Dependent Session Types for Certified 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 the 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. All reported results are formalized in Coq. 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 [11] 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.

$$\boxed{\text{client}} \longleftrightarrow p_1 \longleftrightarrow p_2 \longleftrightarrow p_3 \longleftrightarrow \cdots \longleftrightarrow p_n$$

Each of the p_i nodes here represents a queue cell which holds a value and are linked together by bidirectional channels of type queue_A. As indicated by the type constructor &, the first queue node q_1 first receives either an ins or 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

2022. ACM 2475-1421/2022/1-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) [7] with session-typed concurrency. In TLL_C, one could define the queues through the following dependent session type:

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

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:

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

When a queue server receives an $\mathsf{ins}(v)$ value, the type of the server becomes $\mathsf{queue}(\mathsf{snoc}(xs, v))$ were 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 describe 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:

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

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 exist 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 [8, 25] have successfully combined *classical* session types with functional languages, its is well known that classical session types do not easily support recursive session types [9] (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 [2] eschew the dual operator and define dual *interpretations* of session types based 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, the protocol $!A.P$ would be interpreted dually as follows:

$$\begin{array}{ll} \mathbf{ch}\langle !A.P\rangle & \text{(send message of type } A\text{)} \\ \mathbf{hc}\langle !A.P\rangle & \text{(receive message of type } A\text{)} \end{array}$$

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.

All lemmas and theorems reported in the this paper are formalized in Coq [18]. All examples can be compiled into C programs using our prototype compiler where concurrent processes are implemented using POSIX threads. The compiler implements advanced language features such dependent pattern matching and functional in-place programming [12] for linear types. Proofs, source code, and examples 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.
- The entire calculus, with its meta-theorems, is formalized in Coq.
- We implement a prototype compiler which compiles TLL_C into safe and efficient C code.

[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 certified 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 $\mathbf{1}$ 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 send of msg to provide such a proof ensures that the protocol is followed correctly.

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 take inspiration from binary session types [8, 25] and label dependent session types [20], 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 other words, dual channel types interpret protocol actions in opposite ways. These constructors act similarly to the duality of left and right rules for intuitionistic session types [2]. 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 [6] 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, the 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 message that are erased prior to runtime. Basically, the DH protocol forces communication between Alice and Bob to be encrypted and maintain secrecy at runtime.

```
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 to 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 $\mathbf{hc}\langle\mathsf{DH}(p, g)\rangle$ ensure 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 Diffie-Hellman 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 certified 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.

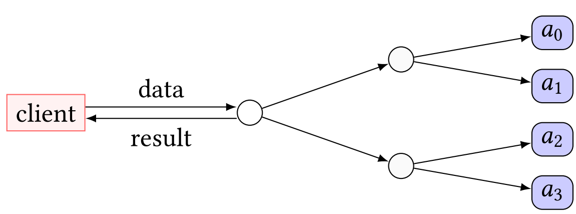

Fig. 1. Tree Diagram of Map-Reduce

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:

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

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

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

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, we must define the kinds of operations that can be performed. This requires the protocol of map-reduce to branch depending on what operation the client requests to perform. Unlike many prior session type systems [2, 5] which provide built-in constructs (e.g. $\oplus$ 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 session type for the map-reduce protocol. The following treeP protocol is used to describe the interactions between nodes in the map-reduce tree.

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

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 o of type opr A from its parent. The protocol then branches, via type-level pattern matching on o, into three cases. If o 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 has changed from A to B to reflect the fact that the data stored at the leaves of the sub-tree has been 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 o 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 o is Free.

Using the treeP protocol, we can now implement the worker processes that run at each node of the concurrent tree. The implementation of a leaf worker is shown below. We have elided uninteresting technical details regarding dependent pattern matching.

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

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.

```
def nodeWorker {A : U} {l r : tree A}
    (c_l : hc⟨treeP A l⟩) (c_r : hc⟨treeP A r⟩) (c : ch⟨treeP A (Node l r)⟩) : C(unit) :=
  let ⟨o, c⟩ := recv c in
  match o with
  | Map _ f ⇒
    let c_l ⇐ send c_l (Map f) in
    let c_r ⇐ send c_r (Map f) in
    let c ⇐ send c (just unit) in
    nodeWorker {B} {(map f l) (map f r)} c_l c_r c
  | Reduce _ f g ⇒
    let c_l ⇐ send c_l (Reduce f g) in
    let c_r ⇐ send c_r (Reduce f g) in
    let ⟨just v_l, c_l⟩ ⇐ recv c_l in
    let ⟨just v_r, c_r⟩ ⇐ recv c_r in
    let c ⇐ send c (just (g v_l v_r)) in
    nodeWorker {A} {l r} c_l c_r c
  | Free ⇒
    let c_l ⇐ send c_l Free in
    let c_r ⇐ send c_r Free in
    wait(c_l); wait(c_r); close(c)
```

Given the signature of nodeWorker and the definition of the treeP protocol, it is not hard 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 $\mathbf{ch}\langle \text{treeP } B\ (\text{map } f\ (\text{Node } l\ r))\rangle$ which simplifies to $\mathbf{ch}\langle \text{treeP } B\ (\text{Node } (\text{map } f\ l)\ (\text{map } f\ r))\rangle$. 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 $\mathbf{hc}\langle \text{treeP } B\ (\text{map } f\ l)\rangle$ and $\mathbf{hc}\langle \text{treeP } B\ (\text{map } f\ r)\rangle$.

3.2 A Certified 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.

```
type cTree (A : U) (t : tree A) := C(hc⟨treeP t⟩)

def cLeaf {A : U} (x : A) : cTree A (Leaf x) :=
  fork(c : ch⟨treeP A (Leaf x)⟩) with leafWorker x c

def cNode {A : U}{l r : tree A} (c_l : cTree A l) (c_r : cTree A r) : cTree (Node l r) :=
  let c_l ⇐ c_l in
  let c_r ⇐ c_r in
  fork(c : ch⟨treeP A (Node l r)⟩) with nodeWorker c_l c_r c
```

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 some 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 **hc**$\langle P \rangle$ and spawn a new process that runs the worker with the other end of the channel at type **ch**$\langle P \rangle$. The duality of the channels 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.

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

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 allows 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 [4].

```
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.

```
def splittingTree (xs : list int) : tree (list int) := match xs with
  | nil ⇒ Leaf nil
  | x :: nil ⇒ Leaf (x :: nil)
  | zs ⇒ let ⟨xs, ys⟩ := split zs in Node (splittingTree xs) (splittingTree ys)
```

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.

```
def splittingCTree (xs : list int) : ch⟨!(cTree (list int) (splittingTree xs)). 1⟩ → C(unit) :=
  match xs with
  | nil ⇒ let c ⇐ send c (cLeaf nil) in close(c); return ()
  | x :: nil ⇒ let c ⇐ send c (cLeaf (x :: nil)) in close(c); return ()
  | zs ⇒
    let ⟨xs, ys⟩ := split zs in
    let c_l ⇐ fork(c) with splittingCTree xs c in
    let c_r ⇐ fork(c) with splittingCTree ys c in
    ...
```

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

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

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.

```
theorem reduceSplittingTree :
  ∀(xs : list int). reduce (λ(x).x) merge (splittingTree xs) = msort xs
```

Using this theorem, we can rewrite the singleton value 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] 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) [7]. TLL is a dependent type theory that combines Martin-Löf-style dependent types [14] with linear types [10, 23]. Notably, TLL supports *essential linearity* [13] 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 as follows:

$$\begin{tikzcd} \boxed{\Gamma \vdash m : A \text{ (Logical Typing)}} \arrow[r, shift left, "\text{provides types}"] & \boxed{\Gamma ; \Delta \vdash m : A \text{ (Program Typing)}} \arrow[l, shift left, "\text{subjects to verify}"] \end{tikzcd}$$

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:

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

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:

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

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 are type parameters in polymorphic functions. For example, the polymorphic identity function can be implemented as

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

which has the type $\forall\{A : \mathsf{U}\} \to_{\mathsf{U}} \forall(x : A) \to_{\mathsf{U}} A$. 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 [19] 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:

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

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 [1, 3, 13, 15, 22] 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 [10] and bunched implications [16, 17] 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 [1, 15]), then there may be other aliases of v which can be used outside of f.

Wadler, in his seminal work [24], 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 [21] 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$_C$.

Protocols and Channel Types. As touched on in Section 2.1, 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.

Recursive protocols can be formed using the $\mu(x : A).m$ construct which is typed as follows:

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

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 and not proofs for arbitrary propositions. To ensure that protocols defined through $\mu(x : A).m$ are always productive, recursive usages of x must be syntactically *guarded* behind a protocol action in m. This enforces the *contractiveness* condition for recursive session types [8]. 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.

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.

Concurrency Monad.

REFERENCES

[1] 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*. https://doi.org/10.1145/3209108.3209189

[2] Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. 222–236. https://doi.org/10.1007/978-3-642-15375-4_16

[3] Iliano Cervesato and Frank Pfenning. 2002. A Linear Logical Framework. *Information and Computation* 179, 1 (2002), 19–75. https://doi.org/10.1006/inco.2001.2951

[4] Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* The MIT Press.

[5] 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. https://doi.org/10.1145/3414080.3414087

[6] W. Diffie and M. Hellman. 1976. New directions in cryptography. *IEEE Transactions on Information Theory* 22, 6 (1976), 644–654. https://doi.org/10.1109/TIT.1976.1055638

[7] Qiancheng Fu and Hongwei Xi. 2023. A Two-Level Linear Dependent Type Theory. arXiv:2309.08673 [cs.PL]

[8] Simon Gay and Vasco Vasconcelos. 2010. Linear type theory for asynchronous session types. *J. Funct. Program.* 20 (01 2010), 19–50. https://doi.org/10.1017/S0956796809990268

[9] Simon J. Gay, Peter Thiemann, and Vasco Thudichum Vasconcelos. 2020. Duality of Session Types: The Final Cut. *ArXiv* abs/2004.01322 (2020), 23–33.

[10] Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science* 50, 1 (1987), 1–101. https://doi.org/10.1016/0304-3975(87)90045-4

[11] Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR*.

[12] Anton Lorenzen, Daan Leijen, and Wouter Swierstra. 2023. FP²: Fully in-Place Functional Programming. *Proc. ACM Program. Lang.* 7, ICFP, Article 198 (aug 2023), 30 pages. https://doi.org/10.1145/3607840

[13] Zhaohui Luo and Y Zhang. 2016. *A Linear Dependent Type Theory.* 69–70.

[14] 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. https://doi.org/10.1016/S0049-237X(08)71945-1

[15] Conor McBride. 2016. I Got Plenty o' Nuttin'. In *A List of Successes That Can Change the World.*

[16] Peter O'Hearn. 2003. On bunched typing. *Journal of Functional Programming* 13, 4 (July 2003), 747–796. https://doi.org/10.1017/S0956796802004495

[17] Peter W. O'Hearn and David J. Pym. 1999. The Logic of Bunched Implications. *Bulletin of Symbolic Logic* 5, 2 (June 1999), 215–244. https://doi.org/10.2307/421090

[18] The Coq Development Team. 2020. *The Coq Proof Assistant, version 8.11.0.* https://doi.org/10.5281/ZENODO.3744225

[19] The Rust teams. 2022. *Rust Programming Language.* http://www.rust-lang.org/

[20] Peter Thiemann and Vasco T. Vasconcelos. 2019. Label-Dependent Session Types. *Proc. ACM Program. Lang.* 4, POPL, Article 67 (dec 2019), 29 pages. https://doi.org/10.1145/3371135

[21] David N. Turner and Philip Wadler. 1999. Operational interpretations of linear logic. *Theoretical Computer Science* 227, 1 (1999), 231–248. https://doi.org/10.1016/S0304-3975(99)00054-7

[22] Matthijs Vákár. 2014. Syntax and Semantics of Linear Dependent Types. *CoRR* abs/1405.0033 (2014). arXiv:1405.0033 http://arxiv.org/abs/1405.0033

[23] P. Wadler. 1990. Linear Types can Change the World!. In *Programming Concepts and Methods.*

[24] Philip Wadler. 1991. Is There a Use for Linear Logic? *SIGPLAN Not.* 26, 9 (May 1991), 255–273. https://doi.org/10.1145/115866.115894

[25] 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. https://doi.org/10.1145/2364527.2364568