# Dependent Session Types for Certified Concurrent Programming

ANONYMOUS AUTHOR(S)

We present $\text{TLL}_C$ 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 $\text{TLL}_C$ 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 $\text{TLL}_C$. 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 $\text{TLL}_C$ programs into concurrent C code is implemented and freely available.

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

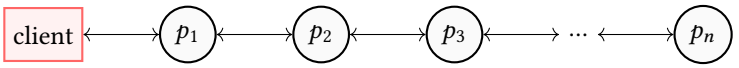
## 1 INTRODUCTION

Session types [Honda 1993] 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:

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

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 are linked together by bidirectional channels of type $\text{queue}_A$. 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 $\text{queue}_A$ 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 $\text{queue}_A$ 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 $\text{TLL}_C$, a dependent session type system which extends the Two-Level Linear dependent type theory (TLL) [Fu and Xi 2023] with session-typed concurrency. In $\text{TLL}_C$, one could define the queues through the following dependent session type:

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

Here, the type $\text{queue}(xs)$ 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 $\ell$ received from the client. The opr type which $\ell$ inhabits is defined as a simple inductive type with two constructors:

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

When a queue server receives an $\text{ins}(v)$ value, the type of the server becomes $\text{queue}(\text{snoc}(xs, v))$ 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* $\text{sing } x$) along with the channel $\textbf{hc}\langle\text{queue}(xs')\rangle$ 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:

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

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 [Gay and Vasconcelos 2010; Wadler 2012] have successfully combined *classical* session types with functional languages, its is well known that classical session types do not easily support recursive session types [Gay et al. 2020]

(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 [Caires and Pfenning 2010] 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 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:

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

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 $\text{TLL}_C$ and is widely applicable for integrating intuitionistic session types with other functional languages.

In order to show that $\text{TLL}_C$ ensures communication safety, we develop a process calculus based concurrency semantics. Process configurations in the calculus are collections of $\text{TLL}_C$ 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 $\text{TLL}_C$ 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 [The Coq Development Team 2020]. 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 [Lorenzen et al. 2023] 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 $\text{TLL}_C$. $\text{TLL}_C$ 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 $\text{TLL}_C$. We show that $\text{TLL}_C$, 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 $\text{TLL}_C$ into safe and efficient C code.

[1] **TODO**

## 2 OVERVIEW OF DEPENDENT SESSION TYPES

Session types in $\text{TLL}_C$ 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 $\text{TLL}_C$.

### 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 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 $\text{TLL}_C$: the $!(x : A).B$ and $?(x : A).B$ *protocol actions*. The syntax of these constructs take inspiration from binary session types [Gay and Vasconcelos 2010; Wadler 2012] and label dependent session types [Thiemann and Vasconcelos 2019], however the meaning of these constructs in $\text{TLL}_C$ 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 $\text{TLL}_C$, 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 [Caires and Pfenning 2010]. 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 $\text{TLL}_C$. 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 [Diffie and Hellman 1976] 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 $\text{TLL}_C$. 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.

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.

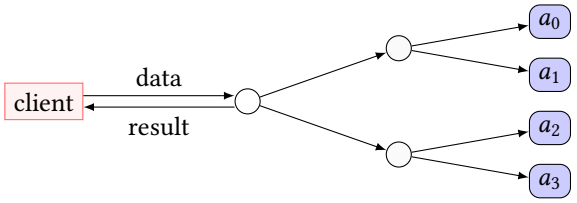



Fig. 1. Tree Diagram of Map-Reduce

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 $\text{TLL}_C$. 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 [Caires and Pfenning 2010; Das and Pfenning 2020] 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:

```
inductive opr(A : U) := Map : ∀{B : U} (f : A → B) → opr(A)
                      | Reduce : ∀{B : U} (f : A → B) (g : B → B → B) → opr(A)
                      | Free : opr(A)
```

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.

```
def treeP (A : U) (t : tree A) := ?(o : opr A).
  match o with Map _ f ⇒  treeP B (map f t)
               | Reduce _ f g ⇒ !(sing (reduce f g t)). treeP t
               | Free ⇒ 1
```

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.

```
def leafWorker {A : U} (x : A) (c : ch⟨treeP A (Leaf x)⟩) : C(unit) :=
  let ⟨o, c⟩ := recv c in
  match o with
  | Map ⇒ leafWorker B (f x) c
  | Reduce ⇒ let c ⇐ send c (just (f x)) in leafWorker A x c
  | Free ⇒ close(c)
```

## REFERENCES


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

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

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

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

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

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

Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR*.

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. https://doi.org/10.1145/3607840

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

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

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