ e \\
 & \mid & \textbf{atomic}\ [\![\, e \rhd_k e \,]\!] \mid \textbf{commit}\ k \\
op & ::= & \textbf{fst} \mid \textbf{snd} \mid \textbf{add} \mid \textbf{sub} \mid \textbf{mul} \mid \textbf{leq} \\
 & & \\
P & ::= & e \mid P \parallel P \mid \nu c.P \mid [\![\, P \rhd_k P \,]\!] \mid \textbf{co}\ k \\
 & & \\
\mathrm{A} & ::= & \textbf{unit} \mid \textbf{bool} \mid \textbf{int} \mid A \times A \mid A \rightarrow A \mid A\,\textbf{chan}
\end{array}$$

where $n \in \mathbb{N}$, $x \in Var$, $c \in Chan$, $k \in \mathcal{K}$

Table 1: TCML syntax.

Let us briefly describe the functional core first. Values comprise Natural numbers, boolean values, a unit value, pairs and variables; functions are first-class values too. The language provides conditional and let expressions, function application and a small standard set of operation on pairs (**fst** and **snd**), numbers (**add**, **mul** and **sub**) and booleans (**leq**).

We introduce some syntactical conventions for functions and sequencing of terms. We write **fun** $f() = e$ for **fun** $f(x) = e$, where x does not occur free in e. We also write $\textbf{fun}\ f(x_0, x_1, \ldots, x_n) = e$ for $\textbf{fun}\ f(x_0) = \textbf{fun}\ f_1(x_1) = \ldots \textbf{fun}\ f_n(x_n) = e$, where $f_1, \ldots, f_n$ do not occur free in e. We abbreviate $\textbf{let}\ x = e_1\ \textbf{in}\ e_2$ to $e_1;\ e_2$, where x does not occur free in e_2.

Primitives to spawn new processes, send and receive messages over channels and to create channels are provided too. In particular, new processes can be spawned by the **spawn** expressions. A value can be sent over a channel from one process to another by the **send** expression. A process can receive a value over a channel from another process using the **recv** expression; two processes can communicate with each other with the **send** and **recv** primitives only if they are using the same channel. New channels can be created by the $\textbf{newChan}_A$ expression, where A is the type of value that the generated channel transports.

Two additional expressions are available, **atomic** and **commit**, to start a new transaction and to create a new commit point respectively. We will shortly describe them at greater length.

We now describe processes. In TCML, processes run concurrently, are composed by the parallel construct $\|$, and communicate using synchronous channels. Processes send and receive values over channels using **send** and **recv**. For example, the following two processes will send integer 1 over channel c and receive it on the same channel:

$$\textbf{send}\ c\ 1 \parallel \textbf{recv}\ c$$

New processes can be spawned using the **spawn** construct. The following term will evolve to the two processes described above:

$$\textbf{let}\ z = \textbf{spawn}\,(\textbf{fun}\ f() = \textbf{send}\ c\ 1)\ \textbf{in}\ \textbf{recv}\ c$$

As already mentioned, TCML's evaluation strategy is eager. Thus the expression bound to z in the above example must be fully reduced to a value before the body of the **let** expression is evaluated. In this case, when the evaluation of the **spawn** expression will generate a new process and then reduce to the unit value (). This value only indicates that process generation has been successful. Since the value itself is not interesting for the rest of the expression, it is discarded by not binding z anywhere in the body of the **let** expression. We can abbreviate this example with the ';′ abbreviation introduced earlier as follows:

$$\textbf{spawn}\,(\textbf{fun}\ f() = \textbf{send}\ c\ 1);\ \textbf{recv}\ c$$

Channels can be either public or private. As mentioned earlier, we assume to have an infinite supply of channels *Chan*. Private channels are marked by $\nu c.e$ syntax, meaning that the channel name c is bound in the term e and cannot be used outside the scope of e. New private channels can be created by the expression $\textbf{newChan}_A$, where A is the type of values that can be communicated over that channel.

If we want the earlier conversation to be private, so that no other process can interfere with it, we can create a new private channel as follows:

$$\textbf{let}\ c = \textbf{newChan}_{\textbf{int}}\ \textbf{in}\ \textbf{spawn}\,(\textbf{fun}\ f() = \textbf{send}\ c\ 1);\ \textbf{recv}\ c$$

As already mentioned in the introduction, TCML provides *transactions* and *commit points* in place of CML's *Events.*

A running transaction has the form $[\![\, P_1 \ \rhd_k \ P_2 \,]\!]$, where P_1 is called the *default* process of transaction k, whereas P_2 is called the *alternative* process. Since a process P_1 can be the parallel composition of processes, we might also refer to P_1 and P_2 as the default and alternative *processes.* Running transactions can also contain *commit points* $\textbf{co}\ k$, where k is a variable bound to the transaction. Commit points only exist within a transaction; the type system disallows commit points to appear outside of transactions.

A new transaction k can be started from an $\textbf{atomic}\ [\![\, e_1 \rhd_k e_2 \,]\!]$ expression, where the *default expression* e_1 will constitute the default process of the transaction, and the *alternative expression* e_2 will constitute its alternative process. Commit points for a transaction k can be spawned by $\textbf{commit}\ k$ expressions. Up to the moment when an $\textbf{atomic}\ [\![\, e_1 \rhd_k e_2 \,]\!]$ expression has been evaluated, no transaction $[\![\, e_1 \ \rhd_k \ e_2 \,]\!]$ is started, and neither can the default expression be evaluated. Similarly, a commit point $\textbf{co}\ k$ is only created after a $\textbf{commit}\ k$ is evaluated. Expressions **atomic** and **commit** activate, or trigger, a transaction and a commit point, respectively.

Consider the following example:

$$[\![\, \textbf{send}\ c\ 0;\ \textbf{send}\ c\ 1;\ \textbf{send}\ c\ 2; \textbf{commit}\ k \ \rhd_k \ () \,]\!]$$

Intuitively, the default part of a transaction is an expression that is tentatively executed, until the transaction is finalized by a commit point. The alternative is an expression that can replace the default process at any time before committing. In this example, the transaction tries to send three numbers over channel c and then commits. In order for the transaction to commit, the three communications must have taken place. Since communications are tentative before committing, the behaviour of this example transaction is to either send exactly three numbers over c, or do nothing.

Transactions can also be programmed to repeatedly try the default expression in case of abortion, until they are committed. For example, the following example will repeatedly try to send exactly two numbers over channel c:

$$\textbf{fun}\ f() = \textbf{atomic}\ [\![\, \textbf{send}\ c\ 0;\ \textbf{send}\ c\ 1 \rhd_k \ f\ () \,]\!]$$

This transaction will try to send both numbers 0 and 1 over channel c. If the transaction is aborted, the alternative expression will recursively run the same transaction we started with. These kind of transactions are referred to as *restarting transactions.* We will abbreviate restarting transactions as follows:

$$\textbf{atomic}_{rec\ k}\, [\![\, e \,]\!] = \textbf{fun}\ f() = \textbf{atomic}\ [\![\, e \rhd_k f\ () \,]\!]$$

2.1.1 Syntax operators

Before presenting the type system and the reduction semantics of TCML, we need to define some predicates over processes and expressions. The *free variables*

$FV(()) = \emptyset$	$FV(x \in Var) = \{x\}$
$FV(n) = \emptyset$	$FV(\textbf{fun } f(x) = e) = FV(e)/\{f, x\}$
$FV(\textbf{true}) = \emptyset$	$FV((v_1, v_2)) = FV(v_1) \cup FV(v_2)$
$FV(\textbf{false}) = \emptyset$	$FV(c \in Chan) = \emptyset$
$FV(e_1 e_2) = FV(e_1) \cup FV(e_2)$	$FV(op\ e) = FV(e)$
$FV(\textbf{let } x = e_1 \textbf{ in } e_2) =$ $FV(e_1) \cup FV(e_2) \backslash x$	$FV(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) =$ $FV(e_1) \cup FV(e_2) \cup FV(e_3)$
$FV(\textbf{ send } e_1\ e_2) = FV(e_1 \cup FV(e_2))$	$FV(\textbf{ recv } e) = FV(e)$
$FV(\textbf{newChan}_A) = \emptyset$	$FV(\textbf{spawn } e) = FV(e)$
$FV(\textbf{commit } k) = \emptyset$	$FV(\textbf{atomic } \llbracket e_1 \rhd_k e_2 \rrbracket) =$ $FV(e_1) \cup FV(e_2)$
$FV(\nu c.P) = FV(P)$	$FV(P_1 \parallel P_2) = FV(P_1) \cup FV(P_2)$
$FV(\llbracket P_1 \rhd_k P_2 \rrbracket) = FV(P_1) \cup FV(P_2)$	$FV(\textbf{ co } k) = \emptyset$

Table 2: Free variables predicate FV.

predicate $FV \in Proc \longrightarrow \mathcal{P}(Var)$ is defined inductively over syntax expressions in Table 2.

An expression is called *closed* if and only if $FV(e) = \emptyset$. We define the set of closed values as $CVal = \{v | FV(v) = \emptyset\}$, the set of closed expressions as $CExp = \{e | FV(e) = \emptyset\}$ and the set of closed processes as $CProc = \{P | FV(P) = \emptyset\}$. We also say that a variable x has a *free occurrence* in a term e if $x \in FV(e)$. If a variable x occurs in e but $x \notin FV(e)$, then we will call it a *bound occurrence* of variable x in e.

Along with the FV predicate, we thus define the *free channels* predicate FC inductively over syntax expressions in Table 3. This predicate is necessary to avoid variable-capture when creating channels and to provide a correct definition of restriction.

We use the *Barendregt's variable convention*, or just Barendregt's convention, to deal with bound variables: in any mathematical context (e.g. definitions, proofs etc), all bound occurrences of variables and channel names and transaction names in an expression are different from all free occurrences of variables and channel names and transaction names in that context, and we identify terms up to alpha-conversion.

Consider for example the following expression:

$$\textbf{let } x = 1 \textbf{ in let } x = x + 1 \textbf{ in } x$$

It is not clear which one of the two **let** expression binds the variable x in the innermost body of the **let** expressions. With Barendregt's convention we can consider the above expression equivalent to the following one:

$$\textbf{let } x = 1 \textbf{ in let } y = x + 1 \textbf{ in } y$$

which is not ambiguous.

We also define the *free transaction names* predicate FTN inductively over syntax expressions in Table 4.

The substitution function $-[-/-] : Proc \times Val \times Var \to Proc$ is defined inductively over syntax expressions in Table 5.

$FC(()) = \emptyset$	$FC(x \in Var) = \emptyset$
$FC(n) = \emptyset$	$FC(\textbf{fun } f(x) = e) = FC(e)$
$FC(\textbf{true}) = \emptyset$	$FC((v_1, v_2)) = FC(v_1) \cup FC(v_2)$
$FC(\textbf{false}) = \emptyset$	$FC(c \in Chan) = c$
$FC(e_1 e_2) = FC(e_1) \cup FC(e_2)$	$FC(op\ e) = FC(e)$
$FC(\textbf{let } x = e_1 \textbf{ in } e_2) = FC(e_1) \cup FC(e_2)$	$FC(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) = FC(e_1) \cup FC(e_2) \cup FC(e_3)$
$FC(\textbf{send } e_1\ e_2) = FC(e_1) \cup FC(e_2)$	$FC(\textbf{recv } e) = FC(e)$
$FC(\textbf{newChan}_A) = \emptyset$	$FC(\textbf{spawn } e) = FC(e)$
$FC(\textbf{commit } k) = \emptyset$	$FC(\textbf{atomic } [\![e_1 \rhd_k e_2]\!]) = FC(e_1) \cup FC(e_2)$
$FC(\nu c.P) = FC(P) \backslash c$	$FC(P_1 \parallel P_2) = FC(P_1) \cup FC(P_2)$
$FC([\![P_1 \rhd_k P_2]\!]) = FC(P_1) \cup FC(P_2)$	$FC(\textbf{co } k) = \emptyset$

Table 3: Free channels predicate *FC*.

$$FTN(()) = \emptyset \qquad FTN(x \in Var) = \emptyset$$
$$FTN(n) = \emptyset \qquad FTN(\mathbf{fun}\ f(x) = e) = FTN(e)$$
$$FTN(\mathbf{true}) = \emptyset \qquad FTN((v_1, v_2)) = FTN(v_1) \cup FTN(v_2)$$
$$FTN(\mathbf{false}) = \emptyset \qquad FTN(c \in Chan) = \emptyset$$

$$FTN(e_1\ e_2) = FTN(e_1) \cup FTN(e_2) \qquad FTN(op\ e) = FTN(e)$$
$$FTN(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) = FTN(e_1) \cup FTN(e_2)/x \qquad FTN(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) = FTN(e_1) \cup FTN(e_2) \cup FTN(e_3)$$
$$FTN(\mathbf{send}\ e_1\ e_2) = FTN(e_1) \cup FTN(e_2) \qquad FTN(\mathbf{recv}\ e) = FTN(e)$$
$$FTN(\mathbf{newChan}_A) = \emptyset \qquad FTN(\mathbf{spawn}\ e) = FTN(e)$$
$$FTN(\mathbf{commit}\ k) = \{k\} \qquad FTN(\mathbf{atomic}\ [\![e_1 \rhd_k e_2]\!]) = (FTN(e_1) \backslash k) \cup FTN(e_2)$$

$$FTN(\nu c.P) = FTN(P) \qquad FTN(P_1 \parallel P_2) = FTN(P_1) \cup FTN(P_2)$$
$$FTN([\![P_1 \rhd_k P_2]\!]) = FTN(P_1) \cup FTN(P_2) \backslash k \qquad FTN(\mathbf{co}\ k) = \{k\}$$

Table 4: Free transaction names predicate *FTN*.

$$()[v/x] = () \qquad x[v/x] = v$$
$$n[v/x] = n \qquad x'[v/x] = x' \text{ if } x \neq x'$$
$$\mathbf{false}[v/x] = \mathbf{false} \qquad (e_1, e_2)[v/x] = (e_1[v/x], e_2[v/x])$$
$$\mathbf{true}[v/x] = \mathbf{true} \qquad c[v/x] = c$$

$$(\mathbf{fun}\ f(x') = e)[v/x] = \mathbf{fun}\ f(x') = e[v/x], \text{ if } x \neq x',\ x \neq f \text{ and } f, x' \notin FV(v)$$
$$(\mathbf{let}\ x' = e_1\ \mathbf{in}\ e_2)[v/x] = \mathbf{let}\ x' = e_1[v/x]\ \mathbf{in}\ e_2[v/x], \text{ if } x \neq x' \text{ and } x' \notin FV(v)$$

$$(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3)[v/x] = \mathbf{if}\ e_1[v/x]\ \mathbf{then}\ e_2[v/x]\ \mathbf{else}\ e_3[v/x] \qquad (e_1\ e_2)[v/x] = (e_1[v/x]\ e_2[v/x])$$
$$(\mathbf{send}\ e_1\ e_2)[v/x] = \mathbf{send}\ e_1[v/x]\ e_2[v/x] \qquad op\ e[v/x] = op\ e[v/x]$$
$$(\mathbf{recv}\ e)[v/x] = \mathbf{recv}\ e[v/x] \qquad \mathbf{spawn}\ e[v/x] = \mathbf{spawn}\ e[v/x]$$
$$\mathbf{newChan}_A[v/x] = \mathbf{newChan}_A \qquad \mathbf{atomic}\ [\![e_1 \rhd_k e_2]\!][v/x] = \mathbf{atomic}\ [\![e_1[v/x] \rhd_k e_2[v/x]]\!]$$
$$\mathbf{commit}\ k[v/x] = \mathbf{commit}\ k$$

$$(\nu c.e)[v/x] = \nu c.e[v/x], \text{ if } c \notin FC(v) \qquad (e_1 \parallel e_2)[v/x] = e_1[v/x] \parallel e_2[v/x]$$
$$[\![e_1 \rhd_k e_2]\!][v/x] = [\![e_1[v/x] \rhd_k e_2[v/x]]\!] \qquad \mathbf{co}\ k[v/x] = \mathbf{co}\ k$$

Table 5: Substitution function.

2.2 Reduction semantics

TCML's evaluation strategy is eager, in that expressions can take a functional reduction step only if the required arguments are values. In particular, it is left-to-right and call-by-value. We can categorize evaluation rules in expression rules and process rules. There are four main sets of rules, respectively three sets for sequential, concurrency, transactions reduction rules, and one set for structural equivalence.

Before further introducing the reduction semantics, we extend the syntax in Table 1 with the notion of *evaluation contexts*.

$$\begin{array}{lcl} E & ::= & [] \mid op\ E \mid (E, e) \mid (v, E) \mid E\ e \mid v\ E \\ & \mid & \mathbf{if}\ E\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid \mathbf{let}\ x = E\ \mathbf{in}\ e \\ & \mid & \mathbf{send}\ E\ e \mid \mathbf{send}\ v\ E \mid \mathbf{recv}\ E \mid \mathbf{spawn}\ E \end{array}$$

Table 6: Evaluation contexts syntax.

As in CML [16], we use Felleisen's evaluation contexts [17] in the definition of TCML's reduction semantics. An evaluation context E is a function from an expression to an expression, inductively defined by a grammar syntax. For example, [] is the identity function, mapping an expression to itself; $op\ E$ maps an expression e to $op(E\ e)$. TCML's evaluation strategy is eager and left-to-right, thus evaluation contexts are defined left-to-right too. Intuitively, an evaluation context uniquely identifies a "hole" in an expression. The hole contains an expression that can take a reduction step within the larger expression. Note that there an evaluation context never falls under the scope of a binder, thus there is no danger of free variable capture when inserting an expression e into the hole of any evaluation context. Evaluation contexts are very useful to separate expression

Sequential reduction rules are marked by $\hookrightarrow$: $CExp \rightharpoonup CExp$ transitions. Concurrency and transactions reduction rules are marked by $\longrightarrow : CProc \rightharpoonup CProc$.

Table 7 present the set of reduction rules for the sequential part of TCML. Sequential reduction steps are represented by the transition arrow $\hookrightarrow$.

In rule [E-App] , function application can be reduced only if it contains a **fun** $f(x) = e$ value as an operator and another value as its operand; in the body of the function every occurrence of f is substituted by the operand, and every occurrence of x is substituted by the operator. Similarly in rule [E-Let] , **let** expressions can be reduced only if the argument bound to the variable is a value. Its reduction substitutes that value for occurrences of x in the body of the expression. Conditional expressions are reduced as expected in rules [E-If-True] and [E-If-False] . According to rule [E-Op] , operations can only be reduced on values as well.

The actual evaluation of operations is summarized in the separate Table 8, where the function $\delta \in op \times Val \longrightarrow Val$ assigns to each operation and its argu-

[E-If-True]

$$\frac{}{\textbf{if true then } e_1 \textbf{ else } e_2 \hookrightarrow e_1}$$

[E-If-False]

$$\frac{}{\textbf{if false then } e_1 \textbf{ else } e_2 \hookrightarrow e_2}$$

[E-Let]

$$\frac{}{\textbf{let } x = v \textbf{ in } e \hookrightarrow e[v/x]}$$

[E-Op]

$$\frac{}{op\ v \hookrightarrow \delta(op, v)}$$

[E-App]

$$\frac{}{\textbf{fun } f(x) = e\ v_2 \hookrightarrow e[\textbf{fun } f(x) = e/f][v_2/x]}$$

Table 7: Sequential reduction rules

$$\begin{array}{ll}
\delta(\textbf{fst}, (v, w)) = v & \delta(\textbf{snd}, (v, w)) = w \\
\delta(\textbf{add}, (n, m)) = n + m & \delta(\textbf{sub}\,, (n, m)) = n - m \\
\delta(\textbf{mul}, (n, m)) = n * m & \delta(\textbf{leq}\,(n, m)) = \textbf{true} \text{ if } n \leq m, \\
 & \qquad\qquad\qquad \textbf{false}\ \textit{otherwise}
\end{array}$$

Table 8: Operations relation.

ment a corresponding value. Operations on values have the expected semantics. Operations **fst** and **snd** project the first and second component of a pair value. Operations **add** , **mul** and **sub** respectively add, subtract and multiply pairs of values. Operation **leq** returns a **true** if the first component of a pair is lesser or equal than the second component, and **false** otherwise.

[C-Spawn]

$$\frac{}{E[\textbf{spawn } v] \longrightarrow v\ () \parallel E[()]}$$

[C-NewChan]

$$\frac{}{E[\textbf{newChan}_A] \longrightarrow \nu c.E[c]} \quad \begin{array}{l} ty(c) = A\,\textbf{chan} \\ c \notin FC(E[()]) \end{array}$$

[Tr-Commit]

$$\frac{}{E[\textbf{commit } k] \longrightarrow \textbf{co } k \parallel E[()]}$$

[Tr-Atomic]

$$\frac{}{E[\textbf{atomic } [\![\, e_1 \rhd_k e_2 \,]\!]] \longrightarrow [\![\, E[e_1] \rhd_k E[e_2] \,]\!]}$$

Table 9: Expression reduction rules

The set of rules contained in Table 9 describes how to generate TCML processes starting from expressions. Remember from Table 1 that there are five kinds of processes in TCML's syntax: expressions, parallel composition of processes, channel restriction, transactions and commit points.

New processes can be spawned by the **spawn** expression, whose semantics is given by rule [C-Spawn]. The new process starts as a function application, in which a value v is applied to a unit value $()$. The new process will start executing as soon as the function application $(v\ ())$ is reduced. The originating process receives a unit value $()$, as a sign that a new process has been spawned. New channels can be generated by the $\mathbf{newChan}_A$ expression, as described by the [C-NewChan] rule. Notice how the first side condition on the rule enforces that the new channel c has the same type as the explicit type contained in the $\mathbf{newChan}_A$ expression. The second side condition avoids the problem of variable capture, so that no free channel is mistakenly bound by restriction νc.

Given a default and an alternative expression, new transactions can be spawned by the **atomic** expression as described by rule [Tr-Atomic]. For example, suppose that we wanted a process to send the integer 1 over channel c atomically, or alternatively to send the integer 2 over channel c. This process can be written as follows:

$$\mathbf{atomic}\ \llbracket(\mathbf{send}\ c\ 1; \mathbf{commit}\ k) \rhd_k \mathbf{send}\ c\ 2\rrbracket$$

The default expression of this transaction is $\mathbf{send}\ c\ 1; \mathbf{commit}\ k$, the alternative part is $\mathbf{send}\ c\ 1$. The role of the $\mathbf{commit}\ k$ expression will be explained later. After applying rule [Tr-Atomic], the above expression generates a new transaction as follows:

$$\llbracket\ \mathbf{send}\ c\ 1; \mathbf{commit}\ k\ \ \rhd_k\ \ \mathbf{send}\ c\ 2\rrbracket$$

Notice that rule [Tr-Atomic] applies the evaluation context in which the **atomic** expression is, to both the default and alternative expressions. This will allow the original evaluation context, containing the **atomic** expression, to be restored in case the transaction is aborted. For example, consider the following expression:

$$\mathbf{let}\ x = \mathbf{atomic}\ \llbracket\ \mathbf{recv}\ c \rhd_k \mathbf{recv}\ d\rrbracket\ \mathbf{in}\ x+1$$

The evaluation context of this expression is $\mathbf{let}\ x = []\ \mathbf{in}\ x+1$. The transaction that will be created by rule [Tr-Atomic] is the following:

$$\llbracket\mathbf{let}\ x =\ \mathbf{recv}\ c\ \mathbf{in}\ x+1\ \ \rhd_k\ \ \mathbf{let}\ x =\ \mathbf{recv}\ d\ \mathbf{in}\ x+1\rrbracket$$

When the **atomic** expression is reduced, the **let** expression is copied to both the default and alternative part.

Lastly, commit points can be spawned by rule [Tr-Commit] when a **commit** expression is reached. Let us consider the following process:

$$\llbracket\mathbf{commit}\ k\ \ \rhd_k\ \ \mathbf{send}\ c\ 2\rrbracket$$

After applying rule [Tr-Commit], the process will reduce to the following:

$$\llbracket\ \mathbf{co}\ k \parallel ()\ \ \rhd_k\ \ \mathbf{send}\ c\ 2\rrbracket$$

$$\frac{e \hookrightarrow e'}{E[e] \longrightarrow E[e']}\ [\textsc{C-Step}] \qquad \frac{P_1 \longrightarrow P_1'}{P_1 \parallel P_2 \longrightarrow P_1' \parallel P_2}\ [\textsc{C-Par}] \qquad \frac{P \longrightarrow P'}{\nu c.P \longrightarrow \nu c.P'}\ [\textsc{C-Chan}]$$

$$\frac{}{E_1[\,\mathbf{recv}\ c] \parallel E_2[\,\mathbf{send}\ c\ v] \longrightarrow E_1[v] \parallel E_2[()]}\ [\textsc{C-Sync}]$$

Table 10: Concurrency reduction rules

Notice how a transaction contains processes, such as a parallel composition of a commit point and a unit value, rather than an expression. Before the expression **commit** k was evaluated, the transaction could not commit. Only after creating a commit point **co** k, transaction k can commit.

Table 10 contains the set of rules describing concurrency in TCML. Rule [C-Step] allows functions to be reduced. This rule lifts functional reductions to processes and reduces it.

A process can communicate a value v to another process a channel c. This situation is captured in rule [C-Sync], where two parallel processes contain **send** $c\ v$ and **recv** c in their respective evaluation contexts. After the reduction step is performed, value v is passed to the receiving process, whereas the sending process receives a unit value $()$. Communication is thus synchronous, because in one step both processes send and receive a value over channel c.

Parallel processes and processes under channel restriction can each take reduction step independently if they can, as shown in rules [C-Chan] and [C-Par].

$$\frac{P \longrightarrow P'}{[\![\, P\ \rhd_k\ P_2 \,]\!] \longrightarrow [\![\, P'\ \rhd_k\ P_2 \,]\!]}\ [\textsc{Tr-Step}] \qquad \frac{}{[\![\, P_1 \rhd_k P_2 \,]\!] \longrightarrow P_2}\ [\textsc{Tr-Abort}]$$

$$\frac{}{P_1 \parallel [\![\, P_2\ \rhd_k\ P_3 \,]\!] \longrightarrow [\![\, P_1 \parallel P_2\ \rhd_k\ P_1 \parallel P_3 \,]\!]}\ [\textsc{Tr-Emb}] \qquad \frac{P_1 \equiv\ \mathbf{co}\ k \parallel P_1'}{[\![\, P_1 \rhd_k P_2 \,]\!] \longrightarrow P_1'/k}\ [\textsc{Tr-Co}]$$

Table 11: Transactions reduction rules

Table 11 describes reduction rules for transactions.

Only processes in the default part of a transaction are allowed to take reduction steps, according to rule [Tr-Step]. Transactions can be aborted at any time, as shown in rule [Tr-Abort]. A process parallel to a transaction can be embedded into it by rule [Tr-Emb]. When a process is embedded, it joins both the processes in the default part of the transaction and those in the alternative part. Remember however that the alternative processes never take any reduction step as long as the transaction is not aborted. In summary, the embedding mechanism stores a process in the alternative part of a transaction and puts it in parallel with the processes in its default part.

Let us see how transactions work with an example. Suppose we had the following processes:

$$\llbracket\ \textbf{send}\ c\ 1; \textbf{send}\ c\ 2; \textbf{commit}\ k\ \rhd_k\ \ \textbf{send}\ c\ 2\ \rrbracket \parallel \textbf{recv}\ c$$

The first process is a transaction that sends two numbers over channel c, 1 and 2, and then commits. Alternatively it sends the number 2 over channel c. The other process just receives a single number from channel c.

An important point to make is that transactions are not evaluation contexts. Because of this, processes within a transaction cannot communicate with processes outside of it, until they are embedded in the transaction. In fact, consider rule [C-Sync] from Table 10: channel communication only happens between parallel processes if **send** and **recv** expressions are available in their respective evaluation contexts. But transactions are not evaluation contexts, thus processes that could communicate without transactions cannot do so unless they belong to the same transaction.

In our example, the process within the transaction cannot communicate with the process outside of it. In order for them to communicate, we can embed the process on the right into the transaction using rule [Tr-Emb]:

$$\llbracket\ \textbf{send}\ c\ 1; \textbf{send}\ c\ 2; \textbf{commit}\ k \parallel \textbf{recv}\ c\ \ \rhd_k\ \ \textbf{send}\ c\ 2 \parallel \textbf{recv}\ c\,\rrbracket$$

The processes can now synchronize over channel c using rule [Tr-Step], which in turn is enabled by rule [C-Sync]:

$$\llbracket\ \textbf{send}\ c\ 2; \textbf{commit}\ k \parallel 1\ \ \rhd_k\ \ \textbf{send}\ 2\ \ \parallel \textbf{recv}\ c\,\rrbracket$$

Evaluation is now stuck in the default part of the transaction, since there is no other process with which to synchronize over c. The only transition now available for the system is to abort the transaction using rule [Tr-Abort]:

$$\textbf{send}\ 2\ \ \parallel \textbf{recv}\ c$$

The embedding mechanism constitutes the core of communicating transactions. Processes that want to collaborate with a transaction can do so, but only if they register their own state in the alternative part of that transaction first. In this way, if the transaction is aborted, all reductions and communications among default processes are discarded in case of a transaction abort. The stored processes are then run instead.

The committing process can continue evaluation, while the transaction can be committed at any time by rule [Tr-Co]. When the transaction is committed, all references to transaction k are removed from the default processes by the *strip* function $\backslash k$.

Consider the following example:

$$\llbracket\ \textbf{send}\ c\ 1; \textbf{commit}\ k \parallel \textbf{recv}\ c\ \rhd_k\ \ \textbf{send}\ 2\ \ \parallel \textbf{recv}\ c \rrbracket$$

After performing a synchronization step with rule [C-SYNC] and generating a commit point by rule [TR-COMMIT], the above system evolves to the following:

$$\llbracket\ \textbf{co}\ k \parallel () \parallel 1\ \ \rhd_k\ \ \textbf{send}\ 2\ \ \parallel \textbf{recv}\ c \rrbracket$$

Transaction k is ready to commit now. Applying rule [TR-CO], the system will finally reduce to the following:

$$() \parallel 1$$

Upon commit of transaction k, the alternative processes stored in it have been erased.

The strip function $-/- : Proc \times TName \longrightarrow Proc$ is defined inductively over syntax expressions in Table 12. Multiple commit points might be generated for the same transaction. When one of these commit points is selected to commit the transaction, the other ones become a dangling reference to a transaction that does not exist any more. The strip function takes care of removing these dangling reference. For example, consider the following transaction:

$$\llbracket\ \textbf{co}\ k \parallel\ \textbf{co}\ k \parallel ()\ \ \rhd_k\ \ \textbf{send}\ c\ 2 \rrbracket$$

This transaction is ready to commit. Suppose that we chose to commit it using the first commit point. Then, according to rule [TR-CO] the transaction will reduce to:

$$(\ \textbf{co}\ k \parallel ())\backslash k$$

which is equal to $(\ \textbf{co}\ k)\backslash k \parallel ()\backslash k$, which in turn is equal to $() \parallel ()$.

Finally we present the structural equivalence rules. Rules [EQ-ASSOC] and [EQ-COM] introduce associativity and commutativity of parallel processes, allowing processes in parallel composition to be rearranged. Rule [EQ-RESTR] allows restrictions to be moved across parallel processes, provided that variable capture is avoided, as specified in the side condition.

Intuitively, a TCML program is structurally equivalent to one where a sequence of channel restrictions contains all parallel processes. Processes within a transaction can also be put in this form by structural equivalence.

$$
\begin{array}{ll}
()/k = () & n/k = n \\
\textbf{false}\ /k = \textbf{false} & \textbf{true}\ /k = \textbf{true} \\
c/k = c & (e_1, e_2)/k = (e_1/k, e_2/k) \\
x/k = x & (\textbf{fun}\ f(x) = e)/k = \\
 & \quad \textbf{fun}\ f(x) = e/k \\
 & \\
(\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3)/k = & \textbf{let}\ x = e_1\ \textbf{in}\ e_2)/k = \\
\quad \textbf{if}\ e_1/k\ \textbf{then}\ e_2/k\ \textbf{else}\ e_3/k & \quad \textbf{let}\ x = e_1/k\ \textbf{in}\ e_2/k \\
(e_1\ e_2)/k = (e_1/k\ e_2/k) & (\mathit{op}\ e)/k = \mathit{op}\,(e/k) \\
(\,\textbf{send}\ e_1\ e_2)/k = \ \textbf{send}\ e_1/k\ e_2/k & (\,\textbf{recv}\ e)/k = \ \textbf{recv}\ e/k \\
\textbf{newChan}_{\,A}\ /k = \textbf{newChan}_{\,A} & (\textbf{spawn}\ e)/k = \textbf{spawn}\ e/k \\
\textbf{commit}\ k/k = () & \textbf{atomic}\ [\![\, e_1 \rhd_l e_2 \,]\!]\ /k = \\
\textbf{commit}\ l/k = \textbf{commit}\ k, \text{ if } k \neq l & \quad \textbf{atomic}\ [\![\, e_1/k \rhd_l e_2/k \,]\!], \text{ if } k \neq l \\
 & \\
(\,\nu c.e)/k = \ \nu c.(e/k) & (e_1 \parallel e_2)/k = e_1/k \parallel e_2/k \\
\ \textbf{co}\ k/k = () & [\![\, e_1 \ \ \rhd_l \ \ e_2 \,]\!]\ /k = \\
\ \textbf{co}\ k'/k = \ \textbf{co}\ k', \text{ if } k \neq k' & \quad [\![\, e_1/k \ \ \rhd_l \ \ e_2/k \,]\!] \text{ if } k \neq l
\end{array}
$$

Table 12: Strip function.

[C-Eq]

$$\frac{P_1 \equiv P_1' \quad P_1' \longrightarrow P_2' \quad P_2' \equiv P_2}{P_1 \longrightarrow P_2}$$

[Eq-Restr]

$$\frac{}{(\nu c.P_1) \parallel P_2 \equiv \nu c.(P_1 \parallel P_2)}\ c \notin FC(P_2)$$

[Eq-Assoc]

$$\frac{}{P_1 \parallel (P_2 \parallel P_3) \equiv (P_1 \parallel P_2) \parallel P_3}$$

[Eq-Com]

$$\frac{}{P_1 \parallel P_2 \equiv P_2 \parallel P_1}$$

Table 13: Structural equivalence rules.

2.3 Type system

We start the description of TCML's type system by the formal definition of type judgements.

We define the set of all types $Types$, specified by induction in Table 1. We also define the set of variable bindings $\mathcal{B} = \{x : A | x \in Var \wedge A \in Types\}$. A typing context $\Gamma \in Contexts = \mathcal{P}(\mathcal{K}) \times \mathcal{P}(\mathcal{B})$ is a pair containing a set of transaction names in the first component and a set of bindings in the second component. The operation that projects the first component of a typing context Γ is $\pi_1 : Contexts \longrightarrow \mathcal{P}(\mathcal{K})$ whereas $\pi_2 : Contexts \longrightarrow \mathcal{P}(\mathcal{B})$ is the operation that projects the variable bindings associated to a typing context. We define the *domain* of a typing context Γ as the set of both transaction names and variable names in Γ. We denote this function as $dom(\Gamma)$. Thus $dom(\Gamma) = \pi_1(\Gamma) \cup \{x | \exists A.\{x : A\} \in \pi_2(\Gamma)\}$. We also define two operators $,_{\mathcal{K}} : Contexts \times \mathcal{K} \rightharpoonup Contexts$ and $,_{\mathcal{B}} : Contexts \times \mathcal{B} \rightharpoonup Contexts$, defined as follows:

- $\Gamma,_{\mathcal{K}} k = (\pi_1(\Gamma) \cup \{k\}, \pi_2(\Gamma))$, if $k \notin \pi_1(\Gamma)$
- $\Gamma,_{\mathcal{B}} x : A = (\pi_1(\Gamma), \pi_2(\Gamma) \cup \{x : A\})$, if $x \notin dom(\Gamma)$

We will only write Γ, k for $\Gamma,_{\mathcal{K}} k$ and $\Gamma, x : A$ for $\Gamma,_{\mathcal{B}} x : A$ whenever the difference is obvious from the context.

[T-Unit]
$$\frac{}{\Gamma \vdash () : \mathbf{unit}}$$

[T-Int]
$$\frac{}{\Gamma \vdash n : \mathbf{int}}$$

[T-Bool]
$$\frac{}{\Gamma \vdash b : \mathbf{bool}}$$

[T-Var]
$$\frac{}{\Gamma, x : A \vdash x : A}$$

[T-Tup]
$$\frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : B}{\Gamma \vdash (e_1,\ e_2) : A \times B}$$

[T-Rec]
$$\frac{\Gamma, f : A \to B, x : A \vdash e : B}{\Gamma \vdash \mathbf{fun}\ f(x : A) : B = e : A \to B}$$

[T-Chan]
$$\frac{}{\Gamma \vdash c : A\,\mathbf{chan}} \quad \begin{array}{l} ty(c) = A\,\mathbf{chan}, \\ A \in BaseType \end{array}$$

Table 14: Types for values

Values in TCML are typed either by primitive types **unit**, **int** and **bool**, or by composite types $A \longrightarrow A'$, $A \times A'$ and $A\,\mathbf{chan}$. The former kind applies to the unit value, natural numbers and boolean values. The latter kind applies to functions, tuples and channels carrying values of type A. The corresponding typing rules are defined in Table 14. The only rule we discuss in depth is rule [T-Chan] in Table 14.

According to this rule, channels are well-typed if and only if two side conditions are met. The first condition is that $ty(c) = A\,\mathbf{chan}$. The pre-defined function $ty \in Chan \longrightarrow Types$ is defined as a total function that maps channel names to a channel type $A\,\mathbf{chan}$. The second condition imposes that type of channel c must be in the set of types $BaseType$, which can be defined as the least set of types such that:

i. $\{\mathbf{unit}, \mathbf{int}, \mathbf{bool}\} \in BaseType$

ii. $a, b \in BaseType \Rightarrow (a, b) \in BaseType$

iii. $a, b \in BaseType \Rightarrow a \to b \in BaseType$

Informally, base types are those types that do not comprise channel types. This restriction prevents scope extrusion, the phenomenon in which a a process P communicates a private channel c to a process Q without that channel's restriction scope. For example, consider the following case:

$$Q[\,\mathbf{recv}\ a] \parallel \nu c P[\,\mathbf{send}\ a\ c].$$

If scope extrusion is allowed, then channel c can be transmitted from process P to process Q over channel a. In this case, channel c is no longer private in P, but can be used by Q as well. In this report we will not consider scope extrusion in TCML for sake of simplicity.

$$\text{[T-IF]}\quad \frac{\Gamma \vdash e_1 : \mathbf{bool} \quad \Gamma \vdash e_2 : A \quad \Gamma \vdash e_3 : A}{\Gamma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : A} \qquad \text{[T-APP]}\quad \frac{\Gamma \vdash e_1 : A \to B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : B}$$

$$\text{[T-LET]}\quad \frac{\Gamma \vdash e_1 : A \quad \Gamma,\ x : A \vdash e_2 : B}{\Gamma \vdash \mathbf{let}\ x\ =\ e_1\ \mathbf{in}\ e_2 : B} \qquad \text{[T-OP]}\quad \frac{\Gamma \vdash e : A}{\Gamma \vdash op\ e : B}\ ty_o(op) = A \to B$$

$$\text{[T-SEND]}\quad \frac{\Gamma \vdash e_1 : A\,\mathbf{chan} \quad \Gamma \vdash e_2 : A}{\Gamma \vdash \mathbf{send}\ e_1\ e_2 : \mathbf{Unit}} \qquad \text{[T-RECV]}\quad \frac{\Gamma \vdash e : A\,\mathbf{chan}}{\Gamma \vdash \mathbf{recv}\ e : A}$$

$$\text{[T-NEWCHAN]}\quad \frac{}{\Gamma \vdash \mathbf{newChan}_{A} : A\,\mathbf{chan}} \qquad \text{[T-SPAWN]}\quad \frac{\Gamma \vdash e : \mathbf{unit} \to \mathbf{unit}}{\Gamma \vdash \mathbf{spawn}\ e : \mathbf{unit}}$$

$$\text{[T-ATOMIC]}\quad \frac{\Gamma, k \vdash e_1 : A \quad \Gamma \vdash e_2 : A}{\Gamma \vdash \mathbf{atomic}\ [\![\, e_1 \rhd_k e_2\,]\!] : A} \qquad \text{[T-COMMIT]}\quad \frac{}{\Gamma, k \vdash \mathbf{commit}\ k : \mathbf{unit}}$$

Table 15: Types for expressions

Typing rules for expressions are provided in Table 15. The rules for the functional part of TCML, such as **if** and **let** expressions, are the standard ones from typed lambda calculi. For example, the expression **if** 1 **then** e_1 **else** e_2 is not allowed by the type system, since there is no reduction rule to evaluate this expression. Rules [E-If-True] and [E-If-False] in fact only reduce an if expression if the conditional expression in it is either the boolean value **true** or **false**. Both branches of the **if** expression must have the same type, in order to avoid expressions that cannot be reduced.

Typing rules [T-Send] and [T-Recv] guarantee that expressions **send** and **recv** will only operate on channels of the appropriate type. For example, consider the following process:

$$\textbf{send}\ c\ 1 \parallel \textbf{if}\,(\textbf{recv}\ c)\ \textbf{then}\ e_1\ \textbf{else}\ e_2$$

After reducing the process with rule [C-Sync], we have:

$$() \parallel \textbf{if}\ 1\ \textbf{then}\ e_1\ \textbf{else}\ e_2$$

which is not allowed by the type system, for the reasons aforementioned. Thus the typing rules [T-Send], [T-Recv] and [T-Chan] enforce that only values with an expected type are passed from one process to another.

Expression $\textbf{newChan}_A$ is typed by the explicit type A annotation in rule [T-NewChan].

As mentioned in section 2.1, new processes are created by the **spawn** expression. According to rule [T-Spawn], the argument of **spawn** must be a function of the form **unit** $\longrightarrow$ **unit**. The rationale of this rule is to first reduce **spawn**'s argument into a function, and then resume its reduction later in a newly created process by applying a unit value to it. For example, consider the following process:

$$\textbf{spawn}\,(\textbf{fun}\ f(x) = \textbf{send}\ c\ 1; f())$$

which spawns a process that repeatedly sends number 1 over channel c. According to rule [C-Spawn], it will reduce to the following expression:

$$(\textbf{fun}\ f(x) = \textbf{send}\ c\ 1; f())\ () \parallel ()$$

As you can notice, the spawned process is an application of the function $f(x)$ to the unit value (), which explain why typing rule [T-Spawn] requires the argument to the **spawn** expression to be a function that only accepts **unit** values as input. The reason why its return type is also **unit** is explained later, when discussing rules [T-Par-L] and [T-Par-R], but intuitevely TCML processes cannot use the value into which other processes reduce, so the return type of the argument is chosen to be **unit**.

In rule [T-Atomic], the default and alternative expressions of a transaction are required to have the same type. This ensures that, whether a transaction is committed or aborted, the expression containing **atomic** $[\![\, e_1 \rhd_k e_2 \,]\!]$ will be able to continue reduction. This is the same rational behind rules such as [T-If]. Consider for example the following process:

$$\textbf{if atomic}\ [\![\, \textbf{true} \rhd_k 1 \,]\!]\ \textbf{then}\ e_1\ \textbf{else}\ e_2$$

If transaction k is activated by rule [Tr-Atomic], the process will reduce to the following:

$$\textbf{if}\ [\![\,\textbf{true}\ \rhd_k 1\,]\!]\ \textbf{then}\ e_1\ \textbf{else}\ e_2$$

If transaction k is aborted by rule [Tr-Abort], the resulting expression **if** 1 **then** e_1 **else** e_2 is not well-typed. This situation is remedied by forcing both the default and alternative expression to have the same type.

Note that in the premises of [T-Atomic] the default expression's typing context contains transaction name k, whereas the typing context of the alternative expression does not contain k, since k is only bound in the default of the transaction. On the one hand, this allows the default expression to define commit points to transaction k. On the other hand, it disallows the alternative expression to have any reference to the same transaction k. When a transaction is aborted, the default process is replaced by the alternative and the transaction ceases to exist.

Rule [T-Commit] from Table 15 ensures that an expression **commit** k is well-typed if its typing context contains the transaction name k. For example, the following process:

$$\textbf{atomic}\ [\![\,\textbf{commit}\ j \rhd_k ()\,]\!]$$

is not well-typed, because the **commit** j expression refers to a transaction name j, which is not bound to any transaction. The following process is well-typed:

$$\textbf{atomic}\ [\![\,\textbf{commit}\ k \rhd_k ()\,]\!]$$

[T-Par-L]
$$\frac{\Gamma \vdash P_1 : A \quad \Gamma \vdash P_2 : \textbf{unit}}{\Gamma \vdash P_1 \parallel P_2 : A}$$

[T-Par-R]
$$\frac{\Gamma \vdash P_1 : \textbf{unit} \quad \Gamma \vdash P_2 : A}{\Gamma \vdash P_1 \parallel P_2 : A}$$

[T-Restr]
$$\frac{\Gamma \vdash P : B}{\Gamma \vdash \nu c.P : B}$$

[T-Trans]
$$\frac{\Gamma, k \vdash P_1 : A \quad \Gamma \vdash P_2 : A}{\Gamma \vdash [\![\, P_1 \rhd_k P_2 \,]\!] : A}$$

[T-Co]
$$\frac{}{\Gamma, k \vdash\ \textbf{co}\ k : \textbf{unit}}$$

Table 16: Types for processes

Rules [T-Par-L] and [T-Par-R] reflect the fact that spawned TCML processes have type **unit**, as discussed for rule [T-Spawn]. The type A of the main program is preserved across all the spawned processes of type **unit**, so that a TCML program maintains a type A across all parallel programs. For example, the following process is not well-typed:

$$1 \parallel 2$$

but the following process is:

$$() \parallel 2$$

Thus at any time, there is only one process of type A, whereas the other spawned processes all have type **unit**.

Rule [T-Trans] enforces the default and alternative processes of a transaction to have the same type, mirroring rule [T-Atomic]. This condition guarantees that a process maintains the same type in case an abort step is performed. Consider the following process, very similar to the one we have already discussed for rule [T-Atomic]:

$$\textbf{if } [\![\, \textbf{true} \ \rhd_k 1]\!] \textbf{ then } e_1 \textbf{ else } e_2$$

If transaction k is aborted by rule [Tr-Abort], the resulting expression **if** 1 **then** e_1 **else** e_2 is not well-typed. This situation is remedied by forcing both the default and alternative expression to have the same type. Notice also that in rule [T-Atomic] the alternative processes cannot be typed assuming the transaction name k in the typing context. This condition ensures that, when transaction k is aborted, the alternative process contains no reference to transaction k anymore.

Rule [T-Co] types commit point processes **co** k, where transaction name k must be present in the typing context. This ensures that commit points can only be contained by transactions bearing the same transaction name. Commit point processes have type **unit**, just like other spawned processes.

$ty_o(\textbf{fst}) = A \times B \to A$	$ty_o(\textbf{snd}) = A \times B \to B$
$ty_o(\textbf{leq}) = \textbf{int} \times \textbf{int} \to \textbf{bool}$	$ty_o(\textbf{add}) = \textbf{int} \times \textbf{int} \to \textbf{int}$
$ty_o(\textbf{mul}) = \textbf{int} \times \textbf{int} \to \textbf{int}$	$ty_o(\textbf{sub}) = \textbf{int} \times \textbf{int} \to \textbf{int}$

Table 17: Types for operations

Finally, predicate ty_o specifies a type for the usual value operations such as addition and multiplication. Table 17 provides its definition. For any type A and B, pair operations **fst** and **snd** are assigned a matching arrow type, from the pair type $A \times B$ to A or B respectively. Arithmetic and boolean operations are typed as expected, namely they are functions from pairs of integer types to either an integer or a boolean type.

2.3.1 Type soundness

In this section we will prove a soundness theorem for TransCML semantics, intuitively that the evaluation of well-typed TCML processes never "goes wrong", in the spirit of Milner's motto [15]. In order to prove this theorem, we need to prove a type preservation theorem (2.9) and a progress theorem (2.11). We will need a number of lemmas and a Type Uniqueness Theorem (2.3) for the former, and a notion of "stuck" expressions for the second, which will be introduced further on.

We start by proving some useful lemmas on typing contexts and the typing judgement. The first lemma is Strengthening, which let's us drop bindings to the variables in typing context whenever these variables do not occur free in a process:

Lemma 2.1 (Strenghtening)
If $\Gamma, x : A \vdash P : B$ *and* $x \notin FV(P)$*, then* $\Gamma \vdash P : B$

Proof
We need to prove that, $\forall \Delta, P, B.\ \Delta \vdash P : B \Rightarrow \mathcal{P}(\Delta, B, P)$, where $\mathcal{P}(\Delta, B, P) = (\Delta = \Gamma, x : A) \wedge (x \notin FV(P)) \Rightarrow \Gamma \vdash P : B$. Let us prove this by induction over the judgement $\Delta \vdash P : B$. Note that values and expressions are instances of processes, as defined in Table 1.

[T-Var] Let us assume that:

1) $\Delta', y : B \vdash y : B$, where $\Delta = \Delta', y : B$ for some Δ'
2) $\Delta = \Gamma, x : A$
3) $x \notin FV(y)$

We need to show that $\Gamma \vdash y : B$.

By the definition of predicate FV, hypothesis (3) implies that variable x and y are different. Because of this and the side condition on hypothesis (1), the typing context Γ must contain a binding $y : B$; recall the definition of $,_{\mathcal{B}}$. Thus, $\Gamma = \Lambda, y : B$ for some typing context Λ. From rule [T-Var], we have $\Lambda, y : B \vdash y : B$, which is equivalent to $\Gamma \vdash y : B$

[T-Unit] Let us assume that:

1) $\Delta \vdash () : \textbf{unit}$
2) $\Delta = \Gamma, x : A$
3) $x \notin FV(())$

We need to show that $\Gamma \vdash () : \textbf{unit}$.

This can be derived directly by application of rule [T-Unit]. The cases for rules [T-Bool], [T-Int], [T-Chan], [T-NewChan] and [T-Recv] are immediate too.

[T-Rec] Let us assume that:

1) $\Delta, f : B \longrightarrow C, y : B \vdash e : C$
2) $\mathcal{P}((\Delta, f : B \longrightarrow C, y : B), C, e)$
3) $\Delta \vdash \textbf{fun } f(y) = e : B \longrightarrow C$
4) $\Delta = \Gamma, x : A$
5) $x \notin FV(\textbf{fun } f(y) = e)$

We need to show that $\Gamma \vdash \textbf{fun } f(x) = e : B \longrightarrow C$.

Assuming Barendregt's convention, variable x is different from bound variables f and y. Thus from hypothesis (4) we can infer that the typing context $\Gamma, f : B \longrightarrow C, y : B, x : A$ is well defined according to the definition of $,_{\mathcal{B}}$, and that it is equal to because of the definition of $,_{\mathcal{B}}$. From hypothesis (5) we have that $x \notin FV(e)$. Applying the last two consideration on inductive hypothesis (2), we have that $\Delta, f : B \longrightarrow C, y : B \vdash e : B$. Applying rule [T-Rec], we obtain $\Gamma \vdash \textbf{fun } f(x) = e : B \longrightarrow C$.

Case [T-Let] can be proved similarly.

[T-App] Let us assume that:

1) $\Delta \vdash e_1 : C \longrightarrow B$
2) $\Delta \vdash e_2 : C$
3) $\mathcal{P}(\Delta, e_1, C \longrightarrow B)$
4) $\mathcal{P}(\Delta, e_2, C)$
5) $\Delta \vdash e_1\ e_2 : B$
6) $\Delta = \Gamma, x : A$
7) $x \notin FV(e_1\ e_2)$

We need to show that $\Gamma \vdash e_1\ e_2 : B$.

From hypothesis (7) we can derive that $x \notin FV(e_1)$ (*) and $x \notin FV(e_2)$ (**). We can apply hypotheses (6) and (*) on inductive hypothesis (3) to derive $\Gamma \vdash e_1 : C \longrightarrow B$; similarly, we can derive $\Gamma \vdash e_2 : C$ applying hypotheses (6) and (**) on inductive hypothesis (4). From these last two judgements, we can derive $\Gamma \vdash e_1\ e_2$ by rule [T-App].

Cases [T-Tup] , [T-Op], [T-If], [T-Spawn], [T-Send], [T-Restr], [T-Par-L] and [T-Par-R] can be proved similarly.

[T-Atomic] Let us assume that:

1) $\Delta, k \vdash e_1 : B$
2) $\Delta \vdash e_2 : B$
3) $\mathcal{P}((\Delta, k), e_1, C \longrightarrow B)$

4) $\mathcal{P}(\Delta, e_2, C)$

5) $\Delta \vdash$ **atomic** $[\![\, e_1 \rhd_k e_2 \,]\!] : B$

6) $\Delta = \Gamma, x : A$

7) $x \notin FV($**atomic** $[\![\, e_1 \rhd_k e_2 \,]\!])$

We need to show that $\Gamma \vdash$ **atomic** $[\![\, e_1 \rhd_k e_2 \,]\!] : B$.

By the definition of FV and hypothesis (7), we can derive that $x \notin FV(e_1)$ (*) and $x \notin FV(e_2)$ (**). By hypothesis (6) we have that $\Delta, k = \Gamma, x : A, k$, and then that $\Delta, k = \Gamma, k, x : A$ (***) by the the definition of $,_{\mathcal{B}}$ and $,_{\mathcal{K}}$. Applying hypothesis (***) and (*) on inductive hypothesis (3), we have that $\Gamma, k \vdash e_1 : B$. Applying hypotheses (6) and (**) on inductive hypothesis (4), we can derive $\Gamma \vdash e_2 : B$. From these last two judgements, we can derive $\Gamma \vdash$ **atomic** $[\![\, e_1 \rhd_k e_2 \,]\!]$ by rule [T-Atomic].

Case [T-Trans] can be proved similarly.

[T-Commit] Let us assume that:

1) $\Delta, k \vdash$ **commit** $k : B$

2) $\Delta, k = \Gamma, x : A$

3) $x \notin FV($**commit** $k)$

We need to show that $\Gamma \vdash$ **commit** $k : B$.

From hypothesis (2) we can infer that transaction name $k \in \Gamma$ by the definition of $,_{\mathcal{K}}$. Thus $\Gamma = \Lambda, k$ is well-defined, for some typing context Λ. From this consideration, we can apply rule [T-Co] to have $\Lambda, k \vdash$ **commit** k, and thus that $\Gamma \vdash$ **commit** $k : B$.

Case [T-Co] can be proved similarly.

∎

Next we prove the Weakening lemma, which allows us to add assumptions to a typing judgment, whenever this assumptions do not clash with existing variable binding in a typing context:

Lemma 2.2 (Weakening)
If $\Gamma \vdash P : B$ *and* $x \notin dom(\Gamma)$*, then* $\Gamma, x : A \vdash P : B$

Proof
We need to prove that, $\forall \Gamma, P, B.\ \Gamma \vdash P : B \Rightarrow \mathcal{P}(\Gamma, B, P)$, where $\mathcal{P}(\Gamma, B, P) = x \notin dom(\Gamma) \Rightarrow \Gamma, x : A \vdash P : B$, for a given variable x and a given type A. Let us prove this by induction over the judgement $\Gamma \vdash P : B$.

[T-Var] Let us assume that:

1) $\Gamma', y : B \vdash y : B$, where $\Gamma = \Gamma', y : B$ for some Γ'

2) $x \notin dom(\Gamma)$

We need to show that $\Gamma, x : A \vdash y : B$.

Because of hypothesis (2), we have that $\Gamma, x : A$ is well-defined. Thus, from hypothesis (1), $\Gamma, x : A = \Gamma', y : B, x : A$. By the definition of $,_{\mathcal{B}}$, we also have that $\Gamma', y : B, x : A = \Gamma', x : A, y : B$. By rule [T-Var], we have that $\Gamma', x : A, y : B \vdash y : B$. Since $\Gamma', x : A, y : B = \Gamma', y : B, x : A = \Gamma, y : B$, we have that $\Gamma, x : A \vdash y : B$.

[T-Unit] Let us assume that:

1) $\Gamma \vdash () : \textbf{unit}$
2) $x \notin dom(\Gamma)$

We need to show that $\Gamma, x : A \vdash () : \textbf{unit}$.

This follows immediately from rule [T-Unit].

Cases [T-Bool], [T-Int], [T-Chan], [T-NewChan] and [T-Recv] are immediate too.

[T-Rec] Let us assume that:

1) $\Gamma, f : C \longrightarrow B, y : C \vdash e : B$
2) $\mathcal{P}((\Gamma, f : C \longrightarrow B, y : C), e, B)$
3) $\Gamma \vdash \textbf{fun } f(y) = e : B$
4) $x \notin dom(\Gamma)$

We need to show that $\Gamma, x : A \vdash \textbf{fun } f(y) = e : B$.

By the Barendregt convention, we can assume that variable x is different variables f and y. From this consideration and hypothesis (4), we have that $x \notin dom(\Gamma, f : C \longrightarrow B, y : B)$. We can apply this to inductive hypothesis (2) to derive that $\Gamma, f : C \longrightarrow B, y : C, x : A \vdash e : B$. By the definition of $,_{\mathcal{B}}$, $\Gamma, f : C \longrightarrow B, y : C, x : A = \Gamma, x : A, f : C \longrightarrow B, y : C$. Thus our last derivation is equal to $\Gamma, x : A, f : C \longrightarrow B, y : C \vdash e : B$, on which we can apply rule [T-Rec] to get $\Gamma, x : A \vdash \textbf{fun } f(y) = e : B$.

Case [T-Let] can be proved similarly.

[T-App] Let us assume that, for some type C:

1) $\Gamma \vdash e_1 : C \longrightarrow B$
2) $\Gamma \vdash e_2 : C$
3) $\mathcal{P}(\Gamma, e_1, C \longrightarrow B)$
4) $\mathcal{P}(\Gamma, e_2, C)$
5) $\Gamma \vdash e_1 \; e_2 : B$
6) $x \notin dom(\Gamma)$

We need to show that $\Gamma, x : A \vdash e_1\ e_2 : B$.

We can apply directly the inductive hypothesis (6) on both inductive hypotheses (3) and (4) to derive $\Gamma, x : A \vdash e_1 : C \longrightarrow B$ and $\Gamma, x : A \vdash e_2 : C$. Using rule [T-App] on them, we can derive $\Gamma, x : A \vdash e_1\ e_2 : B$.

Cases [T-Tup], [T-Op], [T-If], [T-Spawn], [T-Send], [T-Restr], [T-Par-L] and [T-Par-R] can be proved similarly.

[T-Atomic] Let us assume that:

1) $\Gamma, k \vdash e_1 : B$
2) $\Gamma \vdash e_2 : B$
3) $\mathcal{P}((\Gamma, k), e_1, B)$
4) $\mathcal{P}(\Gamma, e_2, B)$
5) $\Gamma \vdash \textbf{atomic}\ [\![\, e_1 \rhd_k e_2 \,]\!] : B$
6) $x \notin \mathit{dom}(\Gamma)$

We need to show that $\Gamma, x : A \vdash \textbf{atomic}\ [\![\, e_1 \rhd_k e_2 \,]\!] : B$.

We can apply hypothesis (6) on inductive hypothesis (4) to have $\Gamma, x : A \vdash e_2 : B$ (*). Since a transaction name is not a variable, variable x cannot be equal to transaction name k, thus $x \notin \mathit{dom}(\Gamma, k)$. We can apply this to inductive hypothesis (3) to derive $\Gamma, k, x : A \vdash e_1 : B$. By the definition of $,_{\mathcal{B}}$ and $,_{\mathcal{K}}$, $\Gamma, k, x : A = \Gamma, x : A, k$. Hence, $\Gamma, x : A, k \vdash e_1 : B$. From this and (*), we can derive $\Gamma, x : A \vdash \textbf{atomic}\ [\![\, e_1 \rhd_k e_2 \,]\!]$ by rule [T-Atomic].

Case [T-Trans] can be proved similarly.

[T-Commit] Let us assume that:

1) $\Gamma', k \vdash \textbf{commit}\ k : B$, where $\Gamma = \Gamma', k$ for some Γ'
2) $x \notin \mathit{dom}(\Gamma)$

We need to show that $\Gamma, x : A \vdash \textbf{commit}\ k : B$.

Because of hypothesis (2), we have that the typing context $\Gamma, x : A$ is well-defined. Moreover, we can derive by hypothesis (1) that $\Gamma, x : A = \Gamma', k, x : A$, and, by the definition of $,_{\mathcal{B}}$ and $,_{\mathcal{K}}$, we have that $\Gamma, x : A = \Gamma', x : A, k$. We can now apply rule [T-Commit] to derive $\Gamma', x : A, k \vdash \textbf{commit}\ k$, that is, $\Gamma, x : A \vdash \textbf{commit}\ k$.

Case [T-Co] can be proved similarly.

∎

Another useful theorem is the Type Uniqueness theorem, which states that well-typed processes can only be assigned a single type:

Theorem 2.3 (Type Uniqueness)
If $\Gamma \vdash P : A$ then, for all types B, $\Gamma \vdash P : B$ implies $A = B$.

Proof
We need to prove that $\forall \Gamma, P, A,\ \Gamma \vdash P : A \Rightarrow \mathcal{P}(\Gamma, P, A)$, where $\mathcal{P}(\Gamma, P, A) = \forall B.\, \Gamma \vdash P : B \Rightarrow A = B$. Let us prove this theorem by induction on derivations on judgement $\Gamma \vdash P : A$.

[T-UNIT] Let us assume that:

1) $\Gamma \vdash () : \textbf{unit}$

We need to show that $\forall B.\, \Gamma \vdash P : B \Rightarrow \textbf{unit} = B$

Let us assume a type B such that $\Gamma \vdash () : B$. The only rule that can derive this last judgement is rule [T-UNIT], from which we can derive that $B = \textbf{unit}$. We have $\Gamma \vdash () : B \Rightarrow \textbf{unit} = B$ by deduction, and $\forall B.\, \Gamma \vdash () : B \Rightarrow \textbf{unit} = B$ by generalisation.

The cases for rules [T-INT] and [T-BOOL] are very similar.

[T-CHAN] Let us assume that:

1) $\Gamma \vdash c : A\,\textbf{chan}$
2) $ty(c) = A\,\textbf{chan}$
3) $A \in BaseType$

We need to show that $\forall B.\, \Gamma \vdash c : B \Rightarrow A\,\textbf{chan} = B$.

Let us assume a type B such that $\Gamma \vdash c : B$. The only rule that can derive this last judgment is rule [T-CHAN], from whose side conditions we can derive that $ty(c) = B$. Since ty is a total function, it is also injective, thus $ty(c) = A\,\textbf{chan} = B$. Using deduction and generalizing over B on this last equation, we have that $\forall B.\, \Gamma \vdash c : B \Rightarrow A\,\textbf{chan} = B$.

The case for rule [T-NEWCHAN] is similar, with the exception that we can infer $A = B$ from the type annotation embedded into the **newChan** syntax. The cases for rules [T-COMMIT] and [T-CO] can be proved similarly.

[T-VAR] Let us assume that:

1) $\Gamma, x : A \vdash x : A$

We need to show that $\forall B. \Gamma, x : B \vdash x : B \Rightarrow A = B$.

Let us assume a type B such that $\Gamma, x : A \vdash x : B$. This last judgement can only be derived by rule [T-VAR], according to which the variable in the typing context and in the expression have the same type. Thus we have that $A = B$ and that $\Gamma, x : B \vdash x : B$. Using deduction and generalization, we have that $\forall B. \Gamma, x : B \vdash x : B \Rightarrow A = B$.

[T-TUP] Let us assume that:

1) $\Gamma \vdash (e_1, e_2) : A_1 \times A_2$
2) $\Gamma \vdash e_1 : A_1$

3) $\Gamma \vdash e_2 : A_2$

4) $\mathcal{P}(\Gamma, e_1, A_1)$

5) $\mathcal{P}(\Gamma, e_2, A_2)$

We need to show that $\forall B.\, \Gamma \vdash (e_1, e_2) : B \Rightarrow A_1 \times A_2 = B$.

Let us assume a type B such that $\Gamma \vdash (e_1, e_2) : B$. The only rule that can derive this last judgement is rule [T-Tup], from which we have that $B = B_1 \times B_2$ for some B_1 and B_2, and that $\Gamma \vdash e_1 : B_1$ (*) and $\Gamma \vdash e_2 : B_2$ (**). We can use hypothesis 2) and inductive hypothesis 4) on (*) and hypothesis 3) and inductive hypothesis 5) on (**) to have that $A_1 = B_1$ and that $A_2 = B_2$. Since $B = B_1 \times B_2$, we can now derive $B = A_1 \times A_2$. Using deduction and generalization on B, we have that $\forall B.\, \Gamma \vdash (e_1, e_2) : B \Rightarrow A_1 \times A_2 = B$.

The cases for rules [T-If], [T-Send], [T-Recv], [T-Spawn] and [T-Restr] can be proved similarly. The cases for rules [T-Par-L] and [T-Par-R] are similar too, since we must use one of the hypothesis according to which one of the two parallel processes has type **unit**, to allow inversion.

[T-Rec] Let us assume that:

1) $\Gamma \vdash (\textbf{fun } f(x : A_1) : A_2 = e) : A_1 \to A_2$

2) $\Gamma, f : A_1 \to A_2, x : A_1 \vdash e : A_2$

3) $\mathcal{P}((\Gamma, f : A_1 \to A_2, x : A_1), e, A_2)$

We need to show that $\forall B.\, \Gamma \vdash (\textbf{fun } f(x : A_1) : A_2 = e) : B \Rightarrow A_1 \to A_2 = B$.

Let us assume a type B such that $\textbf{fun } f(x : A_1) : A_2 = e : B$. The only rule that can derive this judgement is rule [T-Rec], thus we have that $B = A_1 \to A_2$. Using deduction and generalization on B, we have that $\forall B.\, \Gamma \vdash (\textbf{fun } f(x : A_1) : A_2 = e) : B \Rightarrow A_1 \to A_2 = B$.

[T-Let] Let us assume that:

1) $\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : A$

2) $\Gamma \vdash e_1 : A'$

3) $\Gamma, x : A' \vdash e_2 : A$

4) $\mathcal{P}(\Gamma, e_1, A')$

5) $\mathcal{P}((\Gamma, x : A'), e_2, A)$

We need to show that $\forall B.\, \Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : B \Rightarrow A = B$.

Let us assume a type B such that $\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : B$. The only rule that can derive this last judgement is rule [T-Let], from which we can derive $\Gamma \vdash e_1 : B'$ (*) and $\Gamma, x : B' \vdash B$ (**). From inductive hypothesis 4) and (*), we can derive that $A' = B'$. Because of this, we can consider (**)

as $\Gamma, x : A' \vdash B$ (***). Instantiating inductive hypothesis 5) with type B, we can derive $A = B$ from (***). Using deduction and generalization on B, we have that $\forall B. \Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : B \Rightarrow A = B$.

[T-APP] Let us assume that:

1) $\Gamma \vdash e_1\ e_2 : A$
2) $\Gamma \vdash e_1 : A'$
3) $\Gamma \vdash e_2 : A' \rightarrow A$
4) $\mathcal{P}(\Gamma, e_1, A')$
5) $\mathcal{P}(\Gamma, e_2, A' \rightarrow A)$

We need to prove that $\forall B. \Gamma \vdash e_1\ e_2 : B \Rightarrow A = B$.

Let us assume a type B such that $\Gamma \vdash e_1\ e_2 : B$. The only typing rule that can derive this judgement is [T-APP], from which we can deduce that $\Gamma \vdash e_1 : B'$ (*) and $\Gamma \vdash e_2 : B' \rightarrow B$ (**) for some type B'. We can apply inductive hypothesis 4) on hypothesis 2) and (*), to derive that $B' = A'$. Similarly we can apply inductive hypothesis 5) on hypothesis 3) and (**) to obtain that $A' \rightarrow A = B' \rightarrow B$. Because of this and $A' = B'$, it follows that $A = B$. Using deduction on our initial assumption and generalization on B, we have that $\forall B. \Gamma \vdash e_1\ e_2 : B \Rightarrow A = B$.

The case for rule [T-OP] can be proved similarly.

[T-ATOMIC] Let us assume that:

1) $\Gamma \vdash \textbf{atomic } [\![\, e_1 \rhd_k e_2 \,]\!] : A$
2) $\Gamma, k \vdash e_1 : A$
3) $\Gamma \vdash e_2 : A$
4) $\mathcal{P}((\Gamma, k), e_1, A)$
5) $\mathcal{P}(\Gamma, e_2, A)$

We need to prove that $\forall B. \Gamma \vdash \textbf{atomic } [\![\, e_1 \rhd_k e_2 \,]\!] : B \Rightarrow A = B$.

Let us assume a type B such that $\Gamma \vdash \textbf{atomic } [\![\, e_1 \rhd_k e_2 \,]\!] : B$. The only rule that can derive this judgement is [T-ATOMIC], from which we can derive $\Gamma \vdash e_2 : B$. We can apply inductive hypothesis 5) on hypothesis 3) and the last derived judgment, to have that $A = B$. Using deduction on our initial assumption and generalization on B, we have that $\forall B. \Gamma \vdash \textbf{atomic } [\![\, e_1 \rhd_k e_2 \,]\!] : B \Rightarrow A = B$.

The case for rule [T-TRANS] is similar.

∎

We will now try to prove the type preservation theorem. In order to do that, first of all we need to prove the Substitution lemma, also known as subject reduction. This lemma proves that substitution of free variables for values of the same type preserves typing of the overall process:

Lemma 2.4 (Substitution)
If $\Gamma, x : A \vdash P : B$ *and* $\Gamma \vdash v : A$*, then* $\Gamma \vdash P[v/x] : B$

Proof
We need to prove that $\forall \Delta, P, B.\ \Delta \vdash P : B \Rightarrow \mathcal{P}(\Delta, B, P)$, where $\mathcal{P}(\Delta, B, P) = (\Delta = \Gamma, x : A) \wedge (\Gamma \vdash v : A) \Rightarrow \Gamma \vdash P[v/x] : B$, for any given variable x, value v and type A. Let us prove this by induction over judgement $\Delta \vdash P : B$.

[T-Var] Let us assume that:

1) $\Delta \vdash y : B$
2) $\Delta = \Gamma, x : A$
3) $\Gamma \vdash v : A$

We need to show that $\Gamma \vdash y[v/x] : B$.

Only two cases are possible: either $x = y$ or $x \neq y$.

$[x = y]$ Since $x = y$, we can infer that $\Delta = \Gamma, y : A$ from hypothesis 2), and then $\Delta \vdash y : A$ from rule [T-Var] (*). Notice that variable y cannot be already defined in Γ, because the operation $,_{\mathcal{B}}$ is defined on a typing context Γ and a variable x only if $x \notin dom(\Gamma)$. Applying hypothesis (1) and (*) to Theorem 2.3, we can infer that $A = B$. Hypothesis (3) is thus $\Gamma \vdash v : B$. By the definition of substitution, we have that $v = x[v/x]$, and, since $x = y$, that $v = y[v/x]$. From this last fact and hypothesis (3), we have that $\Gamma \vdash y[v/x] : B$.

$[x \neq y]$ In the latter case, if $x \neq y$ then $y[v/x] = y$ by the definition of the substitution function in Table 5. From the definition of the free variable predicate FV in Table 2, we can also derive that $x \notin FV(y)$. We can apply these considerations and hypothesis (3) to Lemma 2.1 (Strengthening) to derive $\Gamma \vdash y[v/x] : B$

[T-Unit] Let us assume that:

1) $\Delta \vdash () : \textbf{unit}$
2) $\Delta = \Gamma, x : A$
3) $\Gamma \vdash v : A$

We need to show that $\Gamma \vdash ()[v/x] : \textbf{unit}$.

Rule [T-Unit] states that $\Gamma \vdash () : \textbf{unit}$. From the definition of substitution, we have directly that $\Gamma \vdash ()[v/x] : \textbf{unit}$, since $()[v/x] = ()$ for any x and v.

Cases [T-Bool], [T-Int], [T-Chan], [T-NewChan] and [T-Recv] can be proved similarly.

[T-Rec] Let us assume that, for some type C:

1) $\Delta, f : C \longrightarrow B, y : C \vdash e : B$

2) $\mathcal{P}((\Delta, f : C \longrightarrow B, y : C), B, e)$

3) $\Delta \vdash \textbf{fun } f(y) = e : C \longrightarrow B$

4) $\Delta = \Gamma, x : A$

5) $\Gamma \vdash v : A$

We need to show that $\Gamma \vdash (\textbf{fun } f(y) = e)[v/x] : C \longrightarrow B$.

From hypotheses 1) and 4), we have that the typing context $\Gamma, x : A, f : C \longrightarrow B, y : C$ is well-defined. Because of the definition of operator $,_{\mathcal{B}}$, we can write this typing context as $\Gamma, f : C \longrightarrow B, y : C, x : A$. We can apply this last fact and hypothesis (1) to inductive hypothesis (2) to derive $\Gamma, f : C \longrightarrow B, y : C \vdash e[v/x] : B$. We can apply rule [T-Rec] to derive $\Gamma \vdash \textbf{fun } f(y) = e[v/x] : B$. Finally, by the definition of substitution and by the Barendregt's convention, we can derive $\Gamma \vdash (\textbf{fun } f(y) = e)[v/x] : B$.

Case [T-Let] can be proved similarly.

[T-App] Let us assume that, for some type C:

1) $\Delta \vdash e_1 : C \longrightarrow B$

2) $\Delta \vdash e_2 : C$

3) $\mathcal{P}(\Delta, C \longrightarrow B, e)$

4) $\mathcal{P}(\Delta, C, e)$

5) $\Gamma, x : A \vdash e_1\ e_2 : B$

6) $\Delta = \Gamma, x : A$

7) $\Gamma \vdash v : A$

We need to show that $\Gamma \vdash (e_1\ e_2)[v/x] : B$.

We can apply hypotheses (6) and (7) to inductive hypotheses (3) and (4) to derive that $\Gamma \vdash e_1[v/x] : C \longrightarrow B$ and $\Gamma \vdash e_2[v/x] : C$. By rule [T-App], we have that $\Gamma \vdash e_1[v/x]\ e_2[v/x] : B$ and, by the definition of substitution, that $\Gamma \vdash (e_1\ e_2)[v/x] : B$

[T-Atomic] Let us assume that:

1) $\Delta, k \vdash e_1 : B$

2) $\Delta \vdash e_2 : B$

3) $\mathcal{P}((\Delta, k), C \longrightarrow B, e)$

4) $\mathcal{P}(\Delta, C, e)$

5) $\Gamma, x : A \vdash \textbf{atomic } [\![\, e_1 \rhd_k e_2 \,]\!] : B$

6) $\Delta = \Gamma, x : A$

7) $\Gamma \vdash v : A$

We need to show that $\Gamma \vdash \textbf{atomic} \llbracket e_1 \rhd_k e_2 \rrbracket[v/x] : B$.

By the definition of $,_{\mathcal{B}}$ and $,_{\mathcal{K}}$, and by hypothesis (6), we have that $\Delta, k = \Gamma, k, x : A$. We can apply this consideration and hypothesis (7) to inductive hypothesis (3) to derive that $\Gamma, k \vdash e_1[v/x] : B$. We can also apply hypotheses (6) and (7) to inductive hypothesis (4) to derive $\Gamma \vdash e_2[v/x] : B$. By rule [T-Atomic], we have that $\Gamma \vdash \textbf{atomic} \llbracket e_1[v/x] \rhd_k e_2[v/x] \rrbracket : B$; by the definition of substitution, we have $\Gamma \vdash \textbf{atomic} \llbracket e_1 \rhd_k e_2 \rrbracket[v/x] : B$.

Case [T-Trans] can be proved similarly.

[T-Commit] Let us assume that:

1) $\Delta \vdash \textbf{commit}\ k : \textbf{unit}$, where $\Delta = \Gamma', k$ for some Γ'
2) $\Delta = \Gamma, x : A$
3) $\Gamma \vdash v : A$

We need to show that $\Gamma \vdash \textbf{commit}\ k[v/x] : \textbf{unit}$.

Variables and transaction names belong to different domains, thus $x \neq k$; from this and the side conditions on hypotheses (1) and (2), we can derive that $\Gamma = \Lambda, k$ for some typing context Λ. Thus, we can derive $\Lambda, k \vdash \textbf{commit}\ k : \textbf{unit}$ by rule [T-Commit]. By the definition of substitution, we have that $\textbf{commit}\ k[v/x] = \textbf{commit}\ k$; since $\Gamma = \Lambda, k$, we have that $\Gamma \vdash \textbf{commit}\ k[v/x] : B$.

Case [T-Co] can be proved similarly. ∎

The following lemma, Context Abstraction, allows us to reason about and decompose the evaluation context of an expression. If an well-typed expression is divided into an evaluation context and a sub-expression, then both the evaluation context with a fresh variable in its hole, and the sub-expression are well-typed:

Lemma 2.5 (Context Abstraction)
If $\Gamma \vdash E[e] : B$ *and* $x \notin dom(\Gamma)$*, then* $\exists A. \Gamma \vdash e : A$ *and* $\Gamma, x : A \vdash E[x] : B$

Proof We need to prove that $\forall \Gamma. \forall E.\ \mathcal{P}(\Gamma, E)$, where $\mathcal{P}(\Gamma, E) = \forall B. \forall e. \Gamma \vdash E[e] : B \wedge x \notin dom(\Gamma) \Rightarrow \exists A. \Gamma \vdash e : A \wedge \Gamma, x : A \vdash E[x] : B$, for any given expression e.

Let us prove this lemma by structural induction over the evaluation context E.

[e]: Let us assume that:

1) $\Gamma \vdash [e] : B$
2) $x \notin dom(\Gamma)$

We need to show that: $\exists A. \Gamma \vdash e : A \wedge \Gamma, x : A \vdash [x] : B$.

The empty context applied to an expression is equal to the expression itself. Thus, $[e] = e$ (*) and $[x] = x$ (**). From (*) and hypothesis (1), we have that $\Gamma \vdash e : B$. By axiom [T-Var] and (**), we also have that $\Gamma, x : B \vdash [x] : B$. From the last two judgements, we have $\Gamma \vdash e : B \wedge \Gamma, x : B \vdash [x] : B$. Thus B is a witness to $\exists A. \Gamma \vdash e : A \wedge \Gamma, x : A \vdash [x] : B$

op E[e]: We will prove the case for the **add** operation only; the remaining cases can be proved similarly.

Let us assume that:

1) $\mathcal{P}(\Gamma, E)$
2) $\Gamma \vdash \textbf{add } E[e] : B$
3) $x \notin dom(\Gamma)$

We need to show that: $\exists A. \Gamma \vdash e : A \wedge \Gamma, x : A \vdash \textbf{add } E[x] : B$.

The only rule that can derive hypothesis (2) is [T-Op], from the premises of which we can derive that $B = \textbf{int}$ and $\Gamma \vdash E[e] : \textbf{int} \times \textbf{int}$. Let us instantiate inductive hypothesis (1) with type **int** and expression e and apply it on this last fact and hypothesis (3) to have that $\exists A. \Gamma \vdash e : A \wedge \Gamma, x : A \vdash E[x] : B$. Let us assume that T is such a type. Thus we have that $\Gamma \vdash e : T$ (*) and $\Gamma, x : T \vdash E[x] : \textbf{int} \times \textbf{int}$ (**). From the latter judgement (**) and rule [T-Op], we have $\Gamma, x : T \vdash \textbf{add } E[x] : \textbf{int} \times \textbf{int}$. Taking this last statement together with (*), we have $\Gamma \vdash e :$ and $\Gamma, x : T \vdash \textbf{add } E[x] : \textbf{int} \times \textbf{int}$. Thus, considering that $B = \textbf{int} \times \textbf{int}$, type T is a witness to $\exists A. \Gamma \vdash e : A \wedge \Gamma, x : A \vdash \textbf{add } E[x] : B$

Case **spawn** $E[e]$ and **recv** $E[e]$ can be proved similarly.

$(E[e_1], e_2)$: Let us assume that:

1) $\mathcal{P}(\Gamma, E)$
2) $\Gamma \vdash (E[e_1], e_2) : B$
3) $x \notin dom(\Gamma)$

We need to show that: $\exists A. \Gamma \vdash e_1 : A \wedge \Gamma, x : A \vdash (E[x], e_2) : B$.

The only rule that can derive hypothesis (2) is [T-Tup], from the premises of which we can derive that $\Gamma \vdash E[e_1] : C_1$ (*), $\Gamma \vdash e_2 : C_2$ (**) and $B = C_1 \times C_2$, for some type C_1 and C_2. Let us instantiate hypothesis (1) with type C_1 and expression e_1, and apply (*) and hypothesis (3) to it to have that $\exists A. \Gamma \vdash e_1 : A \wedge \Gamma, x : A \vdash E[x] : C_1$. Let us assume that T is such a type. Then we have $\Gamma \vdash e_1 : T$ (***) and $\Gamma, x : T \vdash E[x] : C_1$. By (**), hypothesis (3) and lemma 2.2, we have that $\Gamma, x : T \vdash e_2 : C_2$. Applying rule [T-VTup], we have $\Gamma, x : T \vdash (E[x], e_2) : C_1 \times C_2$, that is, $\Gamma, x : T \vdash (E[x], e_2) : C$. Combining this derivation with (***), type T is a witness to $\exists A. \Gamma e_1 : A \wedge \Gamma, x : A \vdash (E[x], e_2) : B$.

Cases $(v, E[e])$, $E[e_1]\ e_2$, $v\ E[e]$, **if** $E[e_1]$ **then** e_2 **else** e_3, **send** $E[e_1]\ e_2$ and **send** $v\ E[e]$ can be proved similarly.

let $y = E[e_1]$ **in** e_2: Let us assume that:

1) $\mathcal{P}(\Gamma, E)$
2) $\Gamma \vdash \textbf{let } y = E[e_1] \textbf{ in } e_2 : B$
3) $x \notin dom(\Gamma)$

We need to show that: $\exists A. \Gamma \vdash e : A \wedge \Gamma, x : A \vdash \textbf{let } y = E[x] \textbf{ in } e_2 : B$.

The only rule that can derive hypothesis (2) is [T-Let], from the premises of which we can derive $\Gamma \vdash E[e_1] : C$ (*) and $\Gamma, y : C \vdash e_2 : B$ (**) for some type C. Let us instantiate hypothesis (1) with type C and expression e_1, and apply (*) and hypothesis (3) to it to have that $\exists A. \Gamma \vdash e_1 : A \wedge \Gamma, x : A \vdash E[x] : C$. Let us suppose that T is such a type. Then we have that $\Gamma \vdash e_1 : T$ (***) and $\Gamma, x : T \vdash E[x] : C$ (****). By Lemma 2.2 (Weakening) on (**) and hypothesis (3), and by Barendregt's convention (remember that y is a bound variable in the case in analysis) and the definition of $,_{\mathcal{B}}$, we have that $\Gamma, x : T, y : C \vdash e_2 : B$. We can now apply rule [T-Let] on this last judgement and (****) to derive that $\Gamma, x : T \vdash \textbf{let } y = E[x] \textbf{ in } e_2$. From this and (***), we have that $\Gamma \vdash e_1 : T \wedge \Gamma, x : T \vdash \textbf{let } y = E[x] \textbf{ in } e_2$. Thus, type T is a witness to $\exists A. \Gamma \vdash e_1 : A \wedge \Gamma, x : T \vdash \textbf{let } y = E[x] \textbf{ in } e_2$.

∎

Before attempting to prove Type Preservation, we need to prove a lemma to prove that the Strip function $/k$ preserves type judgements:

Lemma 2.6 (Type Preservation under Strip function)
If $\Gamma, k \vdash P : A$, *then* $\Gamma \vdash P/k : A$.

Proof
We need to prove that, $\forall P. \mathcal{P}(P)$, where $\mathcal{P}(P) = \forall \Gamma. \forall B. \Delta \vdash P : B \wedge (\Delta = \Gamma, k) \Rightarrow \Gamma \vdash P/k : B$, . Let us prove this by structural induction over the process P.

$()$: Let us assume that:

1) $\Delta \vdash () : A$
2) $\Delta = \Gamma, k$

We need to prove that $\Gamma \vdash ()/k : A$.

This follows immediately from hypothesis (1), because $()/k = ()$ by the definition of Strip function $-/-$ and by rule [T-Unit].

The cases for expressions **true**, **false**, $n \in \mathbb{N}$, $x \in Var$, $c \in Chan$ and $\textbf{newChan}_A$ can be proved similary.

fun $f(x) = e$ Let us assume that:

1) $\Delta \vdash \textbf{fun } f(x) = e : B \longrightarrow A$
2) $\Delta = \Gamma, k$
3) $\mathcal{P}(e_1)$

We need to prove that $\Gamma \vdash \textbf{fun } f(x) = e/k : B \longrightarrow A$.

The only typing rule that can derive hypothesis (1) is [T-REC], from the premises of which we can derive that $\Delta, f : B \longrightarrow A, x : B \vdash e : A$ (*). By hypothesis (2) the typing context $\Delta, f : B \longrightarrow A, x : B$ equal to $\Gamma, k, f : B \longrightarrow A, x : B$, which in turn is equal to $\Gamma, f : B \longrightarrow A, x : B, k$ by the definition of operator ",". We can instantiate inductive hypothesis (3) with typing context $\Gamma, f : B \longrightarrow A, x : B, k$ and type A, apply our last consideration and (*) to have $\Gamma, f : B \longrightarrow A, x : B \vdash (e/k) : A$. By rule [T-REC] we have that $\Gamma \vdash \textbf{fun } f(x) = (e/k) : B \longrightarrow A$. By the definition of function $-/-$, $\Gamma \vdash \textbf{fun } f(x) = e/k : B \longrightarrow A$.

$e_1 \parallel e_2$: Let us assume that:

1) $\Delta \vdash e_1 \parallel e_2 : A$
2) $\Delta = \Gamma, k$
3) $\mathcal{P}(e_1)$
4) $\mathcal{P}(e_2)$

We need to prove that $\Gamma \vdash (e_1 \parallel e_2)/k : A$.

The only typing rules that can derive hypothesis (1) are [T-PAR-L] and [T-PAR-R], from the premises of which we can derive that either $\Delta \vdash e_1 : A$ and $\Delta \vdash e_2 : \textbf{unit}$, or $\Delta \vdash e_1 : \textbf{unit}$ and $\Delta \vdash e_2 : A$.

Let us analyze the former case first. We can apply $\Delta \vdash e_1 : A$ and hypothesis (2) on inductive hypothesis (3) to derive $\Gamma \vdash e_1/k : A$. Similarly we can derive $\Gamma \vdash e_2/k : \textbf{unit}$ applying the other judgement and hypothesis (2) on inductive hypothesis (4). From these two judgements, we can derive $\Gamma \vdash e_1/k \parallel e_2/k : A$ by rule [T-PAR-L]. By the definition of the $-/-$ function, $(e_1 \parallel e_2)/k = (e_1/k) \parallel (e_2/k)$. Thus, we have that $\Gamma \vdash (e_1 \parallel e_2)/k : A$. The latter case, where $\Delta \vdash e_1 : \textbf{unit}$ and $\Delta \vdash e_2 : A$, can be proved similarly.

Cases (e_1, e_2), **let** $f = x$ **in** e, $e_1\ e_2$, **send** $v_1\ v_2$, **recv** v, **spawn** e, *op* e, $\nu\, c.\ P$ can be proved similarly.

atomic $[\![\, e_1 \rhd_k e_2 \,]\!]$: Let us assume that:

1) $\Delta \vdash \textbf{atomic } [\![\, e_1 \rhd_k e_2 \,]\!] : A$
2) $\Delta = \Gamma, k$
3) $\mathcal{P}(e_1)$

4) $\mathcal{P}(e_2)$

We need to prove that $\Gamma \vdash (\textbf{atomic}\ [\![\, e_1 \rhd_k e_2 \,]\!])/k : A$.

The only typing rule that can derive hypothesis (1) is [T-Atomic], from the premises of which we can derive that $\Delta \vdash e_1 : A$ and $\Gamma \vdash e_2 : A$. We can apply the first judgement to inductive hypothesis (3), instantiating Γ as Γ, k and B as A. The second judgement to inductive hypothesis (3), where $\Gamma' = \Gamma$ and $B' = A$, to derive $\Gamma, k \vdash e_1/k : A$ and $\Gamma \vdash e_2/k : A$. Using rule [T-Atomic] on the last two judgements we have that $\Gamma \vdash \textbf{atomic}\ [\![\, e_1/k \rhd_k e_2/k \,]\!]$. By the definition of the $-/-$ function, $\textbf{atomic}\ [\![\, e_1/k \rhd_k e_2/k \,]\!] = \textbf{atomic}\ [\![\, e_1 \rhd_k e_2 \,]\!]/k$. Thus, we have that $\Gamma \vdash \textbf{atomic}\ [\![\, e_1 \rhd_k e_2 \,]\!]/k : A$.

Case $[\![\, e_1 \ \rhd_k\ e_2 \,]\!]$ can be proved similarly.

commit k: Let us assume that:

1) $\Delta \vdash \textbf{commit}\ l : \textbf{unit}$
2) $\Delta = \Gamma, k$

We need to prove that $\Gamma \vdash \textbf{commit}\ l/k : A$

Regarding transaction names k and l, either $l = k$ or $l \neq k$. In the former case, $\textbf{commit}\ k/k = ()$ by the definition of the $-/-$ function. Because of this, since $\Gamma \vdash () : A$ by rule [T-Unit], we have that $\Gamma \vdash \textbf{commit}\ k/k : A$. In the latter case, the proof follows directly from hypothesis (1), since $\textbf{commit}\ l/k = \textbf{commit}\ l$ if $l \neq k$.

Case $\textbf{co}\ k$ can be proved similarly.

∎

We also need to prove that type judgements are preserved under the equivalence relation:

Lemma 2.7 (Type Preservation under Equivalence)
If $P \equiv P'$, then $\Gamma \vdash P : A$ if and only if $\Gamma \vdash P' : A$

Proof
We need to prove that $\forall \Gamma, P_1, P_2, A.\ P_1 \equiv P_2 \Rightarrow \mathcal{P}(\Gamma, P_1, P_2, A)$, where $\mathcal{P}(\Gamma, P_1, P_2, A) = \Gamma \vdash P_1 : A \Leftrightarrow \Gamma \vdash P_2 : A$.

Let us prove this lemma by induction over the structural equivalence $\equiv$.

Reflexivity: Let us assume that

1) $\Gamma \vdash P : A$
2) $P \equiv P$

We need to show that: $\Gamma \vdash P : A$.

This follows directly from hypothesis (1).

Symmetry: Let us assume that

1) $P_2 \equiv P_1$
2) $\mathcal{P}(\Gamma, P_2, P_1, A)$
3) $\Gamma \vdash P_1 : A$
4) $P_1 \equiv P_2$

We need to show that $\Gamma \vdash P_1 : A \Leftrightarrow \Gamma \vdash P_2 : A$.

In order to prove the equality in this case, we need to prove the following:

i) $\Gamma \vdash P_1 : A \Rightarrow \Gamma \vdash P_2 : A$.
ii) $\Gamma \vdash P_2 : A \Rightarrow \Gamma \vdash P_1 : A$.

In order to prove i), let us assume that $\Gamma \vdash P_1 : A$. By inductive hypothesis (2), $\Gamma \vdash P_1 : A \Leftrightarrow \Gamma \vdash P_2 : A$. Thus we can apply our assumption on inductive hypothesis (2) to have $\Gamma \vdash P_2 : A$ (*). Thus, since we proved this assuming (*), we can abstract our assumption from this proof and have that $\Gamma \vdash P_1 : A \Rightarrow \Gamma \vdash P_2 : A$.

Case ii) is symmetric and can be proved similarly.

From i) and ii) we can infer that $\Gamma \vdash P_1 : A \Leftrightarrow \Gamma \vdash P_2 : A$.

Transitivity: Let us assume that

1) $P_1 \equiv P_2$
2) $\mathcal{P}(\Gamma, P_1, P_2, A)$
3) $P_2 \equiv P_3$
4) $\mathcal{P}(\Gamma, P_2, P_3, A)$
5) $P_1 \equiv P_3$

We need to show that: $\Gamma \vdash P_1 : A \Leftrightarrow \Gamma \vdash P_3 : A$.

In order to prove the equality in this case, we need to prove the following:

i) $\Gamma \vdash P_1 : A \Rightarrow \Gamma \vdash P_3 : A$.
ii) $\Gamma \vdash P_3 : A \Rightarrow \Gamma \vdash P_1 : A$.

In order to prove i), let us assume that $\Gamma \vdash P_1 : A$. By applying it to inductive hypothesis (2), we have that $\Gamma \vdash P_2 : A$. By applying this to inductive hypothesis (4) we have $\Gamma \vdash P_3 : A$. By abstracting from assumption, we have that $\Gamma \vdash P_1 : A \Rightarrow \Gamma \vdash P_3 : A$.

In order to prove ii), let us assume that $\Gamma \vdash P_3 : A$. By applying it to inductive hypothesis (4), we have that $\Gamma \vdash P_2 : A$. By applying this to inductive hypothesis (2) we have $\Gamma \vdash P_1 : A$. By abstracting from our assumption, we have that $\Gamma \vdash P_3 : A \Rightarrow \Gamma \vdash P_1 : A$.

Combining i) and ii), we have that $\Gamma \vdash P_1 : A \Leftrightarrow \Gamma \vdash P_3 : A$.

[Eq-Com] Let us assume that:

1) $P_1 \parallel P_2 \equiv P_2 \parallel P_1$

We need to show that: $\Gamma \vdash P_2 \parallel P_1 : A \Leftrightarrow \Gamma \vdash P_1 \parallel P_2 : A$.

In order to prove the equality in this case, we need to prove the following:

i) $\Gamma \vdash P_2 \parallel P_1 : A \Rightarrow \Gamma \vdash P_1 \parallel P_2 : A$.

ii) $\Gamma \vdash P_1 \parallel P_2 : A \Rightarrow \Gamma \vdash P_2 \parallel P_1 : A$.

In order to prove i), let us assume that $\Gamma \vdash P_2 \parallel P_1$. Only two rules can derive this judgement: [T-Par-L] or [T-Par-R]. Let us analyze each case in which they were used:

a) Rule [T-Par-L] was used. Then from its premises we have that $\Gamma \vdash P_1 :$ **unit** and $\Gamma \vdash P_2 : A$. We can then prove that $\Gamma \vdash P_2 \parallel P_1 : A$ using rule [T-Par-R].

b) Rule [T-Par-R] was used. Then from its premises we have that $\Gamma \vdash P_1 : A$ and $\Gamma \vdash P_2 :$ **unit**. We can then prove that $\Gamma \vdash P_2 \parallel P_1 : A$ using rule [T-Par-L].

Abstracting from our initial assumption, we have that $\Gamma \vdash P_2 \parallel P_1 : A \Rightarrow \Gamma \vdash P_1 \parallel P_2 : A$.

Case i) can be proved very similarly to case i), starting from assumption $\Gamma \vdash P_1 : A$ instead of assumption $\Gamma \vdash P_2 : A$. We will not reproduce it for brevity's sake.

Combining i) and ii), we have that $\Gamma \vdash P_2 \parallel P_1 : A \Leftrightarrow \Gamma \vdash P_1 \parallel P_2 : A$

[Eq-Assoc] Let us assume that:

1) $P_1 \parallel (P_2 \parallel P_3) \equiv (P_1 \parallel P_2) \parallel P_3$

We need to show that: $\Gamma \vdash (P_1 \parallel P_2) \parallel P_3 : A \Leftrightarrow \Gamma \vdash P_1 \parallel (P_2 \parallel P_3) : A$.

In order to prove the equality in this case, we need to prove the following:

i) $\Gamma \vdash (P_1 \parallel P_2) \parallel P_3 : A \Rightarrow \Gamma \vdash P_1 \parallel (P_2 \parallel P_3) : A$.

ii) $\Gamma \vdash P_1 \parallel (P_2 \parallel P_3) : A \Rightarrow \Gamma \vdash (P_1 \parallel P_2) \parallel P_3 : A$.

In order to prove i), let us assume that $\Gamma \vdash (P_1 \parallel P_2) \parallel P_3 : A$. This assumption can only be derived by a combination of rules [T-Par-L] and [T-Par-R], depending on which process P_i has type A. Let us analyze all possible cases:

a) Rule [T-Par-R] was used first, and then either [T-Par-L] or [T-Par-L]. Then from the premises of these rules, we have that $\Gamma \vdash P_1 :$ **unit**, $\Gamma \vdash P_2 :$ **unit** and $\Gamma \vdash P_3 : A$. We can then prove that $\Gamma \vdash P_2 \parallel P_3 : A$ using rule [T-Par-R] and then that $\Gamma \vdash P_1 \parallel (P_2 \parallel P_3) : A$ again by rule [T-Par-R].

b) Rule [T-Par-L] and then [T-Par-R] were used. Then from all the premises, we get that $\Gamma \vdash P_1 : \textbf{unit}$, $\Gamma \vdash P_2 : A$ and $\Gamma \vdash P_3 : \textbf{unit}$. We can then prove that $\Gamma \vdash P_2 \parallel P_3 : A$ by rule [T-Par-L] and then that $\Gamma \vdash P_1 \parallel (P_2 \parallel P_3) : A$ by rule [T-Par-R].

c) Rule [T-Par-L] and then [T-Par-L] were used. Then from all the premises, we get that $\Gamma \vdash P_1 : \textbf{unit}$, $\Gamma \vdash P_2 : \textbf{unit}$ and $\Gamma \vdash P_3 : \textbf{unit}$. We can then prove that $\Gamma \vdash P_2 \parallel P_3 : \textbf{unit}$ using rule [T-Par-R] and then that $\Gamma \vdash P_1 \parallel (P_2 \parallel P_3) : A$ by rule [T-Par-R].

Abstracting from our previous assumption, we have that $\Gamma \vdash (P_1 \parallel P_2) \parallel P_3 : A \Rightarrow \Gamma \vdash P_1 \parallel (P_2 \parallel P_3) : A$.

Case ii) can be proved with a very similar case analysis as the one above, starting from assumption $\Gamma \vdash P_1 \parallel (P_2 \parallel P_3) : A$ instead of assumption $(P_1 \parallel P_2) \parallel P_3$. We will omit it for brevity's sake.

Combining i) and ii), we have that $\Gamma \vdash (P_1 \parallel P_2) \parallel P_3 : A \Leftrightarrow \Gamma \vdash P_1 \parallel (P_2 \parallel P_3) : A$.

[Eq-Restr] Let us assume that:

1) $\nu c.P_1 \parallel P_2 \equiv \nu c.(P_1 \parallel P_2)$

We need to show that: $\Gamma \vdash \nu c.P_1 \parallel P_2 : A \Leftrightarrow \Gamma \vdash \nu c.(P_1 \parallel P_2) : A$.

In order to prove the equality in this case, we need to prove the following:

i) $\Gamma \vdash \nu c.P_1 \parallel P_2 : A \Leftrightarrow \Gamma \vdash \nu c.(P_1 \parallel P_2) : A$.

ii) $\Gamma \vdash \nu c.(P_1 \parallel P_2) : A \Leftrightarrow \Gamma \vdash \nu c.P_1 \parallel P_2 : A$.

In order to prove i), let us assume that $\Gamma \vdash \nu c.P_1 \parallel P_2 : A$. This assumption can only be derived by either rules [T-Par-L] and [T-Restr], or by rules [T-Par-R] and [T-Restr]. Let us analyze each case:

a) Rules [T-Par-L] and [T-Restr] were used. From their premises we have that $\Gamma \vdash P_1 : A$ and $\Gamma \vdash P_2 : \textbf{unit}$. Then we can first derive that $\Gamma \vdash P_1 \parallel P_2 : A$ by rule [T-Par-L], and then $\Gamma \vdash \nu c.(P_1 \parallel P_2) : A$ from rule [T-Restr].

b) Rule [T-Par-R] and [T-Restr] were used. From their premise we have that $\Gamma \vdash P_1 : \textbf{unit}$ and $\Gamma \vdash P_2 : A$. Then we can first derive that $\Gamma \vdash P_1 \parallel P_2 : A$ by rule [T-Par-R], and then $\Gamma \vdash \nu c.(P_1 \parallel P_2) : A$ from rule [T-Restr].

Abstracting from our previous assumption, we have that $\Gamma \vdash \nu c.P_1 \parallel P_2 : A \Leftrightarrow \Gamma \vdash \nu c.(P_1 \parallel P_2) : A$.

Case ii) can be proved with a very similar case analysis as the one above, starting from assumption $\Gamma \vdash \nu c.(P_1 \parallel P_2) : A$ instead of assumption $\Gamma \vdash \nu c.P_1 \parallel P_2 : A$. We will omit it for brevity's sake.

Combining i) and ii), we have that $\Gamma \vdash \nu c.P_1 \parallel P_2 : A \Leftrightarrow \Gamma \vdash \nu c.(P_1 \parallel P_2) : A$.

∎

We first prove a type preservation lemma for sequential reductions $\hookrightarrow$. The lemma states that if a well-typed expression takes a sequential reduction step, then the resulting expression is also well-typed:

Lemma 2.8 (Sequential Type Preservation)
If $\Gamma \vdash e : A$ *and* $e \hookrightarrow e'$*, then* $\Gamma \vdash e' : A$

Proof
We need to prove that $\forall \Gamma, e, A.\ e \hookrightarrow e' \Rightarrow \mathcal{P}(\Gamma, e, e', A)$, where $\mathcal{P}(\Gamma, e, e', A) = \Gamma \vdash e : A \Rightarrow \Gamma \vdash e' : A$.

Let us prove this lemma by induction on the derivations of $e \hookrightarrow e'$.

[E-App] : Let us assume that:

1) $\Gamma \vdash (\textbf{fun } f(x) = e)\ v_2 : A$
2) $(\textbf{fun } f(x) = e)\ v_2 \hookrightarrow e[(\textbf{fun } f(x) = e)/f][v_2/x]$

We need to show that: $\Gamma \vdash e[(\textbf{fun } f(x) = e)/f][v_2/x] : A$.

Hypothesis (2) can only be typed by rule [T-App]; from its premise we can deduce that $\exists T. \Gamma \vdash e_1 : T \longrightarrow A \wedge \Gamma \vdash e_2 : T$. Let us assume that B is such a type T. We can then infer that $\Gamma \vdash \textbf{fun } f(x) = e : B \longrightarrow A$ (*) and $\Gamma \vdash v_2 : B$ (**). The only typing rule that can infer (*) is [T-Rec], from the premise of which we have that $\Gamma, f : B \longrightarrow A, x : B \vdash e : B$. We can also apply Lemma 2.2 (Weakening) on (*) to have $\Gamma, x : B \vdash \textbf{fun } f(x) = e : B \longrightarrow A$, since $x \notin FV(\textbf{fun } f(x) = e)$, because x is bound in (*). From these last two judgements, and because we can write $\Gamma, f : B \longrightarrow A, x : B$ as $\Gamma, x : B, f : B \longrightarrow A$, we have $\Gamma, x : B \vdash e[(\textbf{fun } f(x) = e)/f] : B$ by Lemma 2.4 (Substitution). From this and (**), we have $\Gamma \vdash e[(\textbf{fun } f(x) = e)/f][v_2/x] : B$ again by Lemma 2.4 (Substitution).

[E-Let] : Let us assume that:

1) $\Gamma \vdash \textbf{let } x = v \textbf{ in } e : B$
2) $\textbf{let } x = v \textbf{ in } e \hookrightarrow e[v/x]$

We need to show that: $\Gamma \vdash e[v/x] : B$.

Hypothesis (1) can only be derived from rule [T-Let]. From its premise, we have that $\Gamma \vdash e : A$ and $\Gamma, x : A \vdash e : B$. By Lemma 2.4 (Substitution), $\Gamma \vdash e : B$.

[E-If-True] : Let us assume that:

1) $\Gamma \vdash \textbf{if true then } e_1 \textbf{ else } e_2 : A$

2) $\textbf{if true then } e_2 \textbf{ else } e_3 \hookrightarrow e_1$

We need to show that: $\Gamma \vdash e_1 : A$.

Hypothesis (1) can only be derived from rule [T-IF]. From its premises, we get directly $\Gamma \vdash e_1 : A$.

[E-IF-FALSE] : Same as [E-IF-TRUE] .

[E-OP] : We will only consider the case for **add** ; the cases for the other operations are very similar. Let us assume that:

1) $\Gamma \vdash \textbf{add}\ v : B$
2) $\textbf{add}\ v \hookrightarrow \delta(v)$

We need to show that: $\Gamma \vdash \delta(v) : B$.

Hypothesis (1) can only be derived from rule [T-OP]. From the its premise, we can infer that $\Gamma \vdash v : A$; from the condition on its premise, we can derive that $B = \textbf{int}$ and $A = \textbf{int} \times \textbf{int}$. Thus, we need to show that $\Gamma \vdash \delta(v) : \textbf{int}$; moreover, we have that $\Gamma \vdash v : \textbf{int} \times \textbf{int}$. Only rule [T-VTUP] can produce such a judgement, thus v must have the form (v_1, v_2). Moreover, from the premises of [T-VTUP], $\Gamma \vdash v_1 : \textbf{int}$ and $\Gamma \vdash v_2 : \textbf{int}$. From rule [T-INT], these values must be integers. Since $\delta(v_1, v_2)$ is equal to a sum of integers, and integers have type **int** from rule [T-INT], we can derive that $\Gamma \vdash \delta(v) : \textbf{int}$. ∎

We are now ready to prove the Type Preservation theorem. According to this theorem, if a well-typed process takes a reduction step of any kind, the resulting process will be well-typed:

Theorem 2.9 (Type Preservation)
If $\Gamma \vdash P : A$ *and* $P \longrightarrow P'$, *then* $\Gamma \vdash P' : A$.

Proof
We need to prove that $\forall P. \forall P'.\ P \longrightarrow P' \Rightarrow \mathcal{P}(P, P')$, where
$\mathcal{P}(P, P') = \forall \Gamma. \forall A.\, \Gamma \vdash P : A \Rightarrow \Gamma \vdash P' : A$.

Let us prove this theorem by induction on derivations of $P \longrightarrow P'$.

[C-STEP] Let us assume that:

1) $e \hookrightarrow e'$
2) $\Gamma \vdash E[e] : A$
3) $E[e] \longrightarrow E[e']$

We need to show that: $\Gamma \vdash E[e'] : A$.

Let us pick a variable $x \notin dom(\Gamma)$. By Lemma 2.5 (Context Abstraction) on hypothesis (2), we have that $\exists B.\, \Gamma \vdash e : B \wedge \Gamma, x : B \vdash E[x] : A$. Let us assume that T is such a type. Then we have that $\Gamma \vdash e : T$ (*) and $\Gamma, x :$

$T \vdash E[x] : A$ (**). By Lemma 2.8 (Sequential Type Preservation) on (*) and hypothesis (1), we have that $\Gamma \vdash e' : T$. By Lemma 2.4 (Substitution), this last derivation and (**), we have that $\Gamma \vdash E[x][e'/x] : A$. Since we have chosen a variable x that does not occur in Γ, we know that x does not occur free in the evaluation context E either. Moreover, remember that a hole in an evaluation context is never caught by a scope of a binder. Thus, by the definition of substitution in Table 5, we have that $E[x][e'/x] = E[x[e'/x]] = E[e']$, and thus that $\Gamma \vdash E[e'] : A$

[C-Sync] Let us assume that:

1) $\Gamma \vdash E_1[\,\mathbf{recv}\ c] \parallel E_2[\,\mathbf{send}\ c\ v] : A$
2) $E_1[\,\mathbf{recv}\ c] \parallel E_2[\,\mathbf{send}\ c\ v] \longrightarrow E_1[v] \parallel E_2[()]$

We need to show that: $\quad \Gamma \vdash E_1[v] \parallel E_2[()] : A$.

Hypothesis (1) can only be derived by either rule [T-Par-L] or [T-Par-R]. We will first analyse the case where rule [T-Par-L] has been used.

From the premises of [T-Par-L], we have that $\Gamma \vdash E_1[\,\mathbf{recv}\ c] : A$ (*) and $\Gamma \vdash E_2[\,\mathbf{send}\ c\ v] : \mathbf{unit}$ (**). Let us pick two variable $x \notin dom(\Gamma)$ and $y \notin dom(\Gamma)$. By Lemma 2.5 (Context Abstraction) on the two previous judgements, we have that $\exists B.\, \Gamma \vdash \mathbf{recv}\ c : B \wedge \Gamma, x : B \vdash E_1[x] : A$ and that $\exists C.\, \Gamma \vdash \mathbf{send}\ c\ v : C \wedge \Gamma, y : C \vdash E_2[y] : \mathbf{unit}$. Let us assume that T and U are such types. Then we have $\Gamma \vdash \mathbf{recv}\ c : T$, $\ \Gamma, x : T \vdash E_1[x] : A$(*), $\Gamma \vdash \mathbf{send}\ c\ v : U$ and $\Gamma, y : U \vdash E_2[y] : \mathbf{unit}$ (**).

The only rule that can derive $\Gamma \vdash \mathbf{recv}\ c : T$ is [T-Recv]; because of the side condition on this rule, we have that $ty(c) = T\,\mathbf{chan}$ (***). The only rule that can derive $\Gamma \vdash \mathbf{send}\ c\ v : U$ is [T-Send] , from the premises of which we have that U is equal to $\mathbf{unit}$, $\Gamma \vdash v : Z$ and $ty(c) = Z\,\mathbf{chan}$. By (***), $Z = T$ and thus $\Gamma \vdash v : T$. By Lemma 2.4 (Substitution) on (*) and this last judgement, we have that $\Gamma \vdash E_1[v] : A$ (****).

We can also directly derive $\Gamma \vdash () : \mathbf{unit}$. By Lemma 2.4 (Substitution) on (**) and this last judgement, we have that $E_2[()] : \mathbf{unit}$. From this last judgement, (****) and rule [T-Par-L], we have that $\Gamma \vdash E_1[v] \parallel E_2[()] : A$.

The case where [T-Par-R] has been used can be proved similarly.

[C-Spawn] Let us assume that:

1) $\Gamma \vdash E[\mathbf{spawn}\ v] : A$
2) $E[\mathbf{spawn}\ v] \longrightarrow v\ () \parallel E[()]$

We need to show that: $\quad \Gamma \vdash v\ () \parallel E[()] : A$.

By Lemma 2.5 (Context Abstraction) on hypothesis (1), we have that $\exists B.(\Gamma \vdash \mathbf{spawn}\ v : B) \wedge (\Gamma, x : B \vdash E[()] : A)$. Let us assume that T is such a type. Then we have $\Gamma \vdash \mathbf{spawn}\ v : T$ (*) and $\Gamma, x : T \vdash E[x] : A$ (**). The only rule that can type (*) is rule [T-Spawn], from which we have that $\Gamma \vdash v : \textbf{unit} \longrightarrow \textbf{unit}$ (***) and that T must be equal to **unit**. Thus (**) becomes $\Gamma, x : \textbf{unit} \vdash E[x] : A$. By rule [T-Unit] we have $\Gamma \vdash () : \textbf{unit}$; applying Lemma 2.4 (Substitution) to this and (**), we have $\Gamma \vdash E[()] : A$. We can also infer that $\Gamma \vdash v\ ()$ from (***) and rule [T-App]. We can finally derive $\Gamma \vdash v\ () \parallel E[()] : A$ from the last two judgements we derived and rule [T-Par-L].

[C-NewChan] Let us assume that:

1) $\Gamma \vdash E[\textbf{newChan}_B] : A$
2) $E[\textbf{newChan}_B] \longrightarrow \nu c.E[c]$
3) $ty(c) = B\,\textbf{chan}$
4) $c \notin FC(E)$

We need to show that: $\quad \Gamma \vdash \nu c.E[c] : A$.

By Lemma 2.5 (Context Abstraction) on hypothesis (1), we have that $\exists C.\, \Gamma \vdash \textbf{newChan}_B : C \wedge \Gamma, x : C \vdash E[()] : A$. Let us assume that T is such a type. Then we have $\Gamma \vdash \textbf{newChan}_B : T$ (*) and $\Gamma, x : T \vdash E[()] : A$. By rule [T-NewChan], we know that $\Gamma \vdash \textbf{newChan}_B : B\,\textbf{chan}$, thus $T = B\,\textbf{chan}$ and $\Gamma, x : B\,\textbf{chan} \vdash E[x] : A$ (**). By rule [T-Chan] and hypothesis (3) we have that $\Gamma \vdash c : B\,\textbf{chan}$. Applying Lemma 2.4 (Substitution) to (**) and this last judgement, we have that $\Gamma \vdash E[c] : A$. Finally, we get $\Gamma \vdash \nu c.E[c] : A$ by rule [T-Restr].

[C-Par] Let us assume that:

1) $P_1 \longrightarrow P_1'$
2) $\mathcal{P}(P_1, P_1')$
3) $\Gamma \vdash P_1 \parallel P_2 : A$
4) $P_1 \parallel P_2 \longrightarrow P_1' \parallel P_2$

We need to show that: $\quad \Gamma \vdash P_1' \parallel P_2 : A$.

Hypothesis (3) can only be derived either by rule [T-Par-L] or by [T-Par-R]. In former case, we have $\Gamma \vdash P_1 : A$ and $\Gamma \vdash P_2 : \textbf{unit}$. Applying the former judgement on inductive hypothesis (2), we have $\Gamma \vdash P' : A$. Applying rule [T-Par-L] to the latter and this last judgement, we have $\Gamma \vdash P_1' \parallel P_2 : A$. Similarly for the latter case.

[C-Chan] Let us assume that:

1) $P \longrightarrow P'$
2) $\mathcal{P}(\Gamma, P, P', A)$
3) $\Gamma \vdash \nu c.P : A$
4) $\nu c.P \longrightarrow \nu c.P'$

We need to show that: $\Gamma \vdash \nu c.P' : A$.

Hypothesis (3) can only be derived by rule [T-Restr], from whose premises we can derive that $\Gamma \vdash P : A$. Applying this to inductive hypothesis (2), we have $\Gamma \vdash P' : A$. Applying rule [T-Restr], we have that $\nu c.P' : A$.

[C-Eq] Let us assume that:

1) $\Gamma \vdash P_1 : A$
2) $P_1 \equiv P_1'$
3) $P_1' \longrightarrow P_2'$
4) $\mathcal{P}(\Gamma, P_1', P_2', A)$
5) $P_2' \equiv P_2$
6) $P_1 \longrightarrow P_2$

We need to show that: $\Gamma \vdash P_2 : A$.

By Lemma 2.7 (Type Preservation Under Equivalence) on hypothesis 1) and 2), we have that $\Gamma \vdash P_1' : A$. Applying inductive hypothesis (4) to this last judgement and hypothesis (3), we have that $\Gamma \vdash P_2' : A$. Applying Lemma 2.7 (Type Preservation Under Equivalence) again on this last judgement and hypothesis (5), we have that $\Gamma \vdash P_2 : A$.

[Tr-Co] Let us assume that:

1) $\Gamma \vdash [\![P_1 \ \rhd_k \ P_2]\!] : A$
2) $P_1 \equiv \mathbf{co}\ k \parallel P_1'$
3) $[\![P_1 \ \rhd_k \ P_2]\!] : A \longrightarrow P_1'/k$

We need to show that: $\Gamma \vdash P_1'/k : A$.

The only rule that can derive hypothesis (1) is [T-Trans], from the premises of which we can derive that $\Gamma, k \vdash P_1 : A$ and $\Gamma \vdash P_2 : A$. By Lemma 2.7 (Type Preservation Under Equivalence) on hypothesis (2) and this last judgement, we have that $\Gamma, k \vdash \mathbf{co}\ k \parallel P_1' : A$. This judgement can be derived by either rule [T-Par-L] or [T-Par-R]. Thus either $\Gamma, k \vdash \mathbf{co}\ k : A$ (*) and $\Gamma, k \vdash P_1' : \mathbf{unit}$, or $\Gamma, k \vdash \mathbf{co}\ k : \mathbf{unit}$ and $\Gamma, k \vdash P_1' : A$. In the former case, we can apply Lemma 2.6 (Type Preservation under Strip Function) to derive $\Gamma \vdash P_1'/k : \mathbf{unit}$. Since we have both $\Gamma, k \vdash \mathbf{co}\ k : \mathbf{unit}$ and (*), by Theorem 2.3 (Type Uniqueness) we have that $A = \mathbf{unit}$, and thus $\Gamma \vdash P_1'/k : A$. In the latter case, we can apply Lemma 2.6 (Type Preservation under Strip Function) to have $\Gamma \vdash P_1'/k : A$.

[Tr-Abort] Let us assume that:

1) $\Gamma \vdash [\![P_1 \ \rhd_k \ P_2]\!] : A$
2) $[\![P_1 \ \rhd_k \ P_2]\!] : A \longrightarrow P_2$

We need to show that: $\Gamma \vdash P_2 : A$.

Hypothesis (1) can only be derived by rule [T-Trans], from whose premises we have $\Gamma \vdash P_2 : A$.

[Tr-Emb] Let us assume that:

1) $\Gamma \vdash P_1 \parallel [\![P_2 \;\rhd_k\; P_3]\!] : A$
2) $P_1 \parallel [\![P_2 \;\rhd_k\; P_3]\!] : A \longrightarrow [\![P_1 \parallel P_2 \;\rhd_k\; P_1 \parallel P_3]\!]$

We need to show that: $\Gamma \vdash [\![P_1 \parallel P_2 \;\rhd_k\; P_1 \parallel P_3]\!] : A$.

Hypothesis (1) can be derived by either by rule [T-Par-L] or [T-Par-R]. Let us examine each case:

a) [T-Par-L] has been used. From the premises of this rule, we have that $\Gamma \vdash P_1 : A$ (*) and $\Gamma \vdash [\![P_2 \;\rhd_k\; P_3]\!] : \textbf{unit}$. Moreover, this last judgement can only be derived from rule [T-Trans], thus from its premises we have $\Gamma \vdash P_2 : \textbf{unit}$ (**) and $\Gamma \vdash P_3 : \textbf{unit}$ (***). By rule [T-Par-L] on (*) and (**) first, and then on (***) and (**), we have that $\Gamma \vdash P_1 \parallel P2 : A$ and $\Gamma \vdash P_1 \parallel P_3 : A$. From these last two judgements, we can derive $\Gamma \vdash [\![P_1 \parallel P_2 \;\rhd_k\; P_1 \parallel P_3]\!] : A$.
b) [T-Par-R] has been used. From the premises of this rule, we have that $\Gamma \vdash P_1 : \textbf{unit}$ (*) and $\Gamma \vdash [\![P_2 \;\rhd_k\; P_3]\!] : A$. Moreover, this last judgement can only be derived from rule [T-Trans], thus from its premises we have $\Gamma \vdash P_2 : A$ (**) and $\Gamma \vdash P_3 : A$ (***). By rule [T-Par-R] on (*) and (**) first, and then on (***) and (**), we have that $\Gamma \vdash P_1 \parallel P2 : A$ and $\Gamma \vdash P_1 \parallel P_3 : A$. From these last two judgements, we can derive $\Gamma \vdash [\![P_1 \parallel P_2 \;\rhd_k\; P_1 \parallel P_3]\!] : A$.

[Tr-Step] Let us assume that:

1) $\Gamma \vdash [\![P_1 \;\rhd_k\; P_2]\!] : A$
2) $P_1 \longrightarrow P_1'$
3) $\mathcal{P}(\Gamma, P_1, P_1', A)$
4) $[\![P_1 \;\rhd_k\; P_2]\!] : A \longrightarrow [\![P_1' \;\rhd_k\; P_2]\!]$

We need to show that: $\Gamma \vdash [\![P_1' \;\rhd_k\; P_2]\!] : A$.

The only rule that can derive hypothesis (1) is [T-Trans], from the premises of which we have that $\Gamma, k \vdash P_1 : A$ (*) and $\Gamma \vdash P_2 : A$ (**). Let us instantiate inductive hypothesis (3) with typing context Γ, k and type A. Applying hypothesis (2) and (*) on the inductive hypothesis, we have that $\Gamma, k \vdash P_1' : A$. From this and (**), we have that $\Gamma \vdash [\![P_1' \;\rhd_k\; P_2]\!] : A$ by rule [T-Trans].

[Tr-Atomic] Let us assume that:

1) $\Gamma \vdash E[\textbf{atomic}\ [\![e_1 \rhd_k e_2]\!]] : A$

2) $E[\textbf{atomic}\ [\![\, e_1 \rhd_k e_2 \,]\!]] \longrightarrow [\![\, E[e_1] \ \rhd_k\ E[e_2] \,]\!]$

We need to show that: $\quad \Gamma \vdash [\![\, E[e_1] \ \rhd_k\ E[e_2] \,]\!] : A$.

Let us pick a variable $x \notin \Gamma$. Then by the Context Abstraction rule and hypothesis (1), we have that $\exists B.\, \Gamma \vdash \textbf{atomic}\ [\![\, e_1 \rhd_k e_2 \,]\!] : B \wedge \Gamma, x : B \vdash E[x] : A$. Let us assume that T is such a type. Then we have that $\Gamma \vdash \textbf{atomic}\ [\![\, e_1 \rhd_k e_2 \,]\!] : T$ and $\Gamma, x : T \vdash E[x] : A$ (*). The only rule that can type $\Gamma \vdash \textbf{atomic}\ [\![\, e_1 \rhd_k e_2 \,]\!] : T$ is [T-Atomic], from the premises of which we have that $\Gamma \vdash e_1 : T$ and $\Gamma \vdash e_2 : T$. Applying Lemma 2.4 (Substitution) to (*) and the last two judgements respectively, we have that $\Gamma \vdash E[e_1] : A$ and $\Gamma \vdash E[e_2] : A$. From these two statements, we can derive $[\![\, E[e_1] \ \rhd_k\ E[e_2] \,]\!] : A$ by rule [T-Trans].

[Tr-Commit] Let us assume that:

1) $\Gamma \vdash E[\textbf{commit}\ k] : A$

2) $E[\textbf{commit}\ k] \longrightarrow \textbf{co}\ k \parallel E[()]$

We need to show that: $\quad \Gamma \vdash \textbf{co}\ k \parallel E[()] : A$.

Let us pick a variable $x \notin \Gamma$. Then by Lemma 2.5 (Context Abstraction), $\exists B.\, \Gamma \vdash \textbf{commit}\ k : B \wedge \Gamma, x : B \vdash E[x] : A$. Let us assume that T is such a type. Then $\Gamma \vdash \textbf{commit}\ k : T$ (*) and $\Gamma, x : T \vdash E[x] : A$ (**). The only rule that can derive (*) is [T-Commit], from which we can infer that $T = \textbf{unit}$ and that $\Gamma = \Delta, k$ (***) for some typing context Δ. Thus (**) is $\Gamma, x : \textbf{unit} \vdash E[x]$. Moreover, by rule [T-Unit], we have that $\Gamma \vdash () : \textbf{unit}$. Thus, from these last two results, we can derive that $\Gamma \vdash E[()]$. By rule [T-Co] and (***) we have that $\Gamma \vdash \textbf{co}\ k : \textbf{unit}$; by rule [T-Par-R] on the last two judgements, we can derive $\Gamma \vdash \textbf{co}\ k \parallel E[()] : A$.

∎

Before proving the progress theorem, we define what it means for an expression to be *stuck*.

Stuck expression An expression e is called *stuck* if and only if:

i) $e \neq v$, that is e is not a value,

ii) $\nexists E, v, c, k. e = E[\,\textbf{send}\ c\ v]$ or
$e = E[\,\textbf{recv}\ c]$ or
$e = E[\textbf{spawn}\ v]$ or
$e = E[\textbf{newChan}_A]$ or
$e = E[\textbf{atomic}\ [\![\, e_1 \rhd_k e_2 \,]\!]]$ or
$e = E[\textbf{commit}\ k]$

iii) $\nexists e'. e \hookrightarrow e'$

An expression is stuck if it is not a value, if it is not deadlocked on sending or receiving over a channel, it is not trying to generate a new process, and cannot take any further reduction step.

The following lemma proves that well-typed expressions cannot be stuck:

Lemma 2.10 (Expression progress)
If $\Gamma \vdash e : A$*, then either* $e = v$ *or* $\exists c.\exists v.e = E[\textbf{\textit{send}}\ c\ v]$ *or* $\exists c.e = E[\textbf{\textit{recv}}\ c]$ *or* $\exists P.e \to P$*.*

Proof
We need to prove that $\forall\Gamma.\forall e.\forall A,\ \Gamma \vdash e : A \Rightarrow \mathcal{P}(\Gamma, e, A)$, where $\mathcal{P}(\Gamma, P, A) = (e = v) \vee (\exists c.\exists v.e = E[\,\textbf{send}\ c\ v]) \vee (\exists c.e = E[\,\textbf{recv}\ c]) \vee (\exists P.e \to P)$. Let us prove this theorem by induction on derivations on judgement $\Gamma \vdash e : A$.

[T-Unit] Let us assume that:

1) $\Gamma \vdash () : \textbf{unit}$

We need to show that $(() = v) \vee (\exists c.\exists v.() = E[\,\textbf{send}\ c\ v]) \vee (\exists c.() = E[\,\textbf{recv}\ c]) \vee (\exists P.() \to P)$.

This case is trivial, since () is a value.

The cases for rules [T-Int], [T-Bool], [T-Var], [T-Rec] and [T-Chan] are trivial too.

[T-Tup] Let us assume that:

1) $\Gamma \vdash (e_1, e_2) : A_1 \times A_2$
2) $\Gamma \vdash e_1 : A_1$
3) $\Gamma \vdash e_2 : A_2$
4) $\mathcal{P}(\Gamma, e_1, A_1)$
5) $\mathcal{P}(\Gamma, e_2, A_2)$

We need to show that $((e_1, e_2) = v) \vee (\exists c.\exists v.(e_1, e_2) = E[\,\textbf{send}\ c\ v]) \vee (\exists c.(e_1, e_2) = E[\,\textbf{recv}\ c]) \vee (\exists P.(e_1, e_2) \to P)$.

From the inductive hypothesis, we can infer that $(e_1 = v) \vee (\exists c.\exists v.e_1 = E[\,\textbf{send}\ c\ v]) \vee (\exists c.e_1 = E[\,\textbf{recv}\ c]) \vee (\exists P_1.e_1 \to P_1)$, and that $(e_2 = v) \vee (\exists c.\exists v.e_2 = E[\,\textbf{send}\ c\ v]) \vee (\exists c.e_2 = E[\,\textbf{recv}\ c]) \vee (\exists P_2.e_2 \to P_2)$.

Let us assume that $e_1 = v_1$. If $e_2 = v_2$, then $e = (v1, v_2)$, which is a value, and the theorem holds. If $\exists c.\exists v.e_2 = E[\,\textbf{send}\ c\ v]$ or $\exists c.e_2 = E[\,\textbf{recv}\ c]$, then $e = (v_1, E[\,\textbf{send}\ c\ v])$ or $e = (v_1, E[\,\textbf{recv}\ c])$ for a given c and v. According to Table 1, (v_1, E) is also an evaluation context, thus we can state that $\exists c.\exists v.e = E[\,\textbf{send}\ c\ v]$ or $\exists c.e = E[\,\textbf{recv}\ c]$, and the theorem holds.

Lastly, let us suppose that $\exists P_2.e_2 \to P_2$. The only rules can reduce an expression to a process are rules [C-Spawn], [C-NewChan], [Tr-Commit], [Tr-Atomic] and [C-Step]. Let us consider rule [C-Spawn]. According to this rule, $e_2 = E[\textbf{spawn}\ v]$ and $e_2 \rightarrow v\ ()\ \|\ E[()]$. According to Table 1, (v_1, E) is also an evaluation context, thus we can define another evaluation context $E' = (v_1, E)$ such that $e = E'[\textbf{spawn}\ v]$. We can now apply rule [C-Spawn] to derive $E'[\textbf{spawn}\ v] \rightarrow v\ ()\ \|\ E'[()]$. This last reduction proves that $\exists P.(e_1, e_2) \rightarrow P$.. The cases for [C-NewChan], [Tr-Commit], [Tr-Atomic] and [C-Step] are similar.

Let us assume that $\exists c.\exists v.e_1 = E[\,\textbf{send}\ c\ v]$ or $\exists c.e_1 = E[\,\textbf{recv}\ c]$. According to Table 1, (E, e_2) is also an evaluation context, thus we can define another evaluation context $E' = (E, e_2)$, for some given c and v, such that $\exists c.\exists v.e = E'[\,\textbf{send}\ c\ v]$ or $\exists c.e = E'[\,\textbf{recv}\ c]$, in which cases the theorem holds.

Lastly, let us assume that $(\exists P_1.e_1 \rightarrow P_1)$. The only rules can reduce an expression to a process are rules [C-Spawn], [C-NewChan], [Tr-Commit], [Tr-Atomic] and [C-Step]. Let us consider rule [C-Spawn]. According to this rule, $e_1 = E[\textbf{spawn}\ v]$ and $e_1 \rightarrow v\ ()\ \|\ E[()]$. Using the evaluation context E' that we just defined, we have that $e = E'(\textbf{spawn}\ v)$. We can now apply rule [C-Spawn] on e to derive $E'[\textbf{spawn}\ v] \rightarrow v\ ()\ \|\ E'[()]$. This last reduction proves that $\exists P.(e_1, e_2) \rightarrow P$. The cases for [C-NewChan], [Tr-Commit], [Tr-Atomic] and [C-Step] are similar.

Having analysed all cases, we have proved that the theorem holds.

[T-If] Let us assume that:

1) $\Gamma \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : A$
2) $\Gamma \vdash e_1 : \textbf{bool}$
3) $\Gamma \vdash e_2 : A$
4) $\Gamma \vdash e_3 : A$
5) $\mathcal{P}(\Gamma, e_1, \textbf{bool})$
6) $\mathcal{P}(\Gamma, e_2, A_1)$
7) $\mathcal{P}(\Gamma, e_3, A_2)$

We need to show that $(\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 = v) \vee (\exists c.\exists v.\,\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 = E[\,\textbf{send}\ c\ v]) \vee (\exists c.\,\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 = E[\,\textbf{recv}\ c]) \vee (\exists P.\,\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \rightarrow P)$.

By inductive hypothesis 5) we know that either $e_1 = v$ or $\exists c.\exists v.e_1 = E[\,\textbf{send}\ c\ v]$ or $\exists c.e_1 = E[\,\textbf{recv}\ c]$ or $\exists P.e_1 \rightarrow P$. Let us analyse each case.

Let us suppose that $e_1 = v$. Hypothesis 2) states that $\Gamma \vdash e_1 : \textbf{bool}$. Thus e_1 is a value of type **bool**. According to Table 14, there are only two values of type **bool**, **true** and **false**. In either case the expression $\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3$ can take a sequential reduction step through either rule [E-If-True] or [E-If-False] , which is also a process reduction step according to rule [C-Step]. Thus the theorem holds for this case.

$PExpr(e) = \{e\}$	$PExpr(P_1 \parallel P_2) = PExpr(P_1) \cup PExpr(P_2)$
$PExpr(\nu c.P) = PExpr(P)$	$PExpr(\llbracket P_1 \ \rhd_k \ P_2 \rrbracket) = PExpr(P_1)$
$PExpr(\textbf{co}\ k) = \{\}$	

Table 18: PExpr predicate.

Let us assume that $\exists c.\exists v.e_1 = E[\textbf{send}\ c\ v]$. According to Table 1, **if** E **then** e_2 **else** e_3 is also an evaluation context. For given c and v, we can define another evaluation context $E' =$ **if** E **then** e_2 **else** e_3 such that $\exists c.\exists v.e = E'[\textbf{send}\ c\ v]$, in which case the theorem holds. The case in which $\exists c.e_1 = E[\textbf{recv}\ c]$ can be proved similarly.

Let us assume that $\exists P.e \to P$. The only reduction steps that apply on an expression are [C-Step], [C-Spawn], [C-NewChan], [Tr-Commit] and [Tr-Atomic]. Let us assume that rule [C-Step] was used to derive that $\exists P.e \to P$. According to this rule, $e = E[e_1]$ and $e_1 \hookrightarrow e_1'$ (*). According to Table 1, **if** E **then** e_1 **else** e_2 is also an evaluation context. Thus we can define another evaluation context $E' =$ **if** E **then** e_2 **else** e_3. We can now derive that $E'[e_1] \to E'[e_1']$ by rule [C-Step], which proves that $\exists P.e \to P$. Thus the theorem holds in this case. The cases for rules [C-Spawn], [C-NewChan], [Tr-Commit], [Tr-Atomic] are similar.

The cases for rules [T-Let], [T-Op], [T-Send], [T-Recv] can be proved similarly. The case for [T-App] is similar, but it requires to consider all possible instances of both e_1 and e_2.

[T-NewChan] We need to show that $(\textbf{newChan}_A = v) \vee (\exists c.\exists v.\, \textbf{newChan}_A = E[\textbf{send}\ c\ v]) \vee (\exists c.\, \textbf{newChan}_A = E[\textbf{recv}\ c]) \vee (\exists P.\, \textbf{newChan}_A \to P)$.

According to rule [C-NewChan] and picking the empty context $E = []$, $\textbf{newChan}_A \longrightarrow \nu c.c$. According to Table 1, channel restriction is a process. If we pick $P = \nu c.c$, we have found a process P such that $e \longrightarrow P$, which proves the theorem.

The cases for [T-Spawn], [T-Atomic] and [T-Commit] are similar.

∎

In order to state the final progress theorem, we introduce the predicate $PExpr : CProc \longrightarrow \mathcal{P}(CExpr)$ from closed processes to closed expressions, to collect all expressions running in a process:

Theorem 2.11 (Process progress)
If $\Gamma \vdash P : A$, *then* $\forall e \in PExpr(P)$, *e is not stuck.*

Proof
We need to prove that $\forall \Gamma, P, A., \Gamma \vdash P : A \Rightarrow \mathcal{P}(\Gamma, P, A)$, where
$\mathcal{P}(\Gamma, P, A) = \forall e \in PExpr(P)$, e is not stuck. Let us prove this theorem by induction on derivations on judgement $\Gamma \vdash P : A$.

[T-Unit] Let us assume that:

1) $\Gamma \vdash () : \mathbf{unit}$

We need to show that $PExpr(())$ is not stuck.

This case is trivial, since $PExpr(()) = \{()\}$, and () is a value.

The cases for rules [T-Int], [T-Bool], [T-Var], [T-Rec] and [T-Chan] are trivial too.

[T-Tup] Let us assume that:

1) $\Gamma \vdash (e_1, e_2) : A_1 \times A_2$
2) $\Gamma \vdash e_1 : A_1$
3) $\Gamma \vdash e_2 : A_2$
4) $\mathcal{P}(\Gamma, e_1, A_1)$
5) $\mathcal{P}(\Gamma, e_2, A_2)$

We need to show that $PExpr((e_1, e_2))$ is not stuck.

Since $PExpr((e_1, e_2)) = \{(e_1, e_2)\}$, we only need to show that (e_1, e_2) is not stuck. By applying Lemma 2.10 (Expression Progress) on hypothesis 1), we have that $(e_1, e_2) = v$ or $\exists c, v.(e_1, e_2) = E[\,\mathbf{send}\ c\ v]$ or $\exists c.(e_1, e_2) = E[\,\mathbf{recv}\ c]$ or $\exists P.(e_1, e_2) \longrightarrow P$. In the former three cases the theorem holds trivially, since in these cases (e_1, e_2) is not stuck by Definition 2.3.1 (Stuck Expression). In the last case, there are only five rules that let an expression take a $\longrightarrow$ reduction step: [C-Step], [C-Spawn], [T-Spawn], [T-Atomic] and [T-Commit].

Let us consider the case with [C-Step] first. By the hypothesis of rule [C-Step], $e \hookrightarrow e'$, and thus e is not stuck because of point iii) in Definition 2.3.1. In each of the other four cases e is not stuck because of point ii) in Definition 2.3.1.

The cases for rules [T-If], [T-App], [T-Let], [T-Op], [T-Send], [T-Recv], [T-NewChan], [T-Spawn], [T-Atomic] and [T-Commit] can be proved similarly.

[T-Par-L] Let us assume that:

1) $\Gamma \vdash P_1 \parallel P_2 : A$
2) $\Gamma \vdash P_1 : A$
3) $\Gamma \vdash P_2 : \mathbf{unit}$
4) $\mathcal{P}(\Gamma, P_1, A)$
5) $\mathcal{P}(\Gamma, P_2, \mathbf{unit})$

We need to show that $\forall e \in PExpr(P_1 \parallel P_2)$, e is not stuck.

According to Table 18, $PExpr(P_1 \parallel P_2) = PExpr(P_1) \cup PExpr(P_2)$. Since, by inductive hypotheses 4) and 5), all the expressions in both sets are not stuck, then also all the expressions in the union of both sets will not be stuck, thus proving the theorem.

The case for [T-Par-R] can be proved similarly

[T-Trans] Let us assume that:

1) $\Gamma \vdash [\![P_1 \ \rhd_k \ P_2]\!] : A$
2) $\Gamma, k \vdash P_1 : A$
3) $\Gamma \vdash P_2 : A$
4) $\mathcal{P}((\Gamma, k), P_1, A)$
5) $\mathcal{P}(\Gamma, P_2, A)$

We need to show that $\forall e \in PExpr([\![P_1 \ \rhd_k \ P_2]\!])$, e is not stuck.

By inductive hypothesis 4), all expressions in $PExpr(P_1)$ are not stuck. Since $PExpr([\![P_1 \rhd_k P_2]\!]) = PExpr(P_1)$ according to Table 18, this case is proved.

The case for [T-Restr] can be proved similarly.

[T-Co] Let us assume that:

1) $\Gamma \vdash \mathbf{co}\ k : \mathbf{unit}$

We need to show that $\forall e \in PExpr(\ \mathbf{co}\ k)$, e is not stuck.

Since $PExpr(\ \mathbf{co}\ k)$ is equivalent to the empty set, this case holds trivially.

∎

We can now state the soundeness theorem and a corollary:

Theorem 2.12 (Soundness) *If* $\Gamma \vdash e : A$, *then* $\forall P$. *if* $e \longrightarrow^* P$ *then* $\forall e' \in PExpr(P).e'$ *is not stuck.*

Proof By Theorem 2.9 (Type Preservation) and Theorem 2.11 (Progress).

Corollary 2.13 *If* $\Gamma \vdash e : A$, *then* $e \longrightarrow^* P \not\longrightarrow$ *implies that* $\forall e' \in PExp(P).e'$ *is not stuck.*

3 Language design choices

We discuss in this section the rationale for transaction rules. In particular, we will show why we allow delays in committing transactions via rule [Tr-Co], why the scope of a transaction can be extended by rule [S-Scope-Ext], why transactions need to be aborted at any time rather than only at specific points encoded in the term, and why transactions cannot be committed if their commit point is nested in another transaction.

We will consider a special channel ω, over which terms can send a unit value $()$ to signal a successful computation. We will mark such transitions with a label $\omega!()$.

3.1 Asynchronous commits

In TransCML, transactions are committed asynchronously. A transaction k can notify the system that it is ready to commit with the **commit** k expression. When this expression is evaluated, the transaction is not committed immediately; instead, **commit** k expressions are reduced to the unit value $()$, and a commit point process **co** k, that refers to transaction k, is spawned. Let's see at the following example to illustrate the reason behind this rule.

Assume that there was no rule [Tr-Co], and that we replace rule [Tr-Commit] with the following one:

$$\text{[Tr-Commit]}\quad \frac{P_1 \equiv E[\textbf{commit}\ k] \parallel P_1'}{[\![\, P_1 \ \rhd_k\ P_2 \,]\!] \to (E[()] \parallel P_1')\backslash k}$$

According to this rule, transactions commit only when a process reaches a commit expression in its functional core. Notice that this is the only rule that can reduce expression **commit** k, thus evaluation of context E cannot proceed unless the transaction is committed.

Suppose now that we had the following term P:

$$\begin{array}{c}[\![\, \textbf{send}\ c\ (); \textbf{commit}\ k; \textbf{send}\ c\ () \ \rhd_k\ () \,]\!] \parallel \\ [\![\, \textbf{recv}\ c; \textbf{recv}\ c; \textbf{commit}\ l; \textbf{send}\ \omega\ () \ \rhd_l\ () \,]\!]\end{array}$$

In this example, the first transaction commits only if it sends at least one value over channel c. The second transaction must receive two values over channel c in order to commit. Depending on the order in which transactions are embedded, P will evaluate to two different results. On the one hand, if transaction k is embedded into transaction l first, the process in transacion k will be able to communicate over channel c with the process in transaction l, and commit transaction k. Afterwards, another synchronization among the two processes will happen, and transaction l can commit. More formally, the following reductions are possible:

$$[\![\, \textbf{send}\ c\ (); \textbf{commit}\ k; \textbf{send}\ c\ () \ \rhd_k\ () \,]\!]$$

$$
\begin{aligned}
& \parallel [\![\ \textbf{recv}\ c;\ \textbf{recv}\ c; \textbf{commit}\ l;\ \textbf{send}\ \omega\ ()\ \ \rhd_l\ \ ()\,]\!] \\
\xrightarrow{[\textsc{Tr-Emb}]}\ & [\![\, [\![\ \textbf{send}\ c\ (); \textbf{commit}\ k;\ \textbf{send}\ c\ ()\ \ \rhd_k\ \ ()\,]\!] \\
& \parallel\ \textbf{recv}\ c;\ \textbf{recv}\ c; \textbf{commit}\ l;\ \textbf{send}\ \omega\ () \\
& \ \rhd_l\ \ [\![\ \textbf{send}\ c\ (); \textbf{commit}\ k;\ \textbf{send}\ c\ ()\ \ \rhd_k\ \ ()\,]\!] \parallel ()\,]\!] \\
\xrightarrow{[\textsc{Tr-Emb}]}\ & [\![\, [\![\ \textbf{send}\ c\ (); \textbf{commit}\ k;\ \textbf{send}\ c\ () \\
& \parallel\ \textbf{recv}\ c;\ \textbf{recv}\ c; \textbf{commit}\ l;\ \textbf{send}\ \omega\ () \\
& \ \rhd_k\ \ () \parallel\ \textbf{recv}\ c;\ \textbf{recv}\ c; \textbf{commit}\ l;\ \textbf{send}\ \omega\ ()\,]\!] \\
& \ \rhd_l\ \ [\![\ \textbf{send}\ c\ (); \textbf{commit}\ k;\ \textbf{send}\ c\ ()\ \ \rhd_k\ \ ()\,]\!] \parallel ()\,]\!] \\
\xrightarrow{[\textsc{Sync}]}\ & [\![\, [\![(); \textbf{commit}\ k;\ \textbf{send}\ c\ () \parallel ();\ \textbf{recv}\ c; \textbf{commit}\ l;\ \textbf{send}\ \omega\ () \\
& \ \rhd_k\ \ () \parallel\ \textbf{recv}\ c;\ \textbf{recv}\ c; \textbf{commit}\ l;\ \textbf{send}\ \omega\ ()\,]\!] \\
& \ \rhd_l\ \ [\![\ \textbf{send}\ c\ (); \textbf{commit}\ k;\ \textbf{send}\ c\ ()\ \ \rhd_k\ \ ()\,]\!] \parallel ()\,]\!] \\
\xrightarrow{[\textsc{E-Let}]}\xrightarrow{[\textsc{E-Let}]}\ & [\![\, [\![\textbf{commit}\ k;\ \textbf{send}\ c\ () \parallel\ \textbf{recv}\ c; \textbf{commit}\ l;\ \textbf{send}\ \omega\ () \\
& \ \rhd_k\ \ () \parallel\ \textbf{recv}\ c;\ \textbf{recv}\ c; \textbf{commit}\ l;\ \textbf{send}\ \omega\ ()\,]\!] \\
& \ \rhd_l\ \ [\![\ \textbf{send}\ c\ (); \textbf{commit}\ k;\ \textbf{send}\ c\ ()\ \ \rhd_k\ \ ()\,]\!] \parallel ()\,]\!] \\
\xrightarrow{[\textsc{Tr-Commit}]}\xrightarrow{[\textsc{E-Let}]}\ & [\![\ \textbf{send}\ c\ () \parallel\ \textbf{recv}\ c; \textbf{commit}\ l;\ \textbf{send}\ \omega\ () \\
& \ \rhd_l\ \ [\![\ \textbf{send}\ c\ (); \textbf{commit}\ k;\ \textbf{send}\ c\ ()\ \ \rhd_k\ \ ()\,]\!] \parallel ()\,]\!] \\
\xrightarrow{[\textsc{Sync}]}\xrightarrow{[\textsc{E-Let}]}\ & [\![() \parallel\ \textbf{commit}\ l;\ \textbf{send}\ \omega\ () \\
& \ \rhd_l\ \ [\![\ \textbf{send}\ c\ (); \textbf{commit}\ k;\ \textbf{send}\ c\ ()\ \ \rhd_k\ \ ()\,]\!] \parallel ()\,]\!] \\
\xrightarrow{[\textsc{Tr-Commit}]}\xrightarrow{[\textsc{E-Let}]}\ & () \parallel\ \textbf{send}\ \omega\ () \\
\xrightarrow{[\textsc{Send}]\ \omega!()}\ & \nu c.() \parallel ()
\end{aligned}
$$

On the other hand, if transaction l is embedded into transaction k first, a synchronization among the processes in the two transactions can happen. After the synchronization, the expression **commit** k is available for reduction. Because, as a result of applying the embedding rules, transaction l is now the innermost nested transaction, and transaction k is the outermost transaction, rule [Tr-Commit] cannot be applied to commit transaction k from expression **commit** k, because transaction l is in the middle between **commit** k and transaction k. The only option that the system is left in this case is to abort either transaction k or transaction l. We choose to abort transaction l in the following reduction, but the end result would be the same if we had chosen to abort transaction k:

$$[\![\ \textbf{send}\ c\ (); \textbf{commit}\ k;\ \textbf{send}\ c\ ()\ \ \rhd_k\ \ ()\,]\!]$$

$$
\begin{array}{rl}
& \parallel [\![\ \textbf{recv}\ c;\ \textbf{recv}\ c; \textbf{commit}\ l;\ \textbf{send}\ \omega\ ()\ \ \rhd_l\ \ ()\,]\!] \\
\xrightarrow{[\textsc{Tr-Emb}]} & [\![\ \textbf{send}\ c\ (); \textbf{commit}\ k;\ \textbf{send}\ c\ () \\
& \parallel [\![\ \textbf{recv}\ c;\ \textbf{recv}\ c; \textbf{commit}\ l;\ \textbf{send}\ \omega\ ()\ \ \rhd_l\ \ ()\,]\!] \\
& \rhd_k\ \ () \parallel [\![\ \textbf{recv}\ c;\ \textbf{recv}\ c; \textbf{commit}\ l;\ \textbf{send}\ \omega\ ()\ \ \rhd_l\ \ ()\,]\!]\,]\!] \\
\xrightarrow{[\textsc{Tr-Emb}]} & [\![\,[\![\ \textbf{send}\ c\ (); \textbf{commit}\ k;\ \textbf{send}\ c\ () \\
& \parallel\ \textbf{recv}\ c;\ \textbf{recv}\ c; \textbf{commit}\ l;\ \textbf{send}\ \omega\ () \\
& \rhd_l\ \ \textbf{send}\ c\ (); \textbf{commit}\ k;\ \textbf{send}\ c\ () \parallel ()\,]\!] \\
& \rhd_k\ \ () \parallel [\![\ \textbf{recv}\ c;\ \textbf{recv}\ c; \textbf{commit}\ l;\ \textbf{send}\ \omega\ ()\ \ \rhd_l\ \ ()\,]\!]\,]\!] \\
\xrightarrow{[\textsc{Sync}]} & [\![\,[\![(); \textbf{commit}\ k;\ \textbf{send}\ c\ () \parallel ();\ \textbf{recv}\ c; \textbf{commit}\ l;\ \textbf{send}\ \omega\ () \\
& \rhd_l\ \ \textbf{send}\ c\ (); \textbf{commit}\ k;\ \textbf{send}\ c\ () \parallel ()\,]\!] \\
& \rhd_k\ \ () \parallel [\![\ \textbf{recv}\ c;\ \textbf{recv}\ c; \textbf{commit}\ l;\ \textbf{send}\ \omega\ ()\ \ \rhd_l\ \ ()\,]\!]\,]\!] \\
\xrightarrow{[\textsc{E-Let}]}\xrightarrow{[\textsc{E-Let}]} & [\![\,[\![\,\textbf{commit}\ k;\ \textbf{send}\ c\ () \parallel\ \textbf{recv}\ c; \textbf{commit}\ l;\ \textbf{send}\ \omega\ () \\
& \rhd_l\ \ \textbf{send}\ c\ (); \textbf{commit}\ k;\ \textbf{send}\ c\ () \parallel ()\,]\!] \\
& \rhd_k\ \ () \parallel [\![\ \textbf{recv}\ c;\ \textbf{recv}\ c; \textbf{commit}\ l;\ \textbf{send}\ \omega\ ()\ \ \rhd_l\ \ ()\,]\!]\,]\!] \\
\xrightarrow{[\textsc{Tr-Abort}]} & [\![\ \textbf{send}\ c\ (); \textbf{commit}\ k;\ \textbf{send}\ c\ () \parallel () \\
& \rhd_k\ \ () \parallel [\![\ \textbf{recv}\ c;\ \textbf{recv}\ c; \textbf{commit}\ l;\ \textbf{send}\ \omega\ ()\ \ \rhd_l\ \ ()\,]\!]\,]\!] \\
\xrightarrow{[\textsc{Tr-Abort}]} & () \parallel [\![\ \textbf{recv}\ c;\ \textbf{recv}\ c; \textbf{commit}\ l;\ \textbf{send}\ \omega\ ()\ \ \rhd_l\ \ ()\,]\!] \\
\xrightarrow{[\textsc{Tr-Abort}]} & () \parallel ()
\end{array}
$$

After the application of the last [E-Let] rule, transaction l must abort. In fact, transaction l contains **commit** k, which cannot reach the outer transaction k and commit it. There is no other reduction step to perform other than aborting l or k at this point. Even after aborting transaction l, there is no other option than aborting the other transactions. Embedding on the right first will force us to abort all transactions, no matter what we do. Moreover, we will not be able to communicate success on channel ω.

The order in which transactions are embedded has become significant. On one hand, embedding on the right first forces us to abort all transactions. On the other hand, embedding on the left first allows us to commit all transactions and signal success on channel ω. We want to avoid the former case: we want to be assured that successful executions can always be reached, no matter what the order of embeddings is.

Introducing rule [Tr-Co], we modify rule [Tr-Commit] to just spawn a commit point, and let rule [Tr-Co] do the actual commit when it is possible. In this way, commits are delayed and interdependencies between transactions are not as stringent as before, since that the different order of embeddings yield the same result now. Let us resume reduction after the application of the last [E-Let] rule. We will use the new rule [Tr-Co] first to spawn a commit point for transaction k, and resume reduction so that the second synchronization between the two processes can continue. After the second synchronization happens, it is possible to spawn a second commit point for transaction l is spawned, and now both transactions can be committed. The following reduction is thus possible:

$$
\begin{aligned}
& \dots \\
\xrightarrow{[\textsc{E-Let}]}\ & \llbracket\, \llbracket\, \textbf{commit}\ k; \textbf{send}\ c\ () \parallel \textbf{recv}\ c; \textbf{commit}\ l; \textbf{send}\ \omega\ () \\
& \triangleright_l\ \textbf{send}\ c\ (); \textbf{commit}\ k; \textbf{send}\ c\ () \parallel () \,\rrbracket \\
& \triangleright_k\ () \parallel \llbracket\, \textbf{recv}\ c; \textbf{recv}\ c; \textbf{commit}\ l; \textbf{send}\ \omega\ ()\ \triangleright_l\ () \,\rrbracket \,\rrbracket \\
\xrightarrow{[\textsc{Tr-Co}]}\xrightarrow{[\textsc{E-Let}]}\ & \llbracket\, \llbracket\, \textbf{co}\ k \parallel \textbf{send}\ c\ () \parallel \textbf{recv}\ c; \textbf{commit}\ l; \textbf{send}\ \omega\ () \\
& \triangleright_l\ \textbf{send}\ c\ (); \textbf{commit}\ k; \textbf{send}\ c\ () \parallel () \,\rrbracket \\
& \triangleright_k\ () \parallel \llbracket\, \textbf{recv}\ c; \textbf{recv}\ c; \textbf{commit}\ l; \textbf{send}\ \omega\ ()\ \triangleright_l\ () \,\rrbracket \,\rrbracket \\
\xrightarrow{[\textsc{Sync}]}\xrightarrow{[\textsc{E-Let}]}\ & \llbracket\, \llbracket\, \textbf{co}\ k \parallel () \parallel \textbf{commit}\ l; \textbf{send}\ \omega\ () \\
& \triangleright_l\ \textbf{send}\ c\ (); \textbf{commit}\ k; \textbf{send}\ c\ () \parallel () \,\rrbracket \\
& \triangleright_k\ () \parallel \llbracket\, \textbf{recv}\ c; \textbf{recv}\ c; \textbf{commit}\ l; \textbf{send}\ \omega\ ()\ \triangleright_l\ () \,\rrbracket \,\rrbracket \\
\xrightarrow{[\textsc{Tr-Co}]}\ & \llbracket\, \llbracket\, \textbf{co}\ k \parallel () \parallel \textbf{co}\ l \parallel (); \textbf{send}\ \omega\ () \\
& \triangleright_l\ \textbf{send}\ c\ (); \textbf{commit}\ k; \textbf{send}\ c\ () \parallel () \,\rrbracket \\
& \triangleright_k\ () \parallel \llbracket\, \textbf{recv}\ c; \textbf{recv}\ c; \textbf{commit}\ l; \textbf{send}\ \omega\ ()\ \triangleright_l\ () \,\rrbracket \,\rrbracket \\
\xrightarrow{[\textsc{Tr-Commit}]}\ & \llbracket\, \textbf{co}\ k \parallel () \parallel () \parallel (); \textbf{send}\ \omega\ () \\
& \triangleright_k\ () \parallel \llbracket\, \textbf{recv}\ c; \textbf{recv}\ c; \textbf{commit}\ l; \textbf{send}\ \omega\ ()\ \triangleright_l\ () \,\rrbracket \,\rrbracket \\
\xrightarrow{[\textsc{Tr-Commit}]}\xrightarrow{[\textsc{E-Let}]}\ & () \parallel () \parallel () \parallel \textbf{send}\ \omega\ () \\
\xrightarrow{[\textsc{Send}]\ \omega!()}\ & () \parallel () \parallel () \parallel ()
\end{aligned}
$$

A final note worth considering is that it is indeed still possible for transactions to abort at any time and not reach a successful computation. However, a successful computation can be reached no matter what the order of embeddings was, which was not the case before introducing rule [Tr-Co]. We have thus showed why it is not a good idea to force transactions to commit as soon as a commit point is reached.

3.2 Transaction scope extension

The following example justifies the rules to extend the scope of a transaction over expressions in rule [Tr-Atomic]. The discussion in this section is reminiscent of Jeffrey's μCML syntax in [11].

Suppose that we made no distinction between processes and expressions, but that everything was an expression. Then we could add parallels and restrictions as evaluation contexts. Since transactions might now appear in the middle of an expression, we would have no need for **atomic** expressions.

We also need to ensure that reduction of functional expressions is consistent, if its argument is a composition of parallel processes. One way to achieve this is to first restrict parallel processes to be commutative only on the right:

[Eq-Com]

$$\frac{}{(P_1 \parallel P_2) \parallel P_3 \equiv (P_2 \parallel P_1) \parallel P_3}$$

This rule will ensure that the thread on the rightmost position in a parallel composition never changes position. Secondly, we need another structural equivalence rule to move parallel expression from an evaluation context and allow a sequential reduction step:

[Eq-Ctx]

$$\frac{}{E[P_1 \parallel P_2] \equiv P_1 \parallel E[P_2]}$$

Moreover, since the rightmost process cannot be moved, we would need to duplicate a number of rules. For example, transactions could be either on the right hand side or at the left hand side of a parallel expression. If we wanted to perform an embedding step, rule [Tr-Emb] would only allow expressions to be embedded into a transaction if this was on their right-hand side in a parallel composition. The rightmost process of a parallel composition would not be able to be embedded into a transaction on its left. We thus need to duplicate rules to cover both cases, left-hand side and right-hand side. For example:

[Tr-Emb-L]

$$\frac{}{E[\llbracket\, e_1 \;\rhd_k\; e_2 \,\rrbracket] \parallel e_3 \longrightarrow E[\llbracket\, e_1 \parallel e_3 \;\rhd_k\; e_2 \parallel e_3 \,\rrbracket]}$$

We will not described rule [Tr-Emb-R] and other rules that need to be duplicated for brevity's sake.

Suppose now that we had the following term P:

$$\begin{array}{l} \llbracket\, \textbf{recv}\ c; \textbf{recv}\ c; \textbf{commit}\ k \;\rhd_k\; () \,\rrbracket \parallel \\ \textbf{let}\ z = \llbracket\, \textbf{send}\ c\ (); \textbf{commit}\ l \;\rhd_l\; () \,\rrbracket\ \textbf{in}\ \textbf{send}\ c\ (); \textbf{send}\ \omega\ () \end{array}$$

In this example, we are using channel ω in expression **send** ω () to signal success of a computation; when success is reached, we will mark the relative transition with label $\omega!()$. If the scope of transactions were not extensible, we would get different traces depending on the order of embeddings performed. After the first two embeddings, there will be two expressions running in parallel inside the argument of the **let** expression. A synchronization is now available between the two processes. After synchronizing, the innermost transaction l can be committed by rule [Tr-Co]. After committing, we use structural equivalence on the let expression to move the parallel processes into the **let** expression outside of it, so that rule [E-Let] can be applied to the **let** expression. After this rule is applied, another synchronization is available, after which the remaining transaction k can be committed, and success can be signaled. The term reduction we just described is the following:

$$\begin{aligned}
& \llbracket\, \mathbf{recv}\ c;\ \mathbf{recv}\ c; \mathbf{commit}\ k\ \triangleright_k\ ()\,\rrbracket \parallel \\
& \mathbf{let}\ z = \llbracket\, \mathbf{send}\ c\ (); \mathbf{commit}\ l\ \triangleright_l\ ()\,\rrbracket\ \mathbf{in}\ \mathbf{send}\ c\ ();\ \mathbf{send}\ \omega\ () \\
\xrightarrow{[\textsc{Tr-Emb-L}]}\ & \llbracket\, \mathbf{recv}\ c;\ \mathbf{recv}\ c; \mathbf{commit}\ k \parallel \\
& \mathbf{let}\ z = \llbracket\, \mathbf{send}\ c\ (); \mathbf{commit}\ l\ \triangleright_l\ ()\,\rrbracket\ \mathbf{in}\ \mathbf{send}\ c\ ();\ \mathbf{send}\ \omega\ () \\
& \ \triangleright_k\ () \parallel \mathbf{let}\ z = \llbracket\, \mathbf{send}\ c\ (); \mathbf{commit}\ l\ \triangleright_l\ ()\,\rrbracket \\
& \ \mathbf{in}\ \mathbf{send}\ c\ ();\ \mathbf{send}\ \omega\ ()\,\rrbracket \\
\equiv\ & \llbracket\, \mathbf{let}\ z = (\,\mathbf{recv}\ c;\ \mathbf{recv}\ c; \mathbf{commit}\ k \parallel \\
& \llbracket\, \mathbf{send}\ c\ (); \mathbf{commit}\ l\ \triangleright_l\ ()\,\rrbracket)\ \mathbf{in}\ \mathbf{send}\ c\ ();\ \mathbf{send}\ \omega\ () \\
& \ \triangleright_k\ () \parallel \mathbf{let}\ z = \llbracket\, \mathbf{send}\ c\ (); \mathbf{commit}\ l\ \triangleright_l\ ()\,\rrbracket \\
& \ \mathbf{in}\ \mathbf{send}\ c\ ();\ \mathbf{send}\ \omega\ ()\,\rrbracket \\
\xrightarrow{[\textsc{Tr-Emb-R}]}\ & \llbracket\, \mathbf{let}\ z = \llbracket\, \mathbf{recv}\ c;\ \mathbf{recv}\ c; \mathbf{commit}\ k \parallel \mathbf{send}\ c\ (); \mathbf{commit}\ l \\
& \ \triangleright_l\ \ \mathbf{recv}\ c;\ \mathbf{recv}\ c; \mathbf{commit}\ k \parallel ()\,\rrbracket\ \mathbf{in}\ \mathbf{send}\ c\ ();\ \mathbf{send}\ \omega\ () \\
& \ \triangleright_k\ () \parallel \mathbf{let}\ z = \llbracket\, \mathbf{send}\ c\ (); \mathbf{commit}\ l\ \triangleright_l\ ()\,\rrbracket \\
& \ \mathbf{in}\ \mathbf{send}\ c\ ();\ \mathbf{send}\ \omega\ ()\,\rrbracket \\
\xrightarrow{[\textsc{Sync}]}\ & \llbracket\, \mathbf{let}\ z = \llbracket();\ \mathbf{recv}\ c; \mathbf{commit}\ k \parallel (); \mathbf{commit}\ l \\
& \ \triangleright_l\ \ \mathbf{recv}\ c;\ \mathbf{recv}\ c; \mathbf{commit}\ k \parallel ()\,\rrbracket\ \mathbf{in}\ \mathbf{send}\ c\ ();\ \mathbf{send}\ \omega\ () \\
& \ \triangleright_k\ () \parallel \mathbf{let}\ z = \llbracket\, \mathbf{send}\ c\ (); \mathbf{commit}\ l\ \triangleright_l\ ()\,\rrbracket \\
& \ \mathbf{in}\ \mathbf{send}\ c\ ();\ \mathbf{send}\ \omega\ ()\,\rrbracket \\
\xrightarrow{[\textsc{E-Let}]}\xrightarrow{[\textsc{E-Let}]}\ & \llbracket\, \mathbf{let}\ z = \llbracket\, \mathbf{recv}\ c; \mathbf{commit}\ k \parallel \mathbf{commit}\ l \\
& \ \triangleright_l\ \ \mathbf{recv}\ c;\ \mathbf{recv}\ c; \mathbf{commit}\ k \parallel ()\,\rrbracket\ \mathbf{in}\ \mathbf{send}\ c\ ();\ \mathbf{send}\ \omega\ () \\
& \ \triangleright_k\ () \parallel \mathbf{let}\ z = \llbracket\, \mathbf{send}\ c\ (); \mathbf{commit}\ l\ \triangleright_l\ ()\,\rrbracket \\
& \ \mathbf{in}\ \mathbf{send}\ c\ ();\ \mathbf{send}\ \omega\ ()\,\rrbracket \\
\xrightarrow{[\textsc{Tr-Commit}]}\ & \llbracket\, \mathbf{let}\ z = \llbracket\, \mathbf{recv}\ c; \mathbf{commit}\ k \parallel\ \mathbf{co}\ l \parallel () \\
& \ \triangleright_l\ \ \mathbf{recv}\ c;\ \mathbf{recv}\ c; \mathbf{commit}\ k \parallel ()\,\rrbracket\ \mathbf{in}\ \mathbf{send}\ c\ ();\ \mathbf{send}\ \omega\ () \\
& \ \triangleright_k\ () \parallel \mathbf{let}\ z = \llbracket\, \mathbf{send}\ c\ (); \mathbf{commit}\ l\ \triangleright_l\ ()\,\rrbracket \\
& \ \mathbf{in}\ \mathbf{send}\ c\ ();\ \mathbf{send}\ \omega\ ()\,\rrbracket
\end{aligned}$$

$$
\begin{aligned}
&\xrightarrow{[\textsc{Tr-Co}]} [\![\, \textbf{let } z = (\, \textbf{recv } c; \textbf{commit } k \parallel () \parallel ())\ \textbf{in send } c\ (); \textbf{send } \omega\ () \\
&\quad \rhd_k\ () \parallel \textbf{let } z = [\![\ \textbf{send } c\ (); \textbf{commit } l\ \rhd_l\ ()\,]\!] \\
&\quad \textbf{in send } c\ (); \textbf{send } \omega\ ()\,]\!] \\
&\equiv [\![\ \textbf{recv } c; \textbf{commit } k \parallel () \parallel \textbf{let } z = ()\ \textbf{in send } c\ (); \textbf{send } \omega\ () \\
&\quad \rhd_k\ () \parallel \textbf{let } z = [\![\ \textbf{send } c\ (); \textbf{commit } l\ \rhd_l\ ()\,]\!] \\
&\quad \textbf{in send } c\ (); \textbf{send } \omega\ ()\,]\!] \\
&\xrightarrow{[\textsc{E-Let}]} [\![\ \textbf{recv } c; \textbf{commit } k \parallel () \parallel\ \textbf{send } c\ (); \textbf{send } \omega\ () \\
&\quad \rhd_k\ () \parallel \textbf{let } z = [\![\ \textbf{send } c\ (); \textbf{commit } l\ \rhd_l\ ()\,]\!] \\
&\quad \textbf{in send } c\ (); \textbf{send } \omega\ ()\,]\!] \\
&\xrightarrow{[\textsc{Sync}]} [\![(); \textbf{commit } k \parallel () \parallel (); \textbf{send } \omega\ () \\
&\quad \rhd_k\ () \parallel \textbf{let } z = [\![\ \textbf{send } c\ (); \textbf{commit } l\ \rhd_l\ ()\,]\!] \\
&\quad \textbf{in send } c\ (); \textbf{send } \omega\ ()\,]\!] \\
&\xrightarrow{[\textsc{E-Let}]}\xrightarrow{[\textsc{E-Let}]} [\![\, \textbf{commit } k \parallel () \parallel\ \textbf{send } \omega\ () \\
&\quad \rhd_k\ () \parallel \textbf{let } z = [\![\ \textbf{send } c\ (); \textbf{commit } l\ \rhd_l\ ()\,]\!] \\
&\quad \textbf{in send } c\ (); \textbf{send } \omega\ ()\,]\!] \\
&\xrightarrow{[\textsc{Tr-Commit}]} [\![\ \textbf{co } k \parallel () \parallel () \parallel\ \textbf{send } \omega\ () \\
&\quad \rhd_k\ () \parallel \textbf{let } z = [\![\ \textbf{send } c\ (); \textbf{commit } l\ \rhd_l\ ()\,]\!] \\
&\quad \textbf{in send } c\ (); \textbf{send } \omega\ ()\,]\!] \\
&\xrightarrow{[\textsc{Tr-Co}]} () \parallel () \parallel () \parallel\ \textbf{send } \omega\ () \\
&\xrightarrow{[\textsc{Send}]\ \ \omega!()} () \parallel () \parallel () \parallel ()
\end{aligned}
$$

By the structural equivalence rule [Eq-Ctx], term P is equivalent to the following term:

$$
\begin{aligned}
& [\![\ \textbf{recv } c; \textbf{recv } c; \textbf{commit } k\ \rhd_k\ ()\,]\!] \parallel \\
& \textbf{let } z = [\![\ \textbf{send } c\ (); \textbf{commit } l\ \rhd_l\ ()\,]\!]\ \textbf{in send } c\ (); \textbf{send } \omega\ () \\
\equiv\quad & \textbf{let } z = ([\![\ \textbf{recv } c; \textbf{recv } c; \textbf{commit } k\ \rhd_k\ ()\,]\!] \parallel \\
& [\![\ \textbf{send } c\ (); \textbf{commit } l\ \rhd_l\ ()\,]\!])\ \textbf{in send } c\ (); \textbf{send } \omega\ ()
\end{aligned}
$$

Unfortunately, we will now show that no reduction of term Q can execute succesfully. Transaction l can embed into transaction k first, or viceversa, but neither embedding will be able to commit. If we embed transaction l into transaction k first, the two processes inside transaction l can perform one synchronization and spawn a commit point $\mathbf{co}\ l$. Transaction l can be committed by rule [Tr-Co], but transaction k cannot be committed, because commit point k can be spawned only if a process is willing to perform $\mathbf{send}\ c\ ()$. Unfortunately, the system has the form $\mathbf{let}\ z = \llbracket \ldots \rhd_k \ldots \rrbracket\ \mathbf{in}\ \mathbf{send}\ c\ ()$, and the expression $\mathbf{send}\ c\ ()$ cannot be reached from transaction k. Thus transaction k can only be aborted.

The reduction we just described is the following:

$$\begin{aligned}
& \mathbf{let}\ z = (\llbracket\ \mathbf{recv}\ c;\ \mathbf{recv}\ c; \mathbf{commit}\ k\ \rhd_k\ ()\ \rrbracket \parallel \\
& \llbracket\ \mathbf{send}\ c\ (); \mathbf{commit}\ l\ \rhd_l\ ()\ \rrbracket)\ \mathbf{in}\ \mathbf{send}\ c\ ();\ \mathbf{send}\ \omega\ () \\
\xrightarrow{[\textsc{Tr-Emb-L}]}\ & \mathbf{let}\ z = \llbracket\ \mathbf{recv}\ c;\ \mathbf{recv}\ c; \mathbf{commit}\ k \parallel \\
& \llbracket\ \mathbf{send}\ c\ (); \mathbf{commit}\ l\ \rhd_l\ ()\ \rrbracket \\
& \rhd_k\ () \parallel \llbracket\ \mathbf{send}\ c\ (); \mathbf{commit}\ l\ \rhd_l\ ()\ \rrbracket\ \rrbracket \\
& \mathbf{in}\ \mathbf{send}\ c\ ();\ \mathbf{send}\ \omega\ () \\
\xrightarrow{[\textsc{Tr-Emb-R}]}\ & \mathbf{let}\ z = \llbracket\ \llbracket\ \mathbf{recv}\ c;\ \mathbf{recv}\ c; \mathbf{commit}\ k \parallel \mathbf{send}\ c\ (); \mathbf{commit}\ l \\
& \rhd_l\ \mathbf{recv}\ c;\ \mathbf{recv}\ c; \mathbf{commit}\ k \parallel ()\ \rrbracket \\
& \rhd_k\ () \parallel \llbracket\ \mathbf{send}\ c\ (); \mathbf{commit}\ l\ \rhd_l\ ()\ \rrbracket\ \rrbracket \\
& \mathbf{in}\ \mathbf{send}\ c\ ();\ \mathbf{send}\ \omega\ () \\
\xrightarrow{[\textsc{Sync}]}\ & \mathbf{let}\ z = \llbracket\ \llbracket ();\ \mathbf{recv}\ c; \mathbf{commit}\ k \parallel (); \mathbf{commit}\ l \\
& \rhd_l\ \mathbf{recv}\ c;\ \mathbf{recv}\ c; \mathbf{commit}\ k \parallel ()\ \rrbracket \\
& \rhd_k\ () \parallel \llbracket\ \mathbf{send}\ c\ (); \mathbf{commit}\ l\ \rhd_l\ ()\ \rrbracket\ \rrbracket \\
& \mathbf{in}\ \mathbf{send}\ c\ ();\ \mathbf{send}\ \omega\ () \\
\xrightarrow{[\textsc{E-Let}]}\xrightarrow{[\textsc{E-Let}]}\ & \mathbf{let}\ z = \llbracket\ \llbracket\ \mathbf{recv}\ c; \mathbf{commit}\ k \parallel \mathbf{commit}\ l \\
& \rhd_l\ \mathbf{recv}\ c;\ \mathbf{recv}\ c; \mathbf{commit}\ k \parallel ()\ \rrbracket \\
& \rhd_k\ () \parallel \llbracket\ \mathbf{send}\ c\ (); \mathbf{commit}\ l\ \rhd_l\ ()\ \rrbracket\ \rrbracket \\
& \mathbf{in}\ \mathbf{send}\ c\ ();\ \mathbf{send}\ \omega\ () \\
\xrightarrow{[\textsc{Tr-Commit}]}\ & \mathbf{let}\ z = \llbracket\ \llbracket\ \mathbf{recv}\ c; \mathbf{commit}\ k \parallel \mathbf{co}\ l \parallel () \\
& \rhd_l\ \mathbf{recv}\ c;\ \mathbf{recv}\ c; \mathbf{commit}\ k \parallel ()\ \rrbracket \\
& \rhd_k\ () \parallel \llbracket\ \mathbf{send}\ c\ (); \mathbf{commit}\ l\ \rhd_l\ ()\ \rrbracket\ \rrbracket \\
& \mathbf{in}\ \mathbf{send}\ c\ ();\ \mathbf{send}\ \omega\ () \\
\xrightarrow{[\textsc{Tr-Co}]}\ & \mathbf{let}\ z = \llbracket\ \mathbf{recv}\ c; \mathbf{commit}\ k \parallel () \\
& \rhd_k\ () \parallel \llbracket\ \mathbf{send}\ c\ (); \mathbf{commit}\ l\ \rhd_l\ ()\ \rrbracket\ \rrbracket \\
& \mathbf{in}\ \mathbf{send}\ c\ ();\ \mathbf{send}\ \omega\ ()
\end{aligned}$$

$$\xrightarrow{[\textsc{Tr-Abort}]} \textbf{let } z = () \parallel [\![\textbf{ send } c\ (); \textbf{commit } l \ \rhd_l \ ()\,]\!] $$
$$\textbf{in send } c\ (); \textbf{ send } \omega\ ()$$

$$\xrightarrow{[\textsc{Tr-Abort}]} \textbf{let } z = () \parallel () \textbf{ in send } c\ (); \textbf{ send } \omega\ ()$$

$$\xrightarrow{[\textsc{E-Let}]} () \parallel \textbf{ send } c\ (); \textbf{ send } \omega\ ()$$

If we embed transaction k into transaction l first, the two processes inside transaction k can synchronize and a commit point **co** l can be spawned. Unfortunately, commit point l cannot commit transaction k, and a commit point k can only be spawned if a process is willing to perform a **send** c () operation. Again, the only process that can perform that operation is outside transaction k, since the system has again the form **let** $z = [\![\ldots \ \rhd_l \ \ldots]\!]$ **in send** c (); . . .. The only options are to abort either transaction k or l; in the following reduction we abort transaction k first, but the end result would not change in the other case. After a few more necessary transaction aborts, the system reaches a deadlocked state, where it cannot signal success.

If we embed whereas in the latter case:

$$\textbf{let } z = ([\![\textbf{ recv } c; \textbf{ recv } c; \textbf{commit } k \ \rhd_k \ ()\,]\!] \parallel$$
$$[\![\textbf{ send } c\ (); \textbf{commit } l \ \rhd_l \ ()\,]\!]) \textbf{ in send } c\ (); \textbf{ send } \omega\ ()$$

$$\xrightarrow{[\textsc{Tr-Emb-R}]} \textbf{let } z = [\![\, [\![\textbf{ recv } c; \textbf{ recv } c; \textbf{commit } k \ \rhd_k \ ()\,]\!]$$
$$\parallel \textbf{ send } c\ (); \textbf{commit } l$$
$$\rhd_l \ [\![\textbf{ recv } c; \textbf{ recv } c; \textbf{commit } k \ \rhd_k \ ()\,]\!] \parallel ()\,]\!]$$
$$\textbf{in send } c\ (); \textbf{ send } \omega\ ()$$

$$\xrightarrow{[\textsc{Tr-Emb-L}]} \textbf{let } z = [\![\, [\![\textbf{ recv } c; \textbf{ recv } c; \textbf{commit } k \parallel \textbf{ send } c\ (); \textbf{commit } l$$
$$\rhd_k \ () \parallel \textbf{ send } c\ (); \textbf{commit } l\,]\!]$$
$$\rhd_l \ [\![\textbf{ recv } c; \textbf{ recv } c; \textbf{commit } k \ \rhd_k \ ()\,]\!] \parallel ()\,]\!]$$
$$\textbf{in send } c\ (); \textbf{ send } \omega\ ()$$

$$\xrightarrow{[\textsc{Sync}]} \textbf{let } z = [\![\, [\![(); \textbf{ recv } c; \textbf{commit } k \parallel (); \textbf{commit } l$$
$$\rhd_k \ () \parallel \textbf{ send } c\ (); \textbf{commit } l\,]\!]$$
$$\rhd_l \ [\![\textbf{ recv } c; \textbf{ recv } c; \textbf{commit } k \ \rhd_k \ ()\,]\!] \parallel ()\,]\!]$$
$$\textbf{in send } c\ (); \textbf{ send } \omega\ ()$$

$$\xrightarrow{[\textsc{E-Let}]} \xrightarrow{[\textsc{E-Let}]} \textbf{let } z = [\![\, [\![\textbf{ recv } c; \textbf{commit } k \parallel \textbf{commit } l$$
$$\rhd_k \ () \parallel \textbf{ send } c\ (); \textbf{commit } l\,]\!]$$
$$\rhd_l \ [\![\textbf{ recv } c; \textbf{ recv } c; \textbf{commit } k \ \rhd_k \ ()\,]\!] \parallel ()\,]\!]$$
$$\textbf{in send } c\ (); \textbf{ send } \omega\ ()$$

$$
\begin{aligned}
&\xrightarrow{[\textsc{Tr-Commit}]} \textbf{let } z = [\![\, [\![\, \textbf{recv } c; \textbf{commit } k \parallel \textbf{co } l \parallel () \\
&\qquad \rhd_k \; () \parallel \textbf{send } c\ (); \textbf{commit } l \,]\!] \\
&\qquad \rhd_l \; [\![\, \textbf{recv } c; \textbf{recv } c; \textbf{commit } k \; \rhd_k \; () \,]\!] \parallel () \,]\!] \\
&\qquad \textbf{in send } c\ (); \textbf{send } \omega\ () \\
&\xrightarrow{[\textsc{Tr-Abort}]} \textbf{let } z = [\![() \parallel \textbf{send } c\ (); \textbf{commit } l \\
&\qquad \rhd_l \; [\![\, \textbf{recv } c; \textbf{recv } c; \textbf{commit } k \; \rhd_k \; () \,]\!] \parallel () \,]\!] \\
&\qquad \textbf{in send } c\ (); \textbf{send } \omega\ () \\
&\xrightarrow{[\textsc{Tr-Abort}]} \textbf{let } z = [\![\, \textbf{recv } c; \textbf{recv } c; \textbf{commit } k \; \rhd_k \; () \,]\!] \parallel () \\
&\qquad \textbf{in send } c\ (); \textbf{send } \omega\ () \\
&\xrightarrow{[\textsc{Tr-Abort}]} \textbf{let } z = () \parallel () \textbf{ in send } c\ (); \textbf{send } \omega\ () \\
&\xrightarrow{[\textsc{E-Let}]} () \parallel \textbf{send } c\ (); \textbf{send } \omega\ ()
\end{aligned}
$$

We would expect equivalent terms to have the same behaviour. Unlike term P though, Q can never signal success on ω. If we allow the scope of a transaction to be extended, Q can signal success. Such a rule could be:

$$
\textsc{[Tr-Scope-Ext]} \quad \frac{}{E[\textbf{atomic } [\![\, e_1 \rhd_k e_2 \,]\!]] \longrightarrow \textbf{atomic } [\![\, E[e_1] \rhd_k E[e_2] \,]\!]}
$$

Let us consider the case where transaction k was embedded into transaction l first. If we resume reduction just after rule [Tr-Commit] has been applied, it is now possible to apply rule [Tr-Scope-Ext] to reduce the system from expression $\textbf{let } z = [\![\ldots \; \rhd_k \; \ldots]\!] \textbf{ in send } c\ ()$ to expression $[\![\, \textbf{let } z = \ldots \textbf{ in send } c\ (); \ldots \; \rhd_k \; \textbf{let } z = \ldots \textbf{ in send } c\ (); \ldots]\!]$, where the **let** expression inside transaction k (and l) can now be reduced by rule [E-Let] and a new synchronization over channel c is made available. After the synchronization happens, a new commit point for transaction k can be spawned, and both transactions can be committed. The reduction we have just described is the following:

$$
\begin{aligned}
&\ldots \\
&\xrightarrow{[\textsc{Tr-Commit}]} \textbf{let } z = [\![\, [\![\, \textbf{recv } c; \textbf{commit } k \parallel \textbf{co } l \parallel () \\
&\qquad \rhd_k \; () \parallel \textbf{send } c\ (); \textbf{commit } l \,]\!] \\
&\qquad \rhd_l \; [\![\, \textbf{recv } c; \textbf{recv } c; \textbf{commit } k \; \rhd_k \; () \,]\!] \parallel () \,]\!] \\
&\qquad \textbf{in send } c\ (); \textbf{send } \omega\ () \\
&\xrightarrow{[\textsc{Tr-Scope-Ext}]} [\![\, \textbf{let } z = [\![\, \textbf{recv } c; \textbf{commit } k \parallel \textbf{co } l \parallel ()
\end{aligned}
$$

$$
\begin{aligned}
&\quad \triangleright_k \ () \parallel \mathbf{send}\ c\ (); \mathbf{commit}\ l \rrbracket\ \mathbf{in}\ \mathbf{send}\ c\ (); \mathbf{send}\ \omega\ () \\
&\quad \triangleright_l \ \mathbf{let}\ z = \llbracket\ \mathbf{recv}\ c;\ \mathbf{recv}\ c; \mathbf{commit}\ k\ \ \triangleright_k\ \ () \rrbracket \parallel () \\
&\quad \mathbf{in}\ \mathbf{send}\ c\ (); \mathbf{send}\ \omega\ () \rrbracket \\
\xrightarrow{[\text{Tr-Scope-Ext}]}\ & \llbracket \llbracket\ \mathbf{recv}\ c; \mathbf{commit}\ k \parallel \mathbf{co}\ l \parallel \\
& \mathbf{let}\ z = ()\ \mathbf{in}\ \mathbf{send}\ c\ (); \mathbf{send}\ \omega\ () \\
&\quad \triangleright_k \ () \parallel \mathbf{let}\ z = \mathbf{send}\ c\ (); \mathbf{commit}\ l\ \mathbf{in}\ \mathbf{send}\ c\ (); \mathbf{send}\ \omega\ () \rrbracket \\
&\quad \triangleright_l \ \mathbf{let}\ z = \llbracket\ \mathbf{recv}\ c;\ \mathbf{recv}\ c; \mathbf{commit}\ k\ \ \triangleright_k\ \ () \rrbracket \parallel () \\
&\quad \mathbf{in}\ \mathbf{send}\ c\ (); \mathbf{send}\ \omega\ () \rrbracket \\
\xrightarrow{[\text{E-Let}]}\ & \llbracket \llbracket\ \mathbf{recv}\ c; \mathbf{commit}\ k \parallel \mathbf{co}\ l \parallel \mathbf{send}\ c\ (); \mathbf{send}\ \omega\ () \\
&\quad \triangleright_k \ () \parallel \mathbf{let}\ z = \mathbf{send}\ c\ (); \mathbf{commit}\ l\ \mathbf{in}\ \mathbf{send}\ c\ (); \mathbf{send}\ \omega\ () \rrbracket \\
&\quad \triangleright_l \ \mathbf{let}\ z = \llbracket\ \mathbf{recv}\ c;\ \mathbf{recv}\ c; \mathbf{commit}\ k\ \ \triangleright_k\ \ () \rrbracket \parallel () \\
&\quad \mathbf{in}\ \mathbf{send}\ c\ (); \mathbf{send}\ \omega\ () \rrbracket \\
\xrightarrow{[\text{Sync}]}\ & \llbracket \llbracket (); \mathbf{commit}\ k \parallel \mathbf{co}\ l \parallel (); \mathbf{send}\ \omega\ () \\
&\quad \triangleright_k \ () \parallel \mathbf{let}\ z = \mathbf{send}\ c\ (); \mathbf{commit}\ l\ \mathbf{in}\ \mathbf{send}\ c\ (); \mathbf{send}\ \omega\ () \rrbracket \\
&\quad \triangleright_l \ \mathbf{let}\ z = \llbracket\ \mathbf{recv}\ c;\ \mathbf{recv}\ c; \mathbf{commit}\ k\ \ \triangleright_k\ \ () \rrbracket \parallel () \\
&\quad \mathbf{in}\ \mathbf{send}\ c\ (); \mathbf{send}\ \omega\ () \rrbracket \\
\xrightarrow{[\text{E-Let}]}\xrightarrow{[\text{E-Let}]}\ & \llbracket \llbracket \mathbf{commit}\ k \parallel \mathbf{co}\ l \parallel \mathbf{send}\ \omega\ () \\
&\quad \triangleright_k \ () \parallel \mathbf{let}\ z = \mathbf{send}\ c\ (); \mathbf{commit}\ l\ \mathbf{in}\ \mathbf{send}\ c\ (); \mathbf{send}\ \omega\ () \rrbracket \\
&\quad \triangleright_l \ \mathbf{let}\ z = \llbracket\ \mathbf{recv}\ c;\ \mathbf{recv}\ c; \mathbf{commit}\ k\ \ \triangleright_k\ \ () \rrbracket \parallel () \\
&\quad \mathbf{in}\ \mathbf{send}\ c\ (); \mathbf{send}\ \omega\ () \rrbracket \\
\xrightarrow{[\text{Tr-Commit}]}\ & \llbracket \llbracket\ \mathbf{co}\ k \parallel \mathbf{co}\ l \parallel \mathbf{send}\ \omega\ () \\
&\quad \triangleright_k \ () \parallel \mathbf{let}\ z = \mathbf{send}\ c\ (); \mathbf{commit}\ l\ \mathbf{in}\ \mathbf{send}\ c\ (); \mathbf{send}\ \omega\ () \rrbracket \\
&\quad \triangleright_l \ \mathbf{let}\ z = \llbracket\ \mathbf{recv}\ c;\ \mathbf{recv}\ c; \mathbf{commit}\ k\ \ \triangleright_k\ \ () \rrbracket \parallel () \\
&\quad \mathbf{in}\ \mathbf{send}\ c\ (); \mathbf{send}\ \omega\ () \rrbracket \\
\xrightarrow{[\text{Tr-Co}]}\ & \llbracket () \parallel \mathbf{co}\ l \parallel \mathbf{send}\ \omega\ () \\
&\quad \triangleright_l \ \mathbf{let}\ z = \llbracket\ \mathbf{recv}\ c;\ \mathbf{recv}\ c; \mathbf{commit}\ k\ \ \triangleright_k\ \ () \rrbracket \parallel () \\
&\quad \mathbf{in}\ \mathbf{send}\ c\ (); \mathbf{send}\ \omega\ () \rrbracket \\
\xrightarrow{[\text{Tr-Co}]}\ & () \parallel () \parallel \mathbf{send}\ \omega\ () \\
\xrightarrow{[\text{Send}]\ \ \omega!()}\ & () \parallel () \parallel ()
\end{aligned}
$$

As we have seen, process Q has now managed to signal success after extending the scope of the transaction. We will not show the second case for the sake of brevity, but Q can signal success in the latter case too, by extending both transactions over the **let** construct, applying rule [Tr-Co] on term **co** l and using the resulting unit value () to reduce the **let** expression.

Finally, we might question whether a term P has the same atomic, all-or-nothing behaviour after using one of the transaction's scope extension rules. Intuitively, the commit point of a transaction is not moved, but only the transaction's scope is. After a commit point is spawned, the transaction can be committed at any time. Thus, all reductions up to the commit point are still executed atomically, with an all-or-nothing semantics. But in order to prove this property, a formal theory is necessary.

In TCML, instead of having the extra structural equivalence rules and evaluation rules, we keep the distinction between processes and expressions, add the **atomic** expression and keep rule [Tr-Atomic].

3.3 Programmable aborts

According to rule [Tr-Abort] in the reduction semantics of TCML, transactions can be aborted at any time. This might look like an odd choice, which brings unnecessary non-determinism in the system; we will explain why it is necessary.

Let us add a new expression and a new process in our syntax, **abort** k and **ab** k respectively, where k is a transaction name. This new term will be used in TCML programs to explicitly ask the system to abort transaction k, and only then, rather than at any times. To this end, let us replace rule [Tr-Abort] with the following two rules:

$$\text{[Tr-Prog-Ab]}\quad \frac{}{E[\mathbf{abort}\ k] \xrightarrow{\tau} \mathbf{ab}\ k \parallel E[()]} \qquad \text{[Tr-Prog-Abort]}\quad \frac{e_1 \equiv \mathbf{ab}\ k \parallel e_1'}{\llbracket\, e_1 \rhd_k e_2 \,\rrbracket \xrightarrow{\tau} e2}$$

Rule [Tr-Prog-Ab] will avoid that the ordering of embeddings in transaction be relevant, as we have seen for rule [Tr-Co] in section 3.1. Rule [Tr-Prog-Abort] will make sure that a transaction k will be aborted only if an **ab** k term has been spawned in it. Before reaching an abort point, it will not be possible to abort a transaction.

Consider now the following two examples, adapted from [4]:

1. $\llbracket (\,\mathbf{recv}\ b; \mathbf{commit}\ l) \oplus \mathbf{recv}\ a\ \ \rhd_l\ \ ()\,\rrbracket$[1]
2. $\llbracket\ \mathbf{recv}\ b; \mathbf{commit}\ l\ \ \rhd_l\ \ ()\,\rrbracket$

Communication on channel b can become definitive only after reaching the commit point **co** l. Instead, communications on channel a is always tentative and never definitive: there is no commit point after choosing the right branch of the internal choice in example 1). Thus we would expect the two terms to always

[1] $P \oplus Q$ stands for internal choice. Internal choice represents the non-deterministic choice between alternative P and Q. It can defined in TCML as $\nu a.a!() \parallel a?; P \parallel a?; Q$. For clarity's sake, we will assume that internal choice is a language primitive with the obvious semantics.

display the same behaviour, namely that of receiving a value from channel b. We would expect no test to be able to distinguish the two examples.

Surprisingly, it is indeed possible to distinguish them with the following test:

$$[\![\ \textbf{send}\ a\ (); \textbf{abort}\ k\ \rhd_k\ \ \textbf{send}\ b\ (); \textbf{send}\ \omega\ ()\,]\!]$$

Example 1) can pass this test by communicating over ω. It is possible to embed transaction k into transaction l first, and then perform another embedding. If the right branch is chosen in the non-deterministic choice $\oplus$, then the abort point $\textbf{ab}\ k$ can be created after a synchronization over channel a. At this point transaction k is aborted, and the non-deterministic choice $\oplus$ is restored from the alternative process that was stored in transaction k, together with an expression that can signal success on channel ω after performing a synchronization over channel b. If the left branch of the non-deterministic choice $\oplus$ is chosen, then a synchronization over channel b can happen and a commit point $\textbf{co}\ l$ can also be spawned. After committing transaction l, it is possible to signal success over channel ω. The following reduction is thus possible:

$$\begin{aligned}
&[\![(\ \textbf{recv}\ b; \textbf{commit}\ l) \oplus \textbf{recv}\ a\ \ \rhd_l\ \ ()\,]\!]\ \|\\
&[\![\ \textbf{send}\ a\ (); \textbf{abort}\ k\ \ \rhd_k\ \ \textbf{send}\ b\ (); \textbf{send}\ \omega\ ()\,]\!]\\
\xrightarrow{[\textsc{Tr-Emb}]}\ &[\![(\ \textbf{recv}\ b; \textbf{commit}\ l) \oplus \textbf{recv}\ a\ \|\\
&[\![\ \textbf{send}\ a\ (); \textbf{abort}\ k\ \ \rhd_k\ \ \textbf{send}\ b\ (); \textbf{send}\ \omega\ ()\,]\!]\\
&\ \rhd_l\ \ ()\ \|\ [\![\ \textbf{send}\ a\ (); \textbf{abort}\ k\ \ \rhd_k\ \ \textbf{send}\ b\ (); \textbf{send}\ \omega\ ()\,]\!]\,]\!]\\
\xrightarrow{[\textsc{Tr-Emb}]}\ &[\![\,[\![(\ \textbf{recv}\ b; \textbf{commit}\ l) \oplus \textbf{recv}\ a\ \|\ \textbf{send}\ a\ (); \textbf{abort}\ k\\
&\ \rhd_k\ \ (\,\textbf{recv}\ b; \textbf{commit}\ l) \oplus \textbf{recv}\ a\ \|\ \textbf{send}\ b\ (); \textbf{send}\ \omega\ ()\,]\!]\\
&\ \rhd_l\ \ ()\ \|\ [\![\ \textbf{send}\ a\ (); \textbf{abort}\ k\ \ \rhd_k\ \ \textbf{send}\ b\ (); \textbf{send}\ \omega\ ()\,]\!]\,]\!]\\
\xrightarrow{\oplus}\ &[\![\,[\![\ \textbf{recv}\ a\ \|\ \textbf{send}\ a\ (); \textbf{abort}\ k\\
&\ \rhd_k\ \ (\,\textbf{recv}\ b; \textbf{commit}\ l) \oplus \textbf{recv}\ a\ \|\ \textbf{send}\ b\ (); \textbf{send}\ \omega\ ()\,]\!]\\
&\ \rhd_l\ \ ()\ \|\ [\![\ \textbf{send}\ a\ (); \textbf{abort}\ k\ \ \rhd_k\ \ \textbf{send}\ b\ (); \textbf{send}\ \omega\ ()\,]\!]\,]\!]\\
\xrightarrow{[\textsc{Sync}]}\xrightarrow{[\textsc{E-Let}]}\ &[\![\,[\![()\ \|\ \textbf{abort}\ k\\
&\ \rhd_k\ \ (\,\textbf{recv}\ b; \textbf{commit}\ l) \oplus \textbf{recv}\ a\ \|\ \textbf{send}\ b\ (); \textbf{send}\ \omega\ ()\,]\!]\\
&\ \rhd_l\ \ ()\ \|\ [\![\ \textbf{send}\ a\ (); \textbf{abort}\ k\ \ \rhd_k\ \ \textbf{send}\ b\ (); \textbf{send}\ \omega\ ()\,]\!]\,]\!]\\
\xrightarrow{[\textsc{Tr-Prog-Ab}]}\ &[\![\,[\![()\ \|\ \textbf{ab}\ k\ \|\ ()\\
&\ \rhd_k\ \ (\,\textbf{recv}\ b; \textbf{commit}\ l) \oplus \textbf{recv}\ a\ \|\ \textbf{send}\ b\ (); \textbf{send}\ \omega\ ()\,]\!]\\
&\ \rhd_l\ \ ()\ \|\ [\![\ \textbf{send}\ a\ (); \textbf{abort}\ k\ \ \rhd_k\ \ \textbf{send}\ b\ (); \textbf{send}\ \omega\ ()\,]\!]\,]\!]\\
\xrightarrow{[\textsc{Tr-Prog-Abort}]}\ &[\![(\ \textbf{recv}\ b; \textbf{commit}\ l) \oplus \textbf{recv}\ a\ \|\ \textbf{send}\ b\ (); \textbf{send}\ \omega\ ()\\
&\ \rhd_l\ \ ()\ \|\ [\![\ \textbf{send}\ a\ (); \textbf{abort}\ k\ \ \rhd_k\ \ \textbf{send}\ b\ (); \textbf{send}\ \omega\ ()\,]\!]\,]\!]
\end{aligned}$$

$$
\begin{aligned}
&\xrightarrow{\oplus} [\![\, \textbf{recv}\ b; \textbf{commit}\ l \parallel \textbf{send}\ b\ (); \textbf{send}\ \omega\ () \\
&\qquad \rhd_l\ () \parallel [\![\, \textbf{send}\ a\ (); \textbf{abort}\ k\ \rhd_k\ \ \textbf{send}\ b\ (); \textbf{send}\ \omega\ ()\,]\!]\,]\!] \\
&\xrightarrow{[\textsc{Sync}]} [\![(); \textbf{commit}\ l \parallel (); \textbf{send}\ \omega\ () \\
&\qquad \rhd_l\ () \parallel [\![\, \textbf{send}\ a\ (); \textbf{abort}\ k\ \rhd_k\ \ \textbf{send}\ b\ (); \textbf{send}\ \omega\ ()\,]\!]\,]\!] \\
&\xrightarrow{[\textsc{E-Let}]}\xrightarrow{[\textsc{E-Let}]} [\![\, \textbf{commit}\ l \parallel \textbf{send}\ \omega\ () \\
&\qquad \rhd_l\ () \parallel [\![\, \textbf{send}\ a\ (); \textbf{abort}\ k\ \rhd_k\ \ \textbf{send}\ b\ (); \textbf{send}\ \omega\ ()\,]\!]\,]\!] \\
&\xrightarrow{[\textsc{Tr-Commit}]} [\![\, \textbf{co}\ l \parallel () \parallel \textbf{send}\ \omega\ () \\
&\qquad \rhd_l\ () \parallel [\![\, \textbf{send}\ a\ (); \textbf{abort}\ k\ \rhd_k\ \ \textbf{send}\ b\ (); \textbf{send}\ \omega\ ()\,]\!]\,]\!] \\
&\xrightarrow{[\textsc{Tr-Co}]} () \parallel () \parallel \textbf{send}\ \omega\ () \\
&\xrightarrow{[\textsc{Send}]\ \ \omega!()} () \parallel () \parallel ()
\end{aligned}
$$

We have just proved that aborting transaction k enables a new communication on channel b after which we can signal success on ω. On the contrary, transaction l from example 2) cannot enable the abort point of transaction k, simply because it does not communicate over channel a at all.

Rule [Tr-Prog-Abort] has introduced something undesirable in our system: uncommitted actions can bear an effect on the system. In fact, the right-hand side branch of the internal choice enables a successful execution in example 1). But that term can never committed: choosing the right-hand side automatically excludes the left-hand side of the internal choice, which has the only commit point. So example 1) does not have the same behaviour as example 2), which can never commit.

System aborts are also desirable for another reason. Imagine that we wanted to tentatively receive a communication from two given channels. We might write something like the following term:

$$[\![\, \textbf{recv}\ a; \textbf{commit}\ k\ \ \rhd_k\ \ \textbf{recv}\ b; \textbf{send}\ \omega\ ()\,]\!]$$

Even though communication is available on channel b, reduction is blocked because transaction k is waiting on channel a. In fact, such a term could never *programmatically* abort, because there is no mechanism in TCML to detect whether any communication is available over a channel. In general, it is impractical or even impossible to identify all paths that need to be aborted (e.g. an infinite loop inside a transaction). Allowing the system to abort transactions at any time lifts programmers from the responsibility to foresee all possible erroneous situations and only focus on the committable actions the system will perform. Programming in concurrent and distributed systems is a notoriously error-prone task; programmers need not to worry about erroneous states with communicating transactions, provided the system is "smart" enough to identify such states.

It is still possible to add programmable aborts alongside system aborts. For example, programmers might know that some paths will eventually lead to an abort. The system might not be aware of this, so a programmable abort could be a mechanism for programmers to elicit the abortion of a transaction and increase the efficiency of the system.

3.4 Commit dependencies

Whenever an inner transactions reaches and contains the commit point of an outer transaction, the latter cannot be committed unless the former is. This observation might seem obvious, but it is still worth discussing.

Suppose that we added the following structural equivalence rule:

$$\frac{[\text{S-Co}]}{[\![\, \mathbf{co}\ k \parallel P \ \rhd_l\ Q \,]\!] \equiv \mathbf{co}\ k \parallel [\![\, P \ \rhd_l\ Q \,]\!]} \quad k \neq l$$

This rule, together with the other structural equivalence rules, would allow commit points to move from a transaction to another, and to enable transactions to be committed at any level. It is easy to find one of the undesired effects introduced by this choice:

$$\begin{aligned}
& [\![\, \mathbf{send}\ \omega\ () \parallel \mathbf{send}\ a\ (); \mathbf{commit}\ k \ \rhd_k\ () \,]\!] \parallel [\![\, \mathbf{recv}\ a \ \rhd_l\ () \,]\!] \\
\xrightarrow{[\text{Tr-Emb}]}\ & [\![\, \mathbf{send}\ \omega\ () \parallel \mathbf{send}\ a\ (); \mathbf{commit}\ k \parallel [\![\, \mathbf{recv}\ a \ \rhd_l\ () \,]\!] \\
& \rhd_k\ () \parallel [\![\, \mathbf{recv}\ a \ \rhd_l\ () \,]\!] \,]\!] \\
\xrightarrow{[\text{Tr-Emb}]}\ & [\![\, \mathbf{send}\ \omega\ () \parallel [\![\, \mathbf{send}\ a\ (); \mathbf{commit}\ k \parallel \mathbf{recv}\ a \\
& \rhd_l\ \mathbf{send}\ a\ (); \mathbf{commit}\ k \parallel () \,]\!] \ \rhd_k\ () \parallel [\![\, \mathbf{recv}\ a \ \rhd_l\ () \,]\!] \,]\!] \\
\xrightarrow{[\text{Sync}]}\xrightarrow{[\text{E-Let}]}\ & [\![\, \mathbf{send}\ \omega\ () \parallel [\![\, \mathbf{commit}\ k \parallel () \\
& \rhd_l\ \mathbf{send}\ a\ (); \mathbf{commit}\ k \parallel () \,]\!] \ \rhd_k\ () \parallel [\![\, \mathbf{recv}\ a \ \rhd_l\ () \,]\!] \,]\!] \\
\xrightarrow{[\text{Tr-Commit}]}\ & [\![\, \mathbf{send}\ \omega\ () \parallel [\![\, \mathbf{co}\ k \parallel () \parallel () \\
& \rhd_l\ \mathbf{send}\ a\ (); \mathbf{commit}\ k \parallel () \,]\!] \ \rhd_k\ () \parallel [\![\, \mathbf{recv}\ a \ \rhd_l\ () \,]\!] \,]\!] \\
\xrightarrow{[\text{Tr-Co}]}\ & \mathbf{send}\ \omega\ () \parallel [\![() \parallel () \parallel () \ \rhd_l\ \mathbf{send}\ a\ (); \mathbf{commit}\ k \parallel () \,]\!] \\
\xrightarrow{[\text{Send}]\ \omega!()}\ & () \parallel [\![() \parallel () \parallel () \ \rhd_l\ \mathbf{send}\ a\ (); \mathbf{commit}\ k \parallel () \,]\!] \\
\xrightarrow{[\text{Tr-Abort}]}\ & () \parallel \mathbf{send}\ a\ (); \mathbf{commit}\ k \parallel ()
\end{aligned}$$

In this example, transaction k can signal success over ω only if it commits, which can only happen after communicating over a. The only term it can communicate with is in transaction l, which in turn can never be committed, because it lacks a commit point. Thus, transaction k should never be able to signal success. But this is what happens in the last reduction.

Transaction k has managed to signal success, even though we did not expected it to, thus breaking the consistency of transactions. The problem is that the commitment of transaction k depends on transaction l, but we ignore this dependency when we commit k as soon as we reach its commit point.

4 Examples

We will now show the capabilities of the language through some examples. In order to have a better understanding of the behaviour of each system that we are going to present, we informally introduce a Labelled Transition System (LTS). We will not give a formal definition for it. Instead, next to the reduction steps presented in section 2.2, we will introduce two new kinds of transitions, annotated by labels $c?v$ and $c!v$, where $c \in Chan$ and $v \in Val$. A TCML process can take such a transition in the cases described by the following rules:

$$\text{[S{\scriptsize END}]}\quad \frac{}{E[\,\mathbf{send}\ c\ v] \xrightarrow{c!v} E[()]} \qquad \text{[R{\scriptsize ECV}]}\quad \frac{}{E[\,\mathbf{recv}\ c] \xrightarrow{c?v} E[v]}$$

For clarity's sake, we omit the rules that propagate these labelled transitions over parallel composition, channel restriction and transactions. Intuitively, these transitions model the interaction of the system with an external agent performing a complementary receive or send action.

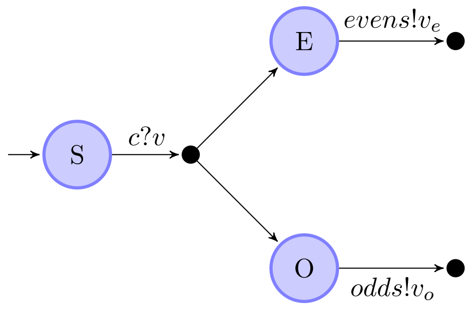
We will also annotate the transition arrow in rules [Tr-Abort] with label **ab** k, rule [Tr-Co] with label **co** k and rule [Tr-Emb] with label **emb** k. These labels will clarify which rule is used to perform a reduction step on transactions.

When a new transaction is started by rule [Tr-Atomic] or a process is embedded into a transaction, we will label the corresponding state with the name of the transaction. Transactions can be aborted at any time by rule [Tr-Abort], so instead of explicitly drawing an arrow from every state in a tentative path to its abort state, we will only draw one from the very start of the transaction to its next state. All tentative transitions and nodes performed in a transaction will be drawn in the orange color, definitive ones in black. Thus each orange node has an implicit *abort* arrow, that points at the first black state that can be reached by traversing the orange arrows backwards. This may be possible for multiple embeddings. When a **co** k transition is performed by a transaction that does not contain nested transactions, the color of transition arrows will be black again.

4.1 Even-Odd consumer

The *Even-Odd Consumer* describes a system containing three public channels, *c*, *evens* and *odds*. A stream of integer numbers is input into the system from channel *c*. Even numbers are output on channel *evens*, and odd numbers on channel *odds*. The system comprises three agents: a producer of integer numbers and two consumers of integer numbers, one for even numbers and one for odd ones. This system describes the common interaction pattern of *single producer/multiple consumers*.

We will first analyse the case in which a single number is sent to the system, and then the more complicated case of a stream of numbers. We will provide three TCML programs for both cases: a *specification* and two *implementations*.



$$\begin{aligned}\nu\, e.\ \nu\, o.\ &\textbf{let}\ n = \textbf{recv}\ c\ \textbf{in}\ \textbf{if}\,(\mathit{isEven}\ n)\ \textbf{then}\,(\textbf{send}\ e\ n)\ \textbf{else}\,(\textbf{send}\ o\ n)\\ &\parallel \textbf{send}\ \mathit{evens}\ (\textbf{recv}\ o)\\ &\parallel \textbf{send}\ \mathit{odds}\ (\textbf{recv}\ e)\end{aligned}$$

Figure 1: Even/Odd consumer specification.

The specification will describe the expected behaviour of the system without using communicating transactions, which will be employed to build the two implementations instead. We will show informally that the specification and implementations describe the same system. In particular, given an informal notion of test on a system, we will show that they all pass the same set of tests.

4.1.1 Channel selection

In this scenario, the system receives a single integer number, which is sent either on channel *evens* if it is even, otherwise on channel *odds*. Figure 1 shows the specification's TCML code and the resulting LTS.

There are three processes running in parallel in the TCML code. We will refer to the first process as the *producer*, to the next one as the *evens consumer*, and the last process as the *odds consumer*. We can also notice two private channels e and o, which are only used within the system. They let the producer connect with the even and odd consumer, respectively.

State S in the LTS refers to the system in this initial state, as just described, where the producer is waiting for a number on channel c, and the consumers are waiting for an input from the producer. The first transition corresponds to the producer receiving a number v on channel c.

The system can then follow one of the two transitions to either state E or O, depending on the value of the v. State E represents the following situation:

$$\nu\, e.\ \nu\, o.\ \ \textbf{send}\ e\ v_e \parallel \textbf{send}\ \mathit{evens}\ (\textbf{recv}\ o) \parallel \textbf{send}\ \mathit{odds}\ (\textbf{recv}\ e)$$

where v is an even number, and the subscript e beneath it represents that v is even. State O represents the situation in which the received number is odd:

$$\nu\, e.\ \nu\, o.\ \ \textbf{send}\ o\ v_o \parallel \textbf{send}\ \mathit{evens}\ (\textbf{recv}\ o) \parallel \textbf{send}\ \mathit{odds}\ (\textbf{recv}\ e)$$

and v_o is the odd number v. From state E the system can output v_e on public channel *evens*; similarly, from state O it can output v_o on public channel *odds*. In either case, the system cannot receive any more inputs at this point. Consequently no more transitions are available.

Observing the LTS, we can deduce that the system can perform three sequences of actions with the external environment:

1. receive a number on channel c.
2. receive a number on channel c and send it on channel *evens* if the number is even.
3. receive a number on channel c and send it on channel *odds* if the number is odd.

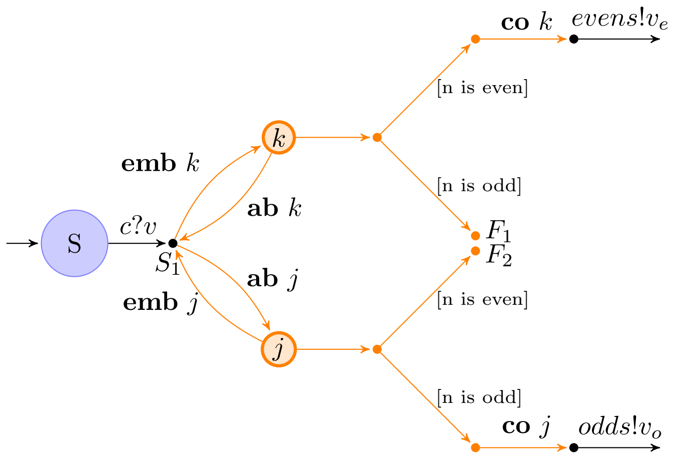
These are all possible sequences of actions that an external agent can observe on the system. We can imagine, for example, that the first test just checks whether the system is running, while the other two tests verify that the system forwards even and odd numbers on the appropriate channels. Assume now that the transactional versions pass the same set of tests. In that case, we would not be able to distinguish the transactional versions from the specification as an external examiner. In that case, we can say that the transactional version implements the specification.

Figure 2 shows the first version of transactional Even/Odd consumer and its LTS. There are still three agents but only one private channel a. The producer receives an integer from public channel c as in the specification, but it does not perform any test on it. Instead, it will send it immediately on private channel a and let the consumers contend to receive it. The consumers are now restarting transactions. Each consumer will tentatively receive a number from channel a. Then they will commit the transaction only if they received the appropriate even or odd number. Otherwise, there will be no other option but to abort the transaction, undo the communication with the producer and try again.

Let us explore the resulting LTS. After the initial state S and receiving a number v, the producer tries to send v over the private channel a. Because transactions are not evaluation contexts, rule [C-SYNC] cannot be applied to two processes unless they are nested within the same transaction. To enable this scenario, the producer can be embedded into either one of the two running transactions that contain a consumer and communicate with it.

In this example, two transitions are available: one in which the producer is embedded into transaction k, which contains the even consumer, and another one in which it is embedded into transaction j, where the odd consumer is. Each transition is marked by the **emb** label. In the first case, the system evolves to:

$$\begin{aligned}&\nu a.[\![\,\textbf{send}\ a\ v \parallel \textbf{let}\ n = \textbf{recv}\ a\ \textbf{in}\\ &\quad \textbf{if}\,(\mathit{isEven}\ n)\ \textbf{then commit}\ k\ ;\ \textbf{send}\ \mathit{evens}\ n\ \textbf{else}\\ &\quad\ \rhd_k\ \ \textbf{send}\ a\ v \parallel \mathit{EvenC}\,]\!] \parallel \mathit{OddC}\end{aligned}$$



$\nu a.($ **send** a $($**recv** $c) \parallel EvenC \parallel OddC)$
where
$EvenC =$ **atomic**$_{rec\ k}$ $[\![$ **let** $n =$ **recv** a **in**
if $(isEven\ n)$ **then commit** k ; **send** $evens\ n$ **else** $()\,]\!]$
$OddC =$ **atomic**$_{rec\ j}$ $[\![$ **let** $n =$ **recv** a **in**
if $(isOdd\ n)$ **then commit** j ; **send** $odds\ n$ **else** $()\,]\!]$

Figure 2: First version of transactional Even/Odd consumer implementation.

As far as reduction semantics is concerned in rule [Tr-Emb], there is no reason to prefer embedding the producer into transaction k rather than transaction j. The choice of which process to embed is entirely non-deterministic, but obviously some embeddings are more useful than others. For example, it would not help to embed the odd consumer into the even consumer here, even though it is a legitimate transition. An actual implementation would favor embeddings that have greater hope of reaching a commit point, taking care at the same time not to alter the semantics of the language by completely disallowing legitimate transitions.

Let us now assume that v is an odd number. Then the system would evolve to the following state:

$\nu a.\, [\![() \parallel () \ \rhd_k \ $ **send** $a\ v \parallel EvenC\,]\!] \parallel EvenC$

This state is represented by node F_1, from which no more transitions are available in the LTS. At this point, the system cannot take any other reduction steps except aborting transaction k. This transition is labelled by **ab** k at the very start of the tentative transitions in transaction k, and ends in node S_1.

The alternative process in transaction k is the following:

send $a\ v \parallel$
atomic$_{rec\ k'}$ $[\![$ **let** $n =$ **recv** a **in if** $(isEven\ n)$ **then commit** k' ; **send** $evens\ n$ **else** $()\,]\!]$

The alternative process contains the producer and even consumer in the same condition they were before exchanging any value. When transaction k is aborted, it is substituted by these processes, which are put in parallel with the odd consumer. The system reaches a state equivalent to state S_1 and is thus effectively rolled back to its original state. The same analysis holds if the producer receives an even number and is embedded into the odd consumer's transaction.

Let us assume that a new number is communicated to the producer over public channel c, and that the producer is embedded in transaction k. Then the consumer will reach expression **commit** k and will be able to spawn a commit point by rule [Tr-Commit]:

$$\nu a.[\![() \parallel \textbf{co}\ k \parallel \textbf{send}\ evens\ v_e \ \rhd_k\ \textbf{send}\ a\ v \parallel EvenC\,]\!] \parallel OddC$$

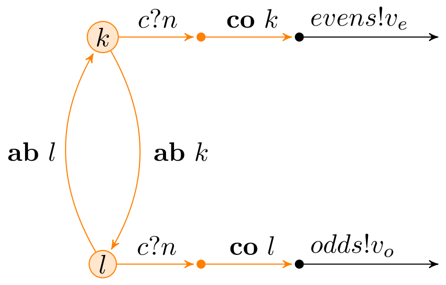
Transaction k can now commit by rule [Tr-Co], since we can move the commit point **co** k to the left by structural equivalence rule [Eq-Assoc]. Then the consumer can communicate even number v_e on public channel *evens*. This sequence of actions is represented by the transition labelled **co** k and $evens!v_e$ in the LTS. After committing, the alternative processes are discarded and the system evolves to the following:

$$\nu a.() \parallel \textbf{send}\ evens\ v_e \parallel OddC$$

After committing transaction k, the system cannot be rolled back anymore, thus all communications among processes in transaction k become definitive. The same analysis holds if number v is odd and the producer is embedded into transaction j.

This system passes the same set of *may* tests [10] as the specification. Obviously the system can receive a single number over channel c, since the producer has not changed. Consider now the paths containing **co** k labels within them in the LTS. There are two such paths, one where an even number is sent to the even consumer, and one where an odd number is sent to odd consumer. In both cases, the system performs a tentative communication on channel c first, and then commits, thus making the communication definitive. If we only consider these kind of paths, and do not consider 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. It is also possible that a transaction infinitely aborts, i.e. is looping forever between state S_1 and state k. If we assume an implementation of TransCML w/ a fair scheduler (e.g. using randomness to some extent to resolve non deterministic choice) the infinitely aborting traces disappear, in the sense that its probability of happening tends to zero, and implementation and specification have the same set of traces.

There are further transition that the system can perform and that we have not described, such as embedding the unit value from the last transition in the last example into transaction j. However, this embedding does not help transaction j to reach a commit point and can only be aborted. Even without



fun $f(x) =$
  atomic $[\![$ **let** $v =$ **recv** c **in if** $isEven\ v$ **then commit** k; **send** $evens\ v$ **else** $() \rhd_k$
    atomic $[\![$ **let** $v =$ **recv** c **in if** $isOdd\ v$ **then commit** j; **send** $odds\ v$ **else** $() \rhd_j$
    $f\ ()\ ]\!]\ ]\!]\ ()$

Figure 3: Second version of transactional Even/Odd consumer implementation.

performing any embedding, transaction j can be aborted infinitely often, since it is a restarting transaction. The LTS does not keep track of tentative transition paths that never reach a commit point, because the only option in these cases is to abort the transaction, and any effect that took place in the system within that transaction will have not be definitively observable by an external examiner.

In the previous example we could have also embedded transaction j in transaction k at any time, resulting in transaction k being the outermost transaction. This embedding does not lead to any committed behaviour either. If the producer is embedded into transaction j and communicates with the odd consumer, then there will be no communication available for the even consumer in transaction k. Transaction k will have to be aborted, even if transaction j was committed. If transaction j is embedded but no process is embedded into it, then transaction k will be able to commit and the system will end up in the same state as that explained in the previous example. The LTS does not consider reduction steps that do not contribute to committing a transaction.

With these considerations about the LTS in mind, let us examine the second version of transactional Even/Odd consumer, shown in Figure 3.

The second version only features a single process containing a recursive function f and no private channels. This function first starts a transaction k, whose alternative is to start another transaction j. The alternative to transaction j is to recursively call f, which starts transaction k again. We can thus argue that either transaction k is active or transaction j is. Aborting transaction k will activate transaction j, and aborting transaction j will activate transaction k.

In short, the behaviour of the system will alternate between that of transaction k and that of transaction j. In particular, transaction k receives a number v on public channel c and sends it on channel *evens* if it is even; similarly does transaction j on channel *odds* if it receives an odd number. If either of these tests is passed, the respective transaction will be committed and the rest of the transaction will be executed definitively. Otherwise the transaction will reach a state where the only possible reduction step is to abort.

Here transactions in TCML look similar to exception-handling blocks, with the difference that aborting exception here are thrown by deadlocks and that thrown exceptions automatically undo previous side effects of the block (the **recv** c in this example). Note that the programmer does not need to specify each individual aborting point in TCML transactions because these are all the points where a transaction deadlocks.

We will not describe the LTS in ulterior length, as it can be easily understood in light of the previous discussion. It is also easy to verify that this system passes all three tests of the specification, including the case of the single communication to the system.

The alternating behaviour of this last version is particularly interesting. This version contains two mutually recursive transactions, that are committed only if one of the tests for even or odd numbers is passed. This reminds very closely *guarded commands*, and it in fact possible to generalize this construct. For example, if expression e_1 is guarded by condition b_1 and expression e_2 is guarded by condition b_2, their composition can be written as:

$$\begin{aligned}\textbf{fun}\ gCom(e_1, e_2) = \textbf{atomic}\ [\![\, & \textbf{if}\ b_1\ \textbf{then commit}\ k; e_1\ \textbf{else}\ () \rhd_k \\ & \textbf{atomic}\ [\![\, \textbf{if}\ b_2\ \textbf{then commit}\ j; e_2\ \textbf{else}\ () \rhd_j gCom\ ()\,]\!]\,]\!]\end{aligned}$$

Guards in guarded commands will be tried until one of them can be committed; their eventual effects will always be rolled back. Similarly, it is possible to exploit the alternating behaviour of the same construct to have external choice. 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:

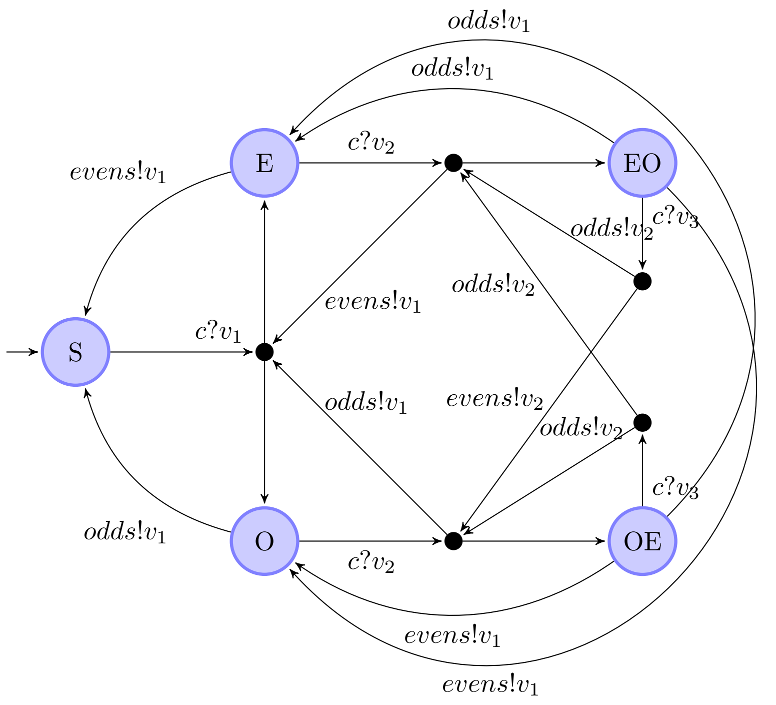
$$\begin{aligned}\textbf{fun}\ extCh(e_1, e_2) = \textbf{atomic}\ [\![\, & \textbf{let}\ x = \textbf{recv}\ c_1\ \textbf{in commit}\ k; e_1 \rhd_k \\ & \textbf{atomic}\ [\![\, \textbf{let}\ x = \textbf{recv}\ c_2\ \textbf{in commit}\ j; e_2 \rhd_j extCh\ ()\,]\!]\,]\!]\end{aligned}$$

If a matching communication is available to synchronize with one of the two branches in the external choice, the construct can abort transactions until the transaction containing the matching branch is available for communication.

4.1.2 Buffered selection

Let us now see the case where the producer receives a stream of numbers from c, rather than a single one. In order to process a potentially infinite number of inputs from c, both producer and consumers must always be able to process a new input after consuming an input; in short, they must be recursive processes. This can be easily achieved by extending the specification in section 4.1.1 with recursive calls. The resulting code and LTS are shown in Figure 4.

While the code needs little explanation, since we have just added recursive calls at the end of the producer and consumers, the resulting LTS has become more complicated. States S , E O represent the same states in which the



$\nu e.\nu o.$
  (**fun** $np(x) =$ **let** $n =$ (**recv** c) **in**
    (**if** ($isEven\ n$) **then** **send** $e\ n$ **else** **send** $o\ n$); np ()) ()
  ∥ (**fun** $ec(x) =$ **send** $evens$ (**recv** e); ec ()) ()
  ∥ (**fun** $oc(x) =$ **send** $odds$ (**recv** o); oc ()) ()

Figure 4: Even/Odd consumer specification

non-recursive system started and then finished after receiving a number v_1 over public channel c. Because all processes are recursive, the system will be brought back to its initial state S after outputting numbers in states E and O, rather than terminate execution.

Being recursive, the producer might also receive a new number v_2 over channel c before a consumer outputs the previous value v_1 on either channel *evens* or *odds*. There are two such transitions labelled $c!v_2$, from state E and state O. Let us assume that number v_1 is even. If number v_2 is even too, the even consumer will not be able to receive a new input without outputting number v_1

first, and the producer will have to wait. In this case, the system will go back to the state right after S. If the number is odd, then it might be communicated to the odd consumer, in which case the system will reach state EO. At this point, both consumers contain a number but the producer does not. There are three possible transitions now: either one of the consumer outputs, or the producer receives a third number from c. At this point, both consumers have to output before receiving any further numbers. The same considerations hold if v_1 is odd and we consider states O and OE.

The system mainly can receive at most three numbers, if they are not all even or all odd, and send them on channels *evens* and *odds* respectively. Notice that the system can behave as a three-place buffer. In fact, that the LTS can perform three $c?v_i$ transitions before any number is output on any channel, thus it can hold three values at the same time before outputting any of them.

A transactional implementation of this specification is shown in figure 5.

This implementation extends the non-recursive transactional version with a recursive call for both producer and consumers. If we consider only committed behaviour and we exclude embeddings that do not directly influence commit paths of a transaction, in light of the discussion about the non-recursive version in Figure 1, the resulting LTS is fairly similar to the specification.

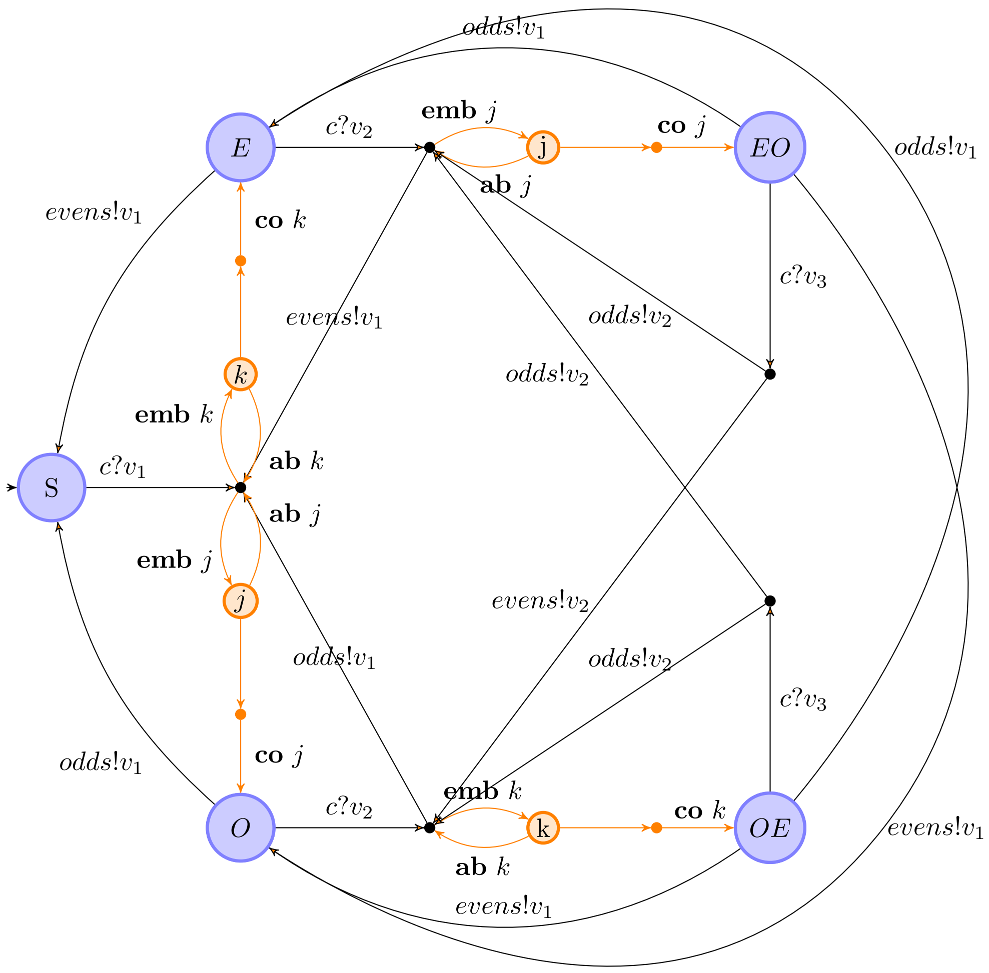
Right after the producer receives a value v_1 in state S, there are two available transactions in which the producer can be embedded. Embedding in the wrong transaction will result in an abort and the system going back to the point where it could choose which embedding to perform. It is also possible to recognize in state S the original LTS of the transactional system, with the only difference that, being recursive, outputting on either channel *evens* or *odds* will bring the system back to its original state.

After the producer receives the second number, there is only one embedding available, namely into the transaction containing the free consumer. Again, if the consumer is not compatible with the choice, the system will be rolled back. When this further transaction is committed, the system can take a third value, before being forced to output.

We will not prove formally that this system can pass the same tests as the specification, but from the shape of the LTS it should not be difficult to be convinced that the implementation follows the specification.

Unfortunately, it is not possible to extend the second transactional version so that it passes the same tests as the specification. Suppose that we had three recursive processes running in parallel, such as the one shown in Figure 3, then the system would be able to consume three consecutive even number, which the specification cannot do. If on the other hand we only spawned two processes to obviate to this problem, then the implementation would not be able to behave as a three-place buffer.

Even though this second system is not equal to the specification, it is interesting to observe that this system is more scalable than the specification. In fact, we can simulate a three-place buffer by just spawning three transactional processes; if we wanted to have more places in the buffer, we could just spawn more processes, with no need to modify any line of code in any of the



$\nu\, a.$

$\quad$ (**fun** $np(x) =$ **send** a (**recv** c); $np()$) ()

$\quad \|$ (**fun** $ec(x) =$ **atomic**$_{rec\ k}$ $[\![$

$\qquad$ **let** $v =$ **recv** c **in if** $isEven\ v$ **then commit** k; **send** $evens\ v$ **else** () $]\!]$; ec ()) ()

$\quad \|$ (**fun** $oc(x) =$ **atomic**$_{rec\ j}$ $[\![$

$\qquad$ **let** $v =$ **recv** c **in if** $isEven\ v$ **then commit** j; **send** $odds\ v$ **else** () $]\!]$; oc ()) ()

Figure 5: Even/Odd consumer implementation

processes. It is not possible to achieve the same scalability in the specification. For example, we might spawn an extra odd consumer, which would compete with the other existing consumer for inputs, but the resulting system would be unbalanced towards odd numbers. In a practical setting, the producer would also be the bottleneck of the system, since it would manage all communications with channel c. This second transactional version is more distributed instead, in that all processes can receive on channel c and they can all output even and odd numbers.

4.2 Three-way rendezvous

In the following section we will demonstrate that TransCCS is expressive enough to define the *three-way rendezvous*, a construct that can synchronize three processes at a time, just like a channel is a rendezvous that can synchronize two processes. The solution presented in this section is inspired by the implementation in Transactional Events from [6].

The following theorem is stated in [16]:

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).

The 2-way rendezvous event constructors that CML offers are **sendEvt** and **recvEvt**. One of the constructors is **choose**, which creates an event from two given events; the resulting event is the non-deterministic choice between the two input events. This theorem does not preclude the implementation of a (n+1)-way rendezvous in particular scenarios. It is in fact possible to implement the 3-way rendezvous in the presence of exactly three processes, as shown in the following CML code:

```
fun twr (c : int chan chan, v0 : int) =
 let
   val c0 : int chan = channel ()
   val (isLeader, c1) = sync (
     choose [wrap (recvEvt c, (fn y => (true, y))),
             wrap (sendEvt (c, c0), (fn () => (false, c0))) ] )
 in if isLeader
     then  let
            val v1 = recv c1
            val c2 = recv c
            val v2 = recv c2
           in
            send (c1, v0); send (c1, v2);
            send (c2, v0); send (c2, v1);
            (v1, v2)
           end
     else  send (c0, v0);
           (recv c0, recv c0)
 end

fun twrTest () =
let val c : (int chan) chan = channel ()
in
 spawn (fn () => twr (c, 1));
 spawn (fn () => twr (c, 2));
 spawn (fn () => twr (c, 3));
end
```

Let us assume that three processes want to rendezvous with each other using the `twr` function. Let us also assume that they all share the same given channel `c`. Each process creates a private channel `c0`, and then each tries either to receive the private channel of another process from channel `c`, or to send its own channel over `c`, in the `choose` event. Upon synchronization of this event, the process that picked the first `wrap` event in the choice, will act as the leader of the rendezvous, indicated by setting the boolean flag `isLeader` to `true`. The other process event will take the role of a follower, thus its `isLeader` flag will be set to `false`. Next, the leader process will try to receive another private channel from channel `c`. At this point, the third process will have no choice but to become a follower. Having gathered the private channels of the followers, the followers communicate their own values to the leader, which in turn proceeds to exchange values among all participants through their private channel.

As we have shown, this piece of CML code does manage to rendezvous three processes over the same channel `c`. But notice that the type of this function is actually `a' chan chan * a' -> unit`, which is not an event type. This means that we cannot compose the three-way rendezvous function we just implemented with other communication primitives. For example, we cannot create a function that non-deterministically performs a three-way rendezvous on a channel `c` or on a channel `d`. Non-deterministic choice is given by the `choose` event constructor, which requires its arguments to be events. The expression `choose [twr c, twr d]` is not well-typed, because `twr` has function type, not an event type.

More importantly, this implementation will not work in the presence of more than three participants. In the case of four participants, two processes might become leaders after synchronizing with the other two followers, and the two rendezvous just started would end up in a deadlock. When Theorem 6.1 states that states CML cannot implement an (n+1)-way rendezvous operation *abstractly*, it refers to the impossibility of creating a (n+1)-way rendezvous of type `event`, which would compose well with the other communication primitives. This also means that it is not possible to create an implementation that works in an arbitrary context. Such a a potential abstract implementation of type other than `event`, for example of functional type, would require the use of timeout expressions and a complex ad-hoc recovery mechanism.

TCML does not have any event combinators, but just primitives for CCS communication and transactions, which can spawned by the **atomic** and the **atomic**$_{\text{rec}}$ expressions (the latter being an composite expression that uses the former). The only 2-way rendezvous provided are the **send** and **recv** expressions. We will first show an implementation of the three-way rendezvous in TCML, and then discuss its composability in TCML and show that Theorem

6.1 does not apply for TCML, that is, there exists an abstract implementation of (n+1) rendezvous.

Let's assume that each client i, where $i \in N$, is associated with a value v_i of type A to communicate, and an integer ID_i that uniquely identifies it. Let us also assume a function *cmap*, that associates a process ID to a unique channel of type A **chan**. This mapping function is necessary to circumvent the restriction imposed on channels on rule [T-CHAN]. Channels are well-typed if and only if the value they carry is of type *BaseType*, that is, if it does not contain any channel names. Instead of sending channel names, we will send process IDs. A fixed assignment of process to a channel will be enough to perform a three-way rendezvous among processes.

The authors in [6] show an implementation of three-way rendezvous using Transactional Events. Apart from the use of transactions instead of Transactional Events, and the use of identifiers, our implementation in TCML is very similar. The code and the LTS are shown in figure 6.

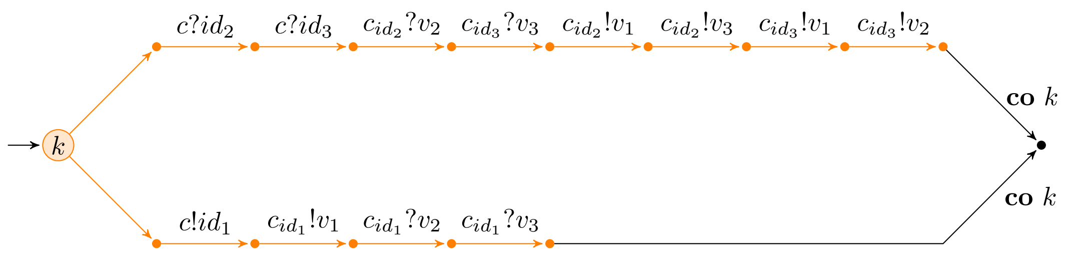
Each participant in the three-way rendezvous non-deterministically decides whether to be a follower or leader in the rendezvous, because of internal choice $\oplus$ in the body of the `rendezvous` function. As shown in the LTS, on the one hand a follower tries to send its own ID over public channel c first, and then it will try to communicate its own value v_i on its own private channel c_{id}. At this point, it will try to receive from that same channel the values of the other participants and commit. On the other hand, the leader will try to receive the IDs of the other participants in the rendezvous, receive their values, swap values accordingly, and commit.

There is an obvious complementarity between follower and leader. Each action performed by the former can be matched by the latter. When the follower sends its own ID over the public channel c, the leader can receive it; then when the follower can send its own value, the leader can receive it, and so on. It is also evident from the LTS that a leader can reach its commit point if it is interacting with two followers. We can in fact recognize for each step that can be matched by a follower, another duplicate step that can be matched by another follower.

Let us consider the scenario depicted in figure 7, in which three participants are trying to rendezvous on channel c, utilizing the three-way rendezvous. Let us assume that their identification numbers are ID_1, ID_2 and ID_3. 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).

There are four states in which the system can evolve. Either all participants decide to become leader, all become followers, one becomes follower and two leaders, or one becomes leader and two followers. The identity of processes does not matter in the rendezvous, so we can consider the scenario in which process ID_1 becomes leader, to be equivalent to the one where it is process ID_2 to become leader.

Figure 7 does not show the full LTS, but we can be easily convinced that the only scenario that leads to a commit point is the one where there is one leader and two followers. Looking again at the LTS for a single participant in Figure



```
fun $rendezvous(v_{id_1}, id_1, c) =$
    let $leader =$
        let $c_{id_2} = cmap\ (\mathbf{recv}\ c)$ in
        let $c_{id_3} = cmap\ (\mathbf{recv}\ c)$ in
        let $v_{id_2} = \mathbf{recv}\ c_{id_2}$ in
        let $v_{id_3} = \mathbf{recv}\ c_{id_3}$ in
        send $c_{id_2}\ v_{id_1}$; send $c_{id_2}\ v_{id_3}$;
        send $c_{id_3}\ v_{id_1}$; send $c_{id_3}\ v_{id_2}$;
        $(v_{id_2}, v_{id_3})$
    in
    let $follower =$
        let $c_{id_1} = cmap\ id$ in
        send $c\ id_1$;
        send $c_{id_1}\ v_{id}$;
        let $v_{id_2} = \mathbf{recv}\ c_{id_2}$ in
        let $v_{id_3} = \mathbf{recv}\ c_{id_3}$ in
        $(v_{id_2}, v_{id_3})$
    in
        atomic$_{rec\ k}$ $[\![$
          (let $rend = leader\ x$ in commit $k; rend) \oplus$
          (let $rend = follower\ x$ in commit $k; rend)\ ]\!]$
```

Figure 6: Three-way rendezvous code and LTS.

6, if there are three followers or three leaders, no synchronization can happen on public channel c. If there are two leaders and only one follower, there will be a race condition for the leaders to receive the ID of the follower, but then both leaders will not be able to proceed because they need at least one more follower to send them its own ID. Each of these cases leads the system to abort the rendezvous and try a different configuration. Notice that it is not necessary to abort the whole rendezvous for it to succeed. For example, if all the processes became followers, it might be enough to abort a single transaction. The process within the aborted transaction will have another chance to become leader next, and the rendezvous might succeed. This consideration should be taken into account in the creation of an efficient implementation.

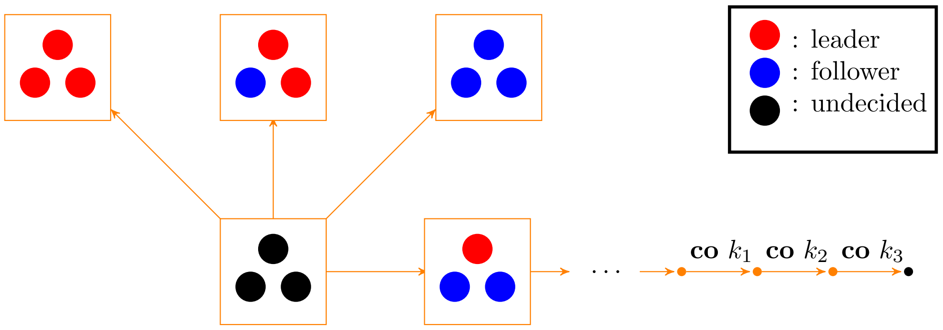


Figure 7: LTS for three participants engaging in the three-way rendezvous.

In the case in which there is one leader and two followers, every action of all participants can be matched by one another, and thus all transactions can reach its commit point. Recalling that we can ignore all paths that lead to an abort, we can state that three participants engaging in a three-way rendezvous will eventually exchange values with each other.

In light of this discussion, let us finally turn our attention to the case in which there are more than three participants engaging in a three-way rendezvous. If there are 4 participants performing a three-way rendezvous, there will be five possible scenarios: either all of them are leaders, or they are all followers, are one, two or three of the will be leaders and the rest followers. Obviously the three-way rendezvous construct will only allow three out of four participants to exchange values. As discussed previously, the rendezvous is successful only if there are two followers and one leader. Therefore, only two cases will succeed: the case with one leader and three followers, and the case with two leaders and two followers. In this last case, the followers might synchronize with different leaders, but the system would not be in a deadlock, because it is always possible to abort one of the transactions and try again.

Similarly, the five participants case comprises six different scenarios, and only three of them will succeed, whenever there are at least one leader and two followers.

The six participants scenario is different. If there are one, three or four leaders, three participants can rendezvous as already discussed. However, after this initial three processes rendezvous, the remaining three processes can perform another rendezvous, after aborting some transactions. In the case in which two participants become leaders and four followers, the system can actually perform two separate rendezvous sequentially. In fact, if we assign the six participant in two separate groups, one of the two sets will first use channel `c` to exchange process IDs. After that, the participants in the other set will be able to exchange their own process IDs as well, while the former set are exchanging values. Each set can be considered as a rendezvous with three participants. Thus we can conceive three scenarios as partially successful, and one scenario as fully successful.

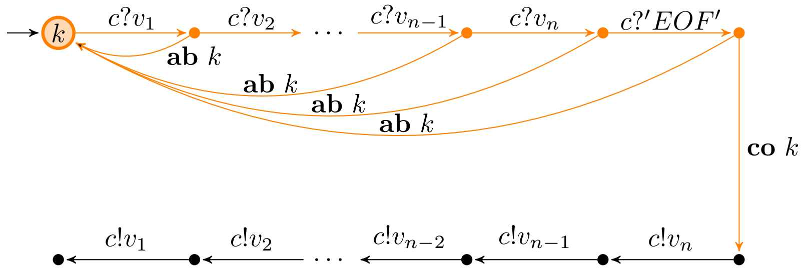
On the light of this discussion, we can note that a system of n participants trying to perform a three-way rendezvous on a public channel c, will eventually reduce to $\lfloor n/3 \rfloor$ sets of pairs of exchanged values, and possibly n modulo 3 processes aborting forever.

TCML's type system does not have the same distinguishing power that CML has. TCML's three way rendezvous has type $(\mathbf{int}, \mathbf{int})$. Nonetheless, the three-way rendezvous composed well with other constructs. As an example, we can non-deterministically choose to a three-way rendezvous over two different channels with the following transaction, in the same vein of the external choice function introduced at the end of section 4.1.1:

```
fun $twrChoice(v_{id_1}, id_1, c, d) =$
    atomic $[\![$ let $x = rendezvous\ c$ in commit $k; x \rhd_k$
      atomic $[\![$ let $y = rendezvous\ d$ in commit $j; y \rhd_j twrChoice\ ()\ ]\!]\ ]\!]$
```

4.3 Communication sequence buffering

We will now consider the effect of moving a commit point within a transaction. Suppose that a process must receive either all values coming from a public channel c until an 'end-of-file' (EOF) token is received, or none of them. Afterwards, all values must be retransmitted in any order over another channel d. Before starting to receive values, the process starts a fresh transaction. Obviously, the commit point must be placed after the termination token is received. Two choice are possible though: the commit point can be place either right after receiving the termination token, or after all values have been sent back. We will call the first version *early commit* and the second version *late commit*. Code listings for both cases are shown in Figure 8 and 9.



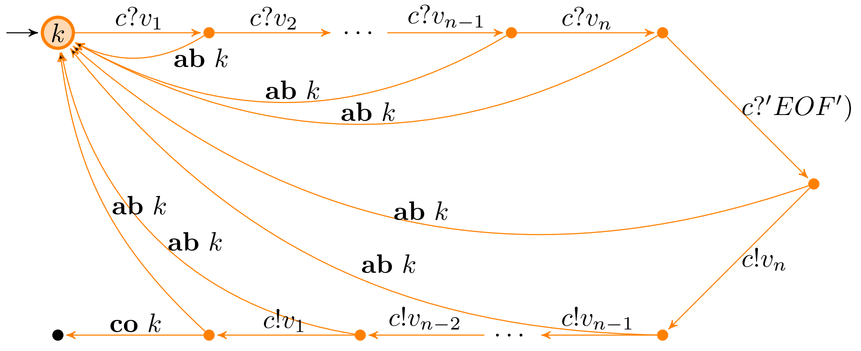
```
atomic$_{rec\ k}$ $[\![$
  (fun $f(x) =$
        let $n =$ recv $c$ in
        (if $n = EOF$ then commit $k$ else $f$ () );
        send $d\ n$) () $]\!]$
```

Figure 8: Communication sequence buffering with early commit.

In the early commit case, the buffering process will recursively receive a communication from channel c until the EOF token is received. Note that each recursive call creates a **let** expression that is bound to **send** expression after the commit point. As soon as the EOF token is received in the **if** $-$ **then** $-$ **else** expression, the commit point **commit** k is reached and then **send** expressions accumulated during the recursive calls and bound by the previous **let** expressions are evaluated.

The transaction enclosing the buffering process can be committed as soon as the commit point is reached. Because of rule [TR-ABORT] from Table 11 in the reduction semantics, transactions can be aborted at any time, and their effect will be undone. Thus the LTS shows an abort transition from any state before the commit transition labelled **co** k is performed. In case of an abort, the system is brought back to the start, because it is a restarting transaction. After the commit transition, subsequent communications on channel d will be definitive and cannot be rolled back anymore, which are drawn in black colour in the LTS.

The communicating transaction in which the buffering process is embedded can thus guarantee its atomic behaviour, since either all values from c will be received first, in order for the transaction to commit.



```
atomic_{rec k} ⟦
  ((fun f(x) =
          let n = recv c in
          (if n = EOF then () else f () );
          send d n) ());
  commit k⟧
```

Figure 9: Communication sequence buffering example with late commit.

In the late commit version, the commit point is placed outside the **if** $-$ **then** $-$ **else** expression and after the **send** expression. The buffering process will then recursively receive values on channel c until the EOF token is received, and then it will proceed to send the values back on channel d. At this point the commit point is reached and the transaction can commit.

The difference from the early version is evident from the resulting LTS. All communications are now tentative, thus affecting the behaviour of the buffering process. It will either receive and send all values, or do nothing. As external observers, we would be able to differentiate the two systems with the following test:

$$\textbf{send}\ c\ v_1;\ \textbf{send}\ c\ v_2;\ \textbf{send}\ c\ EOF;\ \textbf{recv}\ d$$

This test would try to send two values to the system in exam, send an *EOF* token and then try to receive only one of the two values sent. The late commit version of the system will be not able to satisfy this test. In order to complete the transaction, the system needs to reach its commit point. In order to do so, the late version must first send all the values it received from channel c. The same does not hold true for the early commit system, which can complete its transaction as soon as the *EOF* token is received.

4.4 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 10, 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.

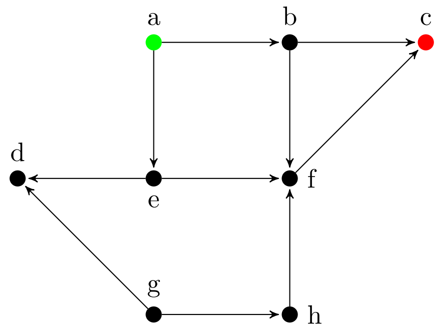


Figure 10: An example graph. The start node 'a' is green, the end node 'c' is red.

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

```
go(X, X, [X]).
go(X, Y, [X|T]) :-
```

```
  link(X, Z),
  go(Z, Y, T).
```

The first argument in the `go` 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 in the node 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 starting node is linked to, and the ending node (second clause). This intermediate node becomes the new starting node, and the `go` 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 `go` 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.

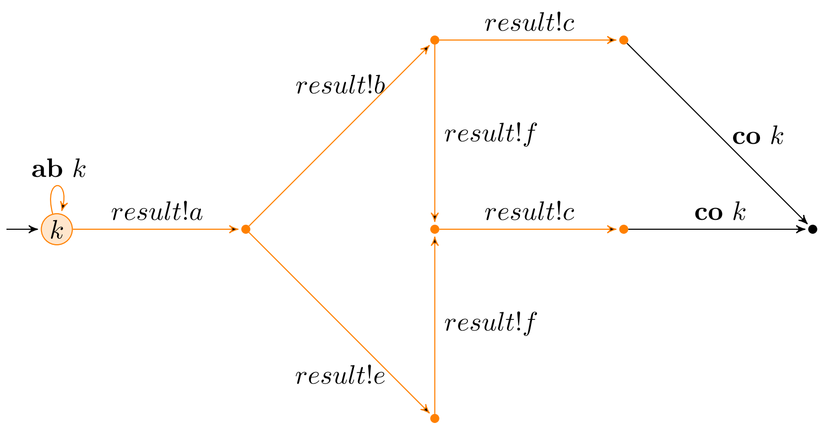
A Prolog machine would find the following solutions:

```
|?- go(a, c, X).
X = [a,e,f,c]
X = [a,b,f,c]
X = [a,b,c]
```

TCML can reproduce the same backtracking behaviour using restarting transactions, as shown in Figure 11. For the sake of the current discussion, let us lift for the *BaseType* restriction in rule [T-CHAN] and let channel names to be sent over channels. Let us also model each edge from node n to node n' in Figure 10 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 *go* 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 *go* on the new node, as in the second clause.

First of all, note that the whole *go* 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 *go* function encounters a node that is not connected to any other node, then the transaction can be aborted and another path can be tried.



```
$\textbf{atomic}_{rec\ k}$ $[\![$
  $(\textbf{fun}\ go(x, e) =$
        $\textbf{if}\,(x = e)$
          $\textbf{then}\ \textbf{send}\ result\ x; \textbf{commit}\ k$
          $\textbf{else}\ \textbf{send}\ result\ x; \textbf{let}\ y = \textbf{recv}\ x\ \textbf{in}\ go\ (y, e))$
  $(startNode, endNode)$ $]\!]$
```

Figure 11: Graph Search algorithm with a single transaction.

From the resulting LTS in Figure 11, we can recognize three sequences of actions that lead to a commit point:

- result!a, result!e, result!f, result!c
- result!a, result!b, result!f, result!c
- result!a, result!b, result!c

which form basically the same set of answers of 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 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.

Another point worth mentioning is that, when a wrong path is taken, the TCML program will restart the whole transaction, whereas the Prolog program would only backtrack up to the latest choice taken. It is possible to obviate to this problem in TCML by opening a new transaction at each invocation of the *go* function:

```
(fun $go(x, e) =$
      atomic$_{rec\ k}$ $[\![$
        if $(x = e)$
          then send $result\ x$; commit $k$
          else send $result\ x$; let $y =$ recv $x$ in $go(y, e)$; commit $k\,]\!])$
  $(startNode, endNode)$
```

Notice that the *go* function declaration and the beginning of the restarting transaction k are swapped. At each recursive call to *go*, a new transaction is started. The system is now free to abort any of the new transactions spawned, so that it can try different y nodes at any point during path searching, thus mimicking backtracking more closely.

Even with the last TCML version, transactions can be aborted at any time, even 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.

5 Towards an LTS

We have discussed the dynamic behaviour of the examples in the previous section, making recourse to an informal notion of LTS. In this chapter we will draft a formal definition of an LTS for TCML's reduction semantics, based on a different syntax for transactions. The current definition is still work in progress, so we will only present intuitions rather than formal arguments.

The syntax for values, expressions, evaluation contexts and types is unchanged. We modify the syntax for processes, and add labels α, β and ℓ as follows:

$$\begin{array}{lcl}
P & ::= & \langle e, ts\rangle_p \ \mid\ \nu c.P \ \mid\ P \parallel P \ \mid\ \langle\, \mathbf{co}\ k, ts\rangle_p \\
ts & ::= & (k, P) :: ts \ \mid\ [] \\
\\
\alpha & ::= & c?v \ \mid\ c!v \\
\beta & ::= & \mathbf{ab}\ k \ \mid\ \mathbf{co}\ k \ \mid\ \mathbf{emb}_{\Delta}\, k \\
\ell & ::= & \tau \ \mid\ \alpha \ \mid\ \beta
\end{array}$$

In the LTS, a process pairs an expression e and a transaction stack ts, and is associated a unique identifier $p \in PID$, the set of all process identifiers. Processes can be composed by parallel composition. Channel restrictions are unchanged. Processes can either take an internal step, communicate with other processes or perform a transaction step. Internal transitions are those marked by the τ label. Transisions involving communication over channels is marked by α labels. Transitions performed by processes in a transaction are marked by β labels. In the $\mathbf{emb}_{\Delta}\, k$ label, the set $\Delta : \mathcal{P}(PID) \times \mathcal{P}(\mathcal{K}^*)$ contains a pair of sets: the first is a subset of the set PID, the secpmd is a subset of the set of all sequences of transaction names. The set of all labels is ℓ.

The sequential part of the language is the same as in the reduction semantics in Table 7 and Table 8. We assume no structural equivalence between processes.

Concurrency rules are presented in Table 19. A process can take an internal step with rule [C-Step]. Communications from the **send** and **recv** expressions can be evaluated by the [Send] and [Recv] rules, and generate transitions labelled by $c?v$ and $c!v$ respectively. Communication is propagated through parallel composition by rule [C-Par] and through channel restriction by rule [C-Restr]. In order to communicate, two processes can synchronize on a channel c if and only if their transaction stacks contain matching names k, as expressed in rule [C-Sync]. Processes spawned by **spawn** are assigned a fresh unique identifier p' and the transaction stack ts of its parent process, after function *killer* has been applied to it. The *killer* function replaces each alternative process in the transaction stack with the process $\langle (), []\rangle_{\emptyset}$, which cannot perform any transition. Let us call this process **NIL**. If a process p is spawned within a transaction and that transaction is aborted, the effect of spawning p must be rolled back. Rather than removing it, the spawned process is replaced by the **NIL** process, which cannot perform any transition, except begin embedded into other transactions.

The LTS transaction rules in Table 20 define how to manipulate the stack of transactions that each process is equipped with. LTS transaction rules are

[C-Step]

$$\frac{e \hookrightarrow e'}{\langle E[e], ks\rangle_p \xrightarrow{ks(\tau)} \langle E[e'], ks\rangle_p}$$

[C-Spawn]

$$\frac{}{\langle E[\textbf{spawn}\ v], ks\rangle_p \xrightarrow{ks(\tau)} \langle v(), killer(ks)\rangle_{p'} \parallel \langle E[()], ks\rangle_p} \quad p' \text{ is globally fresh}$$

[Send]

$$\frac{}{\langle E[\,\textbf{send}\ c\ v], ks\rangle_p \xrightarrow{ks(c!v)} \langle E[()], ks\rangle_p}$$

[Recv]

$$\frac{}{\langle E[\,\textbf{recv}\ c], ks\rangle_p \xrightarrow{ks(c?v)} \langle E[v], ks\rangle_p}$$

[C-Par]

$$\frac{P_1 \xrightarrow{ks(\alpha)} P_1'}{P_1 \parallel P_2 \xrightarrow{ks(\alpha)} P_1' \parallel P_2} \quad \alpha = c?v, c!v$$

[C-Restr]

$$\frac{P_1 \xrightarrow{ks(\alpha)} P_1'}{\nu c.P_1 \xrightarrow{ks(\alpha)} \nu c.P_1'} \quad c \notin \alpha$$

[C-Sync]

$$\frac{P_1 \xrightarrow{ks(c!v)} P_1' \qquad P_2 \xrightarrow{ks(c?v)} P_2'}{P_1 \parallel P_2 \xrightarrow{ks(\tau)} P_1' \parallel P_2'} \quad \pi_1(ks_1) = \pi_2(ks_2)$$

Table 19: LTS Concurrency rules

labelled by either **emb** k, **ab** k or **co** k, and trailed by a list ks of transaction names $k \in \mathcal{K}$. The list of transaction names ks also occurs in the stack of transactions that belongs to each process. For example, rule [Tr-Co] has the following rules:

[Tr-Co]

$$\frac{}{\langle e, ks :: (k, Q) :: ks'\rangle_p \xrightarrow{ks(\,\textbf{co}\ k)} \langle e, ks :: ks'\rangle_p}$$

In this rule, ks indicates both a list of transaction names k in the transition label, and a list of pairs $\mathcal{K} \times Proc$, i.e. a pair composed of a transaction name k and a process Q. In the LTS rules, we omit writing that the list of transaction names ks in the transition label is obtained by projecting the first element of the pair $\mathcal{K} \times Proc$. In an abuse of notation, we omit the $pi_1(ks)$ operation from the transition label, that would need to be specified in most rules in Table 20.

Rule [Tr-Abort] substitutes the current process with an alternative process from the transactions stack of the process itself. In fact, the transaction stack is divided by the :: operator in $ks :: (k, \langle e', ks''\rangle) :: ks'$. When generating a signal $ks(\,\textbf{ab}\ k)$, the alternative $\langle e', ks''\rangle$ that corresponds to k is removed from the stack and substitutes the current default process. The former part of the stack ks is discarded, whereas the latter part ks' is merged with the alternative process' transactions stack ks''. Rule [Tr-Ignore-Abort] allows processes to ignore **ab** k transitions when the transactions stack of the process and the list of

[Tr-Abort]

$$\frac{}{\langle e, ks :: (k, \langle e', ks''\rangle) :: ks'\rangle_p \xrightarrow{ks(\,\mathbf{ab}\ k)} \langle e', ks'' :: ks'\rangle_p}$$

[Tr-Ignore-Abort]

$$\frac{}{\langle e, ks\rangle_p \xrightarrow{ks'(\,\mathbf{ab}\ k)} \langle e, ks\rangle_p} \quad ks' :: k \not\leq ks$$

[Tr-Commit]

$$\frac{}{\langle e, ks :: (k, Q) :: ks'\rangle_p \xrightarrow{ks(\,\mathbf{co}\ k)} \langle e, ks :: ks'\rangle_p}$$

[Tr-Ignore-Commit]

$$\frac{}{\langle e, ks\rangle_p \xrightarrow{ks'(\,\mathbf{co}\ k)} \langle e, ks\rangle_p} \quad ks' :: k \not\leq ks$$

[Tr-Emb-Process]

$$\frac{}{\langle e, ks\rangle_p \xrightarrow{ks(\,\mathbf{emb}_{\,\triangle}\ k)} \langle e, (k, \langle e, ks\rangle) :: ks\rangle_p} \quad p \in \pi_1(\Delta)$$

[Tr-Ignore-P-Emb]

$$\frac{}{\langle e, ks\rangle_p \xrightarrow{ks'(\,\mathbf{emb}_{\,\triangle}\ k)} \langle e, ks\rangle_p} \quad \begin{array}{l} p \notin \pi_1(\Delta), or \\ ks' \not\leq ks \end{array}$$

[Tr-Emb-Transaction]

$$\frac{}{\langle e, ks :: ks'\rangle_p \xrightarrow{ks(\,\mathbf{emb}_{\,\triangle}\ k)} \langle e, (ks :: (k, \langle e, ks\rangle) :: ks'\rangle_p} \quad \begin{array}{l} p \notin \pi_1(\Delta) \\ ks' \in \pi_2(\Delta) \end{array}$$

[Tr-Ignore-T-Emb]

$$\frac{}{\langle e, ks\rangle_p \xrightarrow{ks'(\,\mathbf{emb}_{\,\triangle}\ k)} \langle e, ks\rangle_p} \quad \begin{array}{l} p \notin \pi_1(\Delta) \\ ks' \notin \pi_2(\Delta) \\ ks' :: k \not\leq ks \end{array}$$

[Tr-Co]

$$\frac{}{\langle\,\mathbf{co}\ k, ks\rangle_p \xrightarrow{ks(\,\mathbf{co}\ k)} \langle (), []\rangle}$$

[Tr-Ignore-Co]

$$\frac{}{\langle\,\mathbf{co}\ k, ks\rangle_p \xrightarrow{ks'(\,\mathbf{co}\ k)} \langle\,\mathbf{co}\ k, ks\rangle_p} \quad ks' \not\leq ks$$

[Tr-Broadcast]

$$\frac{P_1 \xrightarrow{ks(\beta)} P_1' \qquad P_2 \xrightarrow{ks(\beta)} P_2'}{P_1 \parallel P_2 \xrightarrow{ks(\beta)} P_1' \parallel P_2'}$$

Table 20: LTS Transaction rules

transaction names in the transition do not match.

Rule [Tr-Commit] simply removes an alternative from the transactions stack, when transactions stack and the list of transaction names in the transition match. The transition is ignore otherwise by rule [Tr-Ignore-Commit].

There are two rules for embedding, one to embed single processes and one to embed transactions. In both cases, the set $\Delta : \mathcal{P}(PID) \times \mathcal{P}(\mathcal{K}^*)$ identifies which processes must be embedded into a transaction either by processes identifier or by matching lists of transaction names in the transaction stack. The two cases are described by rules [Tr-Emb-Process] and [Tr-Emb-Transaction]. Both rules save the transaction ks being embedded on the transactions stack, which thus becomes $(k, \langle e, ks\rangle) :: ks$ for single processes, and $ks :: (k, \langle e, ks\rangle) :: ks'$

for transaction. Rules [TR-IGNORE-P-EMB] and [TR-IGNORE-T-EMB] define in which cases $\mathbf{emb}_{\triangle}\, k$ labels can be ignored.

By rule [TR-CO], a commit point $\mathbf{co}\ k$ can take a $\mathbf{co}\ k$ transition and reduce to a **NIL** process, or ignore any other transition label otherwise by rule [TR-IGNORE-CO].

Lastly, by rule [TR-BROADCAST] processes in parallel composition must take the same β actions. A transition $ks(\beta)$ is *broadcast* to all processes running in parallel.

As already stated in the beginning, the present LTS is work in progress and needs to be finished. Some intuitions about the transactions stack and embedding of processes and transactions have been presented, but need to be investigated in further depth. An implementation of TransCML would surely benefit from the present work, since it presents a distributed version of the reduction semantics. Processes that are part of the same transaction k in the reduction semantics have two separate transactions stack in the LTS, but the transaction name k occurs in both of them. It would be easy then to implement TransCML processes as standard UNIX process or threads, equipped with a transactions stack. An implementation will also have to provide a scheduler, in order to decide into which transactions processes are to be embedded and which transactions are to be aborted. These problems will have to be addressed in future work.

6 Conclusion

We have introduced TransCML, a concurrent functional language enriched with communicating transactions. Inspired to TransCCS, we have shown its reduction semantics and explained the rationale behind its design. We have explored its expressiveness with several examples, and provided a draft of a Labelled Transition System to better capture the distributed behaviour of TCML processes.

Much work remains to be done on TCML. First of all, the drafted Labelled Transition System needs to be completed and defined more formally. It will be necessary to verify that the LTS semantics does actually reflect the reduction semantics presented so far. Namely, we want to prove that if a process P takes a step into process P' in the reduction semantics, then process Q , where $Q = f(P)$ given an appropriate translation function from TCML syntax to the extended LTS syntax, can also take a step into Q' in the LTS semantics, such that $Q' = f(P')$, and vice versa.

After the LTS is defined, it would be interesting to define an equational theory on TCML, to find an appropriate characterization and establish interesting equivalences between transactions. It would certainly be interesting to verify whether the equivalences defined for TransCCS in [4] and [5] still hold in TransCML. This direction of work would justify program transformations to optimize an implementation of TransCML.

Finding an efficient implementation for TransCML is another great challenge, in order to asses the usefullness of TransCML in practice. The LTS might guide such an implementation, given that it more accurately reflects the behaviour of TCML processes; it would make it easier to verify the correctness of the implementation and its faithfulness to the reduction semantics. An essential component for an efficient implementation would surely be a *transaction scheduler*. As we have shown in the three-way rendezvous in section 4.2, there are few computation paths that lead to a commit point, but there are many paths leading to states where the only option is to abort some transaction. Ideally, a transaction scheduler would embed or abort transactions so that it can maximize the number of committed transactions and minimize the number of aborted transactions. In this regard, it would also be interesting to develop larger scale applications to verify both the efficiency of the scheduler and to identify novel and useful programming idioms allowed by communicating transactions.

References

[1] Roberto Bruni, Hernán Melgratti, and Ugo Montanari. cjoin: Join with communicating transactions.

[2] Michael Butler, Tony Hoare, and Carla Ferreira. 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, 2005. Springer-Verlag.

[3] Tushar Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *In Proc. of PODC*, pages 398–407. ACM Press, 2007.

[4] Edsko de Vries, Vasileios Koutavas, and Matthew Hennessy. Communicating transactions. In *Proceedings of the 21st international conference on Concurrency theory*, CONCUR'10, pages 569–583, Berlin, Heidelberg, 2010. Springer-Verlag.

[5] Edsko De Vries, Vasileios Koutavas, and Matthew Hennessy. Liveness of communicating transactions. In *Proceedings of the 8th Asian conference on Programming languages and systems*, APLAS'10, pages 392–407, Berlin, Heidelberg, 2010. Springer-Verlag.

[6] Kevin Donnelly and Matthew Fluet. Transactional events. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, ICFP '06, pages 124–135, New York, NY, USA, 2006. ACM.

[7] J. Field and C. A. Varela. Transactors: A programming model for maintaining globally consistent distributed state in unreliable environments. In *ACM Conference on Principles of Programming Languages (POPL 2005)*, pages 195–208, Long Beach, CA, January 2005.

[8] Dan Grossman. 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, 2007. ACM.

[9] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. *Commun. ACM*, 51(8):91–100, August 2008.

[10] Matthew Hennessy. *Algebraic theory of processes*. MIT Press series in the foundations of computing. MIT Press, 1988.

[11] Alan Jeffrey. Semantics for core concurrent ml using computation types. In *Higher Order Operational Techniques in Semantics, Proceedings*. Cambridge University Press, 1997.

[12] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems.* Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[13] Leslie Lamport. Paxos made simple, fast, and byzantine. In *OPODIS*, pages 7–9, 2002.

[14] Mohsen Lesani and Jens Palsberg. Communicating memory transactions. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 157–168, New York, NY, USA, 2011. ACM.

[15] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[16] John H. Reppy. *Concurrent programming in ML.* Cambridge University Press, New York, NY, USA, 1999.

[17] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.

[18] Lukasz Ziarek, Philip Schatz, and Suresh Jagannathan. Stabilizers: a modular checkpointing abstraction for concurrent functional programs. In *ICFP*, pages 136–147, 2006.