Dependent Session Types for Verified Concurrent Programming

ANONYMOUS AUTHOR(S)

We present TLL_C which extends the Two-Level Linear dependent type theory (TLL) with session type based concurrency. Equipped with Martin-Löf style dependency, the session types of TLL_C allow protocols to specify the properties of communicated messages. When used in conjunction with the dependent type machinery already present in TLL, dependent session types facilitate a form of relational verification by relating concurrent programs with their idealized sequential counterparts. Correctness properties proven for sequential programs can now be easily lifted to their corresponding concurrent programs. Session types now become a powerful tool for intrinsically verifying the correctness of data structures such as queues and concurrent algorithms such as map-reduce. To extend TLL with session types, we develop a novel formulation of intuitionistic session type which we believe to be widely applicable for integrating session types into other type systems beyond the context of TLL_C. We study the meta-theory of our language, proving its soundness as both a term calculus and a process calculus. A prototype compiler which compiles TLL_C programs into concurrent C code is implemented and freely available.

Additional Key Words and Phrases: dependent types, linear types, session types, concurrency

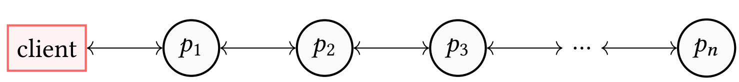
1 Introduction

Session types [23] are an effective typing discipline for coordinating concurrent computation. Through type checking, processes are forced to adhere to communication protocols and maintain synchronization. This allows session type systems to statically rule out runtime bugs for concurrent programs similarly to how standard type systems rule out bugs for sequential programs. While (simple) session type systems guarantee concurrent programs do not crash catastrophically, it remains difficult to write concurrent programs which are semantically correct.

Consider the Pfenning-style concurrent queue which is a common data structure encountered in the session type literature. A queue is described by the following type:

$$\mathsf{queue}_A := \&\{\mathsf{ins} : A \multimap \mathsf{queue}_A, \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A \otimes \mathsf{queue}_A\}\}$$

The following diagram illustrates the channel topology of a client interacting with a queue server.



Each of the p_i nodes here represents a queue cell which holds a value and these nodes are linked together by channels of type queue_A. As indicated by the type constructor &, the first queue node q_1 first receives either an ins or a del label from the client. In the case of an ins label, p_1 receives a value v of type A (indicated by $\multimap$) from the client. The p_1 node then sends an ins label to p_2 and forwards v to it. This forwarding process repeats until the value reaches the end of the queue where a new queue cell p_{n+1} is allocated to store v. On the other hand, if p_1 receives a del label, the type constructor $\oplus$ requires that p_1 send either none or some. The none label is sent to signify that the queue is empty and ready to terminate (indicated by $\mathbf{1}$). The some label is sent along with a value of type A (indicated by $\otimes$) which is the dequeued element. Finally, p_1 forwards its channel, connecting to p_2, to the client so that the client may continue interacting with the rest of the queue.

It is clear from the example above that the session type queue_A only lists what operations a queue should support, but does not specify the expected behavior of these operations. For instance, it does not specify that an ins operation should add an element to the back of the queue or that a

2026. ACM 2475-1421/2026/1-ART
https://doi.org/

del operation should return the element at the front of the queue. A correct implementation needs to maintain all of these additional invariants not captured by the session type. In fact, due to the under specification of the queue_A type, it is possible to implement a "queue" which simply ignores all ins messages and always returns none on del.

To address this issue, we develop TLL_C, a dependent session type system which extends the Two-Level Linear dependent type theory (TLL) [18] with session-typed concurrency. In TLL_C, one could define queues through the following dependent session type:

$$
\begin{aligned}
&\mathsf{queue}(xs : \mathsf{list}\ A) := ?(\ell : \mathsf{opr}).\mathbf{match}\ \ell\ \mathbf{with}\\
&\qquad \mid \mathsf{ins}(v) \Rightarrow \mathsf{queue}(\mathsf{snoc}(xs, v))\\
&\qquad \mid \mathsf{del} \Rightarrow \mathbf{match}\ xs\ \mathbf{with}\ (x :: xs') \Rightarrow\ !(\mathsf{sing}\ x).!(\mathbf{hc}\langle \mathsf{queue}(xs')\rangle).\mathbf{1} \mid [\,] \Rightarrow \mathbf{1}
\end{aligned}
$$

Here, the type $\mathsf{queue}(xs)$ is parameterized by a list xs which represents the current contents of the queue. Notice that the type no longer needs the $\oplus$ and $\&$ type constructors to describe branching behavior. Instead, it uses type-level pattern matching to inspect the label ℓ received from the client. The opr type which ℓ inhabits is defined as a simple inductive type with two constructors:

$$
\text{inductive}\ \mathsf{opr} := \mathsf{ins} : A \longrightarrow \mathsf{opr} \mid \mathsf{del} : \mathsf{opr}
$$

When a queue server receives an $\mathsf{ins}(v)$ value, the type of the server becomes $\mathsf{queue}(\mathsf{snoc}(xs, v))$ where snoc appends v to the end of xs. Conversely, when a del label is received, the type-level pattern matching on xs enforces that if the queue is non-empty (i.e. $x :: xs'$ case), then the server must send the front element x of the queue to the client (indicated by the *singleton type* $\mathsf{sing}\ x$) along with the channel $\mathbf{hc}\langle \mathsf{queue}(xs')\rangle$ connecting to the remainder of the queue. If the queue is empty (i.e. [] case), then the server simply terminates.

Given the queue protocol described above, we can construct queue process nodes and interact with them. The following signatures are of helper functions that wrap interactions with the queue nodes into a convenient interface:

$$
\begin{aligned}
&\mathsf{insert}\ : \forall\{xs : \mathsf{list}\ A\}\ (x : A) \longrightarrow \mathsf{Queue}(xs) \longrightarrow \mathsf{Queue}(\mathsf{snoc}(xs, x))\\
&\mathsf{delete} : \forall\{x : A\}\ \{xs : \mathsf{list}\ A\} \longrightarrow \mathsf{Queue}(x :: xs) \longrightarrow C(\mathsf{sing}\ x \otimes \mathsf{Queue}(xs))\\
&\mathsf{free}\quad\ : \mathsf{Queue}([\,]) \longrightarrow C(\mathsf{unit})
\end{aligned}
$$

The Queue type here is a type alias for the *channel type* of queues (explained later in detail) and the C type constructor here is the *concurrency monad* which encapsulates concurrent computations. Notice in the signature of insert and delete that there are dependent quantifiers surrounded by curly braces. These are the *implicit* quantifiers of TLL which indicate that the corresponding arguments are "ghost" values used for type checking and erased prior to runtime. For our purposes here, such ghost values are especially useful for *relationally* specifying the expected behaviors of queue interactions in terms of sequential list operations. For instance, the signature of insert states that the queue obtained after inserting x is related to the original queue by the list operation snoc. Similarly, the signature of delete states that deleting from a non-empty queue returns the front element x. Even though neither of these xs ghost values exists at runtime, they *statically* ensure that concurrent processes implementing these interfaces behave like actual queues, i.e., are first-in-first-out data structures. In a later section we will show how a generalized map-reduce algorithm can be implemented and verified using similar techniques.

Integrating session typed based concurrency into TLL is non-trivial due to the fact that TLL is a dependently typed functional language. While prior works [19, 45] have successfully combined *classical* session types with functional languages, it is well known that classical session types do not easily support recursive session types [20] (needed to express our queue type). The main issue is that classical session types are defined in terms of a *dual* operator which does not easily commute with recursive type definitions. The addition of arbitrary type-level computations through dependent types further complicates this matter. On the other hand, *intuitionistic* session types [12] eschew the dual operator and define dual *interpretations* of session types based on their *left* or *right* sequent rules. Because intuitionistic session types do not rely on a dual operator, they are able to support recursive session types without commutativity issues. However, intuitionistic session types are often formulated in the context of process calculi without a functional layer. To enjoy the benefits of intuitionistic session types in a functional setting, we develop a novel form of intuitionistic session types where we separate the notion of *protocols* from *channel types*. The $\text{queue}(xs)$ type from before is, in actuality, a protocol whereas $\mathbf{hc}\langle\text{queue}(xs)\rangle$ is a channel type. In general, a channel type is formed by applying the $\mathbf{ch}\langle\cdot\rangle$ and $\mathbf{hc}\langle\cdot\rangle$ type constructors to protocols. These constructors provide dual interpretations to protocols, allowing dual channels of the same protocol to be connected together. For example, $!A.P$ would be interpreted dually as follows:

$$\mathbf{ch}\langle !A.P\rangle \quad \text{(send message of type } A\text{)}$$
$$\mathbf{hc}\langle !A.P\rangle \quad \text{(receive message of type } A\text{)}$$

Such channel types can be naturally included into the contexts of functional type systems without needing to instrument the underlying language into a sequent calculus formulation. We believe our treatment of intuitionistic session types is not specific to TLL_C and is widely applicable for integrating intuitionistic session types with other functional languages.

In order to show that TLL_C ensures communication safety, we develop a process calculus based concurrency semantics. Process configurations in the calculus are collections of TLL_C programs interconnected by channels. At runtime, individual processes are evaluated using the program semantics of base TLL. When two processes at opposing ends (i.e. dually typed) of a channel are synchronized and ready to communicate, the process level semantics transmits their messages across the channel. We study the meta-theory of TLL_C and prove that it is indeed sound at both the level of terms and at the level of process configurations.

We implement a prototype compiler for compiling TLL_C programs into safe C code. The compiler implements advanced language features such as dependent pattern matching and type inference. The unique ownership property of linear types also facilitates optimizations such as in-place programming [27]. All examples presented in this paper can be compiled using our prototype compiler where concurrent processes are implemented as POSIX threads. The compiler source code, and example TLL_C programs are available in our git repository[1].

In summary, we make the following contributions:

- We extend the Two-Level Linear dependent type theory (TLL) with session type based concurrency, forming the language of TLL_C. TLL_C inherits the strengths of TLL such as Martin-Löf style linear dependent types and the ability to control program erasure.
- We develop a novel formulation of intuitionistic session types through a clear separation of protocols and channel types. We believe this formulation to be widely applicable for integrating session types into other functional languages.
- We study the meta-theoretical properties of TLL_C. We show that TLL_C, as a term calculus, possesses desirable properties such as confluence and subject reduction and, as a process calculus, guarantees communication safety.
- We implement a prototype compiler which compiles TLL_C into safe and efficient C code. The compiler implements additional features such as dependent pattern matching, type inference and in-place programming for linear types.

[1] **TODO**

2 Overview of Dependent Session Types

Session types in TLL_C are *minimalistic* in design and yet surprisingly expressive due to the presence of dependent types. Through examples, we provide an overview of how dependent session types facilitate verified concurrent programming in TLL_C.

2.1 Message Specification

An obvious, but important, use of dependent session types is the precise specification of message properties communicated between parties. This is useful in practical network systems where the content of messages may depend on the value of a prior request. Consider the following protocol:

$$!(sz : \mathsf{nat}).\ ?(msg : \mathsf{bytes}).\ ?\{\mathsf{sizeOf}(msg) = sz\}.\ \mathbf{1}$$

Informally speaking, this protocol first expects a natural number *sz* to be sent followed by receiving a byte string *msg*. In simple session type systems without dependency, there would be no way of specifying the relationship between *sz* and *msg*. However, dependent session types allow us to express relations between messages. Notice in the third interaction expected by the protocol, the party sending *msg* must provide a *proof* that the size of *msg* is indeed *sz* according to an agreed upon sizeOf function. Finally, the protocol terminates with **1** and communication ends. Notice that the proof here, as indicated by the curly braces, is a *ghost message*: it is used for type checking and erased prior to runtime. Even though the proof does not participate in actual communication, the necessity for the sender of *msg* to provide such a proof ensures that the protocol is followed.

This example showcases the main primitives for constructing dependent protocols in TLL_C: the $!(x : A).B$ and $?(x : A).B$ *protocol actions*. The syntax of these constructs takes inspiration from binary session types [19, 45] and label dependent session types [38], however the meaning of these constructs in TLL_C is subtly different. In prior works, the ! marker indicates that the channel is to send and the ? marker indicates that the channel is to receive. In TLL_C, neither marker expresses sending or receiving per se, but rather an abstract action that needs to be interpreted through a *channel type*. Hence, the description of the messaging protocol above is stated to be informal. To assign a precise meaning to the protocol, we need to view it through the lenses of channel types:

$$\mathbf{ch}\langle !(sz : \mathsf{nat}).\ ?(msg : \mathsf{bytes}).\ ?\{\mathsf{sizeOf}(msg) = sz\}.\ \mathbf{1}\rangle$$
$$\mathbf{hc}\langle !(sz : \mathsf{nat}).\ ?(msg : \mathsf{bytes}).\ ?\{\mathsf{sizeOf}(msg) = sz\}.\ \mathbf{1}\rangle$$

Here, these two channel types are constructed using *dual* channel type constructors: $\mathbf{ch}\langle\cdot\rangle$ and $\mathbf{hc}\langle\cdot\rangle$. The $\mathbf{ch}\langle\cdot\rangle$ constructor interprets ! as sending and ? as receiving while the $\mathbf{hc}\langle\cdot\rangle$ constructor interprets ! as receiving and ? as sending. In general, dual channel types interpret protocols in opposite ways. These constructors act similarly to the duality of left and right rules for intuitionistic session types [12]. Unlike intuitionistic session types which require the base type system to be based on sequent calculus, our channel types can be integrated into the type systems of functional languages so long as linear types are supported.

2.2 Dependent Ghost Secrets

Dependent ghost messages have interesting applications when it comes to message specification. Consider the following encoding of a idealized Shannon cipher protocol:

$$H(E, D) := \forall\{k : \mathcal{K}\}\ \{m : \mathcal{M}\} \to D(k, E(k, m)) =_{\mathcal{M}} m \qquad \text{(correctness property)}$$
$$\mathcal{E}(E, D) := !\{k : \mathcal{K}\}.\ !\{m : \mathcal{M}\}.\ !(c : C).\ !\{H(E, D) \times (c =_C E(k, m))\}.\ \mathbf{1}$$

Given public encryption and decryption functions $E : \mathcal{K} \times \mathcal{M} \to C$ and $D : \mathcal{K} \times C \to \mathcal{M}$ respectively, the protocol $\mathcal{E}(E, D)$ begins by sending ghost messages: key k of type $\mathcal{K}$ and message m of type $\mathcal{M}$. Next, the ciphertext c of type C, indicated by round parenthesis, is actually sent to the client. Finally, the last ghost message sent is a proof object witnessing the correctness property of the protocol: c is obtained by encrypting m with key k. Observe that for the overall protocol, *only* ciphertext c will be sent at runtime while the other messages (secrets) are erased. The Shannon cipher protocol basically forces communicated messages to always be encrypted and prevents the accidental leakage of plaintext.

It is important to note that ghost messages and proof specifications, by themselves, are *not* sufficient to guaranteeing semantic security. An adversary can simply use a different programming language and circumvent the proof obligations imposed by TLL_C. However, these obligations are useful in ensuring that honest parties correctly follow *trusted* protocols to defend against attackers. For example, in the Shannon cipher protocol above, an honest party is required by the type system to send a ciphertext that is indeed encrypted from the (trusted) algorithm E.

Another, more concrete, example of using ghost messages to specify secrets is the Diffie-Hellman key exchange [17] protocol defined as follows:

$$\begin{aligned}\mathsf{DH}(p\ g : \mathsf{int}) := {} & !\{a : \mathsf{int}\}.\ !(A : \mathsf{int}).\ !\{A = \mathsf{powm}(g, a, p)\}.\\ & ?\{b : \mathsf{int}\}.\ ?(B : \mathsf{int}).\ ?\{B = \mathsf{powm}(g, b, p)\}.\ \mathbf{1}\end{aligned}$$

The DH protocol is parameterized by publicly known integers p and g. Without loss of generality, we refer to the message sender for the first row of the protocol as Alice and the message sender for the second row as Bob. From Alice's perspective, she first sends her secret value a as a dependent ghost message to initialize her half of the protocol. Next, her public value A is sent as a real message to Bob along with a proof that A is correctly computed from values p, g and a (using modular exponentiation powm). At this point, Alice has finished sending messages and waits for message from Bob to complete the key exchange. She first "receives" Bob's secret b as a ghost message which initializes Bob's half of the protocol. Later, Bob' public value B is received as a real message along with a proof that B is correctly computed from p, g and b. Notice that between Alice and Bob, only the real messages A and B will be exchanged at runtime. The secret values a and b and the correctness proofs are all ghost messages that are erased prior to runtime. Basically, the DH protocol forces communication between Alice and Bob to be encrypted and maintain secrecy.

```
def Alice (a p g : int) (c : ch⟨DH(p, g)⟩)
 : C(unit) :=
  let c ⇐ send c {a} in
  let c ⇐ send c (powm(g, a, p)) in
  let c ⇐ send c {refl} in
  let ⟨{b}, c⟩ ⇐ recv c in
  let ⟨B, c⟩ ⇐ recv c in
  let ⟨{pf}, c⟩ ⇐ recv c in
  close(c)
```

```
def Bob (b p g : int) (c : hc⟨DH(p, g)⟩)
 : C(unit) :=
  let ⟨{a}, c⟩ ⇐ recv c in
  let ⟨A, c⟩ ⇐ recv c in
  let ⟨{pf}, c⟩ ⇐ recv c in
  let c ⇐ send c {b} in
  let c ⇐ send c (powm(g, b, p)) in
  let c ⇐ send c {refl} in
  wait(c)
```

The DH key exchange protocol can be implemented through two simple monadic programs Alice and Bob as shown above. The C type constructor here is the concurrency monad for integrating the *effect* of concurrent communication with the *pure* functional core of TLL_C. There are two kinds of send (and respectively recv) operations at play here. The first kind, indicated by **send** c $\{v\}$ is for sending a ghost message v on channel c. After type checking, these ghost sends are compiled to no-ops so that they do not participate in runtime communication. The second kind, indicated by **send** c (v), is for sending a real message v on channel c. These real sends are compiled to actual messages in the generated code. Finally, the close and wait operations synchronize the termination of the protocol. Notice that the duality of channel types $\mathbf{ch}\langle\mathsf{DH}(p, g)\rangle$ and $\mathbf{hc}\langle\mathsf{DH}(p, g)\rangle$ ensures that every send in Alice is matched by a corresponding receive in Bob and vice versa. Moreover, Alice and Bob are enforced by the type checker to correctly carry out the key exchange.

3 Relational Verification via Dependent Session Types

Earlier in the introduction section, we showed a sketch of how dependent session types can be used for verified concurrent programming through the example of a concurrent queue. In this section, we provide a detailed account of how we can use dependent session types to construct a generic map-reduce system. Similarly to the queue example, we will verify the correctness of the map-reduce system by relating it to sequential operations on trees.

3.1 Construction of Map-Reduce

Map-reduce is a commonly used programming model for processing large data sets in parallel. Initially, map-reduce creates a tree of concurrently executing workers as illustrated in Figure 1. The client partitions the data into smaller chunks and sends them to the leaf workers of the tree. Next, each leaf worker applies a user-specified function f to each of its received data chunks and sends the results to its parent worker. When an internal worker receives results from its children, it combines the results using another user-specified binary function g. This procedure continues until the root worker computes the final result and sends it back to the client. Due to the fact that workers without data dependencies can operate concurrently, the overall system can achieve significantly better performance than sequential implementations of the same operations.

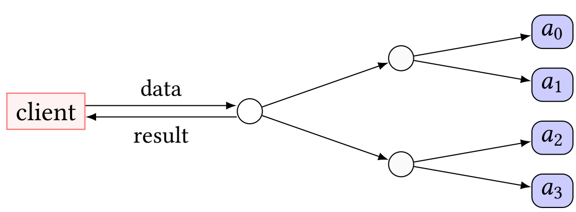


Fig. 1. Tree Diagram of Map-Reduce

The first step in constructing the map-reduce system is to build a model of our desired computation in a sequential setting. For this purpose, we define a simple binary tree inductive type:

$$\textsf{inductive tree } (A : \mathsf{U}) := \ \mathsf{Leaf} : A \to \mathsf{tree}(A) \mid \ \mathsf{Node} : \mathsf{tree}(A) \to \mathsf{tree}(A) \to \mathsf{tree}(A)$$

$$\begin{aligned}&\textsf{def map} : \forall\{A\ B : \mathsf{U}\}\ (f : A \to B) \to \mathsf{tree}(A) \to \mathsf{tree}(B)\\&\quad \mid\ \mathsf{Leaf}\ x \Rightarrow \mathsf{Leaf}\ (f\ x)\\&\quad \mid\ \mathsf{Node}\ l\ r \Rightarrow \mathsf{Node}\ (\mathsf{map}\ f\ l)(\mathsf{map}\ f\ r)\end{aligned}$$

$$\begin{aligned}&\textsf{def reduce} : \forall\{A\ B : \mathsf{U}\}\ (f : A \to B)\ (g : B \to B \to B) \to \mathsf{tree}(A) \to B\\&\quad \mid\ \mathsf{Leaf}\ x \Rightarrow f\ x\\&\quad \mid\ \mathsf{Node}\ l\ r \Rightarrow g\ (\mathsf{reduce}\ f\ g\ l)\ (\mathsf{reduce}\ f\ g\ r)\end{aligned}$$

In this definition, the type U of A is the universe of *unbound* (i.e. non-linear) types in TLL_C. So tree is parameterized by A which represents the type of data stored at the leaf nodes. The *sequential* map and reduce functions for tree are all defined in a standard way.

To construct the concurrent map-reduce system, the protocol of map-reduce must be able to branch depending on what operation the client requests to perform. Unlike many prior session type systems [12, 16] which provide built-in constructs (e.g. ⊕ and &) for internal and external choice, we implement branching protocols using just dependent protocols and type-level pattern matching on sent or received messages. For our map-reduce system, we define the kinds of operations that can be performed through the inductive type opr:

$$\begin{aligned}\textsf{inductive}\ \mathsf{opr}(A : \mathsf{U}) := \ & \mathsf{Map} : \forall\{B : \mathsf{U}\}\ (f : A \to B) \to \mathsf{opr}(A)\\ & \mid \mathsf{Reduce} : \forall\{B : \mathsf{U}\}\ (f : A \to B)\ (g : B \to B \to B) \to \mathsf{opr}(A)\\ & \mid \mathsf{Free} : \mathsf{opr}(A)\end{aligned}$$

The opr type has three constructors:

- Map f represents a map operation that applies the function $f : A \to B$ to each element of type A and produces results of type B.
- Reduce $f\ g$ represents a reduce operation that first applies the function $f : A \to B$ to each element of type A and then combines the results using the binary function $g : B \to B \to B$.
- Free is the command that terminates the concurrent tree.

We are now ready to define the following treeP protocol to describe the interactions between nodes in the map-reduce tree.

$$\begin{aligned}&\textsf{def}\ \mathsf{treeP}\ (A : \mathsf{U})\ (t : \mathsf{tree}\ A) := ?(\ell : \mathsf{opr}\ A).\\ &\quad \textbf{match}\ \ell\ \textbf{with}\ \mathsf{Map}\ \_\ f \Rightarrow\ \mathsf{treeP}\ B\ (\mathsf{map}\ f\ t)\\ &\qquad\qquad\qquad \mid \mathsf{Reduce}\ \_\ f\ g \Rightarrow\ !(\mathsf{sing}\ (\mathsf{reduce}\ f\ g\ t)).\ \mathsf{treeP}\ t\\ &\qquad\qquad\qquad \mid \mathsf{Free} \Rightarrow \mathbf{1}\end{aligned}$$

For each node n in the concurrent tree, it will be providing a channel of type $\mathbf{ch}\langle\mathsf{treeP}\ A\ t\rangle$ to its parent. The parameter t of type tree A represents the shape of the sub-tree rooted at n. The treeP protocol states node n will receive a message ℓ of type opr A from its parent. The protocol then branches, via type-level pattern matching on ℓ, into three cases. If ℓ is of the form Map f, then n will continue the protocol as treeP B (map $f\ t$). Notice that the type parameter of treeP is changed from A to B to reflect the fact that the data stored at the leaves of the sub-tree is transformed from type A to type B. Furthermore, the shape of the sub-tree has also changed from t to map $f\ t$. In the second case where ℓ is of the form Reduce $f\ g$, n will first send the result of type sing (reduce $f\ g\ t$) to its parent. The type sing x is the *singleton type* whose sole inhabitant is the element x. After sending the result, n will continue the protocol as treeP t, i.e. remains unchanged. Finally, n will terminate the protocol when ℓ is Free.

Using the treeP protocol, we implement the processes that run at each leaf of the concurrent tree. We have elided uninteresting technical details regarding dependent pattern matching.

$$\begin{aligned}&\textsf{def}\ \mathsf{leafWorker}\ \{A : \mathsf{U}\}\ (x : A)\ (c : \mathbf{ch}\langle\mathsf{treeP}\ A\ (\mathsf{Leaf}\ x)\rangle) : C(\mathsf{unit}) :=\\ &\quad \textsf{let}\ \langle\ell, c\rangle := \mathbf{recv}\ c\ \textsf{in}\\ &\quad \textbf{match}\ \ell\ \textbf{with}\\ &\quad \mid \mathsf{Map} \Rightarrow \mathsf{leafWorker}\ \{B\}\ (f\ x)\ c\\ &\quad \mid \mathsf{Reduce} \Rightarrow \textsf{let}\ c \Leftarrow \mathbf{send}\ c\ (\mathsf{Just}\ (f\ x))\ \textsf{in}\ \mathsf{leafWorker}\ \{A\}\ x\ c\\ &\quad \mid \mathsf{Free} \Rightarrow \mathbf{close}(c)\end{aligned}$$

The leafWorker function takes two non-ghost arguments: a data element x of type A and a channel c of type $\mathbf{ch}\langle\mathsf{treeP}\ A\ (\mathsf{Leaf}\ x)\rangle$. Through this channel c, the leaf worker will receive requests from its parent and provide responses accordingly. For instance, when the leaf worker receives a Map f request, it will apply $f : A \to B$ to its data element x and continue as a leaf worker with the new data element fx. In this case, the type parameter of leafWorker has changed from A to B to reflect the transformation of the data element.

To represent internal node workers we implement the following nodeWorker function. This function takes (non-ghost) channels c_l and c_r of types $\mathbf{hc}\langle\mathsf{treeP}\ A\ l\rangle$ and $\mathbf{hc}\langle\mathsf{treeP}\ A\ r\rangle$ for communicating with its left and right children. Notice that the types of these channels are indexed by ghost values l and r of type tree A which represent the shapes of the concurrent sub-trees providing c_l

and c_r. The nodeWorker communicates with its parent through the channel c whose type is indexed by the ghost value Node l r.

$$
\begin{array}{l}
\textsf{def}\ \textsf{nodeWorker}\ \{A : \textsf{U}\}\ \{l\ r : \textsf{tree}\ A\} \\
\quad\quad (c_l : \mathbf{hc}\langle \textsf{treeP}\ A\ l\rangle)\ (c_r : \mathbf{hc}\langle \textsf{treeP}\ A\ r\rangle)\ (c : \mathbf{ch}\langle \textsf{treeP}\ A\ (\textsf{Node}\ l\ r)\rangle) : C(\textsf{unit}) := \\
\quad \textsf{let}\ \langle \ell, c\rangle := \mathbf{recv}\ c\ \textsf{in} \\
\quad \mathbf{match}\ \ell\ \mathbf{with} \\
\quad \mid \textsf{Map}\ \_\ f \Rightarrow \\
\quad\quad \textsf{let}\ c_l \Leftarrow \mathbf{send}\ c_l\ (\textsf{Map}\ f)\ \textsf{in} \\
\quad\quad \textsf{let}\ c_r \Leftarrow \mathbf{send}\ c_r\ (\textsf{Map}\ f)\ \textsf{in} \\
\quad\quad \textsf{let}\ c \Leftarrow \mathbf{send}\ c\ (\textsf{Just unit})\ \textsf{in} \\
\quad\quad \textsf{nodeWorker}\ \{B\}\ \{(\textsf{map}\ f\ l)\ (\textsf{map}\ f\ r)\}\ c_l\ c_r\ c \\
\quad \mid \textsf{Reduce}\ \_\ f\ g \Rightarrow \\
\quad\quad \textsf{let}\ c_l \Leftarrow \mathbf{send}\ c_l\ (\textsf{Reduce}\ f\ g)\ \textsf{in} \\
\quad\quad \textsf{let}\ c_r \Leftarrow \mathbf{send}\ c_r\ (\textsf{Reduce}\ f\ g)\ \textsf{in} \\
\quad\quad \textsf{let}\ \langle \textsf{Just}\ v_l, c_l\rangle \Leftarrow \mathbf{recv}\ c_l\ \textsf{in} \\
\quad\quad \textsf{let}\ \langle \textsf{Just}\ v_r, c_r\rangle \Leftarrow \mathbf{recv}\ c_r\ \textsf{in} \\
\quad\quad \textsf{let}\ c \Leftarrow \mathbf{send}\ c\ (\textsf{Just}\ (g\ v_l\ v_r))\ \textsf{in} \\
\quad\quad \textsf{nodeWorker}\ \{A\}\ \{l\ r\}\ c_l\ c_r\ c \\
\quad \mid \textsf{Free} \Rightarrow \\
\quad\quad \textsf{let}\ c_l \Leftarrow \mathbf{send}\ c_l\ \textsf{Free}\ \textsf{in} \\
\quad\quad \textsf{let}\ c_r \Leftarrow \mathbf{send}\ c_r\ \textsf{Free}\ \textsf{in} \\
\quad\quad \mathbf{wait}(c_l); \mathbf{wait}(c_r); \mathbf{close}(c)
\end{array}
$$

Given the signature of nodeWorker and the definition of the treeP protocol, it is easy to see that the implementation of nodeWorker is constrained to function exactly as intended. For instance, in the case where nodeWorker receives a Map f request from its parent, the type of c becomes $\mathbf{ch}\langle \textsf{treeP}\ B\ (\textsf{map}\ f\ (\textsf{Node}\ l\ r))\rangle$ which simplifies to $\mathbf{ch}\langle \textsf{treeP}\ B\ (\textsf{Node}\ (\textsf{map}\ f\ l)\ (\textsf{map}\ f\ r))\rangle$. The shapes of the left and right sub-trees after the map operation need to become map f l and map f r respectively. In other words, the type of c forces the nodeWorker process to recursively send the Map f request to both of its children to transform them into sub-trees of type $\mathbf{hc}\langle \textsf{treeP}\ B\ (\textsf{map}\ f\ l)\rangle$ and $\mathbf{hc}\langle \textsf{treeP}\ B\ (\textsf{map}\ f\ r)\rangle$.

3.2 A Verified Interface for Map-Reduce

Now that we have defined both leaf and internal node workers, we can wrap them up into a more convenient interface as presented below.

$$
\begin{array}{l}
\textsf{type}\ \textsf{cTree}\ (A : \textsf{U})\ (t : \textsf{tree}\ A) := C(\mathbf{hc}\langle \textsf{treeP}\ t\rangle) \\[1ex]
\textsf{def}\ \textsf{cLeaf}\ \{A : \textsf{U}\}\ (x : A) : \textsf{cTree}\ A\ (\textsf{Leaf}\ x) := \\
\quad \mathbf{fork}(c : \mathbf{ch}\langle \textsf{treeP}\ A\ (\textsf{Leaf}\ x)\rangle)\ \mathbf{with}\ \textsf{leafWorker}\ x\ c \\[1ex]
\textsf{def}\ \textsf{cNode}\ \{A : \textsf{U}\}\ \{l\ r : \textsf{tree}\ A\}\ (t_l : \textsf{cTree}\ A\ l)\ (t_r : \textsf{cTree}\ A\ r) : \textsf{cTree}\ (\textsf{Node}\ l\ r) := \\
\quad \textsf{let}\ c_l \Leftarrow t_l\ \textsf{in} \\
\quad \textsf{let}\ c_r \Leftarrow t_r\ \textsf{in} \\
\quad \mathbf{fork}(c : \mathbf{ch}\langle \textsf{treeP}\ A\ (\textsf{Node}\ l\ r)\rangle)\ \mathbf{with}\ \textsf{nodeWorker}\ c_l\ c_r\ c
\end{array}
$$

The type alias cTree is defined to aid in the readability of the interface. The wrapper functions cLeaf and cNode respectively create leaf and internal node workers. This is accomplished by *forking* a new process using the **fork** construct of the concurrency monad. In particular, when given a channel type $\mathbf{ch}\langle P\rangle$, the **fork** construct will create a new channel and give one end of it to the caller at type $\mathbf{hc}\langle P\rangle$ and spawn a new process that runs the worker with the other end of the channel at type $\mathbf{ch}\langle P\rangle$. The duality of the channel types allows the caller and the worker to communicate. Using these wrapper functions, one can construct a concurrent tree in virtually the same way as one would construct a sequential tree. For example, the following code constructs a concurrent tree with four leaf nodes containing integers 0, 1, 2 and 3 respectively.

$$\mathsf{cNode}\ (\mathsf{cNode}\ (\mathsf{cLeaf}\ 0)\ (\mathsf{cLeaf}\ 1))\ (\mathsf{cNode}\ (\mathsf{cLeaf}\ 2)\ (\mathsf{cLeaf}\ 3))$$

The type of this expression is rather verbose to write manually as it contains the full shape of the concurrent tree. This is not a problem in practice as constant type arguments (such as the tree shapes here) can almost always be inferred automatically by the type checker.

Finally, we implement the cMap and cReduce functions that provide the map and reduce operations on concurrent trees. These functions are implemented by simply sending the appropriate requests to the root worker of the concurrent tree.

```
def cMap {A B : U} {t : tree A} (f : A → B) (c : cTree A t) : cTree B (map f t) :=
  let c ⇐ c in
  let c ⇐ send c (Map f) in
  return c

def cReduce {A B : U} {t : tree A} (f : A → B) (g : B → B → B) (c : cTree A t) :
    C(sing (reduce f g t) ⊗ cTree A t) :=
  let c ⇐ c in
  let c ⇐ send c (Reduce f g) in
  let ⟨v, c⟩ ⇐ recv cin
  return ⟨v, return c⟩
```

From the type signature of cMap, we can see that it takes a function f and a concurrent tree of type cTree A t and returns a new concurrent tree of type cTree B (map f t). In other words, the type of cMap guarantees that the shape of the concurrent tree is transformed in the same way as its sequential tree model under the map function. Similarly, the cReduce takes a concurrent tree of type cTree $A\ t$ and returns a (linear) pair consisting of the result of type sing (reduce $f\ g\ t$), and the original concurrent tree. The correctness of cReduce is guaranteed by the singleton type of its result: reducing a concurrent tree results in the same value as reducing its sequential tree model.

3.3 Concurrent Mergesort via Map-Reduce

By properly instantiating the map-reduce interface defined previously, we can implement more complex concurrent algorithms. Moreover, dependent session types allow us to easily verify the correctness of these derived concurrent algorithms relationally through their sequential models. As an extended example, we implement a concurrent version of the mergesort algorithm using the map-reduce interface and verify its correctness.

We define sequential msort, as a model of our concurrent implementation, in the usual way using split and merge functions. We will not go into further details regarding the well-founded recursion of msort or the correctness of sorting as these are textbook results [14, 34].

```
def split (xs : list int) : list int × list int := ...
def merge (xs ys : list int) : list int := ...

def msort (xs : list int) : list int := match xs with
  | nil ⇒ nil
  | x :: nil ⇒ x :: nil
  | zs ⇒ let ⟨xs, ys⟩ := split zs in merge (msort xs) (msort ys)
```

Generally, to implement an algorithm using the map-reduce paradigm, one must first decompose the algorithm and data into a form that is amenable to parallelization. For mergesort, the input list can be recursively split into smaller sub-lists which can be processed in parallel. To make this decomposition *explicit*, we define the following splittingTree function that constructs a binary tree representation of how the input list is split by the mergesort algorithm.

$$
\begin{array}{l}
\textsf{def}\ \textsf{splittingTree}\ (xs : \textsf{list int}) : \textsf{tree}\ (\textsf{list int}) := \textbf{match}\ xs\ \textbf{with} \\
\quad \mid\ \textsf{nil} \Rightarrow \textsf{Leaf nil} \\
\quad \mid\ x :: \textsf{nil} \Rightarrow \textsf{Leaf}\ (x :: \textsf{nil}) \\
\quad \mid\ zs \Rightarrow \textsf{let}\ \langle xs, ys \rangle := \textsf{split}\ zs\ \textsf{in}\ \textsf{Node}\ (\textsf{splittingTree}\ xs)\ (\textsf{splittingTree}\ ys)
\end{array}
$$

To apply map-reduce, we need to construct a concurrent representation of its splitting tree with type cTree (list int) (splittingTree xs). While it is tempting to directly convert the result of splittingTree into a concurrent tree by recursively replacing Leaf with cLeaf and Node with cNode, such an approach would require traversing both the input list (to construct the splitting tree) and the resulting tree (to convert it into a concurrent tree). This would lead to a bottleneck in the performance of the overall algorithm as the traversals would be done sequentially without exploiting parallelism. Instead, we define the splittingCTree function that constructs the concurrent splitting tree in a concurrent manner.

$$
\begin{array}{l}
\textsf{def}\ \textsf{splittingCTree}\ (xs : \textsf{list int}) : \textbf{ch}\langle !(\textsf{cTree}\ (\textsf{list int})\ (\textsf{splittingTree}\ xs)).\ \mathbf{1} \rangle \to C(\textsf{unit}) := \\
\quad \textbf{match}\ xs\ \textbf{with} \\
\quad \mid\ \textsf{nil} \Rightarrow \textsf{let}\ c \Leftarrow \textbf{send}\ c\ (\textsf{cLeaf nil})\ \textsf{in}\ \textbf{close}(c);\ \textbf{return}\ () \\
\quad \mid\ x :: \textsf{nil} \Rightarrow \textsf{let}\ c \Leftarrow \textbf{send}\ c\ (\textsf{cLeaf}\ (x :: \textsf{nil}))\ \textsf{in}\ \textbf{close}(c);\ \textbf{return}\ () \\
\quad \mid\ zs \Rightarrow \\
\qquad \textsf{let}\ \langle xs, ys \rangle := \textsf{split}\ zs\ \textsf{in} \\
\qquad \textsf{let}\ c_l \Leftarrow \textbf{fork}(c)\ \textbf{with}\ \textsf{splittingCTree}\ xs\ c\ \textsf{in} \\
\qquad \textsf{let}\ c_r \Leftarrow \textbf{fork}(c)\ \textbf{with}\ \textsf{splittingCTree}\ ys\ c\ \textsf{in} \\
\qquad \ldots
\end{array}
$$

The splittingCTree function takes an additional channel argument c which is used to send back the constructed concurrent tree to its caller. This small change allows the recursive case to fork two new processes to construct the left and right sub-trees in parallel. After both sub-trees have been constructed, the parent process can then combine them into a single concurrent tree using cNode and send it back to its caller. Notice that splittingCTree never calls the sequential splittingTree function and only uses it at the type level to model the concurrent tree being constructed. The complete implementation of splittingCTree can be found in the supplementary materials but is shortened here for brevity.

Now that we have constructed a concurrent splitting tree of our input list, we can apply the cReduce operation instantiated with $f := \lambda(x).x$ and $g :=$ merge to perform merging in parallel. This gives us an output of type

$$C(\textsf{sing}\ (\textsf{reduce}\ (\lambda(x).x)\ \textsf{merge}\ (\textsf{splittingTree}\ xs)) \otimes \textsf{cTree}\ (\textsf{list int})\ (\textsf{splittingTree}\ xs))$$

The singleton value sing (reduce $(\lambda(x).x)$ merge (splittingTree xs)) returned by the monad relationally describes this series of concurrent computations using just sequential operations. This allows us to easily verify the correctness of our concurrent mergesort implementation by proving the following theorem (in the internal logic of TLL) which states that reducing the splitting tree of a list is equivalent to performing mergesort on this list.

$$
\begin{array}{l}
\textsf{theorem}\ \textsf{reduceSplittingTree} : \\
\quad \forall (xs : \textsf{list int}) \to \textsf{reduce}\ (\lambda(x).x)\ \textsf{merge}\ (\textsf{splittingTree}\ xs) = \textsf{msort}\ xs
\end{array}
$$

Using this theorem, we can rewrite the singleton type returned by cReduce to sing (msort xs). In other words, the result of our concurrent mergesort implementation is guaranteed to be exactly the same as that of the sequential mergesort algorithm, thus completing our verification.

The full pipeline of concurrent mergesort is given in the following cMSort function.

```
def cMSort (xs : list int) : C(sing (msort xs)) :=
  let c ⇐ fork(c) with splittingCTree xs c in
  let ⟨ctree, c⟩ ⇐ recv c in wait c;
  let ⟨v, ctree⟩ ⇐ cReduce (λ(x).x) merge ctree in
  let ctree ⇐ send ctree Free in wait ctree;
  return (rewrite[reduceSplittingTree xs] v)
```

4 Formal Theory of Dependent Session Types

4.1 Core TLL

In this section, we give a brief summary of the Two-Level Linear dependent type theory (TLL) [18]. TLL is a dependent type theory that combines Martin-Löf-style dependent types [29] with linear types [21, 43]. Notably, TLL supports *essential linearity* [28] through the use of a stratified "two-level" typing system: the *logical* level and the *program* level. The typing judgments of the two levels are written and organized as follows:

$$\boxed{\Gamma \vdash m : A \text{ (Logical Typing)}} \underset{\text{subjects to verify}}{\overset{\text{provides types}}{\rightleftarrows}} \boxed{\Gamma;\Delta \vdash m : A \text{ (Program Typing)}}$$

First, the *logical* level is a standard dependent type system that supports unrestricted usage of types and terms. The primary purpose of the logical level is to provide typing rules for types which will be used at the logical level. For example, the rules for dependent function type (Π-types) formation are defined at the logical level as follows:

$$\textsc{Explicit-Fun}\ \frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : r}{\Gamma \vdash \Pi_t(x : A).B : t} \qquad \textsc{Implicit-Fun}\ \frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : r}{\Gamma \vdash \Pi_t\{x : A\}.B : t}$$

The symbols s, r, t range over the *sorts* of type universes, i.e. U or L. These sorts are used to classify types into two categories: unrestricted types (A : U) and linear types (A : L). Program level terms which inhabit unrestricted types can be freely duplicated or discarded, while those which inhabit linear types must be used exactly once. Note that this usage restriction is *not* enforced at the logical level as the logical level typing judgment is completely structural. This is safe because the logical level will never be executed at runtime and is only used for type checking and verification. Thus, multiple uses of a linear resource at the logical level will not lead to any runtime errors.

At the program level, the typing judgment $\Gamma;\Delta \vdash m : A$ is used to exclusively type *terms*. In other words, no rules for forming types are defined at the program level. All the types used in Γ, Δ, m and A must be well-formed according to the logical level typing judgment. This typing judgment possesses two contexts: Γ of all variables in scope, and Δ of all variables that are computationally relevant in program m. Context Δ is crucial for enforcing linearity at the program level. For example, consider the λ-abstraction rules:

$$\textsc{Explicit-Lam}\ \frac{\Gamma, x : A; \Delta, x :_s A \vdash m : B \qquad \Delta \triangleright t}{\Gamma; \Delta \vdash \lambda_t(x : A).m : \Pi_t(x : A).B} \qquad \textsc{Implicit-Lam}\ \frac{\Gamma, x : A; \Delta \vdash m : B \qquad \Delta \triangleright t}{\Gamma; \Delta \vdash \lambda_t\{x : A\}.m : \Pi_t\{x : A\}.B}$$

In Explicit-Lam, we can see that the bound variable x is added to both contexts Γ and Δ. This indicates that x is a variable which can be used both logically (in types and ghost values) through Γ, and computationally (in real values) through Δ. On the other hand, in the Implicit-Lam rule, x is only added to Γ but not Δ. This indicates that x is a ghost variable which can only be used logically. A ubiquitous example of ghost variables is type parameters in polymorphic functions. For instance, the polymorphic identity function can be implemented as

$$\lambda_{\mathsf{U}}\{A : \mathsf{U}\}.\lambda_{\mathsf{U}}(x : A).x$$

which has the type $\Pi_{\mathsf{U}}\{A : \mathsf{U}\}.\Pi_{\mathsf{U}}(x : A).A$. Arguments to implicit functions are typed at the logical level, thus allowing polymorphic functions to be instantiated with a type as an argument. Additionally, as demonstrated in the examples of prior sections, ghost variables also facilitate program verification by statically describing abstractions and invariants of program states.

In the two λ-abstraction rules above, the premise $\Delta \triangleright t$ is a simple side condition that states: if $t = \mathsf{U}$, then all variables in Δ must be unrestricted. In other words, the λ-abstractions that can be applied unrestrictedly (with $t = \mathsf{U}$) are not allowed to capture linearly typed variables from Δ. This is similar to the restriction imposed on closures implementing the `Fn` trait (i.e. those that can be called multiple times) in Rust [37] where capturing of mutable references is prohibited. If such a restriction is not imposed, then evaluating a λ-abstraction (that captures a linear variable) twice may lead to unsafe memory accesses such as double frees or use-after-frees.

The application rules for both explicit and implicit functions are as follows:

$$\textsc{Explicit-App}\ \frac{\Gamma; \Delta_1 \vdash m : \Pi_t(x : A).B \qquad \Gamma; \Delta_2 \vdash n : A}{\Gamma; \Delta_1 \uplus \Delta_2 \vdash m\ n : B[n/x]} \qquad \textsc{Implicit-App}\ \frac{\Gamma; \Delta \vdash m : \Pi_t\{x : A\}.B \qquad \Gamma \vdash n : A}{\Gamma; \Delta \vdash m\ \{n\} : B[n/x]}$$

In Explicit-App, the argument n is a real value which must be typed at the program level. The $\uplus$ operator merges the two program context Δ_1 and Δ_2 by contracting unrestricted variables and requiring that linear variables be disjoint, thus preventing the sharing of linear resources. In Implicit-App, the argument n is a ghost value that is typed at the logical level. Due to the fact that ghost values are erased prior to runtime, the program context Δ in the conclusion only tracks the computationally relevant variables used in m. Notice how in Explicit-App, the argument n is substituted into the return type B. This allows types to depend on program level terms regardless of whether they are of linear or unrestricted types.

Usage vs Uniqueness. Compared to other linear dependent type theories [5, 13, 28, 30, 42] which only enforce the linear *usage* of resources, the TLL type system prevents the *sharing* of linear resources as well. This is similar to the subtle distinction between linear logic [21] and bunched implications [31, 32] described by O'Hearn. Consider a linear function f, in the aforementioned dependent type theories, of some type $A \multimap B$. When function f is applied to some argument v of type A, the argument v is guaranteed to be used exactly once in the *body* of f. Notice that this notion of linearity does not guarantee that f has unique access to v. If v was obtain from some !-exponential or ω-quantity (the sharable quantity in graded systems [5, 30]), then there may be other aliases of v which can be used outside of f.

Wadler, in his seminal work [44], made a similar distinction between linearity and uniqueness in the context of functional programming, noting that implicit uses of *promotion* and *dereliction* in linear logic can lead to violations of uniqueness. He coins the term *steadfast types* to refer to type systems that enforce both linearity and uniqueness. In this sense, TLL is steadfast as its *sort-uniqueness* property (i.e. types uniquely inhabit either U or L) prohibits the implicit promotion and dereliction of linear types, thus preventing the sharing of linear resources. The heap semantics [41] of TLL shows that its programs enjoy the *single-pointer* property which is a consequence of uniqueness at runtime. In the context of concurrency, the steadfast type system of TLL makes it especially suitable for integration with session types: linear usage prevents replaying of communication protocols and uniqueness ensures that a communication channel has a single owner.

4.2 Dependent Session Types of TLL_C

In this section, we formally present the dependent session types of TLL_C.

Basic Protocols and Channel Types. The intuitionistic session types of TLL_C are decoupled into *protocols* and *channel types*. The rule for forming protocols is as follows:

$$\textsc{Proto}\ \frac{\Gamma \vdash}{\Gamma \vdash \textbf{proto} : \mathsf{U}} \qquad \textsc{Explicit-Action}\ \frac{\Gamma, x : A \vdash B : \textbf{proto}}{\Gamma \vdash \rho(x : A).\, B : \textbf{proto}} \qquad \textsc{Implicit-Action}\ \frac{\Gamma, x : A \vdash B : \textbf{proto}}{\Gamma \vdash \rho\{x : A\}.\, B : \textbf{proto}} \qquad \textsc{End}\ \frac{\Gamma \vdash}{\Gamma \vdash \mathbf{1} : \textbf{proto}}$$

$$\text{where } \rho \in \{!, ?\}$$

Here, the Proto rule introduces the **proto** type which is the type of all protocols. Note that **proto** is an unrestricted type, thus protocols can be freely duplicated or discarded. The Explicit-Action and Implicit-Action rules form dependent protocols which inhabit the **proto** type. The End rule marks the termination of a protocol.

Once a protocol is defined, we can form channel types using the following rules:

$$\textsc{ChType}\ \frac{\Gamma \vdash A : \textbf{proto}}{\Gamma \vdash \textbf{ch}\langle A \rangle : \mathsf{L}} \qquad \textsc{HcType}\ \frac{\Gamma \vdash A : \textbf{proto}}{\Gamma \vdash \textbf{hc}\langle A \rangle : \mathsf{L}}$$

Notice that the channel type constructors $\textbf{ch}\langle\cdot\rangle$ and $\textbf{hc}\langle\cdot\rangle$ lift protocols, which are unrestricted values, into linear types. This means that channels must be used exactly once. Furthermore, as explained in the previous section, the unique ownership of linear types in TLL ensures that only a single entity has access to a channel at any point in time, thus preventing race conditions.

Recursive Protocols. Recursive protocols can be formed using the $\mu(x : A).m$ construct:

$$\textsc{RecProto}\ \frac{\Gamma, x : A \vdash m : A \qquad A \text{ is an } \textit{arity} \text{ ending on } \textbf{proto} \qquad x \text{ is } \textit{guarded} \text{ by protocol action in } m}{\Gamma \vdash \mu(x : A).m : A}$$

For a $\mu(x : A).m$ term, we require that A be an *arity* ending on **proto**. This prevents μ from introducing logical inconsistencies as it can only be used to construct protocols but not proofs for arbitrary propositions. To ensure that protocols defined through $\mu(x : A).m$ can be productively unfolded, recursive usages of x must be syntactically *guarded* behind a protocol action in m. This enforces the *contractiveness* condition for recursive session types [19]. Both the arity and guardedness conditions are stable under substitution. Due to space limitations, we present the rules of arities and guardedness in the appendix.

The difficulty of integrating recursive protocols in classical session type systems is well documented [20]. The key challenge is to define a suitable *duality* operator that commutes with recursion. The following example is due to Bernardi and Hennessy [8]. Suppose we define a reasonable, but naive, duality operator $(\cdot)^\perp$ which simply flips ! and ? in protocols. For the dual of recursive protocol $\mu X.?X.X$, if we first apply duality and then unfold the recursion, we get:

$$(\mu X.?X.X)^\perp = \mu X.!X.X = !(\mu X.!X.X).(\mu X.!X.X)$$

On the other hand, if we first unfold the recursion and then apply duality, we get:

$$(\mu X.?X.X)^{\perp} = (?(\mu X.?X.X).(\mu X.?X.X))^{\perp} = !(\mu X.?X.X).(\mu X.!X.X)$$

Notice that the resulting protocols do not agree on the type of the sent message. While solutions have been proposed to address this issue [8, 9], they do not generalize to dependent session types due to the presence of arbitrary type-level computation. In TLL_C, the separation of protocols and channels types allows us to sidestep the duality problem entirely. Suppose we define our previously problematic recursive protocol in TLL_C as follows:

$$T \triangleq \mu(X : \textbf{proto}).?(\_ : X).X = ?\big(\_ : \mu(X : \textbf{proto}).?(\_ : X).X\big).\ \mu(X : \textbf{proto}).?(\_ : X).X$$

When viewed through the lens of channel type constructors $\textbf{ch}\langle\cdot\rangle$ and $\textbf{hc}\langle\cdot\rangle$, the actions specified by the unfolded protocol are correctly dual to each other. More specifically, a channel of type $\textbf{ch}\langle T\rangle$ receives a protocol of type T whereas a channel of type $\textbf{hc}\langle T\rangle$ sends a protocol of type T.

Concurrency Monad. Concurrency is integrated into the pure functional core of TLL through a concurrency monad C. The basic components of the monad are given in the following rules.

$$\textsc{CType}\ \frac{\Gamma \vdash A : s}{\Gamma \vdash C(A) : \mathsf{L}} \qquad \textsc{Return}\ \frac{\Theta; \Gamma; \Delta \vdash m : A}{\Theta; \Gamma; \Delta \vdash \textbf{return}\ m : C(A)} \qquad \textsc{Bind}\ \frac{\Gamma \vdash B : s \qquad \Theta_1; \Gamma; \Delta_1 \vdash m : C(A) \qquad \Theta_2; \Gamma, x : A; \Delta_2 \vdash n : C(B)}{\Theta_1 \cup \Theta_2; \Gamma; \Delta_1 \cup \Delta_2 \vdash \textbf{let}\ x \Leftarrow m\ \textbf{in}\ n : C(B)}$$

To reason about the communication channels that will appear at *runtime*, the program level typing judgment is extended to include a *channel context* Θ which tracks the channels used by the program. It is crucial to understand that the channel context is largely a technical device for analyzing the type safety of TLL_C. Prior to runtime, the channel context is empty as no channels have been created. Programming is carried out using normal variables in Δ. At runtime, channels will be created and substituted for appropriate variables in Δ. It is these runtime channels that occupy the channel context Θ and are typed as follows:

$$\textsc{Channel-CH}\ \frac{\Gamma; \Delta \vdash \qquad \epsilon \vdash A : \textbf{proto} \qquad \Delta \triangleright \mathsf{U}}{c :_{\mathsf{L}} \textbf{ch}\langle A\rangle; \Gamma; \Delta \vdash c : \textbf{ch}\langle A\rangle} \qquad \textsc{Channel-HC}\ \frac{\Gamma; \Delta \vdash \qquad \epsilon \vdash A : \textbf{proto} \qquad \Delta \triangleright \mathsf{U}}{c :_{\mathsf{L}} \textbf{hc}\langle A\rangle; \Gamma; \Delta \vdash c : \textbf{hc}\langle A\rangle}$$

The protocol A used in the channel types here must be *closed*. This is because channels at runtime must follow fully concretized protocols. The Γ and Δ contexts are allowed to be non-empty for the purely technical reason of facilitating proofs for renaming and substitution lemmas.

As explained in Section 2.1, the protocol actions $!(x : A).B$ and $?(x : A).B$ are abstract constructs that need to be interpreted through channel types. Since $\textbf{ch}\langle\cdot\rangle$ and $\textbf{hc}\langle\cdot\rangle$ interpret protocol actions in opposite ways, we only present the typing rules for $\textbf{ch}\langle\cdot\rangle$ below.

$$\textsc{Explicit-Send-CH}\ \frac{\Theta; \Gamma; \Delta \vdash m : \textbf{ch}\langle !(x : A).\ B\rangle}{\Theta; \Gamma; \Delta \vdash \textbf{send}\ m : \Pi_{\mathsf{L}}(x : A).C(\textbf{ch}\langle B\rangle)} \qquad \textsc{Explicit-Recv-CH}\ \frac{\Theta; \Gamma; \Delta \vdash m : \textbf{ch}\langle ?(x : A).\ B\rangle}{\Theta; \Gamma; \Delta \vdash \textbf{recv}\ m : C(\Sigma_{\mathsf{L}}(x : A).\textbf{ch}\langle B\rangle)}$$

$$\textsc{Implicit-Send-CH}\ \frac{\Theta; \Gamma; \Delta \vdash m : \textbf{ch}\langle !\{x : A\}.\ B\rangle}{\Theta; \Gamma; \Delta \vdash \underline{\textbf{send}}\ m : \Pi_{\mathsf{L}}\{x : A\}.C(\textbf{ch}\langle B\rangle)} \qquad \textsc{Implicit-Recv-CH}\ \frac{\Theta; \Gamma; \Delta \vdash m : \textbf{ch}\langle ?\{x : A\}.\ B\rangle}{\Theta; \Gamma; \Delta \vdash \underline{\textbf{recv}}\ m : C(\Sigma_{\mathsf{L}}\{x : A\}.\textbf{ch}\langle B\rangle)}$$

For the Explicit-Send-CH rule, a channel of type $\textbf{ch}\langle !(x : A).B\rangle$ is applied to the **send** operator. This produces a function which takes a real value v of type A and returns a concurrent computation of type $C(\textbf{ch}\langle B[v/x]\rangle)$ which represents the continuation of the protocol after sending a real value of type A. When this monadic value is bound by rule Bind and executed at runtime, the value v will be sent on channel m. The dual Explicit-Recv-HC rule, as shown here,

$$\textsc{Explicit-Recv-HC}\quad \frac{\Theta; \Gamma; \Delta \vdash m : \mathbf{hc}\langle !(x : A).\ B\rangle}{\Theta; \Gamma; \Delta \vdash \mathbf{recv}\ m : C(\Sigma_{\mathsf{L}}(x : A).\mathbf{hc}\langle B\rangle)}$$

receives on a channel of type $\mathbf{hc}\langle !(x : A).B\rangle$, which produces a (monadic) dependent pair (similarly to Explicit-Recv-CH). The first component of the pair is the value of type A that is received, and the second component is a channel of type $\mathbf{hc}\langle B[v/x]\rangle$ representing the continuation of the protocol. Notice that, due to the linearity of the C monad, all of the intermediate monadic values are guaranteed to be bound by the Bind rule and executed.

The implicit send and receive rules are similar to their explicit counterparts, except that they send and receive ghost values instead of real values. This distinction manifests by having the $\underline{\mathbf{send}}$ and $\underline{\mathbf{recv}}$ operators produce implicit functions and implicit pairs respectively. When the implicit function of Implicit-Send-CH is applied to a ghost argument using Implicit-App (Section 4.1), the ghost argument will be erased prior to runtime. Similarly, the first component of the implicit pair produced by Implicit-Recv-CH is also an erased ghost value. The underlying type system of TLL ensures that these ghost values will only be used logically, thus are safe to erase.

The last communication rules govern the creation and termination of channels:

$$\textsc{Fork}\quad \frac{\Theta; \Gamma, x : \mathbf{ch}\langle A\rangle; \Delta, x :_{\mathsf{L}} \mathbf{ch}\langle A\rangle \vdash m : C(\mathsf{unit})}{\Theta; \Gamma; \Delta \vdash \mathbf{fork}\ (x : \mathbf{ch}\langle A\rangle)\ \mathbf{with}\ m : C(\mathbf{hc}\langle A\rangle)}$$

$$\textsc{Close}\quad \frac{\Theta; \Gamma; \Delta \vdash c : \mathbf{ch}\langle \mathbf{1}\rangle}{\Theta; \Gamma; \Delta \vdash \mathbf{close}\ c : C(\mathsf{unit})} \qquad \textsc{Wait}\quad \frac{\Theta; \Gamma; \Delta \vdash c : \mathbf{hc}\langle \mathbf{1}\rangle}{\Theta; \Gamma; \Delta \vdash \mathbf{wait}\ c : C(\mathsf{unit})}$$

Close and Wait are simple rules used to free channels whose protocols have terminated. The Fork rule is used for creating a child process which concurrently executes the monadic computation m. The child process is provided with a fresh channel of type $\mathbf{ch}\langle A\rangle$ which is bound to the variable x in m. Dually, the parent process obtains the channel endpoint of type $\mathbf{hc}\langle A\rangle$, which can be used to communicate with the spawned process. Note that the newly spawned process m is allowed to capture pre-existing channels from Θ and program variables from Δ. Compared to intuitionistic session type systems based on the sequent calculus [12, 16, 33], the $\mathbf{ch}\langle A\rangle$ channel handed to the child process behaves like the right-hand side of a sequent (i.e. the *provided* channel), while the $\mathbf{hc}\langle A\rangle$ channel handed to the parent process behaves like the left-hand side of a sequent (i.e. the *consumed* channels). Essentially, we have embedded intuitionistic session types into a functional language without needing to reorganize the underlying type system into a sequent calculus formulation.

5 Semantics and Meta-Theory

5.1 Process Configurations

In the previous section, we have presented the typing rules for TLL_C terms which form individual processes. To compose multiple processes together, we introduce the process level typing judgment $\Theta \Vdash P$ below. This judgment formally states that a configuration of processes P is well-typed under the context Θ, which tracks the channels used by the processes in P at runtime.

$$\textsc{Expr}\quad \frac{\Theta; \epsilon; \epsilon \vdash m : C(\mathsf{unit})}{\Theta \Vdash \langle m\rangle} \qquad \textsc{Par}\quad \frac{\Theta_1 \Vdash P_1 \qquad \Theta_2 \Vdash P_2}{\Theta_1 \cup \Theta_2 \Vdash P_1 \mid P_2} \qquad \textsc{Scope}\quad \frac{\Theta, c :_{\mathsf{L}} \mathbf{ch}\langle A\rangle, d :_{\mathsf{L}} \mathbf{hc}\langle A\rangle \Vdash P}{\Theta \Vdash \nu cd.P}$$

The process configuration rules are standard. The Expr rule lifts well-typed closed terms of type $C(\mathsf{unit})$ to processes. It is important for the term m to be closed as processes in a configuration cannot rely on external substitutions to resolve free variables. They can only communicate through channels. In the Par rule, well-typed configurations P and Q can be composed in parallel as long as their contexts Θ_1 and Θ_2 can be combined. The Scope rule allows two dual channels to be connected together, allowing processes holding channels c and d to communicate.

The structural congruence of process configurations is defined as the least congruence relation generated by the following standard rules:

$$P \mid Q \equiv Q \mid P \qquad O \mid (P \mid Q) \equiv (O \mid P) \mid Q \qquad P \mid \langle \textbf{return}\ () \rangle \equiv P$$

$$\nu cd.P \mid Q \equiv \nu cd.(P \mid Q) \qquad \nu cd.P \equiv \nu dc.P \qquad \nu cd.\nu c'd'.P \equiv \nu c'd'.\nu cd.P$$

Structural congruence states that parallel composition is commutative and associative and compatible with channel scoping. Processes which terminate with the unit value () can be removed from a configuration. Intuitively, two structurally congruent configurations should be considered equivalent regarding their communication behavior.

5.2 Semantics

Term Reduction. The operational semantics of TLL_C programs is mostly the same as that of call-by-value TLL [18]. The relation $m \rightsquigarrow\!\!\!\!\!\!\rightarrow m'$ is used to denote a single step of *program* level reduction. Due to the monadic formulation of concurrency in TLL_C, the only additional (non-trivial) program reduction rule is the following BindElim rule which reduces a monadic **let**-expression when its bound term is a **return** expression:

$$(\textsc{BindElim}) \qquad \textbf{let}\ x \Leftarrow \textbf{return}\ v\ \textbf{in}\ m \rightsquigarrow\!\!\!\!\!\!\rightarrow m[v/x] \qquad \text{(where } v \text{ is a value)}$$

Values now additionally include channels, partially applied communication operators and thunked monadic expressions. We will use the metavariable v to denote values for the rest of this paper. The full definition of values is presented in the appendix.

Process Reduction. The semantics of processes is defined through the relation $P \Rrightarrow Q$ which states that process configuration P reduces to process configuration Q in one step. The process reduction rules are presented below.

$$\begin{array}{ll}
(\textsc{Proc-Fork}) & \langle \textbf{let}\ x \Leftarrow \textbf{fork}\ (y : A)\ \textbf{with}\ m\ \textbf{in}\ n \rangle \Rrightarrow \nu cd.(\langle n[c/x] \rangle \mid \langle m[d/y] \rangle) \\
(\textsc{Proc-End}) & \nu cd.(\langle \textbf{let}\ x \Leftarrow \textbf{close}\ c\ \textbf{in}\ m \rangle \mid \langle \textbf{let}\ y \Leftarrow \textbf{wait}\ d\ \textbf{in}\ n \rangle) \\
& \quad \Rrightarrow \langle \textbf{let}\ x \Leftarrow \textbf{return}\ ()\ \textbf{in}\ m \rangle \mid \langle \textbf{let}\ y \Leftarrow \textbf{return}\ ()\ \textbf{in}\ n \rangle \\
(\textsc{Proc-Com}) & \nu cd.(\langle \textbf{let}\ x \Leftarrow \textbf{send}\ c\ v\ \textbf{in}\ m \rangle \mid \langle \textbf{let}\ y \Leftarrow \textbf{recv}\ d\ \textbf{in}\ n \rangle) \\
& \quad \Rrightarrow \nu cd.(\langle \textbf{let}\ x \Leftarrow \textbf{return}\ c\ \textbf{in}\ m \rangle \mid \langle \textbf{let}\ y \Leftarrow \textbf{return}\ \langle v, d \rangle_{\mathsf{L}}\ \textbf{in}\ n \rangle) \\
(\textsc{Proc-}\underline{\textsc{Com}}) & \nu cd.(\langle \textbf{let}\ x \Leftarrow \underline{\textbf{send}}\ c\ \{o\}\ \textbf{in}\ m \rangle \mid \langle \textbf{let}\ y \Leftarrow \underline{\textbf{recv}}\ d\ \textbf{in}\ n \rangle) \\
& \quad \Rrightarrow \nu cd.(\langle \textbf{let}\ x \Leftarrow \textbf{return}\ c\ \textbf{in}\ m \rangle \mid \langle \textbf{let}\ y \Leftarrow \textbf{return}\ \langle \{o\}, d \rangle_{\mathsf{L}}\ \textbf{in}\ n \rangle)
\end{array}$$

$$\frac{m \rightsquigarrow\!\!\!\!\!\!\rightarrow m'}{\langle m \rangle \Rrightarrow \langle m' \rangle}\ (\textsc{Proc-Expr}) \qquad \frac{P \Rrightarrow Q}{O \mid P \Rrightarrow O \mid Q}\ (\textsc{Proc-Par}) \qquad \frac{P \Rrightarrow Q}{\nu cd.P \Rrightarrow \nu cd.Q}\ (\textsc{Proc-Scope}) \qquad \frac{P \equiv P' \quad P' \Rrightarrow Q' \quad Q' \equiv Q}{P \Rrightarrow Q}\ (\textsc{Proc-Congr})$$

The first four rules define the synchronous communication semantics of TLL_C. The Proc-Fork rule creates a pair of dual channels c and d to connect the continuation n of the parent process with the newly forked child process m. We can see here that the newly created channels c and d are substituted for the variables x in n and y in m respectively.

The Proc-End rule synchronizes the termination of communicating on dual channels c and d. The resulting process configuration contains two processes which are no longer connected by any channels. Additionally, the close and wait operations are replaced by unit return values once the termination is synchronized.

The PROC-COM rule governs the communication of a real message v from a sender to a receiver. The sending process continues as m with the channel c while the receiving process continues as n with the received message v and the channel d paired together as $\langle v, d\rangle_{\mathsf{L}}$.

The PROC-$\underline{\text{COM}}$ rule is similar to PROC-COM except that it handles the communication of a ghost message o. While this rule seems to indicate that ghost messages are communicated at runtime, we will later show through the erasure safety theorem that ghost messages are always safe to be erased. The exchange of ghost messages here is only for the purpose of establishing a reference point for reasoning about the correctness of erasure safety.

The remaining four rules are standard. The PROC-EXPR rule allows a singleton process to reduce its underlying term. The PROC-PAR and PROC-SCOPE rules allow a process to reduce in parallel composition and under channel scope respectively. Finally, the PROC-CONGR rule allows processes to reduce up to structural congruence.

5.3 Meta-Theory

Compatibility. We first show that the concurrency extensions of TLL_C are compatible with the underlying TLL type system. To this end, we prove that TLL_C enjoys the same meta-theoretical properties as TLL. Due to the fact that these properties do not involve concurrency, their proofs indicate that TLL_C is sound as a term calculus. Here we present a few representative theorems. The full list of theorems and their proofs can be found in our appendix.

The first theorem we present is the validity theorem which states that well-typed terms have well-sorted types. This theorem is important as it ensures that the types appearing in typing judgments are indeed valid (i.e. they inhabit a sort).

THEOREM 5.1 (VALIDITY). *Given $\Theta; \Gamma; \Delta \vdash m : A$, there exists sort s such that $\Gamma \vdash A : s$.*

In TLL and TLL_C, the sort of a type determines whether the type is unrestricted or linear. This means that it is crucial for a type to have a unique sort, otherwise the same type could be interpreted as both unrestricted and linear, leading to unsoundness. To address this concern, we prove the sort uniqueness theorem below which states that a type can have at most one sort. This ensures no ambiguity on whether a type is to be considered unrestricted or linear.

THEOREM 5.2 (SORT UNIQUENESS). *Given $\Gamma \vdash A : s$ and $\Gamma \vdash A : t$, we have $s = t$.*

The next theorem we present is the standard subject reduction theorem which states that types are preserved under term reduction. This theorem is necessary for ensuring that session fidelity holds during process reduction as singleton processes reduce by reducing their underlying terms.

THEOREM 5.3 (SUBJECT REDUCTION). *Given $\Theta; \epsilon; \epsilon \vdash m : A$ and $m \rightsquigarrow\!\!\!\!\rightarrow m'$, we have $\Theta; \epsilon; \epsilon \vdash m' : A$.*

Session Fidelity. The session fidelity theorem ensures that processes adhere to the communication protocols specified by their types. This property guarantees that well-typed processes will not encounter communication mismatches at runtime. Since we consider processes up to structural congruence, we must first show that configuration typing is preserved under structural congruence. This manifests as the following lemma.

LEMMA 5.4 (CONGRUENCE). *Given $\Theta \Vdash P$ and $P \equiv Q$, we have $\Theta \Vdash Q$.*

The session fidelity theorem is then stated as follows.

THEOREM 5.5 (SESSION FIDELITY). *Given $\Theta \Vdash P$ and $P \Rrightarrow Q$, we have $\Theta \Vdash Q$.*

One of the primary challenges in proving session fidelity is to show that typing is preserved during communication steps, specifically in the PROC-COM, and PROC-$\underline{\text{COM}}$ cases. In these cases, the message being communicated is transported from the sender to the receiver without the use of a substitution. We need to show that the message, after communication, is consistently typed with regards to the receiver's context. Unlike simple type systems where one could simply place a value into any context so long as the value has the expected type, dependent type systems require more care. For instance, the evaluation context $\langle [\cdot], \mathsf{refl}\rangle : \Sigma(x : \mathsf{nat}).(x = 1)$ is well-typed if and only if the hole is filled with 1. To address this challenge, we design the monadic Bind rule (Section 4.2) to disallow dependency on the bound value. More specifically, for $\mathbf{let}\ x \Leftarrow m\ \mathbf{in}\ n$ expressions, the type of n *cannot* depend on x. This restriction means that m can be replaced by any other expression of the same type without affecting the type of n. Consider the Proc-Com step below:

$$\begin{aligned} &\nu c d.(\langle \mathbf{let}\ x \Leftarrow \mathbf{send}\ c\ v\ \mathbf{in}\ m\rangle \mid \langle \mathbf{let}\ y \Leftarrow \mathbf{recv}\ d\ \mathbf{in}\ n\rangle) \\ &\quad \Rrightarrow \nu c d.(\langle \mathbf{let}\ x \Leftarrow \mathbf{return}\ c\ \mathbf{in}\ m\rangle \mid \langle \mathbf{let}\ y \Leftarrow \mathbf{return}\ \langle v, d\rangle_{\mathsf{L}}\ \mathbf{in}\ n\rangle)\end{aligned}$$

This operation is carried out between two singleton processes that are evaluating monadic **let**-expressions. Due to the dependency restriction of the Bind rule, we can replace $\mathbf{send}\ c\ v$ with $\mathbf{return}\ c$ and $\mathbf{recv}\ d$ with $\mathbf{return}\ \langle v, d\rangle_{\mathsf{L}}$ without affecting the types of m and n. Due to the fact that all communication operations in TLL_C are carried out on **let**-expressions, the dependency restriction ensures that session fidelity holds during communication steps.

Global Progress. Global progress, i.e. deadlock-freedom, is a desirable property for concurrent programs. Many session type systems [12, 16, 45] guarantee global progress by construction through a disciplined use of channels. However, there are also session type systems [6, 23, 24, 38] that eschew global progress in favor of more expressive session types. TLL_C belongs to the latter category if we consider arbitrary well-typed process configurations. This is because the process type system of TLL_C does not prevent cyclic channel topologies that can lead to deadlocks. However, we can still prove a weaker form of global progress for TLL_C by considering only *reachable* process configurations. Intuitively, reachable configurations are those that can be constructed by **fork** operations starting from a single process. The global progress theorem is then stated as follows.

Theorem 5.6 (Global Progress). *Given $\epsilon \Vdash P$ where P is reachable, either*

- $P \equiv \langle \mathbf{return}\ ()\rangle$, *or*
- *there exists Q such that $P \Rrightarrow Q$.*

Erasure Safety. To show that ghost messages are safe to erase, we define an erasure relation $\Theta; \Gamma; \Delta \vdash m \sim m' : A$. This relation states that all ghost arguments and type annotations in m are replaced by a special opaque value $\square$ in m'. This relation is similar to the one defined for the erasure of *propositions* in standard dependent type theories [7, 26, 36]. The most important erasure rule is shown below. The full set of erasure rules can be found in the appendix.

$$\textsc{Erase-Implicit-App}\quad \frac{\Theta; \Gamma; \Delta \vdash m \sim m' : \Pi_t\{x : A\}.B \qquad \Gamma \vdash n : A}{\Theta; \Gamma; \Delta \vdash m\ \{n\} \sim m'\ \{\square\} : B[n/x]}$$

The Erase-Implicit-App rule states that when erasing an implicit application $m\ \{n\}$, the ghost argument n is replaced by $\square$ in the erased term. Consider the $\underline{\mathbf{send}}\ c$ operator for sending ghost messages on channel c. As defined in Section 4.2, this partially applied operator has a type of the form $\Pi_{\mathsf{L}}\{x : A\}.C(B)$. When fully applied as $\underline{\mathbf{send}}\ c\ \{n\}$, the ghost argument n is erased to $\square$ by Erase-Implicit-App. Since $\square$ is an opaque value, it cannot be inspected or pattern matched on. Thus, if programs can be evaluated soundly after erasing all ghost arguments and type annotations, we can conclude that ghost messages are safe to erase.

The erasure relation is then naturally lifted to the process level as $\Theta \Vdash P \sim P'$ where P' is the erased version of P. The rules for this relation are as follows:

$$\frac{\Theta; \epsilon; \epsilon \vdash m \sim m' : C(\mathsf{unit})}{\Theta \Vdash \langle m \rangle \sim \langle m' \rangle}\ \textsc{Erase-Expr} \qquad \frac{\Theta_1 \Vdash P \sim P' \qquad \Theta_2 \Vdash Q \sim Q'}{\Theta_1 \cup \Theta_2 \Vdash (P \mid Q) \sim (P' \mid Q')}\ \textsc{Erase-Par} \qquad \frac{\Theta, c :_{\mathsf{L}} \mathbf{ch}\langle A \rangle, d :_{\mathsf{L}} \mathbf{hc}\langle A \rangle \Vdash P \sim P'}{\Theta \Vdash \nu c d.P \sim \nu c d.P'}\ \textsc{Erase-Scope}$$

We show that erasure is safe through the following two theorems. These theorems tell us that any possible reduction on an original object (either a term or process) can be simulated on its erased counterpart. Moreover, the erased object obtained after reduction also satisfies the erasure relation with respect to the reduced original object. Basically, these theorems state that any possible evaluation path of the original object remains valid after erasure.

Theorem 5.7 (Term Simulation). *Given $\Theta; \epsilon; \epsilon \vdash m \sim m' : A$ and $m \rightsquigarrow\!\!\!\!\!\twoheadrightarrow n$, there exists n' such that $m' \rightsquigarrow\!\!\!\!\!\twoheadrightarrow^* n'$ and $\Theta; \epsilon; \epsilon \vdash n \sim n' : A$.*

Theorem 5.8 (Process Simulation). *Given $\Theta \Vdash P \sim P'$ and reduction $P \Rrightarrow Q$, there exists Q' such that $P' \Rrightarrow Q'$ and $\Theta \Vdash Q \sim Q'$.*

6 Implementation

We implement a prototype compiler for TLL_C. The main components of the compiler are written in OCaml while a minimalistic runtime library is implemented in C. The compiler takes TLL_C source files as input and generates safe C code which can be further compiled into executable binaries on POSIX compliant systems. In this section, we give an overview of the inference, linearity checking and optimization phases of the compiler.

Inference. To reduce code duplication and type annotation burden, we implement two forms of inference: (1) automatic instantiation of *sort-polymorphic schemes* similarly to the TLL compiler and (2) elaboration of inferred arguments. Consider the identity function below:

$$\texttt{def}\ \mathsf{id}\langle s \rangle\ \%\{A : \mathsf{Type}\langle s \rangle\}\ (x : A) : A := x$$

This function is a sort-polymorphic scheme as it is parameterized over sort variable s. Depending on the universe of A, sort s can be instantiated to either L for linear types or U for unrestricted types. This eliminates the need to define two separate identity functions for linear and unrestricted types. The type A here is marked by % to indicate that it is an inferred argument. Suppose id is applied to a natural number 42. The compiler creates two metavariables $\hat{s}$ and $\hat{\alpha}$ to represent the elided sort and type arguments respectively. Type inference produces the following constraints:

$$\mathsf{id}\ 42 \xRightarrow{\text{desugar}} \mathsf{id}\langle \hat{s} \rangle\ \{\hat{\alpha}\}\ 42 \xRightarrow{\text{infer}} \begin{cases} \hat{s} = \mathsf{U} \\ \hat{\alpha} = \mathsf{nat} \end{cases} \xRightarrow{\text{mono}} \mathsf{id}\langle \mathsf{U} \rangle\ \{\mathsf{nat}\}\ 42$$

Once the constraints are solved through unification [1], the metavariables are replaced by their solutions. The monomorphized code is then passed to the next phase for linearity checking.

Linearity Checking. During the inference phase, the usage of linear variables is not tracked. The type checking algorithm essentially treats TLL_C as a fully structural type system. It is only after all sort-polymorphic schemes and inferred arguments are instantiated that the linearity checking begins. A substructural type checking algorithm is applied to determine if the elaborated program compiles with the actual typing rules of TLL_C. We adopt this two-phase approach to simplify the linearity checking algorithm. Although sort-polymorphism greatly reduces code duplication from the user's perspective, it also obfuscates the classification of types into linear and unrestricted ones. Thus, it is much easier to check linearity after monomorphization.

To support dependent pattern matching, we implement a variation of Cockx's algorithm [15] to type check **match**-expressions and elaborate them into well-formed case trees. Cockx's algorithm forms the basis of Agda's [4] pattern matching mechanism. Several extensions are made to the original algorithm to account for pattern matching on linear inductive types and ghost terms. Our modified algorithm is able to correctly track resource usage in subtle cases such as nested patterns involving linear inductive types. We plan to present the details of our algorithm in a future publication.

Optimization. Once linearity checking is complete, ghost terms are erased in a type directed manner. The intermediate representation (IR) obtained from erasure carries metadata that mark the linearity of certain critical expressions. For example, metadata is attached to **match**-expressions to indicate whether the scrutinee is linear or unrestricted. This information is used to guide further optimizations for improving runtime efficiency.

One of the optimizations performed is constructor unboxing. The layouts of inductive type constructors are analyzed to determine if the inductive type is suitable for unboxing. For example, consider the singleton type defined as follows:

$$\textsf{inductive}\ \mathsf{sing}\langle s\rangle\ \%\{A : \mathsf{Type}\langle s\rangle\}\ (x : A) := \mathsf{Just} : \forall (x : A) \to \mathsf{sing}\ x$$

Here, Just is the only constructor of type sing. This means that pattern matching on a value of type sing is redundant as there is only one possible case. Expressions of the form Just m are unboxed to m to reduce the number of indirections at runtime. In general, an inductive type can be unboxed if it has a single constructor and the constructor has a single non-ghost field.

To reduce the time spent on allocating and deallocating heap objects, we utilize in-place updates for linear values. This optimization is similar to recent works on function in-place programming [27, 35] where allocated heap memory is reused instead of being garbage collected. Unlike these works which utilize reference counting to dynamically check the viability of an in-place update, the metadata in our IR is sufficient to statically determine if an in-place optimization is safe.

7 Related Work

Session types are a class of type systems pioneered by Honda [23] for structuring dyadic communication in the π-calculus. Abramsky notices deep connections between the Linear Logic [21] of Girard and concurrency, predicting that Linear Logic will play a foundation role in future theories of concurrent computation [2, 3]. Caires and Pfenning show an elegant correspondence between session types and Linear Logic [12]. Gay and Vasconcelos integrate session types with λ-calculus [19] which allows one to express concurrent processes using standard functional programming. Wadler further refines the calculus of Gay and Vasconcelos to be deadlock free by construction [45].

Toninho together with Caires and Pfenning develops the first dependent session type systems [33, 39]. These works extend the existing logic of Caires and Pfenning [12] with universal and existential quantifiers to precisely specify properties of communicated messages.

Toninho and Yoshida present an interesting language [40] that integrates both π-calculus style processes and λ-calculus style terms using a contextual monad. Additionally, full λ-calculi are embedded in both functional types and session types to enable large elimination.

Wu and Xi [46] implement session types in the ATS programming language [48] which supports DML style dependent types [47]. This allows them to specify the properties of concurrent programs and verify them using proof automation. While DML style dependency is well suited for automatic reasoning, certain properties can be difficult to encode due to restrictions on the type level language.

Thiemann and Vasconcelos [38] introduce the LDST calculus which utilizes label dependent session types to elegantly describe communication patterns. Communication protocols written in non-dependent session type systems can essentially be simulated through label dependency. On the other hand, LDST's minimalist design limits its capabilities for general verification as label dependency by itself is too weak to express many interesting program properties.

Das and Pfenning develop a refinement session type system [16] where the types of concurrent programs can be refined with logical predicates. Similarly to DML style dependent types, the expressiveness of refinement session types is intentionally limited to facilitate proof automation. The Martin-Löf style dependent session types of TLL_C allow users to express and verify more complex program properties at the cost of decidable proof automation.

Atkey proposes QTT [5] based on initial ideas of McBride [30]. QTT is a dependent type theory which tracks resource usage through semi-ring annotations on binders. By instantiating the semi-ring and its ordering relation correctly, QTT can simulate linear types. The Idris 2 programming language [11] (based on QTT) implements a session typed DSL [10] around its raw communication primitives. The authors do not formalize these session types or study its meta-theory. Unlike TLL_C where a library provider could specify a type (such as channels) as linear and automatically enforce its usage in client code through type checking, the obligation of resource tracking is pushed to the client in QTT where binders must be correctly annotated a priori. User mistakes in the annotations could lead to resources being improperly tracked in a program despite passing type checking.

Hinrichsen et al. develop Actris [22] which extends the Iris [25] separation logic framework with dependent separation protocols. Compared to our work, Actris reasons about concurrent programs at a lower level of abstraction. This gives it greater precision and flexibility when dealing with imperative and unsafe programming features. However, the low level nature of Actris reduces its effectiveness at providing guidance for writing programs. In this regard, the interactivity of type systems is more beneficial to helping users construct correct programs in the first place.

8 Conclusion

TLL_C is a linear dependently typed programming language which extends the TLL type theory with dependent session types. Through examples, we demonstrate how dependent session types can be effectively applied to verify concurrent programs. The expressive power of Martin-Löf style dependency allows TLL_C session types to capture the expected semantics of concurrent programs. This results in greater verification precision and flexibility when compared to other type systems with more restricted forms of dependency. We study the meta-theory of TLL_C and show that it is sound as both a term calculus and also as a process calculus. A prototype compiler is implemented which compiles TLL_C programs into safe concurrent C code.

A direction of research we intend to explore is the integration of dependency with multi-party session types [24]. Protocols expressed through such a session type system will be able to coordinate interactions between processes from a global viewpoint. We predict dependency will again play a key role in verifying the correctness of multi-party concurrent computation.

References

[1] Andreas Abel and Brigitte Pientka. 2011. Higher-Order Dynamic Pattern Unification for Dependent Types and Records. In *Typed Lambda Calculi and Applications*, Luke Ong (Ed.). Vol. 6690. Springer Berlin Heidelberg, Berlin, Heidelberg, 10–26. doi:10.1007/978-3-642-21691-6_5 Series Title: Lecture Notes in Computer Science.

[2] Samson Abramsky. 1993. Computational interpretations of linear logic. *Theoretical Computer Science* 111, 1 (1993), 3–57. doi:10.1016/0304-3975(93)90181-R

[3] Samson Abramsky. 1994. Proofs as processes. *Theoretical Computer Science* 135, 1 (1994), 5–9. doi:10.1016/0304-3975(94)00103-0

[4] Agda development team. 2023. *Agda 2.6.4 documentation.* https://agda.readthedocs.io/en/v2.6.4

[5] Robert Atkey. 2018. The Syntax and Semantics of Quantitative Type Theory. In *LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom.* doi:10.1145/3209108.3209189

[6] Stephanie Balzer and Frank Pfenning. 2017. Manifest sharing with session types. *Proceedings of the ACM on Programming Languages* 1, ICFP (Aug. 2017), 1–29. doi:10.1145/3110281

[7] Bruno Barras and Bruno Bernardo. 2008. The Implicit Calculus of Constructions as a Programming Language with Dependent Types. In *Foundations of Software Science and Computational Structures*, Roberto Amadio (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 365–379.
[8] Giovanni Bernardi and Matthew Hennessy. 2016. Using higher-order contracts to model session types. *Logical Methods in Computer Science* 12 (06 2016). doi:10.2168/LMCS-12(2:10)2016
[9] Giovanni Tito Bernardi, Ornela Dardha, Simon J. Gay, and Dimitrios Kouzapas. 2014. On Duality Relations for Session Types. In *TGC*.
[10] Edwin Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *35th European Conference on Object-Oriented Programming (ECOOP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 9:1–9:26. doi:10.4230/LIPIcs.ECOOP.2021.9
[11] Edwin C. Brady. 2021. Idris 2: Quantitative Type Theory in Practice. *CoRR* abs/2104.00480 (2021). arXiv:2104.00480 https://arxiv.org/abs/2104.00480
[12] Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. 222–236. doi:10.1007/978-3-642-15375-4_16
[13] Iliano Cervesato and Frank Pfenning. 2002. A Linear Logical Framework. *Information and Computation* 179, 1 (2002), 19–75. doi:10.1006/inco.2001.2951
[14] Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press.
[15] Jesper Cockx and Andreas Abel. 2018. Elaborating Dependent (Co)Pattern Matching. *Proc. ACM Program. Lang.* 2, ICFP, Article 75 (jul 2018), 30 pages. doi:10.1145/3236770
[16] Ankush Das and Frank Pfenning. 2020. Verified Linear Session-Typed Concurrent Programming. In *Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming* (Bologna, Italy) *(PPDP '20)*. Association for Computing Machinery, New York, NY, USA, Article 7, 15 pages. doi:10.1145/3414080.3414087
[17] W. Diffie and M. Hellman. 1976. New directions in cryptography. *IEEE Transactions on Information Theory* 22, 6 (1976), 644–654. doi:10.1109/TIT.1976.1055638
[18] Qiancheng Fu and Hongwei Xi. 2025. A Two-Level Linear Dependent Type Theory. arXiv:2309.08673 [cs.PL] https://arxiv.org/abs/2309.08673
[19] Simon Gay and Vasco Vasconcelos. 2010. Linear type theory for asynchronous session types. *J. Funct. Program.* 20 (01 2010), 19–50. doi:10.1017/S0956796809990268
[20] Simon J. Gay, Peter Thiemann, and Vasco Thudichum Vasconcelos. 2020. Duality of Session Types: The Final Cut. *ArXiv* abs/2004.01322 (2020), 23–33.
[21] Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science* 50, 1 (1987), 1–101. doi:10.1016/0304-3975(87)90045-4
[22] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2019. Actris: Session-Type Based Reasoning in Separation Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 6 (dec 2019), 30 pages. doi:10.1145/3371074
[23] Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR*.
[24] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1, Article 9 (mar 2016), 67 pages. doi:10.1145/2827695
[25] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. *SIGPLAN Not.* 50, 1 (Jan. 2015), 637–650. doi:10.1145/2775051.2676980
[26] Pierre Letouzey. 2003. A New Extraction for Coq. In *Types for Proofs and Programs*, Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, Herman Geuvers, and Freek Wiedijk (Eds.). Vol. 2646. Springer Berlin Heidelberg, Berlin, Heidelberg, 200–219. doi:10.1007/3-540-39185-1_12 Series Title: Lecture Notes in Computer Science.
[27] Anton Lorenzen, Daan Leijen, and Wouter Swierstra. 2023. FP^2: Fully in-Place Functional Programming. *Proc. ACM Program. Lang.* 7, ICFP, Article 198 (aug 2023), 30 pages. doi:10.1145/3607840
[28] Zhaohui Luo and Y Zhang. 2016. *A Linear Dependent Type Theory*. 69–70.
[29] Per Martin-Löf. 1975. An Intuitionistic Theory of Types: Predicative Part. In *Logic Colloquium '73*, H.E. Rose and J.C. Shepherdson (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 80. Elsevier, 73–118. doi:10.1016/S0049-237X(08)71945-1
[30] Conor McBride. 2016. I Got Plenty o' Nuttin'. In *A List of Successes That Can Change the World*.
[31] Peter O'Hearn. 2003. On bunched typing. *Journal of Functional Programming* 13, 4 (July 2003), 747–796. doi:10.1017/S0956796802004495
[32] Peter W. O'Hearn and David J. Pym. 1999. The Logic of Bunched Implications. *Bulletin of Symbolic Logic* 5, 2 (June 1999), 215–244. doi:10.2307/421090
[33] Frank Pfenning, Luis Caires, and Bernardo Toninho. 2011. Proof-Carrying Code in a Session-Typed Process Calculus. In *Certified Programs and Proofs*, Jean-Pierre Jouannaud and Zhong Shao (Eds.). Springer Berlin Heidelberg, Berlin,

Heidelberg, 21–36.
[34] Benjamin C Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2010. Software foundations. *Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html* 16 (2010).
[35] Alex Reinking*, Ningning Xie*, Leonardo de Moura, and Daan Leijen. 2020. *Perceus: Garbage Free Reference Counting with Reuse (Extended version).* Technical Report MSR-TR-2020-42. Microsoft. https://www.microsoft.com/en-us/research/publication/perceus-garbage-free-reference-counting-with-reuse/ (*) The first two authors contributed equally to this work. v4, 2021-06-07. Extended version of the PLDI'21 paper..
[36] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. 2020. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 1–28. doi:10.1145/3371076
[37] The Rust teams. 2022. *Rust Programming Language.* http://www.rust-lang.org/
[38] Peter Thiemann and Vasco T. Vasconcelos. 2019. Label-Dependent Session Types. *Proc. ACM Program. Lang.* 4, POPL, Article 67 (dec 2019), 29 pages. doi:10.1145/3371135
[39] Bernardo Toninho, Luís Caires, and Frank Pfenning. 2011. Dependent Session Types via Intuitionistic Linear Type Theory. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming* (Odense, Denmark) *(PPDP '11)*. Association for Computing Machinery, New York, NY, USA, 161–172. doi:10.1145/2003476.2003499
[40] Bernardo Toninho and Nobuko Yoshida. 2018. *Depending on Session-Typed Processes.* 128–145. doi:10.1007/978-3-319-89366-2_7
[41] David N. Turner and Philip Wadler. 1999. Operational interpretations of linear logic. *Theoretical Computer Science* 227, 1 (1999), 231–248. doi:10.1016/S0304-3975(99)00054-7
[42] Matthijs Vákár. 2014. Syntax and Semantics of Linear Dependent Types. *CoRR* abs/1405.0033 (2014). arXiv:1405.0033 http://arxiv.org/abs/1405.0033
[43] P. Wadler. 1990. Linear Types can Change the World!. In *Programming Concepts and Methods.*
[44] Philip Wadler. 1991. Is There a Use for Linear Logic? *SIGPLAN Not.* 26, 9 (May 1991), 255–273. doi:10.1145/115866.115894
[45] Philip Wadler. 2012. Propositions as Sessions. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming* (Copenhagen, Denmark) *(ICFP '12)*. Association for Computing Machinery, New York, NY, USA, 273–286. doi:10.1145/2364527.2364568
[46] Hanwen Wu and Hongwei Xi. 2017. Dependent Session Types. *CoRR* abs/1704.07004 (2017). arXiv:1704.07004 http://arxiv.org/abs/1704.07004
[47] Hongwei Xi. 2007. Dependent ML An approach to practical programming with dependent types. *Journal of Functional Programming* 17, 2 (2007), 215–286. doi:10.1017/S0956796806006216
[48] Hongwei Xi. 2010. *The ATS Programming Language.* http://www.ats-lang.org/

Contents

A Syntax

The full syntax of TLL_C is presented below.

$$
\begin{array}{rlcl}
\text{variables} & x, y, z & & \\
\text{channels} & c, d & & \\
\text{sorts} & s, r, t & ::= & \mathsf{U} \mid \mathsf{L} \\
\text{actions} & \rho & ::= & ! \mid ? \\
\text{terms} & m, n, A, B, C & ::= & x \mid c \mid s \\
& & \mid & \Pi_t(x:A).B \mid \Pi_t\{x:A\}.B \mid \Sigma_t(x:A).B \mid \Sigma_t\{x:A\}.B \\
& & \mid & \lambda_t(x:A).m \mid \lambda_t\{x:A\}.m \mid \langle m,n\rangle_t \mid \langle\{m\},n\rangle_t \\
& & \mid & m\ n \mid m\ \{n\} \mid \mathrm{R}^{\Sigma}_{[z]A}(m,[x,y]n) \mid \mu(x:A).B \\
& & \mid & \mathsf{unit} \mid () \mid \mathsf{bool} \mid \mathsf{true} \mid \mathsf{false} \mid \mathrm{R}^{\mathsf{bool}}_{[z]A}(m,n_1,n_2) \\
& & \mid & C(A) \mid \textbf{return}\ m \mid \textbf{let}\ x \Leftarrow m\ \textbf{in}\ n \\
& & \mid & \textbf{proto} \mid \mathbf{1} \mid \rho(x:A).\ B \mid \rho\{x:A\}.\ B \mid \textbf{ch}\langle A\rangle \mid \textbf{hc}\langle A\rangle \\
& & \mid & \textbf{fork}\ (x:A)\ \textbf{with}\ m \mid \textbf{recv}\ m \mid \underline{\textbf{recv}}\ m \mid \textbf{send}\ m \mid \underline{\textbf{send}}\ m \\
& & \mid & \textbf{close}\ m \mid \textbf{wait}\ m \\
\text{thunks} & \tau & ::= & \textbf{fork}\ (x:A)\ \textbf{with}\ m \mid \textbf{recv}\ v \mid \underline{\textbf{recv}}\ v \\
& & \mid & \textbf{send}\ v\ u \mid \underline{\textbf{send}}\ v\ \{m\} \mid \textbf{close}\ v \mid \textbf{wait}\ v \\
& & \mid & \textbf{let}\ x \Leftarrow \tau\ \textbf{in}\ m \\
\text{values} & u, v & ::= & c \mid \lambda_t(x:A).m \mid \lambda_t\{x:A\}.m \mid \langle u,v\rangle_t \mid \langle\{m\},v\rangle_t \\
& & \mid & () \mid \mathsf{true} \mid \mathsf{false} \mid \textbf{return}\ v \mid \tau \mid \textbf{send}\ v \mid \underline{\textbf{send}}\ v \\
\text{process} & O, P, Q & \mid & \langle m\rangle \mid (P \mid Q) \mid \nu cd.P
\end{array}
$$

B Auxiliary Operators/Judgments

In this section, we define several auxiliary operators/judgments used in the formalization of TLL_C.

Sort Ordering. The sort ordering relation $\sqsubseteq$ is defined as follows:

$$(\textsc{Ord-U})\quad \mathsf{U} \sqsubseteq s \qquad\qquad (\textsc{Ord-L})\quad \mathsf{L} \sqsubseteq \mathsf{L}$$

This relation is useful when defining the typing rules of dependent pairs by ensuring that pairs only contain values of a lower or equal sort.

Context Merge. The context merge operator $\uplus$ is a partial function that combines two contexts into one by selective applying the contraction rule on unrestricted variables. The operator is undefined if both two contexts contain overlapping linear variables.

$$\frac{}{\epsilon \uplus \epsilon = \epsilon}\ \textsc{Merge-Empty} \qquad \frac{\Delta_1 \uplus \Delta_2 = \Delta \qquad x \notin \Delta}{(\Delta_1, x :_{\mathsf{U}} A) \uplus (\Delta_2, x :_{\mathsf{U}} A) = (\Delta, x :_{\mathsf{U}} A)}\ \textsc{Merge-U}$$

$$\frac{\Delta_1 \uplus \Delta_2 = \Delta \qquad x \notin \Delta}{(\Delta_1, x :_{\mathsf{L}} A) \uplus \Delta_2 = (\Delta, x :_{\mathsf{L}} A)}\ \textsc{Merge-L}_1 \qquad \frac{\Delta_1 \uplus \Delta_2 = \Delta \qquad x \notin \Delta}{\Delta_1 \uplus (\Delta_2, x :_{\mathsf{L}} A) = (\Delta, x :_{\mathsf{L}} A)}\ \textsc{Merge-L}_2$$

Context Restriction. The context restriction operator $\triangleright$ is a predicate that is useful for defining the typing rules of λ-expressions. In particular, it prevents unrestricted functions from capturing linear variables in their closures.

$$\frac{}{\epsilon \triangleright s}\ \textsc{ReEmpty} \qquad \frac{\Delta \triangleright \mathsf{U}}{\Delta, x :_{\mathsf{U}} A \triangleright \mathsf{U}}\ \textsc{Re-U} \qquad \frac{\Delta \triangleright \mathsf{L}}{\Delta, x :_s A \triangleright \mathsf{L}}\ \textsc{Re-L}$$

Arity. For types A and X, we say that A is an *arity* ending on X if it is either X itself or a Π-type whose codomain is an arity ending on X. Formally, we define the judgment A arity(X) as follows:

$$\frac{}{X \text{ arity}(X)}\ \textsc{Arity-Base} \qquad \frac{B \text{ arity}(X)}{\Pi_t\{x : A\}.B \text{ arity}(X)}\ \textsc{Arity-Implicit} \qquad \frac{B \text{ arity}(X)}{\Pi_t(x : A).B \text{ arity}(X)}\ \textsc{Arity-Explicit}$$

This judgment is used for defining the typing rule of (parameterized) recursive protocols.

Guarded. For variable x and term m, we say that x is *guarded* in m if the judgment m guard(x) is derivable. Intuitively, this means that every occurrence of x in m appears under a protocol action. This is important for ensuring that recursive protocols do not unfold indefinitely without performing any actions. The judgment is defined as follows:

$$\frac{x \neq y}{y \text{ guard}(x)}\ \textsc{Guard-Var} \qquad \frac{}{s \text{ guard}(x)}\ \textsc{Guard-Sort} \qquad \frac{A \text{ guard}(x) \qquad B \text{ guard}(x)}{\Pi_t\{x' : A\}.B \text{ guard}(x)}\ \textsc{Guard-Implicit-Fun} \qquad \frac{A \text{ guard}(x) \qquad B \text{ guard}(x)}{\Pi_t(x' : A).B \text{ guard}(x)}\ \textsc{Guard-Explicit-Fun}$$

$$\frac{A \text{ guard}(x) \qquad m \text{ guard}(x)}{\lambda_t\{x' : A\}.m \text{ guard}(x)}\ \textsc{Guard-Implicit-Lam} \qquad \frac{A \text{ guard}(x) \qquad m \text{ guard}(x)}{\lambda_t(x' : A).m \text{ guard}(x)}\ \textsc{Guard-Explicit-Lam} \qquad \frac{m \text{ guard}(x) \qquad n \text{ guard}(x)}{m\ \{n\} \text{ guard}(x)}\ \textsc{Guard-Implicit-App}$$

$$\frac{m \text{ guard}(x) \qquad n \text{ guard}(x)}{m\ n \text{ guard}(x)}\ \textsc{Guard-Explicit-App} \qquad \frac{A \text{ guard}(x) \qquad B \text{ guard}(x)}{\Sigma_t\{x' : A\}.B \text{ guard}(x)}\ \textsc{Guard-Implicit-Sum} \qquad \frac{A \text{ guard}(x) \qquad B \text{ guard}(x)}{\Sigma_t(x' : A).B \text{ guard}(x)}\ \textsc{Guard-Explicit-Sum}$$

$$\frac{m \text{ guard}(x) \qquad n \text{ guard}(x)}{\langle\{m\}, n\rangle_t \text{ guard}(x)}\ \textsc{Guard-Implicit-Pair} \qquad \frac{m \text{ guard}(x) \qquad n \text{ guard}(x)}{\langle m, n\rangle \text{ guard}(x)}\ \textsc{Guard-Explicit-Pair} \qquad \frac{A \text{ guard}(x) \qquad m \text{ guard}(x) \qquad n \text{ guard}(x)}{\mathrm{R}^{\Sigma}_{[z]A}(m, [x', y']n)}\ \textsc{Guard-Explicit-SumElim}$$

$$\frac{}{\mathsf{unit}\ \mathrm{guard}(x)}\ \textsc{Guard-Unit} \qquad \frac{}{()\ \mathrm{guard}(x)}\ \textsc{Guard-UnitVal} \qquad \frac{}{\mathsf{bool}\ \mathrm{guard}(x)}\ \textsc{Guard-Bool} \qquad \frac{}{\mathsf{true}\ \mathrm{guard}(x)}\ \textsc{Guard-True} \qquad \frac{}{\mathsf{false}\ \mathrm{guard}(x)}\ \textsc{Guard-False}$$

$$\frac{A\ \mathrm{guard}(x) \qquad m\ \mathrm{guard}(x) \qquad n_1\ \mathrm{guard}(x) \qquad n_2\ \mathrm{guard}(x)}{\mathrm{R}^{\mathsf{bool}}_{[z]A}(m, n_1, n_2)\ \mathrm{guard}(x)}\ \textsc{Guard-BoolElim} \qquad \frac{A\ \mathrm{guard}(x)}{C(A)\ \mathrm{guard}(x)}\ \textsc{Guard-}C\textsc{Type} \qquad \frac{m\ \mathrm{guard}(x)}{\mathbf{return}\ m\ \mathrm{guard}(x)}\ \textsc{Guard-Return}$$

$$\frac{m\ \mathrm{guard}(x) \qquad n\ \mathrm{guard}(x)}{\mathbf{let}\ x' \Leftarrow m\ \mathbf{in}\ n\ \mathrm{guard}(x)}\ \textsc{Guard-Bind} \qquad \frac{}{\mathbf{proto}\ \mathrm{guard}(x)}\ \textsc{Guard-Proto} \qquad \frac{}{\mathbf{1}\ \mathrm{guard}(x)}\ \textsc{Guard-End} \qquad \frac{A\ \mathrm{guard}(x) \qquad m\ \mathrm{guard}(x)}{\mu(x' : A).m\ \mathrm{guard}(x)}\ \textsc{Guard-RecProto}$$

$$\frac{A\ \mathrm{guard}(x)}{\rho\{x' : A\}.\ B\ \mathrm{guard}(x)}\ \textsc{Guard-Implicit-Action} \qquad \frac{A\ \mathrm{guard}(x)}{\rho(x' : A).\ B\ \mathrm{guard}(x)}\ \textsc{Guard-Explicit-Action} \qquad \frac{A\ \mathrm{guard}(x)}{\mathbf{ch}\langle A\rangle\ \mathrm{guard}(x)}\ \textsc{Guard-CH} \qquad \frac{A\ \mathrm{guard}(x)}{\mathbf{hc}\langle A\rangle\ \mathrm{guard}(x)}\ \textsc{Guard-HC}$$

$$\frac{}{c\ \mathrm{guard}(x)}\ \textsc{Guard-Channel} \qquad \frac{A\ \mathrm{guard}(x) \qquad m\ \mathrm{guard}(x)}{\mathbf{fork}\ (x' : A)\ \mathbf{with}\ m\ \mathrm{guard}(x)}\ \textsc{Guard-Fork} \qquad \frac{m\ \mathrm{guard}(x)}{\underline{\mathbf{recv}}\ m\ \mathrm{guard}(x)}\ \textsc{Guard-Implicit-Recv} \qquad \frac{m\ \mathrm{guard}(x)}{\mathbf{recv}\ m\ \mathrm{guard}(x)}\ \textsc{Guard-Explicit-Recv}$$

$$\frac{m\ \mathrm{guard}(x)}{\underline{\mathbf{send}}\ m\ \mathrm{guard}(x)}\ \textsc{Guard-Implicit-Send} \qquad \frac{m\ \mathrm{guard}(x)}{\mathbf{send}\ m\ \mathrm{guard}(x)}\ \textsc{Guard-Explicit-Send} \qquad \frac{m\ \mathrm{guard}(x)}{\mathbf{wait}\ m\ \mathrm{guard}(x)}\ \textsc{Guard-Wait} \qquad \frac{m\ \mathrm{guard}(x)}{\mathbf{close}\ m\ \mathrm{guard}(x)}\ \textsc{Guard-Close}$$

C Formal Typing Rules

In this section, we present the full typing rules of TLL_C. We organize the typing rules into logical level, program level and process level.

C.1 Logical Level

The typing judgment for the logical level has the form $\Gamma \vdash m : A$. This judgment states that under the *logical context* Γ, term m has type A. The logical level is completely *structural.*

Logical Context. The logical context Γ is a sequence of variable bindings of the form $x_0 : A_0, x_1 : A_1, \ldots, x_n : A_n$. Each variable x_i is bound to a type A_i. Variables in the logical context are unrestricted and can be used arbitrarily many times. The empty context is denoted by ϵ. To ensure the validity of types in the logical context, we define the context validity judgment $\Gamma \vdash$.

$$\textsc{Ctx-Empty}\ \frac{}{\epsilon \vdash} \qquad \textsc{Ctx-Var}\ \frac{\Gamma \vdash \quad \Gamma \vdash A : s \quad x \notin \Gamma}{\Gamma, x : A \vdash}$$

Note that the context validity judgment is *mutually inductively* defined with the typing judgment.

Core Typing. The core typing rules is responsible for the functional fragment of TLL_C. The convertibility relation $A \simeq B$ is used in the conversion rule to allow type equivalence up to β-reduction. We will present the definition of the convertibility relation in Section D.1.

$$\textsc{Sort}\ \frac{\Gamma \vdash}{\Gamma \vdash s : \mathrm{U}} \qquad \textsc{Var}\ \frac{\Gamma, x : A \vdash}{\Gamma, x : A \vdash x : A} \qquad \textsc{Conversion}\ \frac{\Gamma \vdash B : s \quad \Gamma \vdash m : A \quad A \simeq B}{\Gamma \vdash m : B} \qquad \textsc{Explicit-Fun}\ \frac{\Gamma \vdash A : s \quad \Gamma, x : A \vdash B : r}{\Gamma \vdash \Pi_t(x : A).B : t}$$

$$\textsc{Implicit-Fun}\ \frac{\Gamma \vdash A : s \quad \Gamma, x : A \vdash B : r}{\Gamma \vdash \Pi_t\{x : A\}.B : t} \qquad \textsc{Explicit-Lam}\ \frac{\Gamma, x : A \vdash m : B}{\Gamma \vdash \lambda_t(x : A).m : \Pi_t(x : A).B} \qquad \textsc{Implicit-Lam}\ \frac{\Gamma, x : A \vdash m : B}{\Gamma \vdash \lambda_t\{x : A\}.m : \Pi_t\{x : A\}.B}$$

$$\textsc{Explicit-App}\ \frac{\Gamma \vdash m : \Pi_t(x : A).B \quad \Gamma \vdash n : A}{\Gamma \vdash m\ n : B[n/x]} \qquad \textsc{Implicit-App}\ \frac{\Gamma \vdash m : \Pi_t\{x : A\}.B \quad \Gamma \vdash n : A}{\Gamma \vdash m\ \{n\} : B[n/x]}$$

$$\textsc{Explicit-Sum}\ \frac{s \sqsubseteq t \quad r \sqsubseteq t \quad \Gamma \vdash A : s \quad \Gamma, x : A \vdash B : r}{\Gamma \vdash \Sigma_t(x : A).B : t} \qquad \textsc{Implicit-Sum}\ \frac{r \sqsubseteq t \quad \Gamma \vdash A : s \quad \Gamma, x : A \vdash B : r}{\Gamma \vdash \Sigma_t\{x : A\}.B : t}$$

$$\textsc{Explicit-Pair}\ \frac{\Gamma \vdash \Sigma_t(x : A).B : t \quad \Gamma \vdash m : A \quad \Gamma \vdash n : B[m/x]}{\Gamma \vdash \langle m, n\rangle_t : \Sigma_t(x : A).B} \qquad \textsc{Implicit-Pair}\ \frac{\Gamma \vdash \Sigma_t\{x : A\}.B : t \quad \Gamma \vdash m : A \quad \Gamma \vdash n : B[m/x]}{\Gamma \vdash \langle \{m\}, n\rangle_t : \Sigma_t\{x : A\}.B}$$

$$\textsc{Explicit-SumElim}\ \frac{\Gamma, z : \Sigma_t(x : A).B \vdash C : s \quad \Gamma \vdash m : \Sigma_t(x : A).B \quad \Gamma, x : A, y : B \vdash n : C[\langle x, y\rangle_t/z]}{\Gamma \vdash \mathrm{R}^{\Sigma}_{[z]C}(m, [x, y]n) : C[m/z]}$$

$$\textsc{Implicit-SumElim}\ \frac{\Gamma, z : \Sigma_t\{x : A\}.B \vdash C : s \quad \Gamma \vdash m : \Sigma_t\{x : A\}.B \quad \Gamma, x : A, y : B \vdash n : C[\langle \{x\}, y\rangle_t/z]}{\Gamma \vdash \mathrm{R}^{\Sigma}_{[z]C}(m, [x, y]n) : C[m/z]}$$

Data Typing. The data typing rules govern the typing of base types such as the unit type and the boolean type. The rules are presented below.

$$\textsc{Unit}\ \frac{\Gamma \vdash}{\Gamma \vdash \mathsf{unit} : \mathsf{U}} \qquad \textsc{UnitVal}\ \frac{\Gamma \vdash}{\Gamma \vdash () : \mathsf{unit}} \qquad \textsc{Bool}\ \frac{\Gamma \vdash}{\Gamma \vdash \mathsf{bool} : \mathsf{U}} \qquad \textsc{True}\ \frac{\Gamma \vdash}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \qquad \textsc{False}\ \frac{\Gamma \vdash}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}$$

$$\textsc{BoolElim}\ \frac{\Gamma, z : \mathsf{bool} \vdash A : s \qquad \Gamma \vdash m : \mathsf{bool} \qquad \Gamma \vdash n_1 : A[\mathsf{true}/z] \qquad \Gamma \vdash n_2 : A[\mathsf{false}/z]}{\Gamma \vdash \mathrm{R}^{\mathsf{bool}}_{[z]A}(m, n_1, n_2) : A[m/z]}$$

Monadic Typing. The monadic typing rules govern the composition of monadic computations. The standard rules for monadic return and bind are presented below.

$$C\textsc{Type}\ \frac{\Gamma \vdash A : s}{\Gamma \vdash C(A) : \mathsf{L}} \qquad \textsc{Return}\ \frac{\Gamma \vdash m : A}{\Gamma \vdash \textbf{return}\, m : C(A)} \qquad \textsc{Bind}\ \frac{\Gamma \vdash B : s \qquad \Gamma \vdash m : C(A) \qquad \Gamma, x : A \vdash n : C(B)}{\Gamma \vdash \textbf{let}\ x \Leftarrow m\ \textbf{in}\ n : C(B)}$$

Session Typing. The session typing rules govern the typing of protocol, channels and concurrency primitives. The rules are presented below.

$$\textsc{Proto}\ \frac{\Gamma \vdash}{\Gamma \vdash \textbf{proto} : \mathsf{U}} \qquad \textsc{End}\ \frac{\Gamma \vdash}{\Gamma \vdash \mathbf{1} : \textbf{proto}} \qquad \textsc{Explicit-Action}\ \frac{\Gamma, x : A \vdash B : \textbf{proto}}{\Gamma \vdash \rho(x : A).\, B : \textbf{proto}} \qquad \textsc{Implicit-Action}\ \frac{\Gamma, x : A \vdash B : \textbf{proto}}{\Gamma \vdash \rho\{x : A\}.\, B : \textbf{proto}}$$

$$\textsc{RecProto}\ \frac{\Gamma, x : A \vdash m : A \qquad A\ \mathrm{arity}(\textbf{proto}) \qquad m\ \mathrm{guard}(x)}{\Gamma \vdash \mu(x : A).m : A} \qquad \textsc{ChType}\ \frac{\Gamma \vdash A : \textbf{proto}}{\Gamma \vdash \textbf{ch}\langle A \rangle : \mathsf{L}} \qquad \textsc{HcType}\ \frac{\Gamma \vdash A : \textbf{proto}}{\Gamma \vdash \textbf{hc}\langle A \rangle : \mathsf{L}} \qquad \textsc{Channel-CH}\ \frac{\Gamma \vdash \qquad \epsilon \vdash A : \textbf{proto}}{\Gamma \vdash c : \textbf{ch}\langle A \rangle}$$

$$\textsc{Channel-HC}\ \frac{\Gamma \vdash \qquad \epsilon \vdash A : \textbf{proto}}{\Gamma \vdash c : \textbf{hc}\langle A \rangle} \qquad \textsc{Explicit-Send-CH}\ \frac{\Gamma \vdash m : \textbf{ch}\langle !(x : A).\, B \rangle}{\Gamma \vdash \textbf{send}\, m : \Pi_{\mathsf{L}}(x : A).C(\textbf{ch}\langle B \rangle)} \qquad \textsc{Explicit-Send-HC}\ \frac{\Gamma \vdash m : \textbf{hc}\langle ?(x : A).\, B \rangle}{\Gamma \vdash \textbf{send}\, m : \Pi_{\mathsf{L}}(x : A).C(\textbf{hc}\langle B \rangle)}$$

$$\textsc{Implicit-Send-CH}\ \frac{\Gamma \vdash m : \textbf{ch}\langle !\{x : A\}.\, B \rangle}{\Gamma \vdash \underline{\textbf{send}}\, m : \Pi_{\mathsf{L}}\{x : A\}.C(\textbf{ch}\langle B \rangle)} \qquad \textsc{Implicit-Send-HC}\ \frac{\Gamma \vdash m : \textbf{hc}\langle ?\{x : A\}.\, B \rangle}{\Gamma \vdash \underline{\textbf{send}}\, m : \Pi_{\mathsf{L}}\{x : A\}.C(\textbf{hc}\langle B \rangle)} \qquad \textsc{Explicit-Recv-CH}\ \frac{\Gamma \vdash m : \textbf{ch}\langle ?(x : A).\, B \rangle}{\Gamma \vdash \textbf{recv}\, m : C(\Sigma_{\mathsf{L}}(x : A).\textbf{ch}\langle B \rangle)}$$

$$\textsc{Explicit-Recv-HC}\ \frac{\Gamma \vdash m : \textbf{hc}\langle !(x : A).\, B \rangle}{\Gamma \vdash \textbf{recv}\, m : C(\Sigma_{\mathsf{L}}(x : A).\textbf{hc}\langle B \rangle)} \qquad \textsc{Implicit-Recv-CH}\ \frac{\Gamma \vdash m : \textbf{ch}\langle ?\{x : A\}.\, B \rangle}{\Gamma \vdash \underline{\textbf{recv}}\, m : C(\Sigma_{\mathsf{L}}\{x : A\}.\textbf{ch}\langle B \rangle)} \qquad \textsc{Implicit-Recv-HC}\ \frac{\Gamma \vdash m : \textbf{hc}\langle !\{x : A\}.\, B \rangle}{\Gamma \vdash \underline{\textbf{recv}}\, m : C(\Sigma_{\mathsf{L}}\{x : A\}.\textbf{hc}\langle B \rangle)}$$

$$\textsc{Fork}\ \frac{\Gamma, x : \textbf{ch}\langle A \rangle \vdash m : C(\mathsf{unit})}{\Gamma \vdash \textbf{fork}\ (x : \textbf{ch}\langle A \rangle)\ \textbf{with}\ m : C(\textbf{hc}\langle A \rangle)} \qquad \textsc{Close}\ \frac{\Gamma \vdash m : \textbf{ch}\langle \mathbf{1} \rangle}{\Gamma \vdash \textbf{close}\, m : C(\mathsf{unit})} \qquad \textsc{Wait}\ \frac{\Gamma \vdash m : \textbf{hc}\langle \mathbf{1} \rangle}{\Gamma \vdash \textbf{wait}\, m : C(\mathsf{unit})}$$

C.2 Program Level

The typing judgment for the program level has the form $\Theta; \Gamma; \Delta \vdash m : A$. This judgment states that under the channel context Θ, logical context Γ and the *program context* Δ, term m has type A. The program level is *substructural* as the usage of variables in the program context is tracked.

Program Context. The program context Δ is a sequence of variable bindings of the form $x_0 :_{s_0} A_0, x_1 :_{s_1} A_1, \ldots, x_n :_{s_n} A_n$. Each variable x_i is bound to a type A_i with a sort annotation s_i. The variables in the program context are allowed to appear in computationally relevant positions inside m. To ensure that all types appear in the program context are well-formed, we define the program context validity judgment $\Gamma; \Delta \vdash$. The rules for this judgment are presented below.

$$\textsc{Ctx-Empty}\quad \frac{}{\epsilon; \epsilon \vdash} \qquad \textsc{Ctx-Implicit-Var}\quad \frac{\Gamma; \Delta \vdash \qquad \Gamma \vdash A : s \qquad x \notin \Gamma}{\Gamma, x : A; \Delta \vdash} \qquad \textsc{Ctx-Explicit-Var}\quad \frac{\Gamma; \Delta \vdash \qquad \Gamma \vdash A : s \qquad x \notin \Gamma}{\Gamma, x : A; \Delta, x :_s A \vdash}$$

From these rules we can see that $\mathrm{dom}(\Delta)$ is a subset of $\mathrm{dom}(\Gamma)$. Additionally, the sort annotation s in each program context binding $x :_s A$ is the sort of the associated A type.

Core Typing. The core typing rules is responsible for the functional fragment of TLL_C.

$$\textsc{Var}\quad \frac{\epsilon; \Gamma, x : A; \Delta, x :_s A \vdash \qquad \Delta \triangleright \mathsf{U}}{\epsilon; \Gamma, x : A; \Delta, x :_s A \vdash x : A} \qquad \textsc{Conversion}\quad \frac{\Gamma \vdash B : s \qquad \Theta; \Gamma; \Delta \vdash m : A \qquad A \simeq B}{\Theta; \Gamma; \Delta \vdash m : B}$$

$$\textsc{Explicit-Lam}\quad \frac{\Theta; \Gamma, x : A; \Delta, x :_s A \vdash m : B \qquad \Theta \triangleright t \qquad \Delta \triangleright t}{\Theta; \Gamma; \Delta \vdash \lambda_t(x : A).m : \Pi_t(x : A).B} \qquad \textsc{Implicit-Lam}\quad \frac{\Theta; \Gamma, x : A; \Delta \vdash m : B \qquad \Theta \triangleright t \qquad \Delta \triangleright t}{\Theta; \Gamma; \Delta \vdash \lambda_t\{x : A\}.m : \Pi_t\{x : A\}.B}$$

$$\textsc{Explicit-App}\quad \frac{\Theta_1; \Gamma; \Delta_1 \vdash m : \Pi_t(x : A).B \qquad \Theta_2; \Gamma; \Delta_2 \vdash n : A}{\Theta_1 \cup \Theta_2; \Gamma; \Delta_1 \cup \Delta_2 \vdash m\ n : B[n/x]} \qquad \textsc{Implicit-App}\quad \frac{\Theta; \Gamma; \Delta \vdash m : \Pi_t\{x : A\}.B \qquad \Gamma \vdash n : A}{\Theta; \Gamma; \Delta \vdash m\ \{n\} : B[n/x]}$$

$$\textsc{Explicit-Pair}\quad \frac{\begin{array}{c}\Gamma \vdash \Sigma_t(x : A).B : t \\ \Theta_1; \Gamma; \Delta_1 \vdash m : A \qquad \Theta_2; \Gamma; \Delta_2 \vdash n : B[m/x]\end{array}}{\Theta_1 \cup \Theta_2; \Gamma; \Delta_1 \cup \Delta_2 \vdash \langle m, n\rangle_t : \Sigma_t(x : A).B} \qquad \textsc{Implicit-Pair}\quad \frac{\begin{array}{c}\Gamma \vdash \Sigma_t\{x : A\}.B : t \\ \Gamma \vdash m : A \qquad \Theta; \Gamma; \Delta \vdash n : B[m/x]\end{array}}{\Theta; \Gamma; \Delta \vdash \langle \{m\}, n\rangle_t : \Sigma_t\{x : A\}.B}$$

$$\textsc{Explicit-SumElim}\quad \frac{\begin{array}{c}\Gamma, z : \Sigma_t(x : A).B \vdash C : s \qquad \Theta_1; \Gamma; \Delta_1 \vdash m : \Sigma_t(x : A).B \\ \Theta_2; \Gamma, x : A, y : B; \Delta_2, x :_{r1} A, y :_{r2} B \vdash n : C[\langle x, y\rangle_t/z]\end{array}}{\Theta_1 \cup \Theta_2; \Gamma; \Delta_1 \cup \Delta_2 \vdash \mathrm{R}^{\Sigma}_{[z]C}(m, [x, y]n) : C[m/z]}$$

$$\textsc{Implicit-SumElim}\quad \frac{\begin{array}{c}\Gamma, z : \Sigma_t\{x : A\}.B \vdash C : s \qquad \Theta_1; \Gamma; \Delta_1 \vdash m : \Sigma_t\{x : A\}.B \\ \Theta_2; \Gamma, x : A, y : B; \Delta_2, y :_r B \vdash n : C[\langle \{x\}, y\rangle_t/z]\end{array}}{\Theta_1 \cup \Theta_2; \Gamma; \Delta_1 \cup \Delta_2 \vdash \mathrm{R}^{\Sigma}_{[z]C}(m, [x, y]n) : C[m/z]}$$

Data Typing. The data typing rules govern the typing of base types such as the unit type and the boolean type. The rules are presented below.

$$\textsc{UnitVal}\quad \frac{\Gamma; \Delta \vdash \qquad \Delta \triangleright \mathsf{U}}{\epsilon; \Gamma; \Delta \vdash () : \mathsf{unit}} \qquad \textsc{True}\quad \frac{\Gamma; \Delta \vdash \qquad \Delta \triangleright \mathsf{U}}{\epsilon; \Gamma; \Delta \vdash \mathsf{true} : \mathsf{bool}} \qquad \textsc{False}\quad \frac{\Gamma; \Delta \vdash \qquad \Delta \triangleright \mathsf{U}}{\epsilon; \Gamma; \Delta \vdash \mathsf{false} : \mathsf{bool}}$$

$$\textsc{BoolElim}\quad \frac{\Gamma, z : \mathsf{bool} \vdash A : s \qquad \Theta_1; \Gamma; \Delta_1 \vdash m : \mathsf{bool} \qquad \Theta_2; \Gamma; \Delta_2 \vdash n_1 : A[\mathsf{true}/z] \qquad \Theta_2; \Gamma; \Delta_2 \vdash n_2 : A[\mathsf{false}/z]}{\Theta_1 \cup \Theta_2; \Gamma; \Delta_1 \cup \Delta_2 \vdash \mathrm{R}^{\mathsf{bool}}_{[z]A}(m, n_1, n_2) : A[m/z]}$$

Monadic Typing. The monadic typing rules govern the composition of monadic computations. The standard rules for monadic return and bind are presented below.

$$\textsc{Return}\ \frac{\Theta;\Gamma;\Delta \vdash m : A}{\Theta;\Gamma;\Delta \vdash \textbf{return}\ m : C(A)} \qquad \textsc{Bind}\ \frac{\Gamma \vdash B : s \qquad \Theta_1;\Gamma;\Delta_1 \vdash m : C(A) \qquad \Theta_2;\Gamma, x : A;\Delta_2, x :_r A \vdash n : C(B)}{\Theta_1 \cup \Theta_2;\Gamma;\Delta_1 \cup \Delta_2 \vdash \textbf{let}\ x \Leftarrow m\ \textbf{in}\ n : C(B)}$$

Session Typing. The session typing rules govern the typing of channels and concurrency primitives. The rules are presented below.

$$\textsc{Channel-CH}\ \frac{\Gamma;\Delta \vdash \qquad \epsilon \vdash A : \textbf{proto} \qquad \Delta \triangleright \mathsf{U}}{c :_{\mathsf{L}} \textbf{ch}\langle A\rangle;\Gamma;\Delta \vdash c : \textbf{ch}\langle A\rangle} \qquad \textsc{Channel-HC}\ \frac{\Gamma;\Delta \vdash \qquad \epsilon \vdash A : \textbf{proto} \qquad \Delta \triangleright \mathsf{U}}{c :_{\mathsf{L}} \textbf{hc}\langle A\rangle;\Gamma;\Delta \vdash c : \textbf{hc}\langle A\rangle}$$

$$\textsc{Explicit-Send-CH}\ \frac{\Theta;\Gamma;\Delta \vdash m : \textbf{ch}\langle !(x : A).\ B\rangle}{\Theta;\Gamma;\Delta \vdash \textbf{send}\ m : \Pi_{\mathsf{L}}(x : A).C(\textbf{ch}\langle B\rangle)} \qquad \textsc{Explicit-Send-HC}\ \frac{\Theta;\Gamma;\Delta \vdash m : \textbf{hc}\langle ?(x : A).\ B\rangle}{\Theta;\Gamma;\Delta \vdash \textbf{send}\ m : \Pi_{\mathsf{L}}(x : A).C(\textbf{hc}\langle B\rangle)}$$

$$\textsc{Implicit-Send-CH}\ \frac{\Theta;\Gamma;\Delta \vdash m : \textbf{ch}\langle !\{x : A\}.\ B\rangle}{\Theta;\Gamma;\Delta \vdash \underline{\textbf{send}}\ m : \Pi_{\mathsf{L}}\{x : A\}.C(\textbf{ch}\langle B\rangle)} \qquad \textsc{Implicit-Send-HC}\ \frac{\Theta;\Gamma;\Delta \vdash m : \textbf{hc}\langle ?\{x : A\}.\ B\rangle}{\Theta;\Gamma;\Delta \vdash \underline{\textbf{send}}\ m : \Pi_{\mathsf{L}}\{x : A\}.C(\textbf{hc}\langle B\rangle)}$$

$$\textsc{Explicit-Recv-CH}\ \frac{\Theta;\Gamma;\Delta \vdash m : \textbf{ch}\langle ?(x : A).\ B\rangle}{\Theta;\Gamma;\Delta \vdash \textbf{recv}\ m : C(\Sigma_{\mathsf{L}}(x : A).\textbf{ch}\langle B\rangle)} \qquad \textsc{Explicit-Recv-HC}\ \frac{\Theta;\Gamma;\Delta \vdash m : \textbf{hc}\langle !(x : A).\ B\rangle}{\Theta;\Gamma;\Delta \vdash \textbf{recv}\ m : C(\Sigma_{\mathsf{L}}(x : A).\textbf{hc}\langle B\rangle)}$$

$$\textsc{Implicit-Recv-CH}\ \frac{\Theta;\Gamma;\Delta \vdash m : \textbf{ch}\langle ?\{x : A\}.\ B\rangle}{\Theta;\Gamma;\Delta \vdash \underline{\textbf{recv}}\ m : C(\Sigma_{\mathsf{L}}\{x : A\}.\textbf{ch}\langle B\rangle)} \qquad \textsc{Implicit-Recv-HC}\ \frac{\Theta;\Gamma;\Delta \vdash m : \textbf{hc}\langle !\{x : A\}.\ B\rangle}{\Theta;\Gamma;\Delta \vdash \underline{\textbf{recv}}\ m : C(\Sigma_{\mathsf{L}}\{x : A\}.\textbf{hc}\langle B\rangle)}$$

$$\textsc{Fork}\ \frac{\Theta;\Gamma, x : \textbf{ch}\langle A\rangle;\Delta, x :_{\mathsf{L}} \textbf{ch}\langle A\rangle \vdash m : C(\text{unit})}{\Theta;\Gamma;\Delta \vdash \textbf{fork}\ (x : \textbf{ch}\langle A\rangle)\ \textbf{with}\ m : C(\textbf{hc}\langle A\rangle)} \qquad \textsc{Close}\ \frac{\Theta;\Gamma;\Delta \vdash m : \textbf{ch}\langle \mathbf{1}\rangle}{\Theta;\Gamma;\Delta \vdash \textbf{close}\ m : C(\text{unit})} \qquad \textsc{Wait}\ \frac{\Theta;\Gamma;\Delta \vdash m : \textbf{hc}\langle \mathbf{1}\rangle}{\Theta;\Gamma;\Delta \vdash \textbf{wait}\ m : C(\text{unit})}$$

C.3 Process Level

The typing judgment for the process level has the form $\Theta \Vdash P$. This judgment states that under the channel context Θ, process P is well-typed. Unlike the logical and program levels which can type term that contain free variables, the process level only types processes whose terms are closed. Hence, there are no logical or program contexts in the process typing judgment.

$$\textsc{Expr}\ \frac{\Theta;\epsilon;\epsilon \vdash m : C(\text{unit})}{\Theta \Vdash \langle m\rangle} \qquad \textsc{Par}\ \frac{\Theta_1 \Vdash P \qquad \Theta_2 \Vdash Q}{\Theta_1 \cup \Theta_2 \Vdash P \mid Q} \qquad \textsc{Scope}\ \frac{\Theta, c :_{\mathsf{L}} \textbf{ch}\langle A\rangle, d :_{\mathsf{L}} \textbf{hc}\langle A\rangle \Vdash P}{\Theta \Vdash \nu cd.P}$$

D Operational Semantics

In this section, we present the operational semantics of TLL_C. Similarly to the typing rules, we organize the presentation of the semantics into the logical level, program level, and process level.

D.1 Logical Level

The semantics of the logical level is defined in terms of the *parallel reduction* relation $m \Rrightarrow m'$. This relation allows multiple redexes to be reduced simultaneously.

Core Reduction. The parallel reduction for core functional terms is defined as follows:

$$\textsc{PStep-Var}\quad \frac{}{x \Rrightarrow x} \qquad \textsc{PStep-Sort}\quad \frac{}{s \Rrightarrow s}$$

$$\textsc{PStep-Explicit-Fun}\quad \frac{A \Rrightarrow A' \qquad B \Rrightarrow B'}{\Pi_s(x : A).B \Rrightarrow \Pi_s(x : A').B'} \qquad \textsc{PStep-Implicit-Fun}\quad \frac{A \Rrightarrow A' \qquad B \Rrightarrow B'}{\Pi_s\{x : A\}.B \Rrightarrow \Pi_s\{x : A'\}.B'}$$

$$\textsc{PStep-Explicit-Lam}\quad \frac{A \Rrightarrow A' \qquad B \Rrightarrow B'}{\lambda_s(x : A).B \Rrightarrow \lambda_s(x : A').B'} \qquad \textsc{PStep-Implicit-Lam}\quad \frac{A \Rrightarrow A' \qquad B \Rrightarrow B'}{\lambda_s\{x : A\}.B \Rrightarrow \lambda_s\{x : A'\}.B'}$$

$$\textsc{PStep-Explicit-App}\quad \frac{m \Rrightarrow m' \qquad n \Rrightarrow n'}{m\ n \Rrightarrow m'\ n'} \qquad \textsc{PStep-Implicit-App}\quad \frac{m \Rrightarrow m' \qquad n \Rrightarrow n'}{m\ \{n\} \Rrightarrow m'\ \{n'\}}$$

$$\textsc{PStep-Explicit-}\beta\quad \frac{m \Rrightarrow m' \qquad n \Rrightarrow n'}{(\lambda_s(x : A).m)\ n \Rrightarrow m'[n'/x]} \qquad \textsc{PStep-Implicit-}\beta\quad \frac{m \Rrightarrow m' \qquad n \Rrightarrow n'}{(\lambda_s\{x : A\}.m)\ \{n\} \Rrightarrow m'[n'/x]}$$

$$\textsc{PStep-Explicit-Sum}\quad \frac{A \Rrightarrow A' \qquad B \Rrightarrow B'}{\Sigma_s(x : A).B \Rrightarrow \Sigma_s(x : A').B'} \qquad \textsc{PStep-Implicit-Sum}\quad \frac{A \Rrightarrow A' \qquad B \Rrightarrow B'}{\Sigma_s\{x : A\}.B \Rrightarrow \Sigma_s\{x : A'\}.B'}$$

$$\textsc{PStep-Explicit-Pair}\quad \frac{m \Rrightarrow m' \qquad n \Rrightarrow n'}{\langle m, n\rangle_s \Rrightarrow \langle m', n'\rangle_s} \qquad \textsc{PStep-Implicit-Pair}\quad \frac{m \Rrightarrow m' \qquad n \Rrightarrow n'}{\langle\{m\}, n\rangle_s \Rrightarrow \langle\{m'\}, n'\rangle_s}$$

$$\textsc{PStep-SumElim}\quad \frac{A \Rrightarrow A' \qquad m \Rrightarrow m' \qquad n \Rrightarrow n'}{\mathrm{R}^{\Sigma}_{[z]A}(m, [x, y]n) \Rrightarrow \mathrm{R}^{\Sigma}_{[z]A'}(m', [x, y]n')}$$

$$\textsc{PStep-Explicit-PairElim}\quad \frac{m_1 \Rrightarrow m_1' \qquad m_2 \Rrightarrow m_2' \qquad n \Rrightarrow n'}{\mathrm{R}^{\Sigma}_{[z]A}(\langle m_1, m_2\rangle_s, [x, y]n) \Rrightarrow n'[m_1/x, m_2/y]}$$

$$\textsc{PStep-Implicit-PairElim}\quad \frac{m_1 \Rrightarrow m_1' \qquad m_2 \Rrightarrow m_2' \qquad n \Rrightarrow n'}{\mathrm{R}^{\Sigma}_{[z]A}(\langle\{m_1\}, m_2\rangle_s, [x, y]n) \Rrightarrow n'[m_1/x, m_2/y]}$$

Data Reduction. The parallel reduction for data terms is defined as follows:

$$\textsc{PStep-Unit}\quad \frac{}{\mathsf{unit} \Rrightarrow \mathsf{unit}} \qquad \textsc{PStep-UnitVal}\quad \frac{}{() \Rrightarrow ()} \qquad \textsc{PStep-Bool}\quad \frac{}{\mathsf{bool} \Rrightarrow \mathsf{bool}} \qquad \textsc{PStep-True}\quad \frac{}{\mathsf{true} \Rrightarrow \mathsf{true}} \qquad \textsc{PStep-False}\quad \frac{}{\mathsf{false} \Rrightarrow \mathsf{false}}$$

$$\textsc{PStep-BoolElim}\quad \frac{A \Rrightarrow A' \qquad m \Rrightarrow m' \qquad n_1 \Rrightarrow n_1' \qquad n_2 \Rrightarrow n_2'}{\mathrm{R}^{\mathsf{bool}}_{[z]A}(m, n_1, n_2) \Rrightarrow \mathrm{R}^{\mathsf{bool}}_{[z]A'}(m', n_1', n_2')}$$

$$\textsc{PStep-TrueElim}\quad \frac{n_1 \Rrightarrow n_1'}{\mathrm{R}^{\mathsf{bool}}_{[z]A}(\mathsf{true}, n_1, n_2) \Rrightarrow n_1'} \qquad \textsc{PStep-FalseElim}\quad \frac{n_2 \Rrightarrow n_2'}{\mathrm{R}^{\mathsf{bool}}_{[z]A}(\mathsf{false}, n_1, n_2) \Rrightarrow n_2'}$$

Monadic Reduction. The parallel reduction for monadic terms is defined as follows:

$$\textsc{PStep-}C\textsc{Type}\quad \frac{A \Rrightarrow A'}{C(A) \Rrightarrow C(A')} \qquad \textsc{PStep-Return}\quad \frac{m \Rrightarrow m'}{\mathbf{return}\ m \Rrightarrow \mathbf{return}\ m'}$$

$$\textsc{PStep-Bind}\quad \frac{m \Rrightarrow m' \qquad n \Rrightarrow n'}{\mathbf{let}\ x \Leftarrow m\ \mathbf{in}\ n \Rrightarrow \mathbf{let}\ x \Leftarrow m'\ \mathbf{in}\ n'}$$

$$\textsc{PStep-ReturnBind}\quad \frac{m \Rrightarrow m' \qquad n \Rrightarrow n'}{\mathbf{let}\ x \Leftarrow \mathbf{return}\ m\ \mathbf{in}\ n \Rrightarrow n'[m'/x]}$$

Session Reduction. The parallel reduction for protocols, channels and concurrency primitives are defined as follows:

$$\textsc{PStep-Proto}\quad \frac{}{\textbf{proto} \Rightarrow \textbf{proto}} \qquad \textsc{PStep-End}\quad \frac{}{\mathbf{1} \Rightarrow \mathbf{1}}$$

$$\textsc{PStep-RecProto}\quad \frac{A \Rightarrow A' \qquad m \Rightarrow m'}{\mu(x : A).m \Rightarrow \mu(x : A').m'} \qquad \textsc{PStep-RecUnfold}\quad \frac{A \Rightarrow A' \qquad m \Rightarrow m'}{\mu(x : A).m \Rightarrow m'[(\mu(x : A').m')/x]}$$

$$\textsc{PStep-Explicit-Action}\quad \frac{A \Rightarrow A' \qquad B \Rightarrow B'}{\rho(x : A).\, B \Rightarrow \rho(x : A').\, B'} \qquad \textsc{PStep-Implicit-Action}\quad \frac{A \Rightarrow A' \qquad B \Rightarrow B'}{\rho\{x : A\}.\, B \Rightarrow \rho\{x : A'\}.\, B'}$$

$$\textsc{PStep-CH}\quad \frac{A \Rightarrow A'}{\textbf{ch}\langle A\rangle \Rightarrow \textbf{ch}\langle A'\rangle} \qquad \textsc{PStep-HC}\quad \frac{A \Rightarrow A'}{\textbf{hc}\langle A\rangle \Rightarrow \textbf{hc}\langle A'\rangle}$$

$$\textsc{PStep-Channel}\quad \frac{}{c \Rightarrow d} \qquad \textsc{PStep-Fork}\quad \frac{A \Rightarrow A' \qquad m \Rightarrow m'}{\textbf{fork}\ (x : A)\ \textbf{with}\ m \Rightarrow \textbf{fork}\ (x : A')\ \textbf{with}\ m'} \qquad \textsc{PStep-Explicit-Send}\quad \frac{m \Rightarrow m'}{\textbf{send}\, m \Rightarrow \textbf{send}\, m'}$$

$$\textsc{PStep-Implicit-Send}\quad \frac{m \Rightarrow m'}{\underline{\textbf{send}}\, m \Rightarrow \underline{\textbf{send}}\, m'} \qquad \textsc{PStep-Explicit-Recv}\quad \frac{m \Rightarrow m'}{\textbf{recv}\, m \Rightarrow \textbf{recv}\, m'}$$

$$\textsc{PStep-Implicit-Recv}\quad \frac{m \Rightarrow m'}{\underline{\textbf{recv}}\, m \Rightarrow \underline{\textbf{recv}}\, m'} \qquad \textsc{PStep-Close}\quad \frac{m \Rightarrow m'}{\textbf{close}\, m \Rightarrow \textbf{close}\, m'}$$

$$\textsc{PStep-Wait}\quad \frac{m \Rightarrow m'}{\textbf{wait}\, m \Rightarrow \textbf{wait}\, m'}$$

Convertibility Relation. The convertibility relation $A \simeq B$ is the reflexive, symmetric and transitive closure of the parallel reduction relation. It can be inductively defined as follows:

$$\textsc{Conv-Refl}\quad \frac{}{A \simeq A} \qquad \textsc{Conv-PStep}\quad \frac{A \simeq B \qquad B \Rightarrow C}{A \simeq C} \qquad \textsc{Conv-PStep-Rev}\quad \frac{A \simeq B \qquad C \Rightarrow B}{A \simeq C}$$

Note that the program level CONVERSION rule (Section C.2) also uses this convertibility relation.

D.2 Program Level

The semantics of the program level is defined in terms of a small-step reduction relation $m \rightsquigarrow\!\!\!\!\rightarrow m'$. Unlike the logical level which has a non-deterministic reduction strategy, the program level follows call-by-value evaluation. Arguments are fully evaluated before substitution into functions.

Core Reduction. The small-step reduction for core functional terms is defined as follows:

$$\frac{m \rightsquigarrow\!\!\!\!\rightarrow m'}{m\ n \rightsquigarrow\!\!\!\!\rightarrow m'\ n}\ \textsc{Step-Explicit-App}_1 \qquad \frac{n \rightsquigarrow\!\!\!\!\rightarrow n'}{m\ n \rightsquigarrow\!\!\!\!\rightarrow m\ n'}\ \textsc{Step-Explicit-App}_2 \qquad \frac{m \rightsquigarrow\!\!\!\!\rightarrow m'}{m\ \{n\} \rightsquigarrow\!\!\!\!\rightarrow m'\ \{n\}}\ \textsc{Step-Implicit-App}_1 \qquad \frac{}{(\lambda_s(x:A).m)\ v \rightsquigarrow\!\!\!\!\rightarrow m[v/x]}\ \textsc{Step-Explicit-}\beta$$

$$\frac{}{(\lambda_s\{x:A\}.m)\ \{n\} \rightsquigarrow\!\!\!\!\rightarrow m[n/x]}\ \textsc{Step-Implicit-}\beta \qquad \frac{m \rightsquigarrow\!\!\!\!\rightarrow m'}{\langle m, n\rangle_s \rightsquigarrow\!\!\!\!\rightarrow \langle m', n\rangle_s}\ \textsc{Step-Explicit-Pair}_1 \qquad \frac{n \rightsquigarrow\!\!\!\!\rightarrow n'}{\langle m, n\rangle_s \rightsquigarrow\!\!\!\!\rightarrow \langle m, n'\rangle_s}\ \textsc{Step-Explicit-Pair}_2$$

$$\frac{n \rightsquigarrow\!\!\!\!\rightarrow n'}{\langle \{m\}, n\rangle_s \rightsquigarrow\!\!\!\!\rightarrow \langle \{m\}, n'\rangle_s}\ \textsc{Step-Implicit-Pair}_2 \qquad \frac{m \rightsquigarrow\!\!\!\!\rightarrow m'}{\mathrm{R}^{\Sigma}_{[z]A}(m, [x,y]n) \rightsquigarrow\!\!\!\!\rightarrow \mathrm{R}^{\Sigma}_{[z]A}(m', [x,y]n)}\ \textsc{Step-SumElim}_1$$

$$\frac{}{\mathrm{R}^{\Sigma}_{[z]A}(\langle u, v\rangle_s, [x,y]n) \rightsquigarrow\!\!\!\!\rightarrow n[u/x, v/y]}\ \textsc{Step-Explicit-PairElim} \qquad \frac{}{\mathrm{R}^{\Sigma}_{[z]A}(\langle \{m\}, v\rangle_s, [x,y]n) \rightsquigarrow\!\!\!\!\rightarrow n[m/x, v/y]}\ \textsc{Step-Implicit-PairElim}$$

Data Reduction. The small-step reduction for data terms is defined as follows:

$$\frac{m \rightsquigarrow\!\!\!\!\rightarrow m'}{\mathrm{R}^{\mathsf{bool}}_{[z]A}(m, n_1, n_2) \rightsquigarrow\!\!\!\!\rightarrow \mathrm{R}^{\mathsf{bool}}_{[z]A}(m', n_1, n_2)}\ \textsc{Step-BoolElim} \qquad \frac{}{\mathrm{R}^{\mathsf{bool}}_{[z]A}(\mathsf{true}, n_1, n_2) \rightsquigarrow\!\!\!\!\rightarrow n_1}\ \textsc{Step-TrueElim} \qquad \frac{}{\mathrm{R}^{\mathsf{bool}}_{[z]A}(\mathsf{false}, n_1, n_2) \rightsquigarrow\!\!\!\!\rightarrow n_2}\ \textsc{Step-FalseElim}$$

Monadic Reduction. The small-step reduction for monadic terms is defined as follows:

$$\frac{m \rightsquigarrow\!\!\!\!\rightarrow m'}{\textbf{return}\ m \rightsquigarrow\!\!\!\!\rightarrow \textbf{return}\ m'}\ \textsc{Step-Return} \qquad \frac{m \rightsquigarrow\!\!\!\!\rightarrow m'}{\textbf{let}\ x \Leftarrow m\ \textbf{in}\ n \rightsquigarrow\!\!\!\!\rightarrow \textbf{let}\ x \Leftarrow m'\ \textbf{in}\ n}\ \textsc{Step-Bind} \qquad \frac{}{\textbf{let}\ x \Leftarrow \textbf{return}\ v\ \textbf{in}\ n \rightsquigarrow\!\!\!\!\rightarrow n[v/x]}\ \textsc{Step-ReturnBind}$$

Session Reduction. The small-step reduction for session terms is defined as follows:

$$\frac{m \rightsquigarrow\!\!\!\!\rightarrow m'}{\textbf{send}\ m \rightsquigarrow\!\!\!\!\rightarrow \textbf{send}\ m'}\ \textsc{Step-Explicit-Send} \qquad \frac{m \rightsquigarrow\!\!\!\!\rightarrow m'}{\underline{\textbf{send}}\ m \rightsquigarrow\!\!\!\!\rightarrow \underline{\textbf{send}}\ m'}\ \textsc{Step-Implicit-Send} \qquad \frac{m \rightsquigarrow\!\!\!\!\rightarrow m'}{\textbf{recv}\ m \rightsquigarrow\!\!\!\!\rightarrow \textbf{recv}\ m'}\ \textsc{Step-Explicit-Recv} \qquad \frac{m \rightsquigarrow\!\!\!\!\rightarrow m'}{\underline{\textbf{recv}}\ m \rightsquigarrow\!\!\!\!\rightarrow \underline{\textbf{recv}}\ m'}\ \textsc{Step-Implicit-Recv}$$

$$\frac{m \rightsquigarrow\!\!\!\!\rightarrow m'}{\textbf{close}\ m \rightsquigarrow\!\!\!\!\rightarrow \textbf{close}\ m'}\ \textsc{Step-Close} \qquad \frac{m \rightsquigarrow\!\!\!\!\rightarrow m'}{\textbf{wait}\ m \rightsquigarrow\!\!\!\!\rightarrow \textbf{wait}\ m'}\ \textsc{Step-Wait}$$

D.3 Process Level

The semantics of the process level is defined in terms of a small-step reduction relation $P \Rrightarrow Q$. This relation is what gives TLL_C its concurrent behavior. Before we present the reduction rules, we first define the notion of *structural congruence* $\equiv$ which identifies processes that are the same up to reordering of parallel components and renaming of bound channels.

Structural Congruence. The structural congruence relation $\equiv$ is defined as follows:

$$P \mid Q \equiv Q \mid P \qquad O \mid (P \mid Q) \equiv (O \mid P) \mid Q \qquad P \mid \langle \textbf{return}\ () \rangle \equiv P$$

$$\nu cd.P \mid Q \equiv \nu cd.(P \mid Q) \qquad \nu cd.P \equiv \nu dc.P \qquad \nu cd.\nu c'd'.P \equiv \nu c'd'.\nu cd.P$$

Process Reduction. The small-step reduction for processes is defined as follows:

$$
\begin{array}{ll}
(\textsc{Proc-Fork}) & \langle \textbf{let}\ x \Leftarrow \textbf{fork}\ (y : A)\ \textbf{with}\ m\ \textbf{in}\ n \rangle \Rrightarrow \nu cd.(\langle n[c/x] \rangle \mid \langle m[d/y] \rangle) \\
(\textsc{Proc-End}) & \nu cd.(\langle \textbf{let}\ x \Leftarrow \textbf{close}\ c\ \textbf{in}\ m \rangle \mid \langle \textbf{let}\ y \Leftarrow \textbf{wait}\ d\ \textbf{in}\ n \rangle) \\
& \quad \Rrightarrow \langle \textbf{let}\ x \Leftarrow \textbf{return}\ ()\ \textbf{in}\ m \rangle \mid \langle \textbf{let}\ y \Leftarrow \textbf{return}\ ()\ \textbf{in}\ n \rangle \\
(\textsc{Proc-Com}) & \nu cd.(\langle \textbf{let}\ x \Leftarrow \textbf{send}\ c\ v\ \textbf{in}\ m \rangle \mid \langle \textbf{let}\ y \Leftarrow \textbf{recv}\ d\ \textbf{in}\ n \rangle) \\
& \quad \Rrightarrow \nu cd.(\langle \textbf{let}\ x \Leftarrow \textbf{return}\ c\ \textbf{in}\ m \rangle \mid \langle \textbf{let}\ y \Leftarrow \textbf{return}\ \langle v, d \rangle_{\mathsf{L}}\ \textbf{in}\ n \rangle) \\
(\textsc{Proc-}\underline{\textsc{Com}}) & \nu cd.(\langle \textbf{let}\ x \Leftarrow \underline{\textbf{send}}\ c\ \{o\}\ \textbf{in}\ m \rangle \mid \langle \textbf{let}\ y \Leftarrow \underline{\textbf{recv}}\ d\ \textbf{in}\ n \rangle) \\
& \quad \Rrightarrow \nu cd.(\langle \textbf{let}\ x \Leftarrow \textbf{return}\ c\ \textbf{in}\ m \rangle \mid \langle \textbf{let}\ y \Leftarrow \textbf{return}\ \langle \{o\}, d \rangle_{\mathsf{L}}\ \textbf{in}\ n \rangle)
\end{array}
$$

$$
\begin{array}{cccc}
(\textsc{Proc-Expr}) & (\textsc{Proc-Par}) & (\textsc{Proc-Scope}) & (\textsc{Proc-Congr}) \\
\dfrac{m \rightsquigarrow\!\!\!\!\!\rightarrow m'}{\langle m \rangle \Rrightarrow \langle m' \rangle} &
\dfrac{P \Rrightarrow Q}{O \mid P \Rrightarrow O \mid Q} &
\dfrac{P \Rrightarrow Q}{\nu cd.P \Rrightarrow \nu cd.Q} &
\dfrac{P \equiv P' \qquad P' \Rrightarrow Q' \qquad Q' \equiv Q}{P \Rrightarrow Q}
\end{array}
$$

E Meta-Theory

In this section, we study the meta-theory of TLL_C. The results are classified into 4 categories.

(1) **Compatibility**: the extensions of TLL_C are compatible with the underlying TLL theory.
(2) **Session Fidelity**: processes follow the protocols specified by their session types.
(3) **Global Progress**: well-typed reachable processes do not deadlock.
(4) **Erasure Safety**: evaluation of erased terms and processes is safe.

E.1 Compatibility

To show that the TLL_C extensions made are compatible with the theory of TLL, we prove the TLL_C terms enjoy the same properties as TLL ones.

Confluence. We show that the logical reduction relation is confluent. This property is important as it ensures that type convertibility can be checked regardless of the order in which reductions are applied. Confluence is easy to prove here as the parallel reduction satisfies the diamond property.

Lemma E.1 (Diamond Property). *If $m \Rightarrow m_1$ and $m \Rightarrow m_2$, then there exists m' such that $m_1 \Rightarrow m'$ and $m_2 \Rightarrow m'$.*

Proof. By induction on the structure of the parallel reduction. □

Lemma E.2. *If $m \Rightarrow m_1$ and $m \Rightarrow^* m_2$, then there exists m' such that $m_1 \Rightarrow^* m'$ and $m_2 \Rightarrow m'$.*

Proof. By induction on the derivation of $m \Rightarrow^* m_2$ and Lemma E.1. □

Theorem E.3 (Confluence). *If $m \Rightarrow^* m_1$ and $m \Rightarrow^* m_2$, then there exists m' such that $m_1 \Rightarrow^* m'$ and $m_2 \Rightarrow^* m'$.*

Proof. By induction on the derivation of $m \Rightarrow^* m_2$ and Lemma E.2. □

The validity of the confluence property allows us to prove the injectivity of the convertibility relation for types.

Corollary E.4. *$s_1 \simeq s_2$ implies $s_1 = s_2$.*

Corollary E.5. *$\Pi_s\{x : A\}.B \simeq \Pi_{s'}\{x : A'\}.B'$ implies $s = s'$, $A \simeq A'$, and $B \simeq B'$.*

Corollary E.6. *$\Pi_s(x : A).B \simeq \Pi_{s'}(x : A').B'$ implies $s = s'$, $A \simeq A'$, and $B \simeq B'$.*

Corollary E.7. *$\Sigma_s\{x : A\}.B \simeq \Sigma_{s'}\{x : A'\}.B'$ implies $s = s'$, $A \simeq A'$, and $B \simeq B'$.*

Corollary E.8. *$\Sigma_s(x : A).B \simeq \Sigma_{s'}(x : A').B'$ implies $s = s'$, $A \simeq A'$, and $B \simeq B'$.*

Corollary E.9. *$C(A) \simeq C(B)$ implies $A \simeq B$.*

Corollary E.10. *$\rho\{x : A\}.\ B \simeq \rho'\{x : A'\}.\ B'$ implies $\rho = \rho'$, $A \simeq A'$, and $B \simeq B'$.*

Corollary E.11. *$\rho(x : A).\ B \simeq \rho'(x : A').\ B'$ implies $\rho = \rho'$, $A \simeq A'$, and $B \simeq B'$.*

Corollary E.12. *$\boldsymbol{ch}\langle A\rangle \simeq \boldsymbol{ch}\langle B\rangle$ implies $A \simeq B$.*

Corollary E.13. *$\boldsymbol{hc}\langle A\rangle \simeq \boldsymbol{hc}\langle B\rangle$ implies $A \simeq B$.*

Weakening. Weakening allows for the addition of unused variables to a typing context. The logical level type system allows weakening as it is a fully structural type system. On the other hand, the program level type system only allows weakening of unrestricted variables, i.e. variables whose types inhabit U.

Lemma E.14 (Renaming Arity). *Given renaming ξ, if there is A arity(**proto**), then there is $A[\xi]$ arity(**proto**).*

Proof. By induction on the structure of A. □

Lemma E.15 (Renaming Guarded). *Given renaming ξ, if there is $\forall x, y, \xi(x) = \xi(y) \implies x = y$, then given variable x and A guard(x), there is $A[\xi]$ guard$(\xi(x))$.*

Proof. By induction on the structure of A. □

Lemma E.16 (Logical Weakening). *If $\Gamma \vdash m : A$ and $\Gamma \vdash B : s$, then $\Gamma, x : B \vdash m : A$ where $x \notin \Gamma$.*

Proof. By induction on the derivation of $\Gamma \vdash m : A$. For more details, see file sta_weak.v of our Rocq development which uses a De Bruijn indices representation for variables. □

Lemma E.17 (Program Weakening (Explicit)). *If $\Theta; \Gamma; \Delta \vdash m : A$ and $\Gamma \vdash B : U$, then $\Theta; \Gamma, x : B; \Delta, x :_U B \vdash m : A$ where $x \notin \Gamma$.*

Proof. By induction on the derivation of $\Theta; \Gamma; \Delta \vdash m : A$. For more details, see file dyn_weak.v of our Rocq development which uses a De Bruijn indices representation for variables. □

Lemma E.18 (Program Weakening (Implicit)). *If $\Theta; \Gamma; \Delta \vdash m : A$ and $\Delta \vdash B : L$, then $\Theta; \Gamma, x : B; \Delta \vdash m : A$ where $x \notin \Delta$.*

Proof. By induction on the derivation of $\Theta; \Gamma; \Delta \vdash m : A$. For more details, see file dyn_weak.v of our Rocq development which uses a De Bruijn indices representation for variables. □

Substitution. The substitution lemma at the logical level is standard as the logical type system is completely structural. The substitution lemma at the program level is more involved as it needs to track linear variables in the program context.

Lemma E.19 (Substitution Arity). *Given substitution σ, if there is A arity(**proto**), then there is $A[\sigma]$ arity(**proto**).*

Proof. By induction on the structure of A. □

Lemma E.20 (Substitution Guarded). *Given substitution σ and variables x, y and term A, if there is $\forall z, x \neq z \implies (\sigma\ z)$ guard(y) and A guard(x), then $A[\sigma]$ guard(y).*

Proof. By induction on the structure of A. □

Lemma E.21 (Logical Substitution). *If $\Gamma, x : B \vdash m : A$ and $\Gamma \vdash n : B$, then $\Gamma \vdash m[n/x] : A[n/x]$.*

Proof. This lemma is proved through a more general lemma involving simultaneous substitutions. For more details, see file sta_subst.v of our Rocq development. □

Corollary E.22. *If $\Gamma, x : A \vdash m : C$ and $A \simeq B$, then $\Gamma, x : B \vdash m : C$.*

Lemma E.23 (Program Substitution (Explicit)). *If $\Theta_1; \Gamma, x : B; \Delta_1, x :_s B \vdash m : A$ and $\Theta_2; \Gamma; \Delta_2 \vdash n : B$ and $\Theta_2 \triangleright s$ and $\Delta_2 \triangleright s$, then $\Theta_1 \cup \Theta_2; \Gamma; \Delta_1 \cup \Delta_2 \vdash m[n/x] : A[n/x]$.*

Proof. This lemma is proved through a more general lemma involving simultaneous substitutions. For more details, see file dyn_subst.v of our Rocq development. □

Corollary E.24. *If* $\Theta; \Gamma, x : A; \Delta \vdash m : C$ *and* $\Gamma \vdash B : s$ *and* $A \simeq B$*, then* $\Theta; \Gamma, x : B; \Delta \vdash m : C$.

Lemma E.25 (Program Substitution (Implicit)). *If* $\Theta; \Gamma, x : B; \Delta \vdash m : A$ *and* $\Gamma \vdash n : B$*, then* $\Theta; \Gamma; \Delta \vdash m[n/x] : A[n/x]$.

Proof. This lemma is proved through a more general lemma involving simultaneous substitutions. For more details, see file dyn_subst.v of our Rocq development. □

Corollary E.26. *If* $\Theta; \Gamma, x : A; \Delta, x :_s A \vdash m : C$ *and* $A \simeq B$*, then* $\Theta; \Gamma, x : B; \Delta, x :_s B \vdash m : C$.

Sort Uniqueness. Due to the fact that TLL_C utilizes the sort of types to determine (sub)structural properties of their inhabitants, it is important for types to have unique sorts. If a type could have multiple sorts, then it would be ambiguous whether its inhabitants are linear or unrestricted.

One of the main challenges in proving sort uniqueness is that there is no uniqueness of types in general. In particular, dependent pairs like $\langle \{m\}, n\rangle_s$ do not have unique typing. For this reason, we prove a weaker property of *type similarity* instead of type uniqueness. Then from type similarity we can derive sort uniqueness. We begin by defining the *head similarity* relation as follows:

$$\frac{}{\text{headSim}(x, x)} \qquad \frac{}{\text{headSim}(s, s)} \qquad \frac{\text{headSim}(B_1, B_2)}{\text{headSim}(\Pi_s(x : A_1).B_1, \Pi_s(x : A_2).B_2)}$$

$$\frac{\text{headSim}(B_1, B_2)}{\text{headSim}(\Pi_s\{x : A_1\}.B_1, \Pi_s\{x : A_2\}.B_2)} \qquad \frac{}{\text{headSim}(\lambda_s(x : A).m, \lambda_s(x : A).m)}$$

$$\frac{}{\text{headSim}(\lambda_s\{x : A\}.m, \lambda_s\{x : A\}.m)} \qquad \frac{}{\text{headSim}(m\ n, m\ n)} \qquad \frac{}{\text{headSim}(m\ \{n\}, m\ \{n\})}$$

$$\frac{}{\text{headSim}(\Sigma_s(x : A_1).B_1, \Sigma_s(x : A_2).B_2)} \qquad \frac{}{\text{headSim}(\Sigma_s\{x : A_1\}.B_1, \Sigma_s\{x : A_2\}.B_2)}$$

$$\frac{}{\text{headSim}(\langle m, n\rangle_s, \langle m, n\rangle_s)} \qquad \frac{}{\text{headSim}(\langle \{m\}, n\rangle_s, \langle \{m\}, n\rangle_s)}$$

$$\frac{}{\text{headSim}(\mathrm{R}^{\Sigma}_{[z]A}(m, [x, y]n), \mathrm{R}^{\Sigma}_{[z]A}(m, [x, y]n))} \qquad \frac{}{\text{headSim}(\mu(x : A).m, \mu(x : A).m)} \qquad \frac{}{\text{headSim}(\text{unit}, \text{unit})}$$

$$\frac{}{\text{headSim}((), ())} \qquad \frac{}{\text{headSim}(\text{bool}, \text{bool})} \qquad \frac{}{\text{headSim}(\text{true}, \text{true})} \qquad \frac{}{\text{headSim}(\text{false}, \text{false})}$$

$$\frac{}{\text{headSim}(\mathrm{R}^{\text{bool}}_{[z]A}(m, n_1, n_2), \mathrm{R}^{\text{bool}}_{[z]A}(m, n_1, n_2))} \qquad \frac{}{\text{headSim}(C(A), C(B))} \qquad \frac{}{\text{headSim}(\textbf{return}\ m, \textbf{return}\ m)}$$

$$\frac{}{\text{headSim}(\textbf{let}\ m \Leftarrow x\ \textbf{in}\ n, \textbf{let}\ m \Leftarrow x\ \textbf{in}\ n)} \qquad \frac{}{\text{headSim}(\textbf{proto}, \textbf{proto})} \qquad \frac{}{\text{headSim}(\mathbf{1}, \mathbf{1})}$$

$$\frac{}{\text{headSim}(\rho(x : A).\ B, \rho(x : A).\ B)} \qquad \frac{}{\text{headSim}(\rho\{x : A\}.\ B, \rho\{x : A\}.\ B)} \qquad \frac{}{\text{headSim}(\textbf{ch}\langle A\rangle, \textbf{ch}\langle B\rangle)}$$

$$\frac{}{\text{headSim}(\textbf{hc}\langle A\rangle, \textbf{hc}\langle B\rangle)} \qquad \frac{}{\text{headSim}(c, c)} \qquad \frac{}{\text{headSim}(\textbf{fork}\ (x : A)\ \textbf{with}\ m, \textbf{fork}\ (x : A)\ \textbf{with}\ m)}$$

$$\frac{}{\text{headSim}(\textbf{recv}\ m, \textbf{recv}\ m)} \qquad \frac{}{\text{headSim}(\underline{\textbf{recv}}\ m, \underline{\textbf{recv}}\ m)} \qquad \frac{}{\text{headSim}(\textbf{send}\ m, \textbf{send}\ m)}$$

$$\frac{}{\text{headSim}(\underline{\textbf{send}}\ m, \underline{\textbf{send}}\ m)} \qquad \frac{}{\text{headSim}(\textbf{close}\ m, \textbf{close}\ m)} \qquad \frac{}{\text{headSim}(\textbf{wait}\ m, \textbf{wait}\ m)}$$

We then define the *type similarity* relation as follows:

$$\text{sim}(A, B) \triangleq \exists A', B', A \simeq A' \wedge B \simeq B' \wedge \text{headSim}(A', B')$$

The similarity relation is naturally extended to typing contexts as follows:

$$\frac{}{\mathrm{sim}(\epsilon, \epsilon)} \qquad \frac{\mathrm{sim}(A, B) \qquad \mathrm{sim}(\Gamma_1, \Gamma_2)}{\mathrm{sim}((\Gamma_1, x : A), (\Gamma_2, x : B))}$$

The (head) similarity relation enjoys the following properties.

Lemma E.27 (HeadSim Reflexive). *For any term A, there is headSim(A, A).*

Proof. By induction on the structure of A. □

Lemma E.28 (HeadSim Symmetric). *For any headSim(A, B), there is headSim(B, A).*

Proof. By induction on the derivation of headSim(A, B). □

Lemma E.29 (HeadSim Substitution). *Given substitution σ, if there is headSim(A, B), then there is headSim$(A[\sigma], B[\sigma])$.*

Proof. By induction on the derivation of headSim(A, B). □

Lemma E.30 (Sim Reflexive). *For any term A, there is sim(A, A).*

Proof. By the reflexivity of $\simeq$ and Lemma E.27. □

Lemma E.31 (Sim Transitive Left). *For any sim(A, B) and $B \simeq C$, there is sim(A, C).*

Proof. By the transitivity of $\simeq$. □

Lemma E.32 (Sim Transitive Right). *For any sim(A, B) and $A \simeq C$, there is sim(C, B).*

Proof. By the transitivity of $\simeq$. □

Lemma E.33 (Sim Symmetric). *For any sim(A, B), there is sim(B, A).*

Proof. By the symmetry of $\simeq$ and Lemma E.28. □

Lemma E.34 (Sim Substitution). *Given substitution σ, if there is sim(A, B), then there is sim$(A[\sigma], B[\sigma])$.*

Proof. By the substitutivity of $\simeq$ and Lemma E.29. □

Lemma E.35 (Sim Sort Injective). *If sim(s_1, s_2), then $s_1 = s_2$.*

Proof. By the definition of similarity and Corollary E.4. □

Lemma E.36 (Type Similarity). *Given $\Gamma_1 \vdash m : A$ and $\Gamma_2 \vdash m : B$ and sim(Γ_1, Γ_2), then sim(A, B).*

Proof. By induction on the derivation of $\Gamma_1 \vdash m : A$. □

Theorem E.37 (Sort Uniqueness). *Given $\Gamma \vdash m : s$ and $\Gamma \vdash m : t$, then $s = t$.*

Proof. From Lemma E.27 we have sim(Γ, Γ). Then from Lemma E.36 we have sim(s, t). Finally from Lemma E.35 we have $s = t$. □

Inversion. Due to the presence of type conversion, inversion lemmas are necessary to reason about typing derivations.

Lemma E.38. *If* $\Gamma \vdash \Pi_s(x : A).B : C$*, then there exists* t *such that* $\Gamma, x : A \vdash B : t$ *and* $C \simeq s$*.*

Lemma E.39. *If* $\Gamma \vdash \Pi_s\{x : A\}.B : C$*, then there exists* t *such that* $\Gamma, x : A \vdash B : t$ *and* $C \simeq s$*.*

Lemma E.40. *If* $\Gamma \vdash \lambda_{s_1}(x : A_1).m : \Pi_{s_2}(x : A_2).B$*, then* $\Gamma, x : A_1 \vdash m : B$*.*

Lemma E.41. *If* $\Gamma \vdash \lambda_{s_1}\{x : A_1\}.m : \Pi_{s_2}\{x : A_2\}.B$*, then* $\Gamma, x : A_1 \vdash m : B$*.*

Lemma E.42. *If* $\Gamma \vdash \Sigma_t(x : A).B : C$*, then there exists* s, r *such that* $s \sqsubseteq t$ *and* $r \sqsubseteq t$ *and* $\Gamma \vdash A : s$ *and* $\Gamma, x : A \vdash B : r$ *and* $C \simeq t$*.*

Lemma E.43. *If* $\Gamma \vdash \Sigma_t\{x : A\}.B : C$*, then there exists* s, r *such that* $r \sqsubseteq t$ *and* $\Gamma \vdash A : s$ *and* $\Gamma, x : A \vdash B : r$ *and* $C \simeq t$*.*

Lemma E.44. *If* $\Gamma \vdash C(A) : B$*, then there exists* s *such that* $\Gamma \vdash A : s$ *and* $B \simeq L$*.*

Lemma E.45. *If* $\Gamma \vdash \boldsymbol{ch}\langle A\rangle : B$*, then* $\Gamma \vdash A : \boldsymbol{proto}$ *and* $B \simeq L$*.*

Lemma E.46. *If* $\Gamma \vdash \boldsymbol{hc}\langle A\rangle : B$*, then* $\Gamma \vdash A : \boldsymbol{proto}$ *and* $B \simeq L$*.*

Lemma E.47. *If* $\Gamma \vdash \rho(x : A).\ B : C$*, then* $\Gamma, x : A \vdash B : \boldsymbol{proto}$*.*

Lemma E.48. *If* $\Gamma \vdash \rho\{x : A\}.\ B : C$*, then* $\Gamma, x : A \vdash B : \boldsymbol{proto}$*.*

Lemma E.49. *If* $\Theta; \Gamma; \Delta \vdash \lambda_s(x : A_2).m : \Pi_t(x : A_1).B$*, then* $\Theta; \Gamma, x : A_1; \Delta, x :_r A_1 \vdash m : B$*.*

Lemma E.50. *If* $\Theta; \Gamma; \Delta \vdash m\ n : C$*, then there exists* $A, B, s, \Theta_1, \Theta_2, \Delta_1, \Delta_2$ *such that* $\Theta_1; \Gamma; \Delta_1 \vdash m : \Pi_s(x : A).B$ *and* $\Theta_2; \Gamma; \Delta_2 \vdash n : A$ *and* $\Theta = \Theta_1 \uplus \Theta_2$ *and* $\Delta = \Delta_1 \uplus \Delta_2$ *and* $C \simeq B[n/x]$*.*

Lemma E.51. *If* $\Theta; \Gamma; \Delta \vdash m\ \{n\} : C$*, then there exists* A, B, s *such that* $\Theta; \Gamma; \Delta \vdash m : \Pi_s\{x : A\}.B$ *and* $\Gamma \vdash n : A$ *and* $C \simeq B[n/x]$*.*

Lemma E.52. *If* $\Theta; \Gamma; \Delta \vdash \langle m, n\rangle_s : \Sigma_t(x : A).B$*, then there exists* $\Theta_1, \Theta_2, \Delta_1, \Delta_2$ *such that* $\Theta_1; \Gamma; \Delta_1 \vdash m : A$ *and* $\Theta_2; \Gamma; \Delta_2 \vdash n : B[m/x]$ *and* $\Theta = \Theta_1 \uplus \Theta_2$ *and* $\Delta = \Delta_1 \uplus \Delta_2$ *and* $s = t$*.*

Lemma E.53. *If* $\Theta; \Gamma; \Delta \vdash () : unit$*, then* $\Theta = \epsilon$ *and* $\Delta \triangleright U$*.*

Lemma E.54. *If* $\Theta; \Gamma; \Delta \vdash true : bool$*, then* $\Theta = \epsilon$ *and* $\Delta \triangleright U$*.*

Lemma E.55. *If* $\Theta; \Gamma; \Delta \vdash false : bool$*, then* $\Theta = \epsilon$ *and* $\Delta \triangleright U$*.*

Lemma E.56. *If* $\Theta; \Gamma; \Delta \vdash \boldsymbol{return}\ m : C(A)$*, then* $\Theta; \Gamma; \Delta \vdash m : A$*.*

Lemma E.57. *If* $\Theta; \Gamma; \Delta \vdash \boldsymbol{let}\ m \Leftarrow x\ \boldsymbol{in}\ n : C(B)$*, then there exists* $A, \Theta_1, \Theta_2, \Delta_1, \Delta_2$ *such that* $\Theta_1; \Gamma; \Delta_1 \vdash m : C(A)$ *and* $\Theta_2; \Gamma, x : A; \Delta_2, x :_s A \vdash n : C(B)$ *and* $\Theta = \Theta_1 \uplus \Theta_2$ *and* $\Delta = \Delta_1 \uplus \Delta_2$*. and* $x \notin FV(B)$*.*

Lemma E.58. *If* $\Theta; \Gamma; \Delta \vdash \boldsymbol{fork}\ (x : A)\ \boldsymbol{with}\ m : B$*, then there exists* A' *such that* $\Gamma \vdash A' : \boldsymbol{proto}$ *and* $A = \boldsymbol{ch}\langle A'\rangle$ *and* $B \simeq C(\boldsymbol{hc}\langle A'\rangle)$ *and* $\Theta; \Gamma, x : \boldsymbol{ch}\langle A'\rangle; \Delta, x :_L \boldsymbol{ch}\langle A'\rangle \vdash m : C(unit)$*.*

Lemma E.59. *If* $\Theta; \Gamma; \Delta \vdash \boldsymbol{send}\ m : C$*, then there exists* A, B *such that either* $C \simeq \Pi_L(x : A).C(\boldsymbol{ch}\langle B\rangle)$ *and* $\Theta; \Gamma; \Delta \vdash m : \boldsymbol{ch}\langle !(x : A).\ B\rangle$ *or* $C \simeq \Pi_L(x : A).C(\boldsymbol{hc}\langle B\rangle)$ *and* $\Theta; \Gamma; \Delta \vdash m : \boldsymbol{hc}\langle ?(x : A).\ B\rangle$*.*

Lemma E.60. *If* $\Theta; \Gamma; \Delta \vdash \underline{\boldsymbol{send}}\ m : C$*, then there exists* A, B *such that either* $C \simeq \Pi_L\{x : A\}.C(\boldsymbol{ch}\langle B\rangle)$ *and* $\Theta; \Gamma; \Delta \vdash m : \boldsymbol{ch}\langle !\{x : A\}.\ B\rangle$ *or* $C \simeq \Pi_L\{x : A\}.C(\boldsymbol{hc}\langle B\rangle)$ *and* $\Theta; \Gamma; \Delta \vdash m : \boldsymbol{hc}\langle ?\{x : A\}.\ B\rangle$*.*

Lemma E.61. *If* $\Theta; \Gamma; \Delta \vdash \boldsymbol{recv}\ m : C$*, then there exists* A, B *such that either* $C \simeq C(\Sigma_L(x : A).\boldsymbol{ch}\langle B\rangle)$ *and* $\Theta; \Gamma; \Delta \vdash m : \boldsymbol{ch}\langle ?(x : A).\ B\rangle$ *or* $C \simeq C(\Sigma_L(x : A).\boldsymbol{hc}\langle B\rangle)$ *and* $\Theta; \Gamma; \Delta \vdash m : \boldsymbol{hc}\langle !(x : A).\ B\rangle$*.*

Lemma E.62. *If* $\Theta; \Gamma; \Delta \vdash \underline{\textbf{\textit{recv}}}\, m : C$*, then there exists* A, B *such that either* $C \simeq C(\Sigma_L\{x : A\}.\textbf{\textit{ch}}\langle B\rangle)$ *and* $\Theta; \Gamma; \Delta \vdash m : \textbf{\textit{ch}}\langle ?\{x : A\}.\, B\rangle$ *or* $C \simeq C(\Sigma_L\{x : A\}.\textbf{\textit{hc}}\langle B\rangle)$ *and* $\Theta; \Gamma; \Delta \vdash m : \textbf{\textit{hc}}\langle !\{x : A\}.\, B\rangle$*.*

Lemma E.63. *If* $\Theta; \Gamma; \Delta \vdash \textbf{\textit{close}}\, m : A$*, then* $\Theta; \Gamma; \Delta \vdash m : \textbf{\textit{ch}}\langle \mathit{1}\rangle$ *and* $A \simeq C(\mathit{unit})$*.*

Lemma E.64. *If* $\Theta; \Gamma; \Delta \vdash \textbf{\textit{wait}}\, m : A$*, then* $\Theta; \Gamma; \Delta \vdash m : \textbf{\textit{hc}}\langle \mathit{1}\rangle$ *and* $A \simeq C(\mathit{unit})$*.*

Type Validity. We show that all types appearing in typing judgments are valid, i.e. they are well-sorted at the logical level.

Theorem E.65 (Logical Type Validity). *If* $\Gamma \vdash m : A$*, then there exists* s *such that* $\Gamma \vdash A : s$*.*

Proof. By induction on the derivation of $\Gamma \vdash m : A$. We will show some representative cases.

Case (Var): From the premise we have $\Gamma, x : A \vdash$ which implies $\Gamma \vdash A : s$ for some s.

Case (Explicit-Lam): From the induction hypothesis we have $\Gamma, x : A \vdash B : r$ for some r. The validity of context $\Gamma, x : A \vdash$ implies $\Gamma \vdash A : s$ for some s. Then from Explicit-Fun we have $\Gamma \vdash \Pi_s(x : A).B : s$ which concludes this case.

Case (Explicit-App): From the induction hypothesis we have $\Gamma \vdash \Pi_s(x : A).B : r$ for some r. From Lemma E.38 we have $\Gamma, x : A \vdash B : t$ for some t. By Lemma E.21 we have $\Gamma \vdash B[n/x] : t$ which concludes this case.

Case (Explicit-Recv-CH): From the induction hypothesis we have $\Gamma \vdash \textbf{ch}\langle ?(x : A).\, B\rangle : s$ for some s. By Lemma E.45 we have $\Gamma \vdash ?(x : A).\, B : \textbf{proto}$. By Lemma E.47 we have $\Gamma, x : A \vdash B : \textbf{proto}$. From the validity of context $\Gamma, x : A \vdash$ we have $\Gamma \vdash A : t$ for some t. Applying ChType, we have $\Gamma, x : A \vdash \textbf{ch}\langle B\rangle : \mathsf{L}$. Applying Explicit-Sum, we have $\Gamma \vdash \Sigma_{\mathsf{L}}(x : A).\textbf{ch}\langle B\rangle : \mathsf{L}$. Applying CType, we have $\Gamma \vdash C(\Sigma_{\mathsf{L}}(x : A).\textbf{ch}\langle B\rangle) : \mathsf{L}$ which concludes this case. □

To show that the types appearing in program level typing judgments are valid, we first prove the lifting theorem which allows us to lift programs to the logical level.

Theorem E.66 (Lifting). *If* $\Theta; \Gamma; \Delta \vdash m : A$*, then* $\Gamma \vdash m : A$*.*

Proof. By induction on the derivation of $\Theta; \Gamma; \Delta \vdash m : A$. □

Theorem E.67 (Program Type Validity). *If* $\Theta; \Gamma; \Delta \vdash m : A$*, then there exists* s *such that* $\Gamma \vdash A : s$*.*

Proof. Immediate from Theorem E.66 and Theorem E.65. □

Subject Reduction. We show that both the logical and program level type systems enjoy subject reduction under logical and program reductions respectively.

Lemma E.68 (Arity Preservation). *If* $A \Rrightarrow A'$ *and* A *arity*$(\textbf{\textit{proto}})$*, then* A' *arity*$(\textbf{\textit{proto}})$*.*

Proof. By induction on the derivation of $A \Rrightarrow A'$. □

Lemma E.69 (Guard Preservation). *If* $A \Rrightarrow A'$ *and* A *guard*(x)*, then* A' *guard*(x)*.*

Proof. By induction on the derivation of $A \Rrightarrow A'$ and appealing to Lemma E.20. □

Theorem E.70 (Logical Subject Reduction). *If* $\Gamma \vdash m : A$ *and* $m \Rrightarrow m'$*, then* $\Gamma \vdash m' : A$*.*

Proof. By induction on the derivation of $\Gamma \vdash m : A$ and case analysis on the reduction $m \Rrightarrow m'$. We present the following representative cases.

Case (Explicit-Lam): From case analysis on the reduction, we have $A \Rrightarrow A'$ and $m \Rrightarrow m'$. From the induction hypothesis we have $\Gamma, x : A \vdash m' : B$ and $\Gamma \vdash A' : s$ for some s. By definition of convertibility, we have $A \simeq A'$. By Corollary E.22 we have $\Gamma, x : A' \vdash m' : B$. By Explicit-Lam we have $\Gamma \vdash \lambda_s(x : A').m' : \Pi_s(x : A').B$. By Conversion we have $\Gamma \vdash \lambda_s(x : A').m' : \Pi_s(x : A).B$ which concludes this case.

Case (Explicit-App): From case analysis on the reduction we have two sub-cases: (1) PStep-Explicit-App and (2) PStep-Explicit-β.

In sub-case (1) PStep-Explicit-App, we have $m \Rrightarrow m'$ and $n \Rrightarrow n'$. From the induction hypothesis we have $\Gamma \vdash m' : \Pi_s(x : A).B$ and $\Gamma \vdash n' : A$. By Explicit-App we have $\Gamma \vdash m'\ n' : B[n'/x]$. By definition of convertibility, we have $B[n/x] \simeq B[n'/x]$. By validity we have $\Gamma \vdash \Pi_s(x : A).B : t$ for some t. By Lemma E.38 we have $\Gamma, x : A \vdash B : r$ for some r. By Lemma E.21 we have $\Gamma \vdash B[n/x] : r$. By Conversion we have $\Gamma \vdash m'\ n' : B[n/x]$ which concludes this sub-case.

In sub-case (2) PStep-Explicit-β, we have $m = \lambda_s(x : A).m_0$ for some m_0 and $m_0 \Rrightarrow m_0'$ and $n \Rrightarrow n'$. By Lemma E.40 we have $\Gamma, x : A \vdash m_0 : B$. By the induction hypothesis we have $\Gamma, x : A \vdash m_0' : B$ and $\Gamma \vdash n' : A$. By Lemma E.21 we have $\Gamma \vdash m_0'[n'/x] : B[n'/x]$. By definition of convertibility, we have $B[n/x] \simeq B[n'/x]$ By validity we have $\Gamma \vdash \Pi_s(x : A).B : t$ for some t. By Lemma E.38 we have $\Gamma, x : A \vdash B : r$ for some r. By Lemma E.21 we have $\Gamma \vdash B[n/x] : r$. By Conversion we have $\Gamma \vdash m_0'[n'/x] : B[n/x]$ which concludes this sub-case.

Case (BoolElim) From case analysis on the reduction we have three sub-cases: (1) PStep-BoolElim, (2) PStep-TrueElim, and (3) PStep-FalseElim.

In sub-case (1) PStep-BoolElim, we have $A \Rrightarrow A'$, $m \Rrightarrow m'$, $n_1 \Rrightarrow n_1'$, and $n_2 \Rrightarrow n_2'$. By the induction hypothesis we have $\Gamma, z : \mathsf{bool} \vdash A' : s$, $\Gamma \vdash m' : \mathsf{bool}$, $\Gamma \vdash n_1' : A[\mathsf{true}/z]$, and $\Gamma \vdash n_2' : A[\mathsf{false}/z]$. By definition of convertibility, we have $A \simeq A'$. By Lemma E.21 we have $\Gamma \vdash A'[\mathsf{true}/z] : s$ and $\Gamma \vdash A'[\mathsf{false}/z] : s$. By Conversion we have $\Gamma \vdash n_1' : A'[\mathsf{true}/z]$ and $\Gamma \vdash n_2' : A'[\mathsf{false}/z]$. By BoolElim we have $\Gamma \vdash \mathrm{R}^{\mathsf{bool}}_{[z]A'}(m', n_1', n_2') : A'[m'/z]$. By definition of convertibility, we have $A[m/z] \simeq A'[m'/z]$. By Lemma E.21 we have $\Gamma \vdash A[m/z] : s$. By Conversion we have $\Gamma \vdash \mathrm{R}^{\mathsf{bool}}_{[z]A'}(m', n_1', n_2') : A[m/z]$ which concludes this sub-case.

Case (RecProto): From case analysis on the reduction we have two sub-cases: (1) PStep-RecProto and (2) PStep-RecUnfold.

In sub-case (1) PStep-RecProto, we have $A \Rrightarrow A'$ and $m \Rrightarrow m'$. By validity of context $\Gamma, x : A \vdash$ we have $\Gamma \vdash A : s$ for some s. By the induction hypothesis we have $\Gamma, x : A \vdash m' : A$ and $\Gamma \vdash A' : s$. By definition of convertibility, we have $A \simeq A'$. By Corollary E.22 we have $\Gamma, x : A' \vdash m' : A$. By Conversion we have $\Gamma, x : A' \vdash m' : A'$. By Lemma E.68 we have A' arity(**proto**). By Lemma E.69 we have m' guard(x). By RecProto we have $\Gamma \vdash \mu(x : A').m' : A'$. By Conversion we have $\Gamma \vdash \mu(x : A').m' : A$ which concludes this sub-case.

In sub-case (2) PStep-RecUnfold, we have $A \Rrightarrow A'$ and $m \Rrightarrow m'$. By validity of context $\Gamma, x : A \vdash$ we have $\Gamma \vdash A : s$ for some s. By the induction hypothesis we have $\Gamma, x : A \vdash m' : A$ and $\Gamma \vdash A' : s$. By definition of convertibility, we have $A \simeq A'$. By Corollary E.22 we have $\Gamma, x : A' \vdash m' : A$. By Conversion we have $\Gamma, x : A' \vdash m' : A'$. By Lemma E.68 we have A' arity(**proto**). By Lemma E.69 we have m' guard(x). By RecProto we have $\Gamma \vdash \mu(x : A').m' : A'$. By Lemma E.21 we have $\Gamma \vdash m'[\mu(x : A').m'/x] : A'$. By Conversion we have $\Gamma \vdash m'[\mu(x : A').m'/x] : A$ which concludes this sub-case. □

In order to show subject reduction at the program level, we need to show that reduction of redexes in dependent positions preserves typing. To do so, we prove the following lemma which lifts program reductions into convertibility at the logical level.

Lemma E.71 (Program Step Convertible). *If $\Theta; \epsilon; \epsilon \vdash m : A$ and $m \rightsquigarrow\!\!\!\!\!\twoheadrightarrow n$, then $m \simeq n$.*

The program level substitution lemma (Lemma E.23) requires context restrictions $\Theta_2 \triangleright s$ and $\Delta_2 \triangleright s$ for the substituted term n. To ensure that these restrictions are satisfied for *values*, we prove the following context bound lemma.

Lemma E.72 (Program Context Bound). *Given $\Theta; \Gamma; \Delta \vdash v : A$ and $\Gamma \vdash A : s$, then $\Theta \triangleright s$ and $\Delta \triangleright s$.*

Proof. By induction on the derivation of $\Theta; \Gamma; \Delta \vdash v : A$ where v is a value. We present the following representative cases.

Case (Explicit-Lam): From the premise we have $\Theta \triangleright t$ and $\Delta \triangleright t$. By Lemma E.38 we have $t \simeq s$. By injectivity of sorts (Corollary E.4) we have $t = s$ which concludes this case.

Case (Explicit-Pair): From the assumption that the pair is a value, we have $v = \langle v_1, v_2 \rangle_t$ for some t. Additionally, we have $\Theta_1; \Gamma; \Delta_1 \vdash v_1 : A$ and $\Theta_2; \Gamma; \Delta_2 \vdash v_2 : B[v_1/x]$ and $\Gamma \vdash \Sigma_t(x : A).B : t$. By Theorem E.37 we have $s = t$. By Lemma E.42 we have $\Gamma \vdash A : s$ and $\Gamma, x : A \vdash B : r$ and $s \sqsubseteq t$ and $r \sqsubseteq t$. By the induction hypothesis we have $\Theta_1 \triangleright s$ and $\Delta_1 \triangleright s$ and $\Theta_2 \triangleright r$ and $\Delta_2 \triangleright r$. These context restrictions can then be weakened to $\Theta_1 \triangleright t$ and $\Delta_1 \triangleright t$ and $\Theta_2 \triangleright t$ and $\Delta_2 \triangleright t$. The merged contexts now satisfy $\Theta_1 \cup \Theta_2 \triangleright t$ and $\Delta_1 \cup \Delta_2 \triangleright t$ which concludes this case.

Case (Return): From the premise we have $\Theta; \Gamma; \Delta \vdash m : A$ and $\Gamma \vdash C(A) : s$. By Theorem E.37 we have $s = \mathsf{L}$. The context restrictions $\Theta \triangleright \mathsf{L}$ and $\Delta \triangleright \mathsf{L}$ hold trivially which concludes this case.

Case (Channel-CH) From the premise we have $\Theta \triangleright \mathsf{U}$. From Ord-U we have $\mathsf{U} \sqsubseteq s$ which allows us to weaken the restriction into $\Theta \triangleright s$ and concluding this case.

Case (Explicit-Send-CH): From the premise we have $\Gamma \vdash \Pi_{\mathsf{L}}(x : A).C(\mathbf{ch}\langle B \rangle) : s$. By Theorem E.37 we have $s = \mathsf{L}$. The context restrictions $\Theta \triangleright \mathsf{L}$ and $\Delta \triangleright \mathsf{L}$ hold trivially. □

Theorem E.73 (Program Subject Reduction). *If* $\Theta; \epsilon; \epsilon \vdash m : A$ *and* $m \rightsquigarrow\!\!\!\!\!\rightarrow m'$*, then* $\Theta; \epsilon; \epsilon \vdash m' : A$.

Proof. By induction on the derivation of $\Theta; \epsilon; \epsilon \vdash m : A$ and case analysis on the reduction $m \rightsquigarrow\!\!\!\!\!\rightarrow m'$. We present the following representative cases.

Case (Explicit-App): From case analysis on the reduction we have three sub-cases: (1) Step-Explicit-App$_1$, (2) Step-Explicit-App$_2$, and (3) Step-Explicit-β.

In sub-case (1) Step-Explicit-App$_1$, we have $m \rightsquigarrow\!\!\!\!\!\rightarrow m'$. By the induction hypothesis we have $\Theta_1; \Gamma; \Delta_1 \vdash m' : \Pi_t(x : A).B$. By Explicit-App we have $\Theta_1 \cup \Theta_2; \Gamma; \Delta_1 \cup \Delta_2 \vdash m'\ n : B[n/x]$ which concludes this sub-case.

In sub-case (2) Step-Explicit-App$_2$, we have $n \rightsquigarrow\!\!\!\!\!\rightarrow n'$. By the induction hypothesis we have $\Theta_2; \Gamma; \Delta_2 \vdash n' : A$. By Explicit-App we have $\Theta_1 \cup \Theta_2; \Gamma; \Delta_1 \cup \Delta_2 \vdash m\ n' : B[n'/x]$. By Lemma E.71 we have $n \simeq n'$ and $B[n/x] \simeq B[n'/x]$. By Theorem E.66 we have $\Gamma \vdash n : A$. Applying Theorem E.67 on $\Theta_1; \Gamma; \Delta_1 \vdash m : \Pi_t(x : A).B$ we have $\Gamma \vdash \Pi_t(x : A).B : s$ for some s. By Lemma E.38 we have $\Gamma, x : A \vdash B : r$ for some r. By Lemma E.21 we have $\Gamma \vdash B[n/x] : r$. By Conversion we have $\Theta_1 \cup \Theta_2; \Gamma; \Delta_1 \cup \Delta_2 \vdash m\ n' : B[n/x]$ which concludes this sub-case.

In sub-case (3) Step-Explicit-β, we have $m = \lambda_t(x : A_0).m_0$ for some A_0 and m_0. By Lemma E.49 we have $\Theta_1; \Gamma, x : A; \Delta_1, x :_r A \vdash m_0 : B$. From the validity of context $\Theta_1; \Gamma, x : A; \Delta_1, x :_r A \vdash$ we have $\Gamma \vdash A : r$. By Lemma E.72 and $\Theta_2; \Gamma; \Delta_2 \vdash v : A$ and $\Gamma \vdash A : r$ we have $\Theta_2 \triangleright r$. By Lemma E.23 we have $\Theta_1 \cup \Theta_2; \Gamma; \Delta_1 \cup \Delta_2 \vdash m_0[v/x] : B[v/x]$ which concludes this sub-case.

Case (Explicit-SumElim): From case analysis on the reduction we have two sub-cases: (1) Step-SumElim$_1$ and (2) Step-Explicit-PairElim.

In sub-case (1) Step-SumElim$_1$, we have $m \rightsquigarrow\!\!\!\!\!\rightarrow m'$. By the induction hypothesis we have $\Theta_1; \Gamma; \Delta_1 \vdash m' : \Sigma_t(x : A).B$ By Explicit-SumElim we have $\Theta_1 \cup \Theta_2; \Gamma; \Delta_1 \cup \Delta_2 \vdash \mathrm{R}^{\Sigma}_{[z]C}(m', [x, y]n) : C[m'/z]$. By Lemma E.71 we have $m \simeq m'$ and $C[m/z] \simeq C[m'/z]$. By Theorem E.66 we have $\Gamma \vdash m : \Sigma_t(x : A).B$. Applying Lemma E.21 on assumption $\Gamma, z : \Sigma_t(x : A).B \vdash C : s$ and $\Gamma \vdash m : \Sigma_t(x : A).B$ we have $\Gamma \vdash C[m/z] : s$. By Conversion we have $\Theta_1 \cup \Theta_2; \Gamma; \Delta_1 \cup \Delta_2 \vdash \mathrm{R}^{\Sigma}_{[z]C}(m', [x, y]n) : C[m/z]$ which concludes this sub-case.

In sub-case (2) Step-Explicit-PairElim, we have $m = \langle u, v \rangle_t$ for some u, v and t. By Lemma E.52 we have $\Theta_{11}; \Gamma; \Delta_{11} \vdash u : A$ and $\Theta_{12}; \Gamma; \Delta_{12} \vdash v : B[u/x]$ and $\Theta_1 = \Theta_{11} \cup \Theta_{12}$ and $\Delta_1 = \Delta_{11} \cup \Delta_{12}$ and $s = t$. From the validity of context $\Theta_2; \Gamma, x : A, y : B; \Delta_2, x :_{r_1} A, y :_{r_2} B \vdash$ we have $\Gamma \vdash A : r_1$ and $\Gamma, x : A \vdash B : r_2$. Applying Theorem E.66 to $\Theta_{11}; \Gamma; \Delta_{11} \vdash u : A$ we have $\Gamma \vdash u : A$. Applying Lemma E.21 we have $\Gamma \vdash B[u/x] : r_2$. By Lemma E.72 and $\Theta_{11}; \Gamma; \Delta_{11} \vdash u : A$ and $\Gamma \vdash A : r_1$ we have $\Theta_{11} \triangleright r_1$. By

Lemma E.72 and $\Theta_{12}; \Gamma; \Delta_{12} \vdash v : B[u/x]$ and $\Gamma \vdash B[u/x] : r_2$ we have $\Theta_{12} \triangleright r_2$. By Lemma E.23 we have $\Theta_{11} \uplus \Theta_{12} \uplus \Theta_2; \Gamma; \Delta_{11} \uplus \Delta_{12} \uplus \Delta_2 \vdash n[u/x, v/y] : C[\langle u, v\rangle_t/z]$ which concludes this sub-case.

Case (BoolElim): By case analysis on the reduction we have three sub-cases: (1) Step-BoolElim$_1$, (2) Step-TrueElim, and (3) Step-FalseElim.

In sub-case (1) Step-BoolElim$_1$, we have $m \rightsquigarrow\!\!\!\!\!\twoheadrightarrow m'$. By the induction hypothesis we have $\Theta_1; \Gamma; \Delta_1 \vdash m' : \mathsf{bool}$. By BoolElim we have $\Theta_1 \uplus \Theta_2; \Gamma; \Delta_1 \uplus \Delta_2 \vdash \mathrm{R}^{\mathsf{bool}}_{[z]C}(m', n_1, n_2) : C[m'/z]$. By Lemma E.71 we have $m \simeq m'$ and $C[m/z] \simeq C[m'/z]$. By Theorem E.66 we have $\Gamma \vdash m : \mathsf{bool}$. Applying Lemma E.21 on $\Gamma, z : \mathsf{bool} \vdash C : s$ and $\Gamma \vdash m : \mathsf{bool}$ we have $\Gamma \vdash C[m/z] : s$. By Conversion we have $\Theta_1 \uplus \Theta_2; \Gamma; \Delta_1 \uplus \Delta_2 \vdash \mathrm{R}^{\mathsf{bool}}_{[z]C}(m', n_1, n_2) : C[m/z]$ which concludes this sub-case.

In sub-case (2) Step-TrueElim, we have $m = \mathsf{true}$. By Lemma E.54 we have $\Theta_1 = \epsilon$ and $\Delta_1 \triangleright \mathsf{U}$. Thus we have $\Theta_1 \uplus \Theta_2 = \Theta_2$ and $\Delta_1 \uplus \Delta_2 = \Delta_2$. The assumption $\Theta_2; \Gamma; \Delta_2 \vdash n_1 : C[\mathsf{true}/z]$ gives us the desired result which concludes this sub-case.

In sub-case (3) Step-FalseElim, we have $m = \mathsf{false}$. By Lemma E.55 we have $\Theta_1 = \epsilon$ and $\Delta_1 \triangleright \mathsf{U}$. Thus we have $\Theta_1 \uplus \Theta_2 = \Theta_2$ and $\Delta_1 \uplus \Delta_2 = \Delta_2$. The assumption $\Theta_2; \Gamma; \Delta_2 \vdash n_2 : C[\mathsf{false}/z]$ gives us the desired result which concludes this sub-case.

Case (Bind): By case analysis on the reduction we have two sub-cases: (1) Step-Bind and (2) Step-ReturnBind.

In sub-case (1) Step-Bind, we have $m \rightsquigarrow\!\!\!\!\!\twoheadrightarrow m'$. By the induction hypothesis we have $\Theta_1; \Gamma; \Delta_1 \vdash m' : C(A)$. By Bind we have $\Theta_1 \uplus \Theta_2; \Gamma; \Delta_1 \uplus \Delta_2 \vdash \textbf{let}\ x \Leftarrow m'\ \textbf{in}\ n : C$ which concludes this sub-case.

In sub-case (2) Step-ReturnBind, we have $m = \textbf{return}\, v$ for some value v. By Lemma E.56 we have $\Theta_1; \Gamma; \Delta_1 \vdash v : A$. From the validity of context $\Theta_2; \Gamma, x : A; \Delta_2, x :_r A \vdash$ we have $\Gamma \vdash A : r$. By Lemma E.72 and $\Theta_1; \Gamma; \Delta_1 \vdash v : A$ and $\Gamma \vdash A : r$ we have $\Theta_1 \triangleright r$. By Lemma E.23 we have $\Theta_1 \uplus \Theta_2; \Gamma; \Delta_1 \uplus \Delta_2 \vdash n[v/x] : C(B[v/x])$. From assumption $\Gamma \vdash B : s$ we know that $x \notin \mathrm{FV}(B)$, thus $B[v/x] = B$ and $\Theta_1 \uplus \Theta_2; \Gamma; \Delta_1 \uplus \Delta_2 \vdash n[v/x] : C(B)$. which concludes this sub-case. □

Progress. Due to the presence of concurrency primitives, the values that the program level terms can reduce to are not necessarily canonical forms. They can also be thunked monadic computations. These thunked computations will eventually be reduced by the semantics of the process level.

The following canonical forms lemmas are used to prove program progress (Theorem E.80). They are proved by induction on the typing derivation of the value.

Lemma E.74. *If $\Theta; \epsilon; \epsilon \vdash v : \Pi_s(x : A).B$ then $v = \lambda_s(x : A).m$ or $v =$ **send** u.*

Lemma E.75. *If $\Theta; \epsilon; \epsilon \vdash v : \Pi_s\{x : A\}.B$ then $v = \lambda_s\{x : A\}.m$ or $v = \underline{\textbf{send}}\, u$.*

Lemma E.76. *If $\Theta; \epsilon; \epsilon \vdash v : \Sigma_s(x : A).B$ then $v = \langle v_1, v_2\rangle_s$.*

Lemma E.77. *If $\Theta; \epsilon; \epsilon \vdash v : \Sigma_s\{x : A\}.B$ then $v = \langle\{m\}, v_2\rangle_s$.*

Lemma E.78. *If $\Theta; \epsilon; \epsilon \vdash v : \mathit{bool}$ then $v = \mathit{true}$ or $v = \mathit{false}$.*

Lemma E.79. *If $\Theta; \epsilon; \epsilon \vdash v : C(A)$ then $v =$ **return** u or there exists thunk τ such that $v = \tau$ or $v =$ **let** $x \Leftarrow \tau$ **in** m.*

Theorem E.80 (Program Progress). *If $\Theta; \epsilon; \epsilon \vdash m : A$, then m is a value or there exists m' such that $m \rightsquigarrow\!\!\!\!\!\twoheadrightarrow m'$.*

Proof. By induction on the derivation of $\Theta; \epsilon; \epsilon \vdash m : A$. We present the following cases.

Case (Var): Impossible since the context is empty.

Case (Explicit-Lam): Trivial since $\lambda_t(x : A).m_0$ is a value.

Case (Explicit-App): By the induction hypothesis we have that either m is a value or there exists m' such that $m \rightsquigarrow\!\!\!\!\!\twoheadrightarrow m'$. If $m \rightsquigarrow\!\!\!\!\!\twoheadrightarrow m'$, then we are done by Step-Explicit-App$_1$. If m is a value, by Lemma E.74 we have two sub-cases: (1) $m = \lambda_t(x : A).m_0$ and (2) $m = \textbf{send}\, u$.

In sub-case (1) $m = \lambda_t(x : A).m_0$, by the induction hypothesis on n we have that either n is a value or there exists n' such that $n \rightsquigarrow\!\!\!\!\!\twoheadrightarrow n'$. If $n \rightsquigarrow\!\!\!\!\!\twoheadrightarrow n'$, then we are done by Step-Explicit-App$_2$. If n is a value, then we are done by Step-Explicit-β.

In sub-case (2) $m = \mathbf{send}\, u$, by the induction hypothesis on n we have that either n is a value or there exists n' such that $n \rightsquigarrow\!\!\!\!\!\twoheadrightarrow n'$. If $n \rightsquigarrow\!\!\!\!\!\twoheadrightarrow n'$, then we are done by Step-Explicit-App$_2$. If n is a value v, then we are done as $\mathbf{send}\, u\, v$ is a value.

Case (Explicit-Pair): By assumption we have $\Theta_1; \Gamma; \Delta_1 \vdash m_1 : A$ and $\Theta_2; \Gamma; \Delta_2 \vdash m_2 : B[m_1/x]$. From the induction hypothesis we have that either m_1 is a value or there exists m_1' such that $m_1 \rightsquigarrow\!\!\!\!\!\twoheadrightarrow m_1'$. If $m_1 \rightsquigarrow\!\!\!\!\!\twoheadrightarrow m_1'$, then we are done by Step-Explicit-Pair$_1$. If m_1 is a value u, then by the induction hypothesis on m_2 we have that either m_2 is a value or there exists m_2' such that $m_2 \rightsquigarrow\!\!\!\!\!\twoheadrightarrow m_2'$. If $m_2 \rightsquigarrow\!\!\!\!\!\twoheadrightarrow m_2'$, then we are done by Step-Explicit-Pair$_2$. If m_2 is a value v, then we are done since $\langle u, v\rangle_t$ is a value.

Case (Explicit-SumElim): By the induction hypothesis we have that either m is a value or there exists m' such that $m \rightsquigarrow\!\!\!\!\!\twoheadrightarrow m'$. If $m \rightsquigarrow\!\!\!\!\!\twoheadrightarrow m'$, then we are done by Step-SumElim$_1$. If m is a value, by Lemma E.76 we have $m = \langle u, v\rangle_t$ for some u, v and t. We are done by Step-Explicit-PairElim.

Case (BoolElim): By the induction hypothesis we have that either m is a value or there exists m' such that $m \rightsquigarrow\!\!\!\!\!\twoheadrightarrow m'$. If $m \rightsquigarrow\!\!\!\!\!\twoheadrightarrow m'$, then we are done by Step-BoolElim$_1$. If m is a value, by Lemma E.78 we have two sub-cases: (1) m = true and (2) m = false. In sub-case (1) m = true, we are done by Step-TrueElim. In sub-case (2) m = false, we are done by Step-FalseElim.

Case (Return): By the induction hypothesis we have that either m is a value or there exists m' such that $m \rightsquigarrow\!\!\!\!\!\twoheadrightarrow m'$. If $m \rightsquigarrow\!\!\!\!\!\twoheadrightarrow m'$, then we are done by Step-Return. If m is a value, then $\mathbf{return}\, m$ is a value.

Case (Bind): By the induction hypothesis we have that either m is a value or there exists m' such that $m \rightsquigarrow\!\!\!\!\!\twoheadrightarrow m'$. If $m \rightsquigarrow\!\!\!\!\!\twoheadrightarrow m'$, then we are done by Step-Bind. If m is a value, then $\mathbf{let}\ x \Leftarrow m\ \mathbf{in}\ n$ is a value.

For the session typing rules, the term m is a thunked computation and thus a value. □

E.2 Session Fidelity

The session fidelity property ensures that well-typed processes will adhere to the protocols specified by their session types during execution. To prove this property, we first must prove that structural congruence preserves typing.

Lemma E.81 (Congruence). *Given $\Theta \Vdash P$ and $P \equiv Q$, then $\Theta \Vdash Q$.*

Proof. By induction on the derivation of $\Theta \Vdash P$ and case analysis on the congruence relation.
Case (Expr): Trivial.
Case (Par):

$$\frac{\Theta_1 \Vdash P \qquad \Theta_2 \Vdash Q}{\Theta_1 \uplus \Theta_2 \Vdash P \mid Q}\ \textsc{Par}$$

By case analysis on the congruence relation we have the following sub-cases:

(1) $P \mid Q \equiv Q \mid P$
(2) $P \mid (Q_1 \mid Q_2) \equiv (P \mid Q_1) \mid Q_2$
(3) $P \mid \langle \mathbf{return}\ () \rangle \equiv P$
(4) $vcd.P \mid Q \equiv vcd.(P \mid Q)$

In sub-case (1), by Par we have $\Theta_2 \uplus \Theta_1 \Vdash Q \mid P$. By the commutativity of $\uplus$, we have $\Theta_1 \uplus \Theta_2 \Vdash Q \mid P$ which concludes this sub-case.

In sub-case (2), we have $\Theta_2 \Vdash Q_1 \mid Q_2$. By inversion on its typing derivation, we have $\Theta_{21} \Vdash Q_1$ and $\Theta_{22} \Vdash Q_2$ such that $\Theta_2 = \Theta_{21} \uplus \Theta_{22}$. By Par we have $\Theta_1 \uplus \Theta_{21} \Vdash P \mid Q_1$. By Par again we have $(\Theta_1 \uplus \Theta_{21}) \uplus \Theta_{22} \Vdash (P \mid Q_1) \mid Q_2$. By the associativity of $\uplus$, we have $\Theta_1 \uplus (\Theta_{21} \uplus \Theta_{22}) \Vdash (P \mid Q_1) \mid Q_2$. By substituting Θ_2, we have $\Theta_1 \uplus \Theta_2 \Vdash (P \mid Q_1) \mid Q_2$ which concludes this sub-case.

In sub-case (3), we have $Q = \langle \mathbf{return}\ () \rangle$. By assumption we have $\Theta_2 \Vdash \langle \mathbf{return}\ () \rangle$. By inversion on its typing derivation, we have $\Theta_2; \epsilon; \epsilon \vdash \mathbf{return}\ () : C(\text{unit})$. By Lemma E.56 we have $\Theta_2; \epsilon; \epsilon \vdash () : \text{unit}$. By Lemma E.53 we have $\Theta_2 = \epsilon$ and $\epsilon \triangleright \mathsf{U}$. Thus we have $\Theta_1 \uplus \Theta_2 \Vdash P$.

In sub-case (4), we have $\Theta_1 \Vdash \nu cd.P$ and $\Theta_2 \Vdash Q$ by assumption. By inversion on the typing derivation of $\Theta_1 \Vdash \nu cd.P$, we have $\Theta_1, c :_{\mathsf{L}} \mathbf{ch}\langle A \rangle, d :_{\mathsf{L}} \mathbf{hc}\langle A \rangle \Vdash P$ for some protocol A. By Par we have $\Theta_1, c :_{\mathsf{L}} \mathbf{ch}\langle A \rangle, d :_{\mathsf{L}} \mathbf{hc}\langle A \rangle \uplus \Theta_2 \Vdash P \mid Q$. Since c and d are not in Θ_2, we have $(\Theta_1 \uplus \Theta_2), c :_{\mathsf{L}} \mathbf{ch}\langle A \rangle, d :_{\mathsf{L}} \mathbf{hc}\langle A \rangle \Vdash P \mid Q$. By Scope we have $\Theta_1 \uplus \Theta_2 \Vdash \nu cd.(P \mid Q)$ which concludes this sub-case.

Case (Scope):

$$\frac{\Theta, c :_{\mathsf{L}} \mathbf{ch}\langle A \rangle, d :_{\mathsf{L}} \mathbf{hc}\langle A \rangle \Vdash P}{\Theta \Vdash \nu cd.P}\ \textsc{Scope}$$

By case analysis on the congruence relation we have the following sub-cases:

(1) $\nu cd.(P \mid Q) \equiv \nu cd.P \mid Q$
(2) $\nu cd.P \equiv \nu dc.P$
(3) $\nu cd.\nu c'd'.P \equiv \nu c'd'.\nu cd.P$

In sub-case (1), we have $\Theta, c :_{\mathsf{L}} \mathbf{ch}\langle A \rangle, d :_{\mathsf{L}} \mathbf{hc}\langle A \rangle \Vdash P \mid Q$ by assumption. By inversion on its typing derivation, there exists Θ_1 and Θ_2 such that $\Theta = \Theta_1 \uplus \Theta_2$ and $\Theta_1, c :_{\mathsf{L}} \mathbf{ch}\langle A \rangle, d :_{\mathsf{L}} \mathbf{hc}\langle A \rangle \Vdash P$ and $\Theta_2 \Vdash Q$. Channels c and d must be distributed the typing judgment P as they are linear and do not appear in Q. Applying Scope to $\Theta_1, c :_{\mathsf{L}} \mathbf{ch}\langle A \rangle, d :_{\mathsf{L}} \mathbf{hc}\langle A \rangle \Vdash P$ we have $\Theta_1 \Vdash \nu cd.P$. By Par we have $\Theta_1 \uplus \Theta_2 \Vdash \nu cd.P \mid Q$ which concludes this sub-case.

In sub-case (2), we have $\Theta, c :_{\mathsf{L}} \mathbf{ch}\langle A \rangle, d :_{\mathsf{L}} \mathbf{hc}\langle A \rangle \Vdash P$ by assumption. By exchange, we have $\Theta, d :_{\mathsf{L}} \mathbf{hc}\langle A \rangle, c :_{\mathsf{L}} \mathbf{ch}\langle A \rangle \Vdash P$. By Scope we have $\Theta \Vdash \nu dc.P$ which concludes this sub-case.

In sub-case (3), we have $\Theta, c :_{\mathsf{L}} \mathbf{ch}\langle A \rangle, d :_{\mathsf{L}} \mathbf{hc}\langle A \rangle, c' :_{\mathsf{L}} \mathbf{ch}\langle A' \rangle, d' :_{\mathsf{L}} \mathbf{hc}\langle A' \rangle \Vdash P$ by assumption. By exchange, we have $\Theta, c' :_{\mathsf{L}} \mathbf{ch}\langle A' \rangle, d' :_{\mathsf{L}} \mathbf{hc}\langle A' \rangle, c :_{\mathsf{L}} \mathbf{ch}\langle A \rangle, d :_{\mathsf{L}} \mathbf{hc}\langle A \rangle \Vdash P$. Applying Scope twice we have $\Theta \Vdash \nu c'd'.\nu cd.P$ which concludes this sub-case. □

Theorem E.82 (Session Fidelity). *If $\Theta \Vdash P$ and $P \Rrightarrow Q$, then $\Theta \Vdash Q$.*

Proof. By induction on the derivation of $P \Rrightarrow Q$ and case analysis on the typing judgment.

Case (Proc-Fork):

$$\langle \mathbf{let}\ x \Leftarrow \mathbf{fork}\ (y : A)\ \mathbf{with}\ m\ \mathbf{in}\ n \rangle \Rrightarrow \nu cd.(\langle n[c/x] \rangle \mid \langle m[d/y] \rangle)$$

By inversion on $\Theta \Vdash \langle \mathbf{let}\ x \Leftarrow \mathbf{fork}\ (y : A)\ \mathbf{with}\ m\ \mathbf{in}\ n \rangle$ we have

$$\Theta; \epsilon; \epsilon \vdash \mathbf{let}\ x \Leftarrow \mathbf{fork}\ (y : A)\ \mathbf{with}\ m\ \mathbf{in}\ n : C(\text{unit})$$

By Lemma E.57 we have $\Theta_1; x : B; x :_r B \vdash n : C(\text{unit})$ and $\Theta_2; \epsilon; \epsilon \vdash \mathbf{fork}\ (y : A)\ \mathbf{with}\ m_0 : C(B)$ and $\Theta = \Theta_1 \uplus \Theta_2$. Applying Lemma E.58 to the later we have

$$\Theta_2; y : \mathbf{ch}\langle A' \rangle; y :_t \mathbf{ch}\langle A' \rangle \vdash m_0 : C(\text{unit})$$

and $A = \mathbf{ch}\langle A' \rangle$ and $C(B) \simeq C(\mathbf{hc}\langle A' \rangle)$. By the validity of context $\Theta_2; y : \mathbf{ch}\langle A' \rangle; y :_t \mathbf{ch}\langle A' \rangle \vdash$ we have $\epsilon \vdash \mathbf{ch}\langle A' \rangle : t$. By ChType and Theorem E.37 we have $t = \mathsf{L}$. Applying Corollary E.9 to

$C(B) \simeq C(\mathbf{hc}\langle A' \rangle)$ we have $B \simeq \mathbf{hc}\langle A' \rangle$. Applying Corollary E.24 to $\Theta_1; x : B; x :_r B \vdash n : C(\text{unit})$ using $B \simeq \mathbf{hc}\langle A' \rangle$ we have

$$\Theta_1; x : \mathbf{hc}\langle A' \rangle; x :_r \mathbf{hc}\langle A' \rangle \vdash n : C(\text{unit})$$

By the validity of context $\Theta_1; x : \mathbf{hc}\langle A' \rangle; x :_r \mathbf{hc}\langle A' \rangle \vdash$ we have $\epsilon \vdash \mathbf{hc}\langle A' \rangle : r$. By HcType and Theorem E.37 we have $r = \mathsf{L}$.
By Channel-HC, for some fresh channel c we have $c :_\mathsf{L} \mathbf{hc}\langle A' \rangle; \epsilon; \epsilon \vdash c : \mathbf{hc}\langle A' \rangle$.
By Channel-HC, for some fresh channel d we have $d :_\mathsf{L} \mathbf{ch}\langle A' \rangle; \epsilon; \epsilon \vdash d : \mathbf{ch}\langle A' \rangle$.
By Lemma E.23 we have $\Theta_1, c :_\mathsf{L} \mathbf{hc}\langle A' \rangle; \epsilon; \epsilon \vdash n[c/x] : C(\text{unit})$.
By Lemma E.23 we have $\Theta_2, d :_\mathsf{L} \mathbf{ch}\langle A' \rangle; \epsilon; \epsilon \vdash m_0[d/y] : C(\text{unit})$.
By Expr we have $\Theta_1, c :_\mathsf{L} \mathbf{hc}\langle A' \rangle \Vdash \langle n[c/x] \rangle$ and $\Theta_2, d :_\mathsf{L} \mathbf{ch}\langle A' \rangle \Vdash \langle m_0[d/y] \rangle$.
By Par we have $\Theta_1, c :_\mathsf{L} \mathbf{hc}\langle A' \rangle \uplus \Theta_2, d :_\mathsf{L} \mathbf{ch}\langle A' \rangle \Vdash \langle n[c/x] \rangle \mid \langle m_0[d/y] \rangle$.
Since c and d are fresh, we have $(\Theta_1 \uplus \Theta_2), c :_\mathsf{L} \mathbf{hc}\langle A' \rangle, d :_\mathsf{L} \mathbf{ch}\langle A' \rangle \Vdash \langle n[c/x] \rangle \mid \langle m_0[d/y] \rangle$.
By Scope we have $\Theta_1 \uplus \Theta_2 \Vdash \nu cd.(\langle n[c/x] \rangle \mid \langle m_0[d/y] \rangle)$ which concludes this case.
Case (Proc-End):

$$\begin{aligned} &\nu cd.(\langle \mathbf{let}\ x \Leftarrow \mathbf{close}\, c\ \mathbf{in}\ m \rangle \mid \langle \mathbf{let}\ y \Leftarrow \mathbf{wait}\, d\ \mathbf{in}\ n \rangle) \\ &\quad \Rrightarrow \langle \mathbf{let}\ x \Leftarrow \mathbf{return}\ ()\ \mathbf{in}\ m \rangle \mid \langle \mathbf{let}\ y \Leftarrow \mathbf{return}\ ()\ \mathbf{in}\ n \rangle \end{aligned}$$

By inversion on $\Theta \Vdash \nu cd.(\langle \mathbf{let}\ x \Leftarrow \mathbf{close}\, c\ \mathbf{in}\ m \rangle \mid \langle \mathbf{let}\ y \Leftarrow \mathbf{wait}\, d\ \mathbf{in}\ n \rangle)$ we have either

(1) $\Theta, c :_\mathsf{L} \mathbf{ch}\langle A \rangle, d :_\mathsf{L} \mathbf{hc}\langle A \rangle \Vdash \langle \mathbf{let}\ x \Leftarrow \mathbf{close}\, c\ \mathbf{in}\ m \rangle \mid \langle \mathbf{let}\ y \Leftarrow \mathbf{wait}\, d\ \mathbf{in}\ n \rangle$
(2) $\Theta, c :_\mathsf{L} \mathbf{hc}\langle A \rangle, d :_\mathsf{L} \mathbf{ch}\langle A \rangle \Vdash \langle \mathbf{let}\ x \Leftarrow \mathbf{close}\, c\ \mathbf{in}\ m \rangle \mid \langle \mathbf{let}\ y \Leftarrow \mathbf{wait}\, d\ \mathbf{in}\ n \rangle$

In sub-case (2), by inversion on its typing derivation we have $\Theta_1, c :_\mathsf{L} \mathbf{hc}\langle A \rangle \Vdash \langle \mathbf{let}\ x \Leftarrow \mathbf{close}\, c\ \mathbf{in}\ m \rangle$ and $\Theta_2, d :_\mathsf{L} \mathbf{ch}\langle A \rangle \Vdash \langle \mathbf{let}\ y \Leftarrow \mathbf{wait}\, d\ \mathbf{in}\ n \rangle$ such that $\Theta = \Theta_1 \uplus \Theta_2$.
By inversion on $\Theta_1, c :_\mathsf{L} \mathbf{hc}\langle A \rangle \Vdash \langle \mathbf{let}\ x \Leftarrow \mathbf{close}\, c\ \mathbf{in}\ m \rangle$ we have

$$\Theta_1, c :_\mathsf{L} \mathbf{hc}\langle A \rangle; \epsilon; \epsilon \vdash \mathbf{let}\ x \Leftarrow \mathbf{close}\, c\ \mathbf{in}\ m : C(\text{unit})$$

By Lemma E.57 we have $\Theta_{11}, c :_\mathsf{L} \mathbf{hc}\langle A \rangle; \epsilon; \epsilon \vdash \mathbf{close}\, c : C(B)$.
By Lemma E.63 we have $\Theta_{11}, c :_\mathsf{L} \mathbf{hc}\langle A \rangle; \epsilon; \epsilon \vdash c : \mathbf{ch}\langle \mathbf{1} \rangle$ which is a contradiction since c cannot be $\mathbf{ch}\langle \mathbf{1} \rangle$ in this context. Thus this sub-case is impossible.

In sub-case (1), by inversion on its typing derivation we have $\Theta_1, c :_\mathsf{L} \mathbf{ch}\langle A \rangle \Vdash \langle \mathbf{let}\ x \Leftarrow \mathbf{close}\, c\ \mathbf{in}\ m \rangle$ and $\Theta_2, d :_\mathsf{L} \mathbf{hc}\langle A \rangle \Vdash \langle \mathbf{let}\ y \Leftarrow \mathbf{wait}\, d\ \mathbf{in}\ n \rangle$ such that $\Theta = \Theta_1 \uplus \Theta_2$.
By inversion on $\Theta_1, c :_\mathsf{L} \mathbf{ch}\langle A \rangle \Vdash \langle \mathbf{let}\ x \Leftarrow \mathbf{close}\, c\ \mathbf{in}\ m \rangle$ we have

$$\Theta_1, c :_\mathsf{L} \mathbf{ch}\langle A \rangle; \epsilon; \epsilon \vdash \mathbf{let}\ x \Leftarrow \mathbf{close}\, c\ \mathbf{in}\ m : C(\text{unit})$$

By inversion on $\Theta_2, d :_\mathsf{L} \mathbf{hc}\langle A \rangle \Vdash \langle \mathbf{let}\ y \Leftarrow \mathbf{wait}\, d\ \mathbf{in}\ n \rangle$ we have

$$\Theta_2, d :_\mathsf{L} \mathbf{hc}\langle A \rangle; \epsilon; \epsilon \vdash \mathbf{let}\ y \Leftarrow \mathbf{wait}\, d\ \mathbf{in}\ n : C(\text{unit})$$

By Lemma E.57 we have

$$\Theta_{11}, c :_\mathsf{L} \mathbf{ch}\langle A \rangle; \epsilon; \epsilon \vdash \mathbf{close}\, c : C(B_1) \text{ and } \Theta_{12}; x : B_1; x :_s B_1 \vdash m : C(\text{unit}) \text{ and } \Theta_1 = \Theta_{11} \uplus \Theta_{12}$$

By Lemma E.57 we have

$$\Theta_{21}, d :_\mathsf{L} \mathbf{hc}\langle A \rangle; \epsilon; \epsilon \vdash \mathbf{wait}\, d : C(B_2) \text{ and } \Theta_{22}; y : B_2; y :_t B_2 \vdash n : C(\text{unit}) \text{ and } \Theta_2 = \Theta_{21} \uplus \Theta_{22}$$

Applying Lemma E.63 on $\Theta_{11}, c :_\mathsf{L} \mathbf{ch}\langle A \rangle; \epsilon; \epsilon \vdash \mathbf{close}\, c : C(B_1)$ gives us

$$\Theta_{11}, c :_\mathsf{L} \mathbf{ch}\langle A \rangle; \epsilon; \epsilon \vdash c : \mathbf{ch}\langle \mathbf{1} \rangle \quad \text{and} \quad C(B_1) \simeq C(\text{unit})$$

which implies $A \simeq \mathbf{1}$, $B_1 \simeq \text{unit}$, and $\Theta_{11} = \epsilon$.
Applying Lemma E.64 on $\Theta_{21}, d :_\mathsf{L} \mathbf{hc}\langle A \rangle; \epsilon; \epsilon \vdash \mathbf{wait}\, d : C(B_2)$ gives us

$$\Theta_{21}, d :_\mathsf{L} \mathbf{hc}\langle A \rangle; \epsilon; \epsilon \vdash d : \mathbf{hc}\langle \mathbf{1} \rangle \quad \text{and} \quad C(B_2) \simeq C(\text{unit})$$

which implies $A \simeq \mathbf{1}$, $B_2 \simeq \text{unit}$ and $\Theta_{21} = \epsilon$.
By Corollary E.24 we have

$$\Theta_{12}; x : \text{unit}; x :_{\mathsf{U}} \text{unit} \vdash m : C(\text{unit}) \quad \text{and} \quad \Theta_{22}; y : \text{unit}; y :_{\mathsf{U}} \text{unit} \vdash n : C(\text{unit})$$

Since $\Theta_{11} = \epsilon$ and $\Theta_{21} = \epsilon$, we have

$$\Theta_{11}; \epsilon; \epsilon \vdash \textbf{return}\ () : C(\text{unit}) \quad \text{and} \quad \Theta_{21}; \epsilon; \epsilon \vdash \textbf{return}\ () : C(\text{unit})$$

Applying Bind, we have

$$\Theta_{11} \uplus \Theta_{12}; \epsilon; \epsilon \vdash \textbf{let}\ x \Leftarrow \textbf{return}\ ()\ \textbf{in}\ m : C(\text{unit})$$

and

$$\Theta_{21} \uplus \Theta_{22}; \epsilon; \epsilon \vdash \textbf{let}\ y \Leftarrow \textbf{return}\ ()\ \textbf{in}\ n : C(\text{unit})$$

Applying Expr and Par, we have

$$(\Theta_{11} \uplus \Theta_{12}) \uplus (\Theta_{21} \uplus \Theta_{22}) \Vdash \langle \textbf{let}\ x \Leftarrow \textbf{return}\ ()\ \textbf{in}\ m\rangle \mid \langle \textbf{let}\ y \Leftarrow \textbf{return}\ ()\ \textbf{in}\ n\rangle$$

which concludes this case since $(\Theta_{11} \uplus \Theta_{12}) \uplus (\Theta_{21} \uplus \Theta_{22}) = \Theta$.
Case (Proc-Com):

$$\begin{aligned} &\nu cd.(\langle \textbf{let}\ x \Leftarrow \textbf{send}\ c\ v\ \textbf{in}\ m\rangle \mid \langle \textbf{let}\ y \Leftarrow \textbf{recv}\ d\ \textbf{in}\ n\rangle) \\ &\quad \Rrightarrow \nu cd.(\langle \textbf{let}\ x \Leftarrow \textbf{return}\ c\ \textbf{in}\ m\rangle \mid \langle \textbf{let}\ y \Leftarrow \textbf{return}\ \langle v, d\rangle_{\mathsf{L}}\ \textbf{in}\ n\rangle) \end{aligned}$$

By inversion on $\Theta \Vdash \nu cd.(\langle \textbf{let}\ x \Leftarrow \textbf{send}\ c\ v\ \textbf{in}\ m\rangle \mid \langle \textbf{let}\ y \Leftarrow \textbf{recv}\ d\ \textbf{in}\ n\rangle)$ we have either
(1) $\Theta, c :_{\mathsf{L}} \textbf{ch}\langle A\rangle, d :_{\mathsf{L}} \textbf{hc}\langle A\rangle \Vdash \langle \textbf{let}\ x \Leftarrow \textbf{send}\ c\ v\ \textbf{in}\ m\rangle \mid \langle \textbf{let}\ y \Leftarrow \textbf{recv}\ d\ \textbf{in}\ n\rangle$
(2) $\Theta, c :_{\mathsf{L}} \textbf{hc}\langle A\rangle, d :_{\mathsf{L}} \textbf{ch}\langle A\rangle \Vdash \langle \textbf{let}\ x \Leftarrow \textbf{send}\ c\ v\ \textbf{in}\ m\rangle \mid \langle \textbf{let}\ y \Leftarrow \textbf{recv}\ d\ \textbf{in}\ n\rangle$

In sub-case (1), by inversion on its typing derivation we have $\Theta_1, c :_{\mathsf{L}} \textbf{ch}\langle A\rangle \Vdash \langle \textbf{let}\ x \Leftarrow \textbf{send}\ c\ v\ \textbf{in}\ m\rangle$ and $\Theta_2, d :_{\mathsf{L}} \textbf{hc}\langle A\rangle \Vdash \langle \textbf{let}\ y \Leftarrow \textbf{recv}\ d\ \textbf{in}\ n\rangle$ such that $\Theta = \Theta_1 \uplus \Theta_2$.
By inversion on $\Theta_1, c :_{\mathsf{L}} \textbf{ch}\langle A\rangle \Vdash \langle \textbf{let}\ x \Leftarrow \textbf{send}\ c\ v\ \textbf{in}\ m\rangle$ we have

$$\Theta_1, c :_{\mathsf{L}} \textbf{ch}\langle A\rangle; \epsilon; \epsilon \vdash \textbf{let}\ x \Leftarrow \textbf{send}\ c\ v\ \textbf{in}\ m : C(\text{unit})$$

By inversion on $\Theta_2, d :_{\mathsf{L}} \textbf{hc}\langle A\rangle \Vdash \langle \textbf{let}\ y \Leftarrow \textbf{recv}\ d\ \textbf{in}\ n\rangle$ we have

$$\Theta_2, d :_{\mathsf{L}} \textbf{hc}\langle A\rangle; \epsilon; \epsilon \vdash \textbf{let}\ y \Leftarrow \textbf{recv}\ d\ \textbf{in}\ n : C(\text{unit})$$

Applying Lemma E.57 to $\Theta_1, c :_{\mathsf{L}} \textbf{ch}\langle A\rangle; \epsilon; \epsilon \vdash \textbf{let}\ x \Leftarrow \textbf{send}\ c\ v\ \textbf{in}\ m : C(\text{unit})$ we have

$$\Theta_{11}, c :_{\mathsf{L}} \textbf{ch}\langle A\rangle; \epsilon; \epsilon \vdash \textbf{send}\ c\ v : C(B_1) \text{ and } \Theta_{12}; x : B_1; x :_s B_1 \vdash m : C(\text{unit}) \text{ and } \Theta_1 = \Theta_{11} \uplus \Theta_{12}$$

Applying Lemma E.57 to $\Theta_2, d :_{\mathsf{L}} \textbf{hc}\langle A\rangle; \epsilon; \epsilon \vdash \textbf{let}\ y \Leftarrow \textbf{recv}\ d\ \textbf{in}\ n : C(\text{unit})$ we have

$$\Theta_{21}, d :_{\mathsf{L}} \textbf{hc}\langle A\rangle; \epsilon; \epsilon \vdash \textbf{recv}\ d : C(B_2) \text{ and } \Theta_{22}; y : B_2; y :_t B_2 \vdash n : C(\text{unit}) \text{ and } \Theta_2 = \Theta_{21} \uplus \Theta_{22}$$

Applying Lemma E.50 on $\Theta_{11}, c :_{\mathsf{L}} \textbf{ch}\langle A\rangle; \epsilon; \epsilon \vdash \textbf{send}\ c\ v : C(B_1)$ then there exists A', B_1' such that

$$\begin{aligned} &\Theta_{111}, c :_{\mathsf{L}} \textbf{ch}\langle A\rangle; \epsilon; \epsilon \vdash \textbf{send}\ c : \Pi_t(x : A').B_1' \\ &\Theta_{112}; \epsilon; \epsilon \vdash v : A' \\ &C(B_1) \simeq B_1'[v/x] \quad \text{and} \quad \Theta_{11} = \Theta_{111} \uplus \Theta_{112} \end{aligned}$$

Applying Lemma E.59 on $\Theta_{111}, c :_{\mathsf{L}} \textbf{ch}\langle A\rangle; \epsilon; \epsilon \vdash \textbf{send}\ c : \Pi_t(x : A').B_1'$ gives us either
(a) $\Pi_t(x : A').B_1' \simeq \Pi_{\mathsf{L}}(A'').C(\textbf{ch}\langle B_1''\rangle)$ and $\Theta_{111}, c :_{\mathsf{L}} \textbf{ch}\langle A\rangle; \epsilon; \epsilon \vdash c : \textbf{ch}\langle !(x : A'').\ B_1''\rangle$.
(b) $\Pi_t(x : A').B_1' \simeq \Pi_{\mathsf{L}}(A'').C(\textbf{hc}\langle B_1''\rangle)$ and $\Theta_{111}, c :_{\mathsf{L}} \textbf{ch}\langle A\rangle; \epsilon; \epsilon \vdash c : \textbf{hc}\langle ?(x : A'').\ B_1''\rangle$.

In (b), we have a contradiction since c cannot be $\textbf{hc}\langle ?(x : A'').\ B_1''\rangle$ in this context. Thus (b) is impossible. In (a), we have $A' \simeq A''$ and $B_1' \simeq C(\textbf{ch}\langle B_1''\rangle)$ by Corollary E.6. Additionally, we have $A \simeq !(x : A'').\ B_1''$ and $B_1 \simeq \textbf{ch}\langle B_1''[v/x]\rangle$ and $\Theta_{111} = \epsilon$.
Applying Lemma E.61 on $\Theta_{21}, d :_{\mathsf{L}} \textbf{hc}\langle A\rangle; \epsilon; \epsilon \vdash \textbf{recv}\ d : C(B_2)$ gives us either

(a) $C(B_2) \simeq C(\Sigma_{\mathsf{L}}(x : A''').\mathbf{ch}\langle B_2' \rangle)$ and $\Theta_{21}, d :_{\mathsf{L}} \mathbf{hc}\langle A \rangle; \epsilon; \epsilon \vdash d : \mathbf{ch}\langle ?(x : A'''). B_2' \rangle$.
(b) $C(B_2) \simeq C(\Sigma_{\mathsf{L}}(x : A''').\mathbf{hc}\langle B_2' \rangle)$ and $\Theta_{21}, d :_{\mathsf{L}} \mathbf{hc}\langle A \rangle; \epsilon; \epsilon \vdash d : \mathbf{hc}\langle !(x : A'''). B_2' \rangle$.

In (a), we have a contradiction since d cannot be $\mathbf{ch}\langle ?(x : A'''). \mathbf{ch}\langle B_2' \rangle \rangle$ in this context. Thus (a) is impossible. In (b), we have $B_2 \simeq \Sigma_{\mathsf{L}}(x : A''').\mathbf{hc}\langle B_2' \rangle$ by Corollary E.9. Additionally, we have $A \simeq !(x : A'''). B_2'$ and $\Theta_{21} = \epsilon$.
Since $A \simeq !(x : A''). B_1''$ and $A \simeq !(x : A'''). B_2'$, by transitivity of $\simeq$ we have

$$!(x : A''). B_1'' \simeq !(x : A'''). B_2'$$

By Corollary E.10 we have $A' \simeq A'' \simeq A'''$ and $B_1'' \simeq B_2'$.
By Channel-CH, we have $c :_{\mathsf{L}} \mathbf{ch}\langle B_1''[v/x] \rangle; \epsilon; \epsilon \vdash c : \mathbf{ch}\langle B_1''[v/x] \rangle$.
By Channel-HC, we have $d :_{\mathsf{L}} \mathbf{hc}\langle B_1''[v/x] \rangle; \epsilon; \epsilon \vdash d : \mathbf{hc}\langle B_1''[v/x] \rangle$.
Applying Return to $c :_{\mathsf{L}} \mathbf{ch}\langle B_1''[v/x] \rangle; \epsilon; \epsilon \vdash c : \mathbf{ch}\langle B_1''[v/x] \rangle$ we have

$$c :_{\mathsf{L}} \mathbf{ch}\langle B_1''[v/x] \rangle; \epsilon; \epsilon \vdash \mathbf{return}\, c : C(\mathbf{ch}\langle B_1''[v/x] \rangle)$$

Since there is $B_1 \simeq \mathbf{ch}\langle B_1''[v/x] \rangle$, we can apply Bind to create

$$\Theta_{12}, c :_{\mathsf{L}} \mathbf{ch}\langle B_1''[v/x] \rangle; \epsilon; \epsilon \vdash \mathbf{let}\ x \Leftarrow \mathbf{return}\, c\ \mathbf{in}\ m : C(\text{unit})$$

Pairing with v with d using Explicit-Pair we have

$$\Theta_{112}, d :_{\mathsf{L}} \mathbf{hc}\langle B_1''[v/x] \rangle; \epsilon; \epsilon \vdash \langle v, d \rangle_{\mathsf{L}} : \Sigma_{\mathsf{L}}(x : A').\mathbf{hc}\langle B_1'' \rangle$$

Apply Return to $\Theta_{112}, d :_{\mathsf{L}} \mathbf{hc}\langle B_1''[v/x] \rangle; \epsilon; \epsilon \vdash \langle v, d \rangle_{\mathsf{L}} : \Sigma_{\mathsf{L}}(x : A').\mathbf{hc}\langle B_1'' \rangle$ we have

$$\Theta_{112}, d :_{\mathsf{L}} \mathbf{hc}\langle B_1''[v/x] \rangle; \epsilon; \epsilon \vdash \mathbf{return}\ \langle v, d \rangle_{\mathsf{L}} : C(\Sigma_{\mathsf{L}}(x : A').\mathbf{hc}\langle B_1'' \rangle)$$

Since $B_2 \simeq \Sigma_{\mathsf{L}}(x : A''').\mathbf{hc}\langle B_2' \rangle \simeq \Sigma_{\mathsf{L}}(x : A').\mathbf{hc}\langle B_1'' \rangle$, we have

$$\Theta_{112} \uplus \Theta_{22}, d :_{\mathsf{L}} \mathbf{hc}\langle B_1''[v/x] \rangle; \epsilon; \epsilon \vdash \mathbf{let}\ y \Leftarrow \mathbf{return}\ \langle v, d \rangle_{\mathsf{L}}\ \mathbf{in}\ n : C(\text{unit})$$

By Expr we have

$$\Theta_{12}, c :_{\mathsf{L}} \mathbf{ch}\langle B_1''[v/x] \rangle \Vdash \langle \mathbf{let}\ x \Leftarrow \mathbf{return}\, c\ \mathbf{in}\ m \rangle$$

and

$$\Theta_{112} \uplus \Theta_{22}, d :_{\mathsf{L}} \mathbf{hc}\langle B_1''[v/x] \rangle \Vdash \langle \mathbf{let}\ y \Leftarrow \mathbf{return}\ \langle v, d \rangle_{\mathsf{L}}\ \mathbf{in}\ n \rangle$$

By Par we have

$$\begin{gathered}(\Theta_{112} \uplus \Theta_{12} \uplus \Theta_{22}), c :_{\mathsf{L}} \mathbf{ch}\langle B_1''[v/x] \rangle, d :_{\mathsf{L}} \mathbf{hc}\langle B_1''[v/x] \rangle \Vdash \\ \langle \mathbf{let}\ x \Leftarrow \mathbf{return}\, c\ \mathbf{in}\ m \rangle \mid \langle \mathbf{let}\ y \Leftarrow \mathbf{return}\ \langle v, d \rangle_{\mathsf{L}}\ \mathbf{in}\ n \rangle\end{gathered}$$

By Scope we have

$$\Theta_{112} \uplus \Theta_{12} \uplus \Theta_{22} \Vdash \nu cd.(\langle \mathbf{let}\ x \Leftarrow \mathbf{return}\, c\ \mathbf{in}\ m \rangle \mid \langle \mathbf{let}\ y \Leftarrow \mathbf{return}\ \langle v, d \rangle_{\mathsf{L}}\ \mathbf{in}\ n \rangle)$$

Since $\Theta_{111} = \epsilon$, $\Theta_{21} = \epsilon$, we have $\Theta = ((\Theta_{111} \uplus \Theta_{112}) \uplus \Theta_{12}) \uplus (\Theta_{21} \uplus \Theta_{22})$ which means

$$\Theta \Vdash \nu cd.(\langle \mathbf{let}\ x \Leftarrow \mathbf{return}\, c\ \mathbf{in}\ m \rangle \mid \langle \mathbf{let}\ y \Leftarrow \mathbf{return}\ \langle v, d \rangle_{\mathsf{L}}\ \mathbf{in}\ n \rangle)$$

thus concluding this sub-case.

For sub-case (2), the proof is similar to sub-case (1). The only difference is that the $\mathbf{ch}\langle \cdot \rangle$ and $\mathbf{hc}\langle \cdot \rangle$ types are swapped.

Case (Proc-<u>Com</u>):

$$\begin{aligned}&\nu cd.(\langle \mathbf{let}\ x \Leftarrow \underline{\mathbf{send}}\, c\ \{o\}\ \mathbf{in}\ m \rangle \mid \langle \mathbf{let}\ y \Leftarrow \underline{\mathbf{recv}}\, d\ \mathbf{in}\ n \rangle) \\ &\quad \Rrightarrow \nu cd.(\langle \mathbf{let}\ x \Leftarrow \mathbf{return}\, c\ \mathbf{in}\ m \rangle \mid \langle \mathbf{let}\ y \Leftarrow \mathbf{return}\ \langle \{o\}, d \rangle_{\mathsf{L}}\ \mathbf{in}\ n \rangle)\end{aligned}$$

The proof is similar to the previous case (Proc-Com) with the only difference being explicit applications are replaced with implicit applications.

Case (Proc-Expr):

$$\frac{m \rightsquigarrow\!\!\!\!\twoheadrightarrow m'}{\langle m \rangle \Rrightarrow \langle m' \rangle}$$

By inversion on $\Theta \Vdash \langle m \rangle$ we have $\Theta; \epsilon; \epsilon \vdash m : C(\mathsf{unit})$. By Theorem E.73 we have $\Theta; \epsilon; \epsilon \vdash m' : C(\mathsf{unit})$. By Expr we have $\Theta \Vdash \langle m' \rangle$ which concludes this case.

Case (Proc-Par):

$$\frac{P \Rrightarrow Q}{O \mid P \Rrightarrow O \mid Q}$$

By inversion on $\Theta \Vdash O \mid P$ we have $\Theta_1 \Vdash O$ and $\Theta_2 \Vdash P$ such that $\Theta = \Theta_1 \cup \Theta_2$. By the induction hypothesis we have $\Theta_2 \Vdash Q$. By Par we have $\Theta_1 \cup \Theta_2 \Vdash O \mid Q$ which concludes this case.

Case (Proc-Scope):

$$\frac{P \Rrightarrow Q}{\nu cd.P \Rrightarrow \nu cd.Q}$$

By inversion on $\Theta \Vdash \nu cd.P$ we have either

(1) $\Theta, c :_{\mathsf{L}} \mathbf{ch}\langle A \rangle, d :_{\mathsf{L}} \mathbf{hc}\langle A \rangle \Vdash P$
(2) $\Theta, c :_{\mathsf{L}} \mathbf{hc}\langle A \rangle, d :_{\mathsf{L}} \mathbf{ch}\langle A \rangle \Vdash P$

In case (1), by the induction hypothesis we have $\Theta, c :_{\mathsf{L}} \mathbf{ch}\langle A \rangle, d :_{\mathsf{L}} \mathbf{hc}\langle A \rangle \Vdash Q$. By Scope we have $\Theta \Vdash \nu cd.Q$ which concludes this sub-case.

In case (2), by the induction hypothesis we have $\Theta, c :_{\mathsf{L}} \mathbf{hc}\langle A \rangle, d :_{\mathsf{L}} \mathbf{ch}\langle A \rangle \Vdash Q$. By Scope we have $\Theta \Vdash \nu cd.Q$ which concludes this sub-case.

Case (Proc-Congr):

$$\frac{P \equiv P' \qquad P' \Rrightarrow Q' \qquad Q' \equiv Q}{P \Rrightarrow Q}$$

By Lemma E.81 we have $\Theta \Vdash P'$. By the induction hypothesis we have $\Theta \Vdash Q'$. By Lemma E.81 we have $\Theta \Vdash Q$ which concludes this case. □

E.3 Global Progress

The process level type system of TLL_C is insufficient to ensure that arbitrary process configurations enjoy global progress. This is because cyclic channel topologies are also considered to be well-typed. However, we can still prove a weaker form of progress for a class of configurations we call *reachable configurations*. Intuitively, a reachable configuration is one that can be reached from a well-typed singleton process through **fork**-operations.

Formally, we define the structure of *spawning trees* to capture the spawning relationships between parent-to-children processes. The syntax of spawning trees is given below.

$$\begin{array}{llll} \text{spawning tree} & \mathcal{P}, \mathcal{Q} & ::= & \mathsf{Root}(m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{\mathcal{Q}_j\}_{j \in \mathcal{J}}) \\ & & \mid & \mathsf{Node}(d, m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{\mathcal{Q}_j\}_{j \in \mathcal{J}}) \end{array}$$

Each tree is associated with a term m that performs computation and a set of children processes $\{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}$ where m communicates with each child process $\mathcal{P}_i$ through channel c_i. It also contains a set of subtrees $\{\mathcal{Q}_j\}_{j \in \mathcal{J}}$ that are no longer in communication with m (i.e. they have been detached through **close**/**wait**-operations). In the case of internal nodes, d is the channel which m uses to communicate with its parent process.

The crucial ideal behind the spawning tree structure is that we are going to define an alternative process semantics that operates on spawning trees. We will show that this alternative semantics can be simulated by the original process semantics. Moreover, we will show that the spawning tree semantics enjoys global progress. By defining reachable configurations as those that can be derived from well-typed spawning trees, we can then prove that reachable configurations enjoy global progress (induced by simulation).

To make the typing rules of spawning trees easier to define, we first introduce the following notations for channel types and $\kappa \in \{+, -\}$:

$$\mathsf{ch}^{+}\langle A\rangle = \mathbf{ch}\langle A\rangle \qquad\qquad \neg\mathsf{ch}^{+}\langle A\rangle = \mathsf{ch}^{-}\langle A\rangle$$

$$\mathsf{ch}^{-}\langle A\rangle = \mathbf{hc}\langle A\rangle \qquad\qquad \neg\mathsf{ch}^{-}\langle A\rangle = \mathsf{ch}^{+}\langle A\rangle$$

We define the typing rules for spawning trees as follows:

$$\frac{\overline{c_i :_{\mathsf{L}} \mathsf{ch}^{\kappa_i}\langle A_i\rangle}; \epsilon; \epsilon \vdash m : C(\mathsf{unit}) \qquad \forall i \in \mathcal{I},\ \neg\mathsf{ch}^{\kappa_i}\langle A_i\rangle \Vdash \mathcal{P}_i \qquad \forall j \in \mathcal{J},\ \Vdash \mathcal{Q}_j}{\Vdash \mathsf{Root}(m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}}, \{\mathcal{Q}_j\}_{j\in\mathcal{J}})}\ (\textsc{Valid-Root})$$

$$\frac{\overline{c_i :_{\mathsf{L}} \mathsf{ch}^{\kappa_i}\langle A_i\rangle}, d :_{\mathsf{L}} \mathsf{ch}^{\kappa}\langle A\rangle; \epsilon; \epsilon \vdash m : C(\mathsf{unit}) \qquad \forall i \in \mathcal{I},\ \neg\mathsf{ch}^{\kappa_i}\langle A_i\rangle \Vdash \mathcal{P}_i \qquad \forall j \in \mathcal{J},\ \Vdash \mathcal{Q}_j}{\mathsf{ch}^{\kappa}\langle A\rangle \Vdash \mathsf{Node}(d, m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}}, \{\mathcal{Q}_j\}_{j\in\mathcal{J}})}\ (\textsc{Valid-Node})$$

For the root node, we require that the term m is well-typed in channel context $\overline{c_i :_{\mathsf{L}} \mathsf{ch}^{\kappa_i}\langle A_i\rangle}$ comprised of $\mathsf{ch}^{\kappa_i}\langle A_i\rangle$ channels connecting to its children processes $\mathcal{P}_i$. The dual of each channel $\neg\mathsf{ch}^{\kappa_i}\langle A_i\rangle$ is propagated to typing the corresponding child process $\neg\mathsf{ch}^{\kappa_i}\langle A_i\rangle \Vdash \mathcal{P}_i$. When typing an internal node, we require that m is well-typed in a channel context that also includes $d :_{\mathsf{L}} \mathsf{ch}^{\kappa}\langle A\rangle$, i.e. the channel connecting to its parent process.

We define the *flattening* operation $|\mathcal{P}|$ that converts a spawning tree into a standard process configuration. The operation is defined as follows:

$$\textsc{Flatten-Root}\quad \frac{\forall i \in \mathcal{I},\ |\mathcal{P}_i| = (d_i, P_i) \qquad \forall j \in \mathcal{J},\ |\mathcal{Q}_j| = Q_j}{|\mathsf{Root}(m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}}, \{\mathcal{Q}_j\}_{j\in\mathcal{J}})| = \nu\overline{c_i d_i}.(\langle m\rangle \mid \overline{P_i}) \mid \overline{Q_j}}$$

$$\textsc{Flatten-Node}\quad \frac{\forall i \in \mathcal{I},\ |\mathcal{P}_i| = (d_i, P_i) \qquad \forall j \in \mathcal{J},\ |\mathcal{Q}_j| = Q_j}{|\mathsf{Node}(d, m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}}, \{\mathcal{Q}_j\}_{j\in\mathcal{J}})| = (d, \nu\overline{c_i d_i}.(\langle m\rangle \mid \overline{P_i}) \mid \overline{Q_j})}$$

The flattening operation recursively flattens each child process $\mathcal{P}_i$ into a channel-process pair (d_i, P_i) and each subtree $\mathcal{Q}_j$ into a process configuration Q_j. It then composes m with all the sub-processes in parallel. The channel pairs c_i, d_i are restricted to ensure proper channel scoping.

We now connect the validity of spawning trees to the well-typedness of flattened process configurations through Theorem E.83.

Theorem E.83 (Flatten Valid). *If $\Vdash \mathcal{P}$ and $|\mathcal{P}| = P$, then $\Vdash P$ and if $ch^{\kappa}\langle A\rangle \Vdash \mathcal{P}$ and $|\mathcal{P}| = (d, P)$, then $d :_L ch^{\kappa}\langle A\rangle \Vdash P$.*

Proof. By mutual induction on the derivation of $\Vdash \mathcal{P}$ and $A \Vdash \mathcal{P}$.

Case (Valid-Root): From Flatten-Root, we have $|\mathcal{P}_i| = (d_i, P_i)$ for each child process $\mathcal{P}_i$ and $|\mathcal{Q}_j| = Q_j$ for each subtree $\mathcal{Q}_j$. By the induction hypothesis, we have $d_i :_{\mathsf{L}} \neg\mathsf{ch}^{\kappa_i}\langle A_i\rangle \Vdash P_i$ for each $i \in \mathcal{I}$ and $\Vdash Q_j$ for each $j \in \mathcal{J}$.

From the premise of Valid-Root, we have $\overline{c_i :_{\mathsf{L}} \mathsf{ch}^{\kappa_i}\langle A_i\rangle}; \epsilon; \epsilon \vdash m : C(\mathsf{unit})$.

By Expr, we have $\overline{c_i :_{\mathsf{L}} \mathsf{ch}^{\kappa_i}\langle A_i\rangle} \Vdash \langle m\rangle$.

By applying Par repeated, we have $\overline{c_i :_{\mathsf{L}} \mathsf{ch}^{\kappa_i}\langle A_i\rangle}, \overline{d_i :_{\mathsf{L}} \neg\mathsf{ch}^{\kappa_i}\langle A_i\rangle} \Vdash (\langle m\rangle \mid \overline{P_i})$.

By applying Scope repeatedly, we have $\Vdash \overline{\nu c_i d_i}.(\langle m\rangle \mid \overline{P_i})$.
By applying Par repeatedly, we have $\Vdash \overline{\nu c_i d_i}.(\langle m\rangle \mid \overline{P_i}) \mid \overline{Q_j}$ which concludes this case.

Case (Valid-Node): From Flatten-Node, we have for each child process $\mathcal{P}_i$, $|\mathcal{P}_i| = (d_i, P_i)$ and for each subtree Q_j, $|Q_j| = Q_j$. By the induction hypothesis, we have $d_i :_{\mathsf{L}} \neg\mathsf{ch}^{\kappa_i}\langle A_i\rangle \Vdash P_i$ for each $i \in \mathcal{I}$ and $\Vdash Q_j$ for each $j \in \mathcal{J}$.
From the premise of Valid-Node, we have $\overline{c_i :_{\mathsf{L}} \mathsf{ch}^{\kappa_i}\langle A_i\rangle}, d :_{\mathsf{L}} \mathsf{ch}^{\kappa}\langle A\rangle; \epsilon; \epsilon \vdash m : C(\mathsf{unit})$.
By Expr, we have $\overline{c_i :_{\mathsf{L}} \mathsf{ch}^{\kappa_i}\langle A_i\rangle}, d :_{\mathsf{L}} \mathsf{ch}^{\kappa}\langle A\rangle \Vdash \langle m\rangle$.
By applying Par repeated, we have $\overline{c_i :_{\mathsf{L}} \mathsf{ch}^{\kappa_i}\langle A_i\rangle}, \overline{d_i :_{\mathsf{L}} \neg\mathsf{ch}^{\kappa_i}\langle A_i\rangle}, d :_{\mathsf{L}} \mathsf{ch}^{\kappa}\langle A\rangle \Vdash (\langle m\rangle \mid \overline{P_i})$.
By applying Scope repeatedly, we have $d :_{\mathsf{L}} \mathsf{ch}^{\kappa}\langle A\rangle \Vdash \overline{\nu c_i d_i}.(\langle m\rangle \mid \overline{P_i})$.
By applying Par repeatedly, we have $d :_{\mathsf{L}} \mathsf{ch}^{\kappa}\langle A\rangle \Vdash \overline{\nu c_i d_i}.(\langle m\rangle \mid \overline{P_i}) \mid \overline{Q_j}$ which concludes this case.
□

We now define the spawning tree semantics through the following reduction rules:

Root-Fork
$$\frac{\mathcal{I}' = \{i \in \mathcal{I} \mid c_i \in \mathrm{FC}(m)\}}{\begin{array}{c}\mathsf{Root}(\mathbf{let}\ x \Leftarrow \mathbf{fork}\ (y : A)\ \mathbf{with}\ m\ \mathbf{in}\ n, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}}, \{Q_j\}_{j\in\mathcal{J}}) \\ \Rrightarrow \mathsf{Root}(n[c/x], \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}\setminus\mathcal{I}'} \cup \{(c, \mathsf{Node}(d, m[d/y], \{(c_{i'}, \mathcal{P}_{i'})\}_{i'\in\mathcal{I}'}, \emptyset))\}, \{Q_j\}_{j\in\mathcal{J}})\end{array}}$$

Node-Fork
$$\frac{\mathcal{I}' = \{i \in \mathcal{I} \mid c_i \in \mathrm{FC}(m)\}}{\begin{array}{c}\mathsf{Node}(d, \mathbf{let}\ x \Leftarrow \mathbf{fork}\ (y : A)\ \mathbf{with}\ m\ \mathbf{in}\ n, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}}, \{Q_j\}_{j\in\mathcal{J}}) \\ \Rrightarrow \mathsf{Node}(d, n[c/x], \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}\setminus\mathcal{I}'} \cup \{(c, \mathsf{Node}(d, m[d/y], \{(c_{i'}, \mathcal{P}_{i'})\}_{i'\in\mathcal{I}'}, \emptyset))\}, \{Q_j\}_{j\in\mathcal{J}})\end{array}}$$

Root-Wait
$$\frac{\begin{array}{c}k \in \mathcal{I} \qquad \mathcal{P}_k = \mathsf{Node}(d_k, \mathbf{let}\ y \Leftarrow \mathbf{close}\ d_k\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k\in\mathcal{I}_k}, \{Q_{j_k}\}_{j_k\in\mathcal{J}_k}) \\ Q_k = \mathsf{Root}(\mathbf{let}\ y \Leftarrow \mathbf{return}\ ()\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k\in\mathcal{I}_k}, \{Q_{j_k}\}_{j_k\in\mathcal{J}_k})\end{array}}{\begin{array}{c}\mathsf{Root}(\mathbf{let}\ x \Leftarrow \mathbf{wait}\ c_k\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}}, \{Q_j\}_{j\in\mathcal{J}}) \\ \Rrightarrow \mathsf{Root}(\mathbf{let}\ x \Leftarrow \mathbf{return}\ ()\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}\setminus\{k\}}, \{Q_j\}_{j\in\mathcal{J}\cup\{k\}})\end{array}}$$

Node-Wait
$$\frac{\begin{array}{c}k \in \mathcal{I} \qquad \mathcal{P}_k = \mathsf{Node}(d_k, \mathbf{let}\ y \Leftarrow \mathbf{close}\ d_k\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k\in\mathcal{I}_k}, \{Q_{j_k}\}_{j_k\in\mathcal{J}_k}) \\ Q_k = \mathsf{Root}(\mathbf{let}\ y \Leftarrow \mathbf{return}\ ()\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k\in\mathcal{I}_k}, \{Q_{j_k}\}_{j_k\in\mathcal{J}_k})\end{array}}{\begin{array}{c}\mathsf{Node}(d, \mathbf{let}\ x \Leftarrow \mathbf{wait}\ c_k\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}}, \{Q_j\}_{j\in\mathcal{J}}) \\ \Rrightarrow \mathsf{Node}(d, \mathbf{let}\ x \Leftarrow \mathbf{return}\ ()\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}\setminus\{k\}}, \{Q_j\}_{j\in\mathcal{J}\cup\{k\}})\end{array}}$$

Root-Close
$$\frac{\begin{array}{c}k \in \mathcal{I} \qquad \mathcal{P}_k = \mathsf{Node}(d_k, \mathbf{let}\ y \Leftarrow \mathbf{wait}\ d_k\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k\in\mathcal{I}_k}, \{Q_{j_k}\}_{j_k\in\mathcal{J}_k}) \\ Q_k = \mathsf{Root}(\mathbf{let}\ y \Leftarrow \mathbf{return}\ ()\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k\in\mathcal{I}_k}, \{Q_{j_k}\}_{j_k\in\mathcal{J}_k})\end{array}}{\begin{array}{c}\mathsf{Root}(\mathbf{let}\ x \Leftarrow \mathbf{close}\ c_k\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}}, \{Q_j\}_{j\in\mathcal{J}}) \\ \Rrightarrow \mathsf{Root}(\mathbf{let}\ x \Leftarrow \mathbf{return}\ ()\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}\setminus\{k\}}, \{Q_j\}_{j\in\mathcal{J}\cup\{k\}})\end{array}}$$

Node-Close
$$\frac{\begin{array}{c}k \in \mathcal{I} \qquad \mathcal{P}_k = \mathsf{Node}(d_k, \mathbf{let}\ y \Leftarrow \mathbf{wait}\ d_k\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k\in\mathcal{I}_k}, \{Q_{j_k}\}_{j_k\in\mathcal{J}_k}) \\ Q_k = \mathsf{Root}(\mathbf{let}\ y \Leftarrow \mathbf{return}\ ()\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k\in\mathcal{I}_k}, \{Q_{j_k}\}_{j_k\in\mathcal{J}_k})\end{array}}{\begin{array}{c}\mathsf{Node}(d, \mathbf{let}\ x \Leftarrow \mathbf{close}\ c_k\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}}, \{Q_j\}_{j\in\mathcal{J}}) \\ \Rrightarrow \mathsf{Node}(d, \mathbf{let}\ x \Leftarrow \mathbf{return}\ ()\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}\setminus\{k\}}, \{Q_j\}_{j\in\mathcal{J}\cup\{k\}})\end{array}}$$

Root-Send

$$\frac{\begin{array}{c} k \in \mathcal{I} \qquad \mathcal{P}_k = \mathrm{Node}(d_k, \mathbf{let}\ y \Leftarrow \mathbf{recv}\, d_k\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k \in \mathcal{I}_k}, \{Q_{j_k}\}_{j_k \in \mathcal{J}_k}) \\ \mathcal{I}' = \{i \in \mathcal{I} | c_i \in \mathrm{FC}(v)\} \qquad \mathcal{P}'_k = \mathrm{Node}(d_k, \mathbf{let}\ y \Leftarrow \mathbf{return}\ \langle v, d_k\rangle_{\mathsf{L}}\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k \in \mathcal{I}_k \cup \mathcal{I}'}, \{Q_{j_k}\}_{j_k \in \mathcal{J}_k}) \end{array}}{\begin{array}{c} \mathrm{Root}(\mathbf{let}\ x \Leftarrow \mathbf{send}\, c_k\ v\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}}) \\ \Rrightarrow \mathrm{Root}(\mathbf{let}\ x \Leftarrow \mathbf{return}\, c_k\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I} \setminus (\{k\} \cup \mathcal{I}')} \cup \{(c_k, \mathcal{P}'_k)\}, \{Q_j\}_{j \in \mathcal{J}}) \end{array}}$$

Node-Send

$$\frac{\begin{array}{c} k \in \mathcal{I} \qquad \mathcal{P}_k = \mathrm{Node}(d_k, \mathbf{let}\ y \Leftarrow \mathbf{recv}\, d_k\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k \in \mathcal{I}_k}, \{Q_{j_k}\}_{j_k \in \mathcal{J}_k}) \qquad \mathcal{I}' = \{i \in \mathcal{I} | c_i \in \mathrm{FC}(v)\} \\ d \notin \mathrm{FC}(v) \qquad \mathcal{P}'_k = \mathrm{Node}(d_k, \mathbf{let}\ y \Leftarrow \mathbf{return}\ \langle v, d_k\rangle_{\mathsf{L}}\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k \in \mathcal{I}_k \cup \mathcal{I}'}, \{Q_{j_k}\}_{j_k \in \mathcal{J}_k}) \end{array}}{\begin{array}{c} \mathrm{Node}(d, \mathbf{let}\ x \Leftarrow \mathbf{send}\, c_k\ v\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}}) \\ \Rrightarrow \mathrm{Node}(d, \mathbf{let}\ x \Leftarrow \mathbf{return}\, c_k\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I} \setminus (\{k\} \cup \mathcal{I}')} \cup \{(c_k, \mathcal{P}'_k)\}, \{Q_j\}_{j \in \mathcal{J}}) \end{array}}$$

Root-Recv

$$\frac{\begin{array}{c} k \in \mathcal{I} \qquad \mathcal{P}_k = \mathrm{Node}(d_k, \mathbf{let}\ y \Leftarrow \mathbf{send}\, d_k\ v\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k \in \mathcal{I}_k}, \{Q_{j_k}\}_{j_k \in \mathcal{J}_k}) \\ \mathcal{I}' = \{i \in \mathcal{I}_k | c_i \in \mathrm{FC}(v)\} \qquad \mathcal{P}'_k = \mathrm{Node}(d_k, \mathbf{let}\ y \Leftarrow \mathbf{return}\, d_k\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k \in \mathcal{I}_k \setminus \mathcal{I}'}, \{Q_{j_k}\}_{j_k \in \mathcal{J}_k}) \end{array}}{\begin{array}{c} \mathrm{Root}(\mathbf{let}\ x \Leftarrow \mathbf{recv}\, c_k\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}}) \\ \Rrightarrow \mathrm{Root}(\mathbf{let}\ x \Leftarrow \mathbf{return}\ \langle v, c_k\rangle_{\mathsf{L}}\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i \in (\mathcal{I} \setminus \{k\}) \cup \mathcal{I}'} \cup \{(c_k, \mathcal{P}'_k)\}, \{Q_j\}_{j \in \mathcal{J}}) \end{array}}$$

Node-Recv

$$\frac{\begin{array}{c} k \in \mathcal{I} \qquad \mathcal{P}_k = \mathrm{Node}(d_k, \mathbf{let}\ y \Leftarrow \mathbf{send}\, d_k\ v\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k \in \mathcal{I}_k}, \{Q_{j_k}\}_{j_k \in \mathcal{J}_k}) \\ \mathcal{I}' = \{i \in \mathcal{I}_k | c_i \in \mathrm{FC}(v)\} \qquad \mathcal{P}'_k = \mathrm{Node}(d_k, \mathbf{let}\ y \Leftarrow \mathbf{return}\, d_k\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k \in \mathcal{I}_k \setminus \mathcal{I}'}, \{Q_{j_k}\}_{j_k \in \mathcal{J}_k}) \end{array}}{\begin{array}{c} \mathrm{Node}(d, \mathbf{let}\ x \Leftarrow \mathbf{recv}\, c_k\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}}) \\ \Rrightarrow \mathrm{Node}(d, \mathbf{let}\ x \Leftarrow \mathbf{return}\ \langle v, c_k\rangle_{\mathsf{L}}\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i \in (\mathcal{I} \setminus \{k\}) \cup \mathcal{I}'} \cup \{(c_k, \mathcal{P}'_k)\}, \{Q_j\}_{j \in \mathcal{J}}) \end{array}}$$

Root-<u>Send</u>

$$\frac{\begin{array}{c} k \in \mathcal{I} \qquad \mathcal{P}_k = \mathrm{Node}(d_k, \mathbf{let}\ y \Leftarrow \underline{\mathbf{recv}}\, d_k\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k \in \mathcal{I}_k}, \{Q_{j_k}\}_{j_k \in \mathcal{J}_k}) \\ \mathcal{P}'_k = \mathrm{Node}(d_k, \mathbf{let}\ y \Leftarrow \mathbf{return}\ \langle \{o\}, d_k\rangle_{\mathsf{L}}\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k \in \mathcal{I}_k}, \{Q_{j_k}\}_{j_k \in \mathcal{J}_k}) \end{array}}{\begin{array}{c} \mathrm{Root}(\mathbf{let}\ x \Leftarrow \underline{\mathbf{send}}\, c_k\ \{o\}\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}}) \\ \Rrightarrow \mathrm{Root}(\mathbf{let}\ x \Leftarrow \mathbf{return}\, c_k\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I} \setminus \{k\}} \cup \{(c_k, \mathcal{P}'_k)\}, \{Q_j\}_{j \in \mathcal{J}}) \end{array}}$$

Node-<u>Send</u>

$$\frac{\begin{array}{c} k \in \mathcal{I} \qquad \mathcal{P}_k = \mathrm{Node}(d_k, \mathbf{let}\ y \Leftarrow \underline{\mathbf{recv}}\, d_k\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k \in \mathcal{I}_k}, \{Q_{j_k}\}_{j_k \in \mathcal{J}_k}) \\ \mathcal{P}'_k = \mathrm{Node}(d_k, \mathbf{let}\ y \Leftarrow \mathbf{return}\ \langle \{o\}, d_k\rangle_{\mathsf{L}}\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k \in \mathcal{I}_k}, \{Q_{j_k}\}_{j_k \in \mathcal{J}_k}) \end{array}}{\begin{array}{c} \mathrm{Node}(d, \mathbf{let}\ x \Leftarrow \underline{\mathbf{send}}\, c_k\ \{o\}\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}}) \\ \Rrightarrow \mathrm{Node}(d, \mathbf{let}\ x \Leftarrow \mathbf{return}\, c_k\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I} \setminus \{k\}} \cup \{(c_k, \mathcal{P}'_k)\}, \{Q_j\}_{j \in \mathcal{J}}) \end{array}}$$

Root-<u>Recv</u>

$$\frac{\begin{array}{c} k \in \mathcal{I} \qquad \mathcal{P}_k = \mathrm{Node}(d_k, \mathbf{let}\ y \Leftarrow \underline{\mathbf{send}}\, d_k\ \{o\}\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k \in \mathcal{I}_k}, \{Q_{j_k}\}_{j_k \in \mathcal{J}_k}) \\ \mathcal{P}'_k = \mathrm{Node}(d_k, \mathbf{let}\ y \Leftarrow \mathbf{return}\, d_k\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k \in \mathcal{I}_k}, \{Q_{j_k}\}_{j_k \in \mathcal{J}_k}) \end{array}}{\begin{array}{c} \mathrm{Root}(\mathbf{let}\ x \Leftarrow \underline{\mathbf{recv}}\, c_k\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}}) \\ \Rrightarrow \mathrm{Root}(\mathbf{let}\ x \Leftarrow \mathbf{return}\ \langle \{o\}, c_k\rangle_{\mathsf{L}}\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I} \setminus \{k\}} \cup \{(c_k, \mathcal{P}'_k)\}, \{Q_j\}_{j \in \mathcal{J}}) \end{array}}$$

Node-<u>Recv</u>

$$\frac{\begin{array}{c} k \in \mathcal{I} \qquad \mathcal{P}_k = \mathrm{Node}(d_k, \mathbf{let}\ y \Leftarrow \underline{\mathbf{send}}\, d_k\ \{o\}\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k \in \mathcal{I}_k}, \{Q_{j_k}\}_{j_k \in \mathcal{J}_k}) \\ \mathcal{P}'_k = \mathrm{Node}(d_k, \mathbf{let}\ y \Leftarrow \mathbf{return}\, d_k\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k \in \mathcal{I}_k}, \{Q_{j_k}\}_{j_k \in \mathcal{J}_k}) \end{array}}{\begin{array}{c} \mathrm{Node}(d, \mathbf{let}\ x \Leftarrow \underline{\mathbf{recv}}\, c_k\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}}) \\ \Rrightarrow \mathrm{Node}(d, \mathbf{let}\ x \Leftarrow \mathbf{return}\ \langle \{o\}, c_k\rangle_{\mathsf{L}}\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I} \setminus \{k\}} \cup \{(c_k, \mathcal{P}'_k)\}, \{Q_j\}_{j \in \mathcal{J}}) \end{array}}$$

$$\textsc{Node-Forward}\quad \frac{\begin{array}{l} k \in \mathcal{I} \qquad \mathcal{P}_k = \mathsf{Node}(d_k, \mathbf{let}\ y \Leftarrow \mathbf{recv}\, d_k\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k \in \mathcal{I}_k}, \{Q_{j_k}\}_{j_k \in \mathcal{J}_k}) \qquad \mathcal{I}' = \{i \in \mathcal{I} \mid c_i \in \mathrm{FC}(v)\} \\ \qquad d \in \mathrm{FC}(v) \qquad \mathcal{P}'_k = \mathsf{Node}(c_k, \mathbf{let}\ x \Leftarrow \mathbf{return}\, c_k\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I} \setminus (\{k\} \cup \mathcal{I}')}, \{Q_j\}_{j \in \mathcal{J}}) \end{array}}{\begin{array}{l} \mathsf{Node}(d, \mathbf{let}\ x \Leftarrow \mathbf{send}\, c_k\ v\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}}) \\ \Rrightarrow \mathsf{Node}(d, \mathbf{let}\ y \Leftarrow \mathbf{return}\, \langle v, d_k \rangle_{\mathsf{L}}\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k \in \mathcal{I}_k \cup \mathcal{I}'} \cup \{(d_k, \mathcal{P}'_k)\}, \{Q_{j_k}\}_{j_k \in \mathcal{J}_k}) \end{array}}$$

$$\textsc{Root-Child}\quad \frac{k \in \mathcal{I} \qquad \mathcal{P}_k \Rrightarrow \mathcal{P}'_k}{\mathsf{Root}(m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}}) \Rrightarrow \mathsf{Root}(m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I} \setminus \{k\}} \cup \{(c_k, \mathcal{P}'_k)\}, \{Q_j\}_{j \in \mathcal{J}})}$$

$$\textsc{Node-Child}\quad \frac{k \in \mathcal{I} \qquad \mathcal{P}_k \Rrightarrow \mathcal{P}'_k}{\mathsf{Node}(d, m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}}) \Rrightarrow \mathsf{Node}(d, m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I} \setminus \{k\}} \cup \{(c_k, \mathcal{P}'_k)\}, \{Q_j\}_{j \in \mathcal{J}})}$$

$$\textsc{Root-SubTree}\quad \frac{k \in \mathcal{J} \qquad Q_k \Rrightarrow Q'_k}{\mathsf{Root}(m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}}) \Rrightarrow \mathsf{Root}(m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J} \setminus \{k\}} \cup \{Q'_k\})}$$

$$\textsc{Node-SubTree}\quad \frac{k \in \mathcal{J} \qquad Q_k \Rrightarrow Q'_k}{\mathsf{Node}(d, m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}}) \Rrightarrow \mathsf{Node}(d, m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J} \setminus \{k\}} \cup \{Q'_k\})}$$

$$\textsc{Root-Expr}\quad \frac{m \leadsto\!\!\!\!\!\rightarrow m'}{\mathsf{Root}(m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}}) \Rrightarrow \mathsf{Root}(m', \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}})}$$

$$\textsc{Node-Expr}\quad \frac{m \leadsto\!\!\!\!\!\rightarrow m'}{\mathsf{Node}(d, m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}}) \Rrightarrow \mathsf{Node}(d, m', \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}})}$$

In the rules above, the Root-Fork and Node-Fork rules describe how a **fork**-operation spawns a new child process and adds it to the set of children processes. The new child process is represented as an internal node. The channel connecting the child to its parent is fresh. Child processes that are connected to channels in $\mathrm{FC}(m)$ (the free channels in the spawning expression) are moved to be children of the newly spawned process.

The Root-Wait, Node-Wait, Root-Close and Node-Close rules describe how **close**/**wait**-operations detach a child from its parent. The detached process is moved to the set of subtrees.

The Root-Send and Node-Send rules describe how a **send**-operation sends a value to a child process. The child process must be waiting to receive a value through a **recv**-operation. The sent value may contain channels that are connected to other child processes. It is important to note that Node-Send only applies when the sent value *does not* contain the channel d connecting to the parent, i.e. the side condition $d \notin \mathrm{FC}(v)$. When a value is sent, any child processes connected to channels in $\mathrm{FC}(v)$ are moved to be children of the receiving process.

The Root-Recv and Node-Recv rules describe how a **recv**-operation receives a value from a child process. The child process must be waiting to send a value through a **send**-operation. The received value may contain channels that are connected to other child processes. When a value is received, any child processes connected to channels in $\mathrm{FC}(v)$ are moved to be children of the receiving process. Note that, due to linearity, the channel d_k connecting the sending child to its parent cannot be in $\mathrm{FC}(v)$. Thus, there is no side condition in Node-Recv. Moreover, this means that cyclic channel dependencies cannot arise here.

The Root-Send, Node-Send, Root-Recv, and Node-Recv rules describe the sending and receiving of ghost messages through **send** and **recv** operations. Due to the fact that ghost messages do not contain channels, there are no side conditions or changes to the spawning tree structure.

The Node-Forward rule describes how a **send**-operation can forward a parent channel d to a child process. The child process must be waiting to receive a value. The sent value must contain the parent channel d, i.e. the side condition $d \in \mathrm{FC}(v)$. When this happens, the child process takes over the parent channel d and tree is restructured so that the child process becomes the new parent and the sending process (the original parent) becomes one of its child processes. Other child processes that are connected to channels in $\mathrm{FC}(v)$ are also moved to be children of the receiving process.

The Root-Child, Node-Child, Root-SubTree, and Node-SubTree rules describe how a child process or subtree can take a reduction step.

The Root-Expr and Node-Expr rules describe how the expression in a node can reduce.

The Root-Unit and Node-Unit rules describe how a subtree that has finished (i.e. its expression is **return** () and it has no children or subtrees) can be removed from the spawning tree.

We now state the simulation theorem between the spawning tree semantics and the standard semantics (Section D.3). With slight abuse of notation, we write $|\mathcal{P}|$ to denote just the process P obtained by flattening the spawning tree $\mathcal{P}$.

Theorem E.84 (Spawning Tree Simulation). *If $\Vdash \mathcal{P}$ or $A \Vdash \mathcal{P}$, then given $\mathcal{P} \Rrightarrow \mathcal{P}'$ there is $|\mathcal{P}| \Rrightarrow |\mathcal{P}'|$.*

Proof. By induction on the derivation of $\mathcal{P} \Rrightarrow \mathcal{P}'$.

Case (Root-Fork):

$$\textsc{Root-Fork}\quad \frac{\mathcal{I}' = \{i \in \mathcal{I} | c_i \in \mathrm{FC}(m)\}}{\begin{array}{c}\mathsf{Root}(\mathbf{let}\ x \Leftarrow \mathbf{fork}\ (y : A)\ \mathbf{with}\ m\ \mathbf{in}\ n, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}}) \\ \Rrightarrow \mathsf{Root}(n[c/x], \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I} \setminus \mathcal{I}'} \cup \{(c, \mathsf{Node}(d, m[d/y], \{(c_{i'}, \mathcal{P}_{i'})\}_{i' \in \mathcal{I}'}, \emptyset))\}, \{Q_j\}_{j \in \mathcal{J}})\end{array}}$$

Flattening the LHS, we have:

$$\nu \overline{c_i d_i}.(\langle \mathbf{let}\ x \Leftarrow \mathbf{fork}\ (y : A)\ \mathbf{with}\ m\ \mathbf{in}\ n \rangle \mid \overline{P_i}) \mid \overline{Q_j}$$

Repeated application of Proc-Par and Proc-Scope and then Proc-Fork on the LHS gives us the reduced configuration

$$\nu \overline{c_i d_i}_{(i \in \mathcal{I})}.(\nu cd.(\langle n[c/x] \rangle \mid \langle m[d/y] \rangle) \mid \overline{P_i}) \mid \overline{Q_j}$$

By linearity of channels, we know that $\{c_{i'}, d_{i'}\}_{(i' \in \mathcal{I}')}$ do not appear in $n[c/x]$. Thus, we can apply structural congruence to rearrange the scoping to obtain

$$\nu \overline{c_i d_i}_{(i \in \mathcal{I} \setminus \mathcal{I}')}.(\nu cd.(\langle n[c/x] \rangle \mid \nu \overline{c_{i'} d_{i'}}_{(i' \in \mathcal{I}')}.(\langle m[d/y] \rangle \mid \overline{P_{i'}}_{(i' \in \mathcal{I}')})) \mid \overline{P_i}) \mid \overline{Q_j}$$

which is the flattened RHS.

Case (Node-Fork): Similar to the Root-Fork case.

Case (Root-Wait):

$$\textsc{Root-Wait}\quad \frac{\begin{array}{c} k \in \mathcal{I} \qquad \mathcal{P}_k = \mathsf{Node}(d_k, \mathbf{let}\ y \Leftarrow \mathbf{close}\ d_k\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k \in \mathcal{I}_k}, \{Q_{j_k}\}_{j_k \in \mathcal{J}_k}) \\ Q_{j_k} = \mathsf{Root}(\mathbf{let}\ y \Leftarrow \mathbf{return}\ ()\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k \in \mathcal{I}_k}, \{Q_{j_k}\}_{j_k \in \mathcal{J}_k}) \end{array}}{\begin{array}{c}\mathsf{Root}(\mathbf{let}\ x \Leftarrow \mathbf{wait}\ c_k\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}}) \\ \Rrightarrow \mathsf{Root}(\mathbf{let}\ x \Leftarrow \mathbf{return}\ ()\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I} \setminus \{k\}}, \{Q_j\}_{j \in \mathcal{J} \cup \{k\}})\end{array}}$$

Flattening both sides, we have:

$\nu\overline{c_i d_i}.(\langle \textbf{let } x \Leftarrow \textbf{wait } c_k \textbf{ in } m\rangle \mid \overline{P_{i\,(i\in\mathcal{I}\setminus\{k\})}} \mid (\nu\overline{c_{i_k} d_{i_k}}.(\langle \textbf{let } y \Leftarrow \textbf{close } d_k \textbf{ in } n\rangle \mid \overline{P_{i_k}}) \mid \overline{Q_{j_k}})) \mid \overline{Q_j}$

$\nu\overline{c_i d_{i\,(i\in\mathcal{I}\setminus\{k\})}}.(\langle \textbf{let } x \Leftarrow \textbf{return } () \textbf{ in } m\rangle \mid \overline{P_{i\,(i\in\mathcal{I}\setminus\{k\})}}) \mid \overline{Q_j} \mid (\nu\overline{c_{i_k} d_{i_k}}.(\langle \textbf{let } y \Leftarrow \textbf{return } () \textbf{ in } n\rangle \mid \overline{P_{i_k}}) \mid \overline{Q_{j_k}})$

Apply Proc-Congr to the LHS to rearrange the processes, then apply Proc-Scope and Proc-Par repeatedly to isolate the sub-configuration $\nu c_k d_k.(\langle \textbf{let } x \Leftarrow \textbf{wait } c_k \textbf{ in } m\rangle \mid \langle \textbf{let } y \Leftarrow \textbf{close } d_k \textbf{ in } n\rangle)$. Finally, apply Proc-Wait to this sub-configuration to obtain the reduced configuration, which is structurally congruent to the RHS.

Case (Node-Wait): Similar to the Root-Wait case.

Case (Root-Close): Similar to the Root-Wait case.

Case (Node-Close): Similar to the Root-Wait case.

Case (Root-Send):

$$\textsc{Root-Send}\quad \frac{\begin{array}{cc} k\in\mathcal{I} & \mathcal{P}_k = \mathsf{Node}(d_k, \textbf{let } y \Leftarrow \textbf{recv } d_k \textbf{ in } n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k\in\mathcal{I}_k}, \{Q_{j_k}\}_{j_k\in\mathcal{J}_k}) \\ \mathcal{I}' = \{i\in\mathcal{I} \mid c_i \in \mathrm{FC}(v)\} & \mathcal{P}'_k = \mathsf{Node}(d_k, \textbf{let } y \Leftarrow \textbf{return } \langle v, d_k\rangle_{\mathsf{L}} \textbf{ in } n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k\in\mathcal{I}_k\cup\mathcal{I}'}, \{Q_{j_k}\}_{j_k\in\mathcal{J}_k})\end{array}}{\begin{array}{c}\mathsf{Root}(\textbf{let } x \Leftarrow \textbf{send } c_k\ v \textbf{ in } m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}}, \{Q_j\}_{j\in\mathcal{J}}) \\ \Rrightarrow \mathsf{Root}(\textbf{let } x \Leftarrow \textbf{return } c_k \textbf{ in } m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}\setminus(\{k\}\cup\mathcal{I}')} \cup \{(c_k, \mathcal{P}'_k)\}, \{Q_j\}_{j\in\mathcal{J}})\end{array}}$$

Flattening both sides, we have:

$$\nu\overline{c_i d_i}.(\langle \textbf{let } x \Leftarrow \textbf{send } c_k\ v \textbf{ in } m\rangle \mid \overline{P_{i\,(i\in\mathcal{I}\setminus\{k\})}} \mid (\nu\overline{c_{i_k} d_{i_k}}.(\langle \textbf{let } y \Leftarrow \textbf{recv } d_k \textbf{ in } n\rangle \mid \overline{P_{i_k}}) \mid \overline{Q_{j_k}})) \mid \overline{Q_j}$$

$$\begin{aligned}&\nu\overline{c_i d_{i\,(i\in(\mathcal{I}\setminus(\{k\}\cup\mathcal{I}'))\cup\{k\})}}.(\langle \textbf{let } x \Leftarrow \textbf{return } c_k \textbf{ in } m\rangle \mid \overline{P_{i\,(i\in\mathcal{I}\setminus(\{k\}\cup\mathcal{I}'))}} \\ &\quad \mid \nu\overline{c_{i_k} d_{i_k}}.\nu\overline{c_i d_{i\,(i\in\mathcal{I}')}}.(\langle \textbf{let } y \Leftarrow \textbf{return } \langle v, d_k\rangle_{\mathsf{L}} \textbf{ in } n\rangle \mid \overline{P_{i_k}} \mid \overline{P_{i\,(i\in\mathcal{I}')}}) \mid \overline{Q_{j_k}}) \mid \overline{Q_j}\end{aligned}$$

Apply Proc-Congr to the LHS to rearrange the processes, then apply Proc-Scope and Proc-Par repeatedly to isolate the sub-configuration $\nu c_k d_k.(\langle \textbf{let } x \Leftarrow \textbf{send } c_k\ v \textbf{ in } m\rangle \mid \langle \textbf{let } y \Leftarrow \textbf{recv } d_k \textbf{ in } n\rangle)$. Now, apply Proc-Send to this sub-configuration to obtain the reduced configuration

$$\nu c_k d_k.(\langle \textbf{let } x \Leftarrow \textbf{return } c_k \textbf{ in } m\rangle \mid \langle \textbf{let } y \Leftarrow \textbf{return } \langle v, d_k\rangle_{\mathsf{L}} \textbf{ in } n\rangle)$$

Note that, by linearity, the channels $\{c_i, d_i\}_{(i\in\mathcal{I}')}$ do not occur in $\textbf{let } x \Leftarrow \textbf{return } c_k \textbf{ in } m$. Thus, structural congruence can be applied to move the scope of these channels to

$$\nu\overline{c_i d_{i\,(i\in\mathcal{I}')}}.(\langle \textbf{let } y \Leftarrow \textbf{return } \langle v, d_k\rangle_{\mathsf{L}} \textbf{ in } n\rangle \mid \overline{P_{i\,(i\in\mathcal{I}')}})$$

which gives us the desired result.

Case (Node-Send, Root-Recv Node-Recv): Similar to the Root-Send case.

Case (Root-<u>Send</u> Node-<u>Send</u> Root-<u>Recv</u> Node-<u>Recv</u>): Similar to the Root-Send case. The only difference is that scope restriction does not need to be applied to move any channels since the ghost message o does not contain channels.

Case (Node-Forward):

$$\textsc{Node-Forward}\quad \frac{\begin{array}{ccc} k\in\mathcal{I} & \mathcal{P}_k = \mathsf{Node}(d_k, \textbf{let } y \Leftarrow \textbf{recv } d_k \textbf{ in } n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k\in\mathcal{I}_k}, \{Q_{j_k}\}_{j_k\in\mathcal{J}_k}) & \mathcal{I}' = \{i\in\mathcal{I} \mid c_i\in\mathrm{FC}(v)\} \\ d\in\mathrm{FC}(v) & \mathcal{P}'_k = \mathsf{Node}(c_k, \textbf{let } x \Leftarrow \textbf{return } c_k \textbf{ in } m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}\setminus(\{k\}\cup\mathcal{I}')}, \{Q_j\}_{j\in\mathcal{J}}) & \end{array}}{\begin{array}{c}\mathsf{Node}(d, \textbf{let } x \Leftarrow \textbf{send } c_k\ v \textbf{ in } m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}}, \{Q_j\}_{j\in\mathcal{J}}) \\ \Rrightarrow \mathsf{Node}(d, \textbf{let } y \Leftarrow \textbf{return } \langle v, d_k\rangle_{\mathsf{L}} \textbf{ in } n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k\in\mathcal{I}_k\cup\mathcal{I}'} \cup \{(d_k, \mathcal{P}'_k)\}, \{Q_{j_k}\}_{j_k\in\mathcal{J}_k})\end{array}}$$

Flattening both sides, we have:

$$\nu\overline{c_i d_i}.(\langle \textbf{let } x \Leftarrow \textbf{send } c_k\ v \textbf{ in } m\rangle \mid \overline{P_{i\,(i\in\mathcal{I}\setminus\{k\})}} \mid (\nu\overline{c_{i_k} d_{i_k}}.(\langle \textbf{let } y \Leftarrow \textbf{recv } d_k \textbf{ in } n\rangle \mid \overline{P_{i_k}}) \mid \overline{Q_{j_k}})) \mid \overline{Q_j}$$

$$\begin{aligned}&\nu\overline{c_{i_k} d_{i_k\,(i_k\in\mathcal{I}_k\cup\mathcal{I}'\cup\{k\})}}.(\langle \textbf{let } y \Leftarrow \textbf{return } \langle v, d_k\rangle_{\mathsf{L}} \textbf{ in } n\rangle \mid \overline{P_{i_k\,(i_k\in\mathcal{I}_k\cup\mathcal{I}')}} \\ &\quad \mid \nu\overline{c_i d_{i\,(i\in\mathcal{I}\setminus(\{k\}\cup\mathcal{I}'))}}.(\langle \textbf{let } x \Leftarrow \textbf{return } c_k \textbf{ in } m\rangle \mid \overline{P_{i\,(i\in\mathcal{I}\setminus(\{k\}\cup\mathcal{I}'))}}) \mid \overline{Q_j}) \mid \overline{Q_{j_k}}\end{aligned}$$

Apply Proc-Congr to the LHS to rearrange the processes, then apply Proc-Scope and Proc-Par repeatedly to isolate the sub-configuration $\nu c_k d_k.(\langle \textbf{let } x \Leftarrow \textbf{send } c_k\ v \textbf{ in } m\rangle \mid \langle \textbf{let } y \Leftarrow \textbf{recv } d_k \textbf{ in } n\rangle)$. Now, apply Proc-Send to this sub-configuration to obtain the reduced configuration

$$\nu c_k d_k.(\langle \textbf{let } x \Leftarrow \textbf{return } c_k \textbf{ in } m\rangle \mid \langle \textbf{let } y \Leftarrow \textbf{return } \langle v, d_k\rangle_{\mathsf{L}} \textbf{ in } n\rangle)$$

By symmetry of structural congruence, we have

$$\nu c_k d_k.(\langle \textbf{let } y \Leftarrow \textbf{return } \langle v, d_k\rangle_{\mathsf{L}} \textbf{ in } n\rangle \mid \langle \textbf{let } x \Leftarrow \textbf{return } c_k \textbf{ in } m\rangle)$$

Since $\{c_i, d_i\}_{(i\in\mathcal{I}\setminus(\{k\}\cup\mathcal{I}'))}$ do not occur in $\textbf{let } y \Leftarrow \textbf{return } \langle v, d_k\rangle_{\mathsf{L}} \textbf{ in } n$, we can apply structural congruence to move the scope of these channels to

$$\overline{\nu c_i d_i}_{(i\in\mathcal{I}\setminus(\{k\}\cup\mathcal{I}'))}.(\langle \textbf{let } x \Leftarrow \textbf{return } c_k \textbf{ in } m\rangle \mid \overline{P_i}_{(i\in\mathcal{I}\setminus(\{k\}\cup\mathcal{I}'))})$$

which gives us the desired result.
Case (Root-Child, Node-Child, Root-SubTree, Node-SubTree): By the induction hypothesis, we have $|\mathcal{P}_k| \Rrightarrow |\mathcal{P}'_k|$ or $|\mathcal{Q}_k| \Rrightarrow |\mathcal{Q}'_k|$. Repeated application of Proc-Par and Proc-Scope gives us the desired result.
Case (Root-Expr, Node-Expr): By the assumption, we have $m \rightsquigarrow\!\!\!\!\!\rightarrow m'$. Repeated application of Proc-Par, Proc-Scope and then Proc-Expr gives us the desired result. ◻

In order to show that spawning trees are an adequate characterization of reachability, we prove the following fidelity theorem. This theorem states that if a spawning tree is well-typed and it takes a reduction step, then the resulting spawning tree is also well-typed. Thus, starting from a well-typed singleton $\mathsf{Root}(m, \emptyset, \emptyset)$, the spawning trees reachable from it are all well-typed.

Theorem E.85 (Spawning Tree Fidelity). *If $\Vdash \mathcal{P}$ or $ch^{\kappa}\langle A\rangle \Vdash \mathcal{P}$, then given $\mathcal{P} \Rrightarrow \mathcal{Q}$ there is $\Vdash \mathcal{Q}$ or $ch^{\kappa}\langle A\rangle \Vdash \mathcal{Q}$ respectively.*

Proof. By induction on the derivation of $\mathcal{P} \Rrightarrow \mathcal{Q}$ and by case analysis on the derivation of the typing judgment $\Vdash \mathcal{P}$ or $\mathsf{ch}^{\kappa}\langle A\rangle \Vdash \mathcal{P}$.
Case (Root-Fork):

$$\textsc{Root-Fork}\quad \frac{\mathcal{I}' = \{i \in \mathcal{I} \mid c_i \in \mathsf{FC}(m)\}}{\begin{array}{c}\mathsf{Root}(\textbf{let } x \Leftarrow \textbf{fork } (y : A) \textbf{ with } m \textbf{ in } n, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}}, \{\mathcal{Q}_j\}_{j\in\mathcal{J}}) \\ \Rrightarrow \mathsf{Root}(n[c/x], \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}\setminus\mathcal{I}'} \cup \{(c, \mathsf{Node}(d, m[d/x], \{(c_{i'}, \mathcal{P}_{i'})\}_{i'\in\mathcal{I}'}, \emptyset))\}, \{\mathcal{Q}_j\}_{j\in\mathcal{J}})\end{array}}$$

By inversion on the typing judgment

$$\Vdash \mathsf{Root}(\textbf{let } x \Leftarrow \textbf{fork } (y : A) \textbf{ with } m \textbf{ in } n, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}}, \{\mathcal{Q}_j\}_{j\in\mathcal{J}})$$

we have

$$\begin{array}{l}\overline{c_i :_{\mathsf{L}} \mathsf{ch}^{\kappa_i}\langle A_i\rangle}; \epsilon; \epsilon \vdash \textbf{let } x \Leftarrow \textbf{fork } (y : A) \textbf{ with } m \textbf{ in } n : C(\mathsf{unit}) \\ \forall i \in \mathcal{I}, \neg\mathsf{ch}^{\kappa_i}\langle A_i\rangle \Vdash \mathcal{P}_i \\ \forall j \in \mathcal{J}, \Vdash \mathcal{Q}_j\end{array}$$

Similarly to the reasoning in Theorem E.82 for the Proc-Fork case, by inversion on the term typing judgment we know that $A = \mathbf{ch}\langle A'\rangle$ and the following hold:

$$\begin{array}{l}\overline{c_i :_{\mathsf{L}} \mathsf{ch}^{\kappa_i}\langle A_i\rangle}_{(i\in\mathcal{I}')}, d :_{\mathsf{L}} \mathbf{hc}\langle A'\rangle; \epsilon; \epsilon \vdash m[d/y] : C(\mathsf{unit}) \\ \overline{c_i :_{\mathsf{L}} \mathsf{ch}^{\kappa_i}\langle A_i\rangle}_{(i\in\mathcal{I}\setminus\mathcal{I}')}; c :_{\mathsf{L}} \mathbf{ch}\langle A'\rangle; \epsilon; \epsilon \vdash n[c/x] : C(\mathsf{unit})\end{array}$$

We can partition $\{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}}$ into $\{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}\setminus\mathcal{I}'}$ and $\{(c_{i'}, \mathcal{P}_{i'})\}_{i'\in\mathcal{I}'}$.

We now have well-typed node

$$\mathsf{ch}^-\langle A'\rangle \Vdash \mathsf{Node}(d, m[d/y], \{(c_{i'}, \mathcal{P}_{i'})\}_{i'\in\mathcal{I}'}, \emptyset)$$

which in turn gives us the well-typed root

$$\Vdash \mathsf{Root}(n[c/x], \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}\setminus\mathcal{I}'} \cup \{(c, \mathsf{Node}(d, m[d/x], \{(c_{i'}, \mathcal{P}_{i'})\}_{i'\in\mathcal{I}'}, \emptyset))\}, \{Q_j\}_{j\in\mathcal{J}})$$

and concludes this case.
Case (Node-Fork): Similar to the Root-Fork case.
Case (Root-Wait):

$$\frac{\begin{array}{c}\text{Root-Wait}\\ k\in\mathcal{I} \qquad \mathcal{P}_k = \mathsf{Node}(d_k, \textbf{let } y \Leftarrow \textbf{close}\, d_k \textbf{ in } n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k\in\mathcal{I}_k}, \{Q_{j_k}\}_{j_k\in\mathcal{J}_k})\\ Q_k = \mathsf{Root}(\textbf{let } y \Leftarrow \textbf{return}\, () \textbf{ in } n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k\in\mathcal{I}_k}, \{Q_{j_k}\}_{j_k\in\mathcal{J}_k})\end{array}}{\begin{array}{c}\mathsf{Root}(\textbf{let } x \Leftarrow \textbf{wait}\, c_k \textbf{ in } m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}}, \{Q_j\}_{j\in\mathcal{J}})\\ \Rrightarrow \mathsf{Root}(\textbf{let } x \Leftarrow \textbf{return}\, () \textbf{ in } m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}\setminus\{k\}}, \{Q_j\}_{j\in\mathcal{J}\cup\{k\}})\end{array}}$$

By inversion on the typing judgment

$$\Vdash \mathsf{Root}(\textbf{let } x \Leftarrow \textbf{wait}\, c_k \textbf{ in } m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}}, \{Q_j\}_{j\in\mathcal{J}})$$

we have

$$\begin{array}{l}\overline{c_i :_{\mathsf{L}} \mathsf{ch}^{\kappa_i}\langle A_i\rangle}; \epsilon; \epsilon \vdash \textbf{let } x \Leftarrow \textbf{wait}\, c_k \textbf{ in } m : C(\mathsf{unit})\\ \forall i\in\mathcal{I}, \neg\mathsf{ch}^{\kappa_i}\langle A_i\rangle \Vdash \mathcal{P}_i\\ \forall j\in\mathcal{J}, \Vdash Q_j\end{array}$$

Since $k\in\mathcal{I}$, we know

$$\neg\mathsf{ch}^{\kappa_k}\langle A_k\rangle \Vdash \mathsf{Node}(d_k, \textbf{let } y \Leftarrow \textbf{close}\, d_k \textbf{ in } n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k\in\mathcal{I}_k}, \{Q_{j_k}\}_{j_k\in\mathcal{J}_k})$$

and by inversion on this typing judgment, we have

$$\begin{array}{l}\overline{c_{i_k} :_{\mathsf{L}} \mathsf{ch}^{\kappa_{i_k}}\langle A_{i_k}\rangle}, d_k :_{\mathsf{L}} \neg\mathsf{ch}^{\kappa_k}\langle A_k\rangle; \epsilon; \epsilon \vdash \textbf{let } y \Leftarrow \textbf{close}\, d_k \textbf{ in } n : C(\mathsf{unit})\\ \forall i_k\in\mathcal{I}_k, \neg\mathsf{ch}^{\kappa_{i_k}}\langle A_{i_k}\rangle \Vdash \mathcal{P}_{i_k}\\ \forall j_k\in\mathcal{J}_k, \Vdash Q_{j_k}\end{array}$$

Similarly to the reasoning in Theorem E.82 for the Proc-End case, we know $A_k \simeq \mathbf{1}$ and

$$\begin{array}{l}\overline{c_i :_{\mathsf{L}} \mathsf{ch}^{\kappa_i}\langle A_i\rangle}_{(i\in\mathcal{I}\setminus\{k\})}; \epsilon; \epsilon \vdash \textbf{let } x \Leftarrow \textbf{return}\, () \textbf{ in } m : C(\mathsf{unit})\\ \overline{c_{i_k} :_{\mathsf{L}} \mathsf{ch}^{\kappa_{i_k}}\langle A_{i_k}\rangle}; \epsilon; \epsilon \vdash \textbf{let } y \Leftarrow \textbf{return}\, () \textbf{ in } n : C(\mathsf{unit})\end{array}$$

This gives us the well-typed root

$$\Vdash \mathsf{Root}(\textbf{let } y \Leftarrow \textbf{return}\, () \textbf{ in } n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k\in\mathcal{I}_k}, \{Q_{j_k}\}_{j_k\in\mathcal{J}_k})$$

which in turn gives us the well-typed root

$$\Vdash \mathsf{Root}(\textbf{let } x \Leftarrow \textbf{return}\, () \textbf{ in } m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}\setminus\{k\}}, \{Q_j\}_{j\in\mathcal{J}\cup\{k\}})$$

and concludes this case.
Case (Node-Wait, Root-Close, Node-Close): Similar to the Root-Wait case.
Case (Root-Send):

$$\frac{\begin{array}{c}\text{Root-Send}\\ k\in\mathcal{I} \qquad \mathcal{P}_k = \mathsf{Node}(d_k, \textbf{let } y \Leftarrow \textbf{recv}\, d_k \textbf{ in } n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k\in\mathcal{I}_k}, \{Q_{j_k}\}_{j_k\in\mathcal{J}_k})\\ \mathcal{I}' = \{i\in\mathcal{I} \mid c_i \in \mathrm{FC}(v)\} \qquad \mathcal{P}'_k = \mathsf{Node}(d_k, \textbf{let } y \Leftarrow \textbf{return}\, \langle v, d_k\rangle_{\mathsf{L}} \textbf{ in } n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k\in\mathcal{I}_k\cup\mathcal{I}'}, \{Q_{j_k}\}_{j_k\in\mathcal{J}_k})\end{array}}{\begin{array}{c}\mathsf{Root}(\textbf{let } x \Leftarrow \textbf{send}\, c_k\ v \textbf{ in } m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}}, \{Q_j\}_{j\in\mathcal{J}})\\ \Rrightarrow \mathsf{Root}(\textbf{let } x \Leftarrow \textbf{return}\, c_k \textbf{ in } m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}\setminus(\{k\}\cup\mathcal{I}')} \cup \{(c_k, \mathcal{P}'_k)\}, \{Q_j\}_{j\in\mathcal{J}})\end{array}}$$

By inversion on the typing judgment

$$\Vdash \mathsf{Root}(\mathbf{let}\ x \Leftarrow \mathbf{send}\, c_k\ v\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}})$$

we have

$$\begin{array}{l} \overline{c_i :_{\mathsf{L}} \mathsf{ch}^{\kappa_i}\langle A_i \rangle}; \epsilon; \epsilon \vdash \mathbf{let}\ x \Leftarrow \mathbf{send}\, c_k\ v\ \mathbf{in}\ m : C(\mathsf{unit}) \\ \forall i \in \mathcal{I}, \neg\mathsf{ch}^{\kappa_i}\langle A_i \rangle \Vdash \mathcal{P}_i \\ \forall j \in \mathcal{J}, \Vdash Q_j \end{array}$$

From $k \in \mathcal{I}$, we know

$$\neg\mathsf{ch}^{\kappa_k}\langle A_k \rangle \Vdash \mathsf{Node}(d_k, \mathbf{let}\ y \Leftarrow \mathbf{recv}\, d_k\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k \in \mathcal{I}_k}, \{Q_{j_k}\}_{j_k \in \mathcal{J}_k})$$

and by inversion on this typing judgment, we have

$$\begin{array}{l} \overline{c_{i_k} :_{\mathsf{L}} \mathsf{ch}^{\kappa_{i_k}}\langle A_{i_k} \rangle}, d_k :_{\mathsf{L}} \neg\mathsf{ch}^{\kappa_k}\langle A_k \rangle; \epsilon; \epsilon \vdash \mathbf{let}\ y \Leftarrow \mathbf{recv}\, d_k\ \mathbf{in}\ n : C(\mathsf{unit}) \\ \forall i_k \in \mathcal{I}_k, \neg\mathsf{ch}^{\kappa_{i_k}}\langle A_{i_k} \rangle \Vdash \mathcal{P}_{i_k} \\ \forall j_k \in \mathcal{J}_k, \Vdash Q_{j_k} \end{array}$$

Similarly to the reasoning in Theorem E.82 for the Proc-Comm case, there exists B'' such that

$$\overline{c_i :_{\mathsf{L}} \mathsf{ch}^{\kappa_i}\langle A_i \rangle}_{(i \in \mathcal{I} \setminus (\{k\} \cup \mathcal{I}'))}, c_k :_{\mathsf{L}} \mathsf{ch}^{\kappa_k}\langle B''[v/x] \rangle; \epsilon; \epsilon \vdash \mathbf{let}\ x \Leftarrow \mathbf{return}\, c_k\ \mathbf{in}\ m : C(\mathsf{unit})$$

$$\overline{c_{i_k} :_{\mathsf{L}} \mathsf{ch}^{\kappa_{i_k}}\langle A_{i_k} \rangle}, \overline{c_i :_{\mathsf{L}} \mathsf{ch}^{\kappa_i}\langle A_i \rangle}_{(i \in \mathcal{I}')}, d_k :_{\mathsf{L}} \neg\mathsf{ch}^{\kappa_k}\langle B''[v/x] \rangle; \epsilon; \epsilon \vdash \mathbf{let}\ y \Leftarrow \mathbf{return}\, \langle v, d_k \rangle_{\mathsf{L}}\ \mathbf{in}\ n : C(\mathsf{unit})$$

We can partition $\{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I} \setminus \{k\}}$ into $\{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I} \setminus (\{k\} \cup \mathcal{I}')}$ and $\{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}'}$.
This gives us the well-typed node

$$\neg\mathsf{ch}^{\kappa_k}\langle B''[v/x] \rangle \Vdash \mathsf{Node}(d_k, \mathbf{let}\ y \Leftarrow \mathbf{return}\, \langle v, d_k \rangle_{\mathsf{L}}\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k \in \mathcal{I}_k \cup \mathcal{I}'}, \{Q_{j_k}\}_{j_k \in \mathcal{J}_k})$$

which in turn gives us the well-typed root

$$\Vdash \mathsf{Root}(\mathbf{let}\ x \Leftarrow \mathbf{return}\, c_k\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I} \setminus (\{k\} \cup \mathcal{I}')} \cup \{(c_k, \mathcal{P}'_k)\}, \{Q_j\}_{j \in \mathcal{J}})$$

and concludes this case.
Case (Node-Send, Root-Recv, Node-Recv): Similar to the Root-Send case.
Case (Root-<u>Send</u>):

$$\textsc{Root-}\underline{\textsc{Send}}\ \dfrac{\begin{array}{l} k \in \mathcal{I} \qquad \mathcal{P}_k = \mathsf{Node}(d_k, \mathbf{let}\ y \Leftarrow \underline{\mathbf{recv}}\, d_k\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k \in \mathcal{I}_k}, \{Q_{j_k}\}_{j_k \in \mathcal{J}_k}) \\ \mathcal{P}'_k = \mathsf{Node}(d_k, \mathbf{let}\ y \Leftarrow \mathbf{return}\, \langle \{o\}, d_k \rangle_{\mathsf{L}}\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k \in \mathcal{I}_k}, \{Q_{j_k}\}_{j_k \in \mathcal{J}_k}) \end{array}}{\begin{array}{c} \mathsf{Root}(\mathbf{let}\ x \Leftarrow \underline{\mathbf{send}}\, c_k\ \{o\}\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}}) \\ \Rrightarrow \mathsf{Root}(\mathbf{let}\ x \Leftarrow \mathbf{return}\, c_k\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I} \setminus \{k\}} \cup \{(c_k, \mathcal{P}'_k)\}, \{Q_j\}_{j \in \mathcal{J}}) \end{array}}$$

By inversion on the typing judgment

$$\Vdash \mathsf{Root}(\mathbf{let}\ x \Leftarrow \underline{\mathbf{send}}\, c_k\ \{o\}\ \mathbf{in}\ m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}})$$

we have

$$\begin{array}{l} \overline{c_i :_{\mathsf{L}} \mathsf{ch}^{\kappa_i}\langle A_i \rangle}; \epsilon; \epsilon \vdash \mathbf{let}\ x \Leftarrow \underline{\mathbf{send}}\, c_k\ \{o\}\ \mathbf{in}\ m : C(\mathsf{unit}) \\ \forall i \in \mathcal{I}, \neg\mathsf{ch}^{\kappa_i}\langle A_i \rangle \Vdash \mathcal{P}_i \\ \forall j \in \mathcal{J}, \Vdash Q_j \end{array}$$

From $k \in \mathcal{I}$, we know

$$\neg\mathsf{ch}^{\kappa_k}\langle A_k \rangle \Vdash \mathsf{Node}(d_k, \mathbf{let}\ y \Leftarrow \underline{\mathbf{recv}}\, d_k\ \mathbf{in}\ n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k \in \mathcal{I}_k}, \{Q_{j_k}\}_{j_k \in \mathcal{J}_k})$$

and by inversion on this typing judgment, we have

$$\overline{c_{i_k} :_{\mathsf{L}} \mathsf{ch}^{\kappa_{i_k}}\langle A_{i_k}\rangle}, d_k :_{\mathsf{L}} \neg\mathsf{ch}^{\kappa_k}\langle A_k\rangle; \epsilon; \epsilon \vdash \textbf{let } y \Leftarrow \underline{\textbf{recv}}\, d_k \textbf{ in } n : C(\mathsf{unit})$$
$$\forall i_k \in \mathcal{I}_k, \neg\mathsf{ch}^{\kappa_{i_k}}\langle A_{i_k}\rangle \Vdash \mathcal{P}_{i_k}$$
$$\forall j_k \in \mathcal{J}_k, \Vdash \mathcal{Q}_{j_k}$$

Similarly to the reasoning in Theorem E.82 for the Proc-Comm case, there exists B'' such that

$$\overline{c_i :_{\mathsf{L}} \mathsf{ch}^{\kappa_i}\langle A_i\rangle}_{(i\in\mathcal{I}\setminus\{k\})}, c_k :_{\mathsf{L}} \mathsf{ch}^{\kappa_k}\langle B''[o/x]\rangle; \epsilon; \epsilon \vdash \textbf{let } x \Leftarrow \textbf{return}\, c_k \textbf{ in } m : C(\mathsf{unit})$$
$$\overline{c_{i_k} :_{\mathsf{L}} \mathsf{ch}^{\kappa_{i_k}}\langle A_{i_k}\rangle}, d_k :_{\mathsf{L}} \neg\mathsf{ch}^{\kappa_k}\langle B''[o/x]\rangle; \epsilon; \epsilon \vdash \textbf{let } y \Leftarrow \textbf{return}\, \langle\{o\}, d_k\rangle_{\mathsf{L}} \textbf{ in } n : C(\mathsf{unit})$$

This gives us the well-typed node

$$\neg\mathsf{ch}^{\kappa_k}\langle B''[o/x]\rangle \Vdash \mathsf{Node}(d_k, \textbf{let } y \Leftarrow \textbf{return}\, \langle\{o\}, d_k\rangle_{\mathsf{L}} \textbf{ in } n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k\in\mathcal{I}_k}, \{\mathcal{Q}_{j_k}\}_{j_k\in\mathcal{J}_k})$$

which in turn gives us the well-typed root

$$\Vdash \mathsf{Root}(\textbf{let } x \Leftarrow \textbf{return}\, c_k \textbf{ in } m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}\setminus\{k\}} \cup \{(c_k, \mathcal{P}'_k)\}, \{\mathcal{Q}_j\}_{j\in\mathcal{J}})$$

and concludes this case.
Case (Node-<u>Send</u>, Root-<u>Recv</u>, Node-<u>Recv</u>): Similar to the Root-<u>Send</u> case.
Case (Node-Forward):

$$\textsc{Node-Forward}\quad \frac{\begin{array}{c} k \in \mathcal{I} \qquad \mathcal{P}_k = \mathsf{Node}(d_k, \textbf{let } y \Leftarrow \textbf{recv}\, d_k \textbf{ in } n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k\in\mathcal{I}_k}, \{\mathcal{Q}_{j_k}\}_{j_k\in\mathcal{J}_k}) \qquad \mathcal{I}' = \{i \in \mathcal{I} | c_i \in \mathrm{FC}(v)\} \\ d \in \mathrm{FC}(v) \qquad \mathcal{P}'_k = \mathsf{Node}(c_k, \textbf{let } x \Leftarrow \textbf{return}\, c_k \textbf{ in } m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}\setminus(\{k\}\cup\mathcal{I}')}, \{\mathcal{Q}_j\}_{j\in\mathcal{J}}) \end{array}}{\begin{array}{c} \mathsf{Node}(d, \textbf{let } x \Leftarrow \textbf{send}\, c_k\ v \textbf{ in } m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}}, \{\mathcal{Q}_j\}_{j\in\mathcal{J}}) \\ \Rrightarrow \mathsf{Node}(d, \textbf{let } y \Leftarrow \textbf{return}\, \langle v, d_k\rangle_{\mathsf{L}} \textbf{ in } n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k\in\mathcal{I}_k\cup\mathcal{I}'} \cup \{(d_k, \mathcal{P}'_k)\}, \{\mathcal{Q}_{j_k}\}_{j_k\in\mathcal{J}_k}) \end{array}}$$

By inversion on the typing judgment

$$\mathsf{ch}^{\kappa}\langle A\rangle \Vdash \mathsf{Node}(d, \textbf{let } x \Leftarrow \textbf{send}\, c_k\ v \textbf{ in } m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}}, \{\mathcal{Q}_j\}_{j\in\mathcal{J}})$$

we have

$$\overline{c_i :_{\mathsf{L}} \mathsf{ch}^{\kappa_i}\langle A_i\rangle}, d :_{\mathsf{L}} \mathsf{ch}^{\kappa}\langle A\rangle; \epsilon; \epsilon \vdash \textbf{let } x \Leftarrow \textbf{send}\, c_k\ v \textbf{ in } m : C(\mathsf{unit})$$
$$\forall i \in \mathcal{I}, \neg\mathsf{ch}^{\kappa_i}\langle A_i\rangle \Vdash \mathcal{P}_i$$
$$\forall j \in \mathcal{J}, \Vdash \mathcal{Q}_j$$

From $k \in \mathcal{I}$, we know

$$\neg\mathsf{ch}^{\kappa_k}\langle A_k\rangle \Vdash \mathsf{Node}(d_k, \textbf{let } y \Leftarrow \textbf{recv}\, d_k \textbf{ in } n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k\in\mathcal{I}_k}, \{\mathcal{Q}_{j_k}\}_{j_k\in\mathcal{J}_k})$$

and by inversion on this typing judgment, we have

$$\overline{c_{i_k} :_{\mathsf{L}} \mathsf{ch}^{\kappa_{i_k}}\langle A_{i_k}\rangle}, d_k :_{\mathsf{L}} \neg\mathsf{ch}^{\kappa_k}\langle A_k\rangle; \epsilon; \epsilon \vdash \textbf{let } y \Leftarrow \textbf{recv}\, d_k \textbf{ in } n : C(\mathsf{unit})$$
$$\forall i_k \in \mathcal{I}_k, \neg\mathsf{ch}^{\kappa_{i_k}}\langle A_{i_k}\rangle \Vdash \mathcal{P}_{i_k}$$
$$\forall j_k \in \mathcal{J}_k, \Vdash \mathcal{Q}_{j_k}$$

Similarly to the reasoning in Theorem E.82 for the Proc-Comm case, there exists B'' such that

$$\overline{c_i :_{\mathsf{L}} \mathsf{ch}^{\kappa_i}\langle A_i\rangle}_{(i\in\mathcal{I}\setminus(\{k\}\cup\mathcal{I}'))}, c_k :_{\mathsf{L}} \mathsf{ch}^{\kappa_k}\langle B''[v/x]\rangle; \epsilon; \epsilon \vdash \textbf{let } x \Leftarrow \textbf{return}\, c_k \textbf{ in } m : C(\mathsf{unit})$$
$$\overline{c_{i_k} :_{\mathsf{L}} \mathsf{ch}^{\kappa_{i_k}}\langle A_{i_k}\rangle}, \overline{c_i :_{\mathsf{L}} \mathsf{ch}^{\kappa_i}\langle A_i\rangle}_{(i\in\mathcal{I}')}, d_k :_{\mathsf{L}} \neg\mathsf{ch}^{\kappa_k}\langle B''[v/x]\rangle, d :_{\mathsf{L}} \mathsf{ch}^{\kappa}\langle A\rangle; \epsilon; \epsilon \vdash \textbf{let } y \Leftarrow \textbf{return}\, \langle v, d_k\rangle_{\mathsf{L}} \textbf{ in } n : C(\mathsf{unit})$$

We can partition $\{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}\setminus\{k\}}$ into $\{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}\setminus(\{k\}\cup\mathcal{I}')}$ and $\{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}'}$.
This gives us the well-typed node

$$\mathsf{ch}^{\kappa_k}\langle B''[v/x]\rangle \Vdash \mathsf{Node}(c_k, \textbf{let } x \Leftarrow \textbf{return}\, c_k \textbf{ in } m, \{(c_i, \mathcal{P}_i)\}_{i\in\mathcal{I}\setminus(\{k\}\cup\mathcal{I}')}, \{\mathcal{Q}_j\}_{j\in\mathcal{J}})$$

which in turn gives us the well-typed node

$$\mathsf{ch}^{\kappa}\langle A\rangle \Vdash \mathsf{Node}(d, \textbf{let } y \Leftarrow \textbf{return } \langle v, d_k\rangle_{\mathsf{L}} \textbf{ in } n, \{(c_{i_k}, \mathcal{P}_{i_k})\}_{i_k \in \mathcal{I}_k \cup \mathcal{I}'} \cup \{(d_k, \mathcal{P}'_k)\}, \{Q_{j_k}\}_{j_k \in \mathcal{J}_k})$$

and concludes this case.
Case (Root-Child):

$$\text{Root-Child}\quad \frac{k \in \mathcal{I} \qquad \mathcal{P}_k \Rrightarrow \mathcal{P}'_k}{\mathsf{Root}(m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}}) \Rrightarrow \mathsf{Root}(m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}\setminus\{k\}} \cup \{(c_k, \mathcal{P}'_k)\}, \{Q_j\}_{j \in \mathcal{J}})}$$

By inversion on the typing judgment

$$\Vdash \mathsf{Root}(m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}})$$

we have

$$\begin{aligned}
&\overline{c_i :_{\mathsf{L}} \mathsf{ch}^{\kappa_i}\langle A_i\rangle}; \epsilon; \epsilon \vdash m : C(\mathsf{unit})\\
&\forall i \in \mathcal{I}, \neg\mathsf{ch}^{\kappa_i}\langle A_i\rangle \Vdash \mathcal{P}_i\\
&\forall j \in \mathcal{J}, \Vdash Q_j
\end{aligned}$$

From $k \in \mathcal{I}$ and the typing judgment $\neg\mathsf{ch}^{\kappa_k}\langle A_k\rangle \Vdash \mathcal{P}_k$,
By the induction hypothesis, we have the well-typed process $\neg\mathsf{ch}^{\kappa_k}\langle A_k\rangle \Vdash \mathcal{P}'_k$.
This gives us the well-typed root

$$\Vdash \mathsf{Root}(m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}\setminus\{k\}} \cup \{(c_k, \mathcal{P}'_k)\}, \{Q_j\}_{j \in \mathcal{J}})$$

and concludes this case.
Case (Node-Child, Root-SubTree, Node-SubTree): Similar to the Root-Child case.
Case (Root-Expr):

$$\text{Root-Expr}\quad \frac{m \rightsquigarrow\!\!\!\!\twoheadrightarrow m'}{\mathsf{Root}(m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}}) \Rrightarrow \mathsf{Root}(m', \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}})}$$

By inversion on the typing judgment

$$\Vdash \mathsf{Root}(m, \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}})$$

we have

$$\begin{aligned}
&\overline{c_i :_{\mathsf{L}} \mathsf{ch}^{\kappa_i}\langle A_i\rangle}; \epsilon; \epsilon \vdash m : C(\mathsf{unit})\\
&\forall i \in \mathcal{I}, \neg\mathsf{ch}^{\kappa_i}\langle A_i\rangle \Vdash \mathcal{P}_i\\
&\forall j \in \mathcal{J}, \Vdash Q_j
\end{aligned}$$

By Theorem E.73 we have

$$\overline{c_i :_{\mathsf{L}} \mathsf{ch}^{\kappa_i}\langle A_i\rangle}; \epsilon; \epsilon \vdash m' : C(\mathsf{unit})$$

which gives us the well-typed root

$$\Vdash \mathsf{Root}(m', \{(c_i, \mathcal{P}_i)\}_{i \in \mathcal{I}}, \{Q_j\}_{j \in \mathcal{J}})$$

and concludes this case.
Case (Node-Expr): Similar to the Root-Expr case. □

Now we can define the reachability of a configuration through spawning trees.

Definition E.86 (Reachability). A configuration P is *reachable* if there exists a spawning tree $\mathcal{P}$ such that $\Vdash \mathcal{P}$ and $|\mathcal{P}| = P$.

Now to prove the main progress theorem, we first need to define a few auxiliary judgments that will help us characterize the state of a spawning tree.

$$\frac{\forall j \in \mathcal{J}, \quad Q_j \text{ Terminal}}{\text{Root}(\textbf{return } (), \emptyset, \{Q_j\}_{j \in \mathcal{J}}) \text{ Terminal}}$$

$$\frac{}{\text{Node}(d, \textbf{let } y \Leftarrow \textbf{send } d\ v \textbf{ in } ,)}$$

E.4 Erasure Safety