Dependent Session Types for Certified Concurrent Programming

ANONYMOUS AUTHOR(S)

We present TLL$_C$ which extends the recently developed 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 establishment of precise correspondence relations between concurrent programs and their sequential counterparts. Correctness properties proven for sequential programs can now be easily lifted to their corresponding concurrent programs. Basically, session types become a powerful tool for intrinsically verifying the correctness of concurrent algorithms and data structures. Furthermore, we show how the computational irrelevancy mechanism of TLL$_C$ can be exploited to describe cryptographic protocols as session types. These cryptographic session types prevent secrecy from leaking at runtime and ensure that communication is always encrypted. 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 the Coq Proof Assistant. A prototype compiler which compiles TLL$_C$ programs into optimized 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 tool 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. GV [Wadler 2012] style session types further prevent well-typed concurrent processes from deadlocking. While (simple) session type systems guarantee runtime safety for concurrent programs, it remains difficult to write concurrent programs which are semantically correct. Human errors such as sending messages with incorrect values are still possible and could lead to well-typed programs producing wrong results. The additional complexities induced by concurrency make such errors even more common and insidious than in a purely sequential setting. Simple session types are ultimately too limited in terms of expressiveness to describe deeper correctness properties.

A natural idea to improve the expressiveness of session types is to extend them with term dependency. The hope is that by endowing session types with sufficient proof theoretic strength, protocols could be specified with such precision that semantic errors are no longer possible. While different forms of dependent session types have been proposed in the past such as *label dependent session types* [Thiemann and Vasconcelos 2019], *DML session types* [Wu and Xi 2017], *process dependent types* [Toninho and Yoshida 2018] and many others, this work is the first to fully combine Martin-Löf style dependency with session types. Martin-Löf Type Theory [Martin-Löf 1984] and its descendants [Coquand and Huet 1988; Luo 1994] have seen great success in the field of program verification thanks to their incredible expressiveness and intuitive Curry-Horward correspondence with higher-order logic. The dependent session types we develop inherit these fundamental properties and leverage them for concurrent program verification. In contrast to program logics such as Actris [Hinrichsen et al. 2019] which verify *external* concurrent programs through axiomatic reasoning, concurrency is directly integrated with the semantics of our *internal*

2022. 2475-1421/2022/1-ART $15.00
https://doi.org/

language. This facilitates a form of certified concurrent programming where theorem proving not only justifies the correctness of programs but also provides guidance towards their construction.

A major challenge in the development of dependent session types is the more general problem of integrating substructural typing with dependency. Caires and Pfenning [Caires and Pfenning 2010] show that session types have a canonical correspondence to Linear Logic [Girard 1995]. Languages which host dependent session types must support linear dependent types a priori. Consequently, the dependencies allowed by the host language determine the expressiveness of its session types. For this reason, we extend the Two-Level Linear dependent type theory (TLL) [Fu and Xi 2023] with session type based concurrency, forming the TLL_C (C for concurrent) programming language.

A key feature of TLL_C is *essential linearity* [Luo and Zhang 2016] where types can depend on linear terms. Dependency of this form is achieved in TLL_C by stratifying its type system into a logical level and a program level. The logical level is comprised of compile time objects that facilitate type checking such as proofs and types. Once type checking concludes, the logical level is erased. All that remain are program level terms which get evaluated at runtime. Basically, linear terms appearing in types do not contribute to resource usage. For this reason, the logical level is structural and types can safely depend on linear terms. On the other hand, the program level is substructural in order to control the usage of runtime resources such as references, channels, etc.

Communication protocols in TLL_C are built from basic *protocol actions* of the form $\rho(x : A) \to P$ where $\rho \in \{\Uparrow, \Downarrow\}$. Informally speaking, $\Uparrow(x : A) \to P$ is a protocol which sends message x of type A then continues as protocol P. Conversely, $\Downarrow(x : A) \to P$ receives message x then continues as P. In both cases, continuation P is allowed to depend on message x and other variables in the ambient context. It is through these dependencies that protocols gain their expressiveness.

$$\mathcal{P}(\mathcal{F}) := \Downarrow(x : T) \to \Uparrow(r : R) \to \Uparrow\{p : r =_R \mathcal{F}(x)\} \to \bullet$$

Fig. 1. Algorithmic Protocol

Figure 1 presents a simple dependent protocol $\mathcal{P}(\mathcal{F})$ where $\mathcal{F} : T \to R$ is the sequential implementation of an abstract algorithm $\mathcal{A}$. For a process implementing this protocol, it firsts receives input x (as specified by $\Downarrow(x : T)$) and then sends back message r (as specified by $\Uparrow(r : R)$). The following action $\Uparrow\{p : r =_R \mathcal{F}(x)\}$ requires a proof p asserting that r is equal to $\mathcal{F}(x)$ be sent. Finally, the protocol is terminated by $\bullet$. In other words, executing this protocol to completion produces a value that is provably equal to the result of sequential $\mathcal{F}$. An obvious way to obtain an acceptable value for r is by computing $\mathcal{F}(x)$ directly, however, the process implementing $\mathcal{P}(\mathcal{F})$ is not required to do so. Instead, the process is free to apply any optimization technique (including parallelization) it pleases so long as the protocol obligation $\Uparrow\{p : r =_R \mathcal{F}(x)\}$ is fulfilled. Sequential $\mathcal{F}$ essentially becomes a specification for concurrent implementations of algorithm $\mathcal{A}$.

$$H(E, D) := \Pi_{\text{U}}\{k : \mathcal{K}\}.\Pi_{\text{U}}\{m : \mathcal{M}\}.(D(k, E(k, m)) =_{\mathcal{M}} m)$$
$$\mathcal{E}(E, D) := \Uparrow\{k : \mathcal{K}\} \to \Uparrow\{m : \mathcal{M}\} \to \Uparrow(c : C) \to \Uparrow\{p : H(E, D) \times (c =_C E(k, m))\} \to \bullet$$

Fig. 2. Shannon Cipher Protocol

Designated by surrounding braces, proof p in Figure 1 is considered to be an *implicit message*. Similarly to ICC* [Barras and Bernardo 2008] which erases implicitly quantified arguments, implicit messages in TLL_C are erased and not transmitted at runtime. Despite not participating in actual communication, implicit messages are able to impose constraints on the transmitted messages. For instance, the proof obligation $\Uparrow\{p : r =_R \mathcal{F}(x)\}$ constrains the values of messages x and r in protocol $\mathcal{P}(\mathcal{F})$. The ability for implicit messages to statically refine protocols without runtime communication can be exploited to facilitate a novel encoding of cryptographic protocols as session types. Figure 2 demonstrates how a generalized Shannon cipher protocol can be encoded from the perspective of the encryptor. Given encryption function $E : \mathcal{K} \times \mathcal{M} \rightarrow C$ and decryption function $D : \mathcal{K} \times C \rightarrow \mathcal{M}$, the Shannon cipher protocol $\mathcal{E}(E, D)$ begins by sending implicit messages k and m which are the key and plaintext respectively. Afterwards, ciphertext c is explicitly sent. Finally, implicit message p of type $H(E, D) \times (c =_C E(k, m))$ is sent to complete the protocol. Intuitively, p is a proof that E and D form a proper Shannon cipher (proposition $H(E, D)$) and that c is obtained by encrypting m using function E and key k (proposition $c =_C E(k, m)$). Notice in the entire protocol, only ciphertext c will be transmitted at runtime while k, m and p all get erased. The Shannon cipher protocol basically forces communicated messages to always be encrypted and prevents key and plaintext from being transmitted. We demonstrate in Section 3.2 how the Diffie-Hellman key exchange can be expressed as a session type using this technique.

In order to show that TLL$_C$ ensures communication safety, we develop a process calculus based operational 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 of a channel are synchronized and ready to communicate, the process level semantics transmits their messages across the channel. Sound composition of processes is crucial to ensuring communication safety. TLL$_C$ mediates process composition through a process level type system. Processes communicating over a channel are required by the type system to respect the same protocol. This is similar in principle to intuitionistic session types [Caires and Pfenning 2010] where dual interpretations are given for each protocol through left-rules and right-rules. We adopt the intuitionistic notion of session types as it is comparatively easier to integrate with higher-order recursive protocols than the classical approach [Wadler 2012] which utilizes an explicit duality operator. We study the meta-theory of TLL$_C$ and prove that it is indeed sound as both a term calculus and a process calculus.

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. Proofs, source code, and example 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.
- Through examples, we demonstrate how TLL$_C$ can be applied to effectively construct and verify concurrent programs. We describe a novel method for specifying cryptographic protocols as session types which prevent secrecy from leaking at runtime and ensure communication is always encrypted
- We study the meta-theoretical properties of TLL$_C$ from both a term calculus point of view and a process calculus point of view. We show that TLL$_C$ as a term calculus exhibits 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]https://anonymous.4open.science/r/PLDI24-32BF

2 LINEAR DEPENDENT TYPES

2.1 Core TLL

The Two-Level Linear dependent type theory (TLL) of Fu and Xi [Fu and Xi 2023] is a type theory combing linearity and dependency. TLL is stratified into a level for logical reasoning and a level for constructing programs. While the logical level and program level share the same syntax, they have different typing rules. The logical level judgment $\Gamma \vdash m : A$ is fully structural whereas the program level judgment $\Gamma; \Delta \vdash m : A$ is substructural. TLL allows one to mark terms at the program level as computationally irrelevant using implicit quantifiers of the form $\Pi_s\{x : A\}.B$ similarly to the ICC* calculus of Barras and Bernardo [Barras and Bernardo 2008]. During compilation, terms defined at the logical level and program level arguments under implicit quantification are erased by replacing them with a special $\square$ value devoid of computational behavior.

$$\textsc{Sort}_L\ \frac{\Gamma \vdash}{\Gamma \vdash s : \mathrm{U}} \qquad \textsc{Var}_L\ \frac{\Gamma \vdash \qquad x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{ExplicitProd}_L\ \frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : r}{\Gamma \vdash \Pi_t(x : A).B : t} \qquad \textsc{ImplicitProd}_L\ \frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : r}{\Gamma \vdash \Pi_t\{x : A\}.B : t}$$

Fig. 3. Logical Level (Excerpt)

TLL utilizes two sorts (type of types) L and U for determining whether a type is linear or non-linear respectively. This design is inspired by the LNL_D type theory [Krishnaswami et al. 2015] which also uses sorts to distinguish linear and non-linear types. If type A is of sort L, then A is a linear type and its inhabiting terms must be consumed exactly once if used at the program level. If type B is of sort U, then B is a non-linear type and its inhabitants can be used freely. However, unlike LNL_D, TLL is able to express *essential linearity* advocated by Luo and Zhang [Luo and Zhang 2016] where types can depend on linear terms. Types are able to depend on linear terms because types are considered logical objects in TLL and are always erased during compilation. Terms appearing in types are used for reasoning about programs statically and do not contribute to runtime resource consumption. In this paper, we use variables s, r and t to range over sorts.

$$\textsc{ExplicitLam}_P\ \frac{\Gamma, x : A; \Delta, x :_s A \vdash m : B \qquad \Delta \triangleright t}{\Gamma; \Delta \vdash \lambda_t(x : A).m : \Pi_t(x : A).B} \qquad \textsc{ImplicitLam}_P\ \frac{\Gamma, x : A; \Delta \vdash m : B \qquad \Delta \triangleright t}{\Gamma; \Delta \vdash \lambda_t\{x : A\}.m : \Pi_t\{x : A\}.B}$$

$$\textsc{ExplicitApp}_P\ \frac{\Gamma; \Delta_1 \vdash m : \Pi_t(x : A).B \qquad \Gamma; \Delta_2 \vdash n : A}{\Gamma; \Delta_1 \uplus \Delta_2 \vdash m\ n : B[n/x]} \qquad \textsc{ImplicitApp}_P\ \frac{\Gamma; \Delta \vdash m : \Pi_t\{x : A\}.B \qquad \Gamma \vdash n : A}{\Gamma; \Delta \vdash m\ \{n\} : B[n/x]}$$

Fig. 4. Program Level (Excerpt)

Figure 3 and Figure 4 present excerpts from the typing rules of the logical level and program level respectively. The logical level is essentially MLTT [Martin-Löf 1984] with extra annotations for sorts and implicit quantifiers. All type formation rules are defined exclusively at the logical level since types are regarded as logical objects. For program level typing, the program context Δ forms an annotated subcontext of Γ and is used for tracking resources. Each entry in Δ is of the form $x :_s A$ where s is the sort of A. From the rules presented in Figure 4 we can clearly see the substructural nature of program level typing. Contraction of program contexts is handled by the partial function $\Delta_1 \Cup \Delta_2$ which only merges Δ_1 and Δ_2 if they have no overlapping linear entries. The restriction relation $\Delta \triangleright s$ forbids Δ from containing linear entries if $s = \mathrm{U}$ which ultimately prevents program contexts from being weakened with linear entries. Notice in the $\textsc{ImplicitApp}_P$ rule, the argument n is typed at the logical level without using Δ. This is because n gets erased during compilation and never consumes resources at runtime.

$$\textsc{MergeEmpty}\ \frac{}{\epsilon \Cup \epsilon = \epsilon} \qquad \textsc{MergeU}\ \frac{\Delta_1 \Cup \Delta_2 = \Delta \qquad x \notin \Delta}{(\Delta_1, x :_{\mathrm{U}} A) \Cup (\Delta_2, x :_{\mathrm{U}} A) = (\Delta, x :_{\mathrm{U}} A)} \qquad \textsc{MergeL1}\ \frac{\Delta_1 \Cup \Delta_2 = \Delta \qquad x \notin \Delta}{(\Delta_1, x :_{\mathrm{L}} A) \Cup \Delta_2 = (\Delta, x :_{\mathrm{L}} A)}$$

$$\textsc{MergeL2}\ \frac{\Delta_1 \Cup \Delta_2 = \Delta \qquad x \notin \Delta}{\Delta_1 \Cup (\Delta_2, x :_{\mathrm{L}} A) = (\Delta, x :_{\mathrm{L}} A)} \qquad \textsc{ReEmpty}\ \frac{}{\epsilon \triangleright s} \qquad \textsc{ReU}\ \frac{\Delta \triangleright \mathrm{U}}{\Delta, x :_{\mathrm{U}} A \triangleright \mathrm{U}} \qquad \textsc{ReL}\ \frac{\Delta \triangleright \mathrm{L}}{\Delta, x :_s A \triangleright \mathrm{L}}$$

Fig. 5. Substructural Context Operators

2.2 Concurrent TLL

To adapt TLL for session type based concurrency, we extend the program level typing judgment $\Delta; \Gamma \vdash m : A$ with an extra context Θ, resulting in TLL_C program judgments taking the form $\Theta; \Delta; \Gamma \vdash m : A$. Similarly to Δ, Θ is substructural and uses the same context operators $\Theta_1 \Cup \Theta_2$ and $\Theta \triangleright s$ to manage contraction and weakening. However, unlike Δ, Θ does not form a subcontext of Γ. In fact, Θ is not a context of variables at all. Instead, Θ tracks active channel names created by the process reduction of TLL_C. We make this distinction between channel names and standard variables to support the congruence rules of process calculus such as name restriction and scope extrusion. These congruence rules require renaming operations which do not interact favorably with standard variables due to technical reasons involving dependent types. By shifting the requirements imposed by the process calculus onto channel names, standard variables are able to behave normally. It is also important to note that context Θ is a technical device for studying TLL_C from a process calculus perspective. Θ and the names it contains only arise during intermediate stages of process reductions. Channel-like variables appearing in TLL_C code are actually standard variables which pass around channel names at runtime. Users never directly interact with Θ when programming.

$$\textsc{CtxEmpty}_C\ \frac{}{\epsilon \Vdash} \qquad \textsc{CtxName}_C\ \frac{\Theta \Vdash \qquad \epsilon \vdash A : \mathbf{proto} \qquad c \notin \Theta \qquad \rho \in \{\Uparrow, \Downarrow\}}{\Theta, c :_{\mathrm{L}} \rho\mathbf{ch}\langle A \rangle \Vdash}$$

Fig. 6. Channel Name Context

Figure 6 presents the judgment $\Theta \Vdash$ which formally states that Θ is a well-formed channel name context. In the $\textsc{CtxName}_C$ rule, we see all names c in Θ have types of the form $\rho\mathbf{ch}\langle A \rangle$. We refer to $\rho\mathbf{ch}\langle A \rangle$ as an *endpoint type* which describes the interactions expected on channel c through the indexing protocol A and *role indicator* ρ. Since Θ tracks active channels created during process reduction, it is required by premise $\epsilon \vdash A : \mathbf{proto}$ for the indexing protocol A to be closed. We go into greater detail on TLL_C protocols in Section 3.1 and endpoints types in Section 3.2.

From the perspective of an individual process, sent messages “vanish into nothingness” and received message “appear out of thin air”. In other words, concurrency is an effect. TLL_C uses a monad (written as C) similarly to Toninho et al. [Toninho et al. 2013] to construct programs which perform the concurrency effect. Figure 7 presents basic C monad typing rules. Notice the $C\textsc{Type}_L$

rule is defined exclusively at the logical level without a program level counterpart since $C(A)$ is a linear type and types are logical objects in TLL$_C$. The program level rules Return$_P$ and Bind$_P$ track channel names and resources using Θ and Δ respectively. Structural variants of Return$_P$ and Bind$_P$ also exist at the logical level program in the form of Return$_L$ and Bind$_L$. These logical variants are rather uninteresting and are included to maintain admissibility of TLL's program reflection property where all program level terms can be freely reflected into the logical level.

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

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

Fig. 7. Concurrency Monad

For the remainder of this paper, we mostly refrain from presenting logical variations of program level rules since readers can easily derive them by deleting mentions of Θ and Δ in the program level rules. In examples presented using TLL$_C$ concrete syntax such as Figure 17, `IO` is the C monad. Additionally, the monadic bind **let** $x \Leftarrow m$ **in** n is written using OCaml-like syntax `let* x := m in n`.

3 DEPENDENT SESSION TYPES THROUGH EXAMPLES

In this section, we present examples of how dependent session types in TLL$_C$ facilitate certified concurrent programming. These examples serve as a vehicle for introducing the language of TLL$_C$ and building intuition towards a holistic understanding of its design.

3.1 Message Specification

An immediate use case for dependent session types is the precise specification of messages communicated between parties. This is useful in practical network and systems applications where the contents of messages may depend on the value of a prior request.

```
⇑(sz: nat) → ⇓(msg: bytes) → ⇓{size msg = sz} → •
```

Fig. 8. Simple Dependent Protocol

Figure 8 presents a simple dependent protocol. Informally speaking, this protocol first expects a natural number `sz` to be sent across the channel followed by receiving a byte vector `msg`. In simple session type systems without dependency, there would be no way to specify that the size of `msg` is dependent on the previously communicated `sz`. However, dependent session types allow us to express relations between messages. Notice in the third action expected by the protocol, the party sending `msg` must send a proof that the size of `msg` is indeed equal to `sz` according to an agreed upon `size` function. Finally the protocol terminates and communication ends.

This example showcases the main primitives for constructing dependent protocols in TLL$_C$. The typing rules for protocol primitives are formally presented in Figure 9. From rule Proto$_L$ we see that **proto** (the type of protocols) inhabits the U sort. Protocols in TLL$_C$ are considered non-linear as they represent logical specifications of expected communication. On the other hand, channel endpoints (indexed by protocols) are linear because they are resources provided by the underlying language runtime. The requirement that a channel endpoint always gets consumed is what ensures that communications over the channel follow protocol.

$$\textsc{Proto}_L\ \frac{\Gamma \vdash}{\Gamma \vdash \mathbf{proto} : \mathrm{U}} \qquad \textsc{End}_L\ \frac{\Gamma \vdash}{\Gamma \vdash \bullet : \mathbf{proto}} \qquad \textsc{ExplicitAction}_L\ \frac{\Gamma, x : A \vdash B : \mathbf{proto}}{\Gamma \vdash \rho(x : A) \longrightarrow B : \mathbf{proto}} \qquad \textsc{ImplicitAction}_L\ \frac{\Gamma, x : A \vdash B : \mathbf{proto}}{\Gamma \vdash \rho\{x : A\} \longrightarrow B : \mathbf{proto}}$$

$$\text{where } \rho \in \{\Uparrow, \Downarrow\}$$

Fig. 9. Protocol Actions

Let us take a closer look at the protocols themselves. Written as $\rho(x : A) \longrightarrow B$ where $\rho \in \{\Uparrow, \Downarrow\}$, explicit action indicates that a message x of type A will be communicated across the channel and continue as protocol B. From the rules presented in Figure 9 we can clearly see protocol B depending on message x. The ability to utilize x within B not only facilitates the specification of relations between communicated messages, but also allows B to be computed through type level computations involving x (large elimination). Dedicated selection protocols such as $\oplus\{l : S_l\}_{l \in A}$ and $\&\{l : S_l\}_{l \in A}$ found in non-dependent session type systems are no longer necessary since they can be encoded as type level `match` expressions. Figure 10 demonstrates how boolean variable `b` branches the protocol into `P₁` and `P₂` through large elimination.

```
⇑(b: bool) → match true ⇒ P₁ | false ⇒ P₂
```

Fig. 10. Boolean Protocol Selection

The implicit action, written as $\rho\{x : A\} \longrightarrow B$ is much more subtle when compared to the explicit action. This protocol expects a computationally irrelevant (erasable) term x to be provided then continue as protocol B. Objects that implicit action appears to communicate are erased after type checking and never actually get sent across the channel at runtime. The irrelevancy mechanism of TLL which we build on, guarantees that such objects can be consistently erased without compromising runtime behavior. Although implicit actions seem meaningless as their messages ultimately get erased, it is important to remember that erased messages can still impose proof obligations on other relevant messages during type checking. This application of implicit action is clear from the example depicted in Figure 8 where an irrelevant message of type `size msg = sz` relates relevant messages `msg` and `sz`. The ability for implicit actions to instantiate protocols through irrelevant messages will be explored in greater detail in the next section.

3.2 Secrecy Protection

Previously, we have seen how dependent protocols can be used to enforce relations between communicated messages. Interestingly, implicit actions allow one to specify message relations without actual communication at runtime. In this section, we exploit the erasure of irrelevant messages to specify cryptographic protocols similarly to the encoding of Shannon cipher in Section 1.

Figure 11 demonstrates how the Diffie-Hellman key exchange [Diffie and Hellman 1976] protocol can be encoded as a dependent function. The `#[logical]` modifier indicates to TLL$_C$ that `DH` is an irrelevant definition which will be erased before runtime. Given publicly known integers `p` and `g`, `DH` returns a protocol describing the steps of key exchange. 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 of the protocol as Bob. From Alice's perspective, she first sends her secret value `a` as an irrelevant message to initialize her half of the protocol. Next, her public value `A` is sent as a relevant message to Bob along with a proof that `A` is correctly computed from values `p`, `g` and `a`. At this point, Alice has finished sending messages and waits for messages from Bob to complete the key exchange. She first receives Bob's secret `b` as an irrelevant message which initializes his half of the protocol. Later, Bob's public value `B` is received as a relevant message along with a proof that `B` is correctly computed from value `p`, `g` and `b`. Notice between Alice and Bob, only the relevant messages `A` and `B` will be exchanged at runtime. The secret values `a` and `b` and other correctness proofs will be pruned by erasure because they are applied through implicit actions. Basically, `DH` forces communication between Alice and Bob to be encrypted and maintain secrecy at runtime.

```
#[logical]
def DH (p g: int): proto :=
  ⤊{a: int} → ⤊(A: int) → ⤊{A = (powm g a p)} →
  ⤋{b: int} → ⤋(B: int) → ⤋{B = (powm g b p)} → •
```

Fig. 11. Diffie-Hellman Protocol

As we have alluded to in Section 3.1, protocols such as `DH` are just logical specifications of expected communication. The actual messages are transmitted through channels. Figure 12 illustrates the relationship between channels, endpoints and protocols. Each channel possesses two endpoints of types $\Uparrow\mathbf{ch}\langle A\rangle$ and $\Downarrow\mathbf{ch}\langle A\rangle$ which are indexed by a shared governing protocol A. Written more generally as $\rho\mathbf{ch}\langle A\rangle$, the *role indicator* $\rho \in \{\Uparrow, \Downarrow\}$ of an endpoint determines how processes holding the endpoint interprets A and interacts with it. Basically, if Alice interprets protocol A as send, Bob on the other end of the channel interprets A as receive. By convention, we define a boolean algebra inspired *role-xor* operator in Figure 14 to show how every protocol action is interpreted from every role's perspective. A *role-negation* function is defined in Figure 15 to indicate $\rho\mathbf{ch}\langle A\rangle$ and $\neg\rho\mathbf{ch}\langle A\rangle$ are duals of each other and can be realized as opposite endpoints of a channel. Figure 13 presents the logical level formation rule of endpoint types and the program level rule for introducing channel names. When presenting examples, we often abbreviate $\Uparrow\mathbf{ch}\langle A\rangle$ as $\mathbf{ch}\langle A\rangle$ and $\Downarrow\mathbf{ch}\langle A\rangle$ as $\mathbf{hc}\langle A\rangle$.

$$\begin{tikzcd}\Uparrow\mathbf{ch}\langle A\rangle \arrow[r, leftrightarrow, "A : \mathbf{proto}", "\text{channel}"'] & \Downarrow\mathbf{ch}\langle A\rangle\end{tikzcd}$$

Fig. 12. Channel and Endpoints

$$\textsc{EndpointType}_L\quad \frac{\Gamma \vdash A : \mathbf{proto}}{\Gamma \vdash \rho\mathbf{ch}\langle A\rangle : \mathrm{L}}$$

$$\textsc{ChannelName}_L\quad \frac{\Gamma;\Delta \vdash \qquad \epsilon \vdash A : \mathbf{proto} \qquad \Delta \triangleright \mathrm{U}}{c :_{\mathrm{L}} \rho\mathbf{ch}\langle A\rangle;\Gamma;\Delta \vdash c : \rho\mathbf{ch}\langle A\rangle}$$

Fig. 13. Endpoint Type and Channel Name

In some sense, the endpoints types of TLL$_C$ can be seen as a form of intuitionistic session types [Caires and Pfenning 2010] where protocols have opposite meanings depending on if they are used in left-rules or right-rules. The role indicator on an endpoint type $\rho\mathbf{ch}\langle A\rangle$ essentially determines whether it is a "left-endpoint" or a "right-endpoint". Figure 16 presents communication primitives for sending and receiving messages. We can see from ExplicitSend$_P$ and ExplicitRecv$_P$ (resp. ImplicitSend$_P$ and ImplicitRecv$_P$) how role-xor interprets a protocol $\rho_2(x : A) \to B$ into either sending or receiving based on opposite ρ_1 indicators of endpoints. The erasure of messages applied to implicit actions is also realized through these rules. Observe that ImplicitSend$_P$ constructs $\underline{\mathbf{send}}\ m$ which is an implicitly quantified function. Recall from Section 2.1 that program arguments under implicit quantification are erased during compilation. So for an expression $\underline{\textbf{send}}\, m\ \{n\}$ where n is implicitly applied, n is erased during compilation and does not get transmitted at runtime. Likewise, the left component of $\underline{\textbf{recv}}\, m$ simply defaults to □ at runtime without receiving anything.

$$\Uparrow \wedge \Uparrow = \Uparrow \qquad \Uparrow \wedge \Downarrow = \Downarrow$$
$$\Downarrow \wedge \Uparrow = \Downarrow \qquad \Downarrow \wedge \Downarrow = \Uparrow$$

Fig. 14. Role-xor

$$\neg \Uparrow = \Downarrow$$
$$\neg \Downarrow = \Uparrow$$

Fig. 15. Role-negation

We decide not to use classical style session types in TLL_C because it is unclear to us how the duality $(\_)^\perp$ meta-operator could be extended to accommodate arbitrary expressions occurring inside protocols. Furthermore, it is well known that naive definitions of duality interact poorly with recursive session types [Bernardi and Hennessy 2016; Bernardi et al. 2014; Lindley and Morris 2016]. Our current design allows dependent protocols to be refined incrementally by sent or received messages. Expressions inside protocols are eventually refined into interpretable canonical primitives. The issues concerning classical duality and recursive session types also do not manifest as it is unnecessary to compute the duals of recursive protocols when connecting processes together.

$$\textsc{ExplicitSend}_P\ \frac{\Theta;\Gamma;\Delta \vdash m : \rho_1\textbf{ch}\langle \rho_2(x:A) \to B\rangle \qquad \rho_1 \wedge \rho_2 = \Uparrow}{\Theta;\Gamma;\Delta \vdash \textbf{send}\, m : \Pi_{\text{L}}(x:A).C(\rho_1\textbf{ch}\langle B\rangle)}$$

$$\textsc{ImplicitSend}_P\ \frac{\Theta;\Gamma;\Delta \vdash m : \rho_1\textbf{ch}\langle \rho_2\{x:A\} \to B\rangle \qquad \rho_1 \wedge \rho_2 = \Uparrow}{\Theta;\Gamma;\Delta \vdash \underline{\textbf{send}}\, m : \Pi_{\text{L}}\{x:A\}.C(\rho_1\textbf{ch}\langle B\rangle)}$$

$$\textsc{ExplicitRecv}_P\ \frac{\Theta;\Gamma;\Delta \vdash m : \rho_1\textbf{ch}\langle \rho_2(x:A) \to B\rangle \qquad \rho_1 \wedge \rho_2 = \Downarrow}{\Theta;\Gamma;\Delta \vdash \textbf{recv}\, m : C(\Sigma_{\text{L}}(x:A).\rho_1\textbf{ch}\langle B\rangle)}$$

$$\textsc{ImplicitRecv}_P\ \frac{\Theta;\Gamma;\Delta \vdash m : \rho_1\textbf{ch}\langle \rho_2\{x:A\} \to B\rangle \qquad \rho_1 \wedge \rho_2 = \Downarrow}{\Theta;\Gamma;\Delta \vdash \underline{\textbf{recv}}\, m : C(\Sigma_{\text{L}}\{x:A\}.\rho_1\textbf{ch}\langle B\rangle)}$$

Fig. 16. Communication Primitives

Two simple programs `alice` and `bob` implementing the `DH` key exchange are presented in Figure 17. In this example, the `send` (resp. `recv`) operator is overloaded to fulfill the purposes of both explicit **send** and implicit **send** depending on the type of channel endpoint `c`. The `close` operator for terminating communication is also overload with **close** and **wait** whose rules are formally presented in Figure 18. For `alice`, the `b` message she implicitly receives from `bob` represents his secrecy and is computationally irrelevant (marked by surrounding braces). Due to the fact that `b` will not actually be sent to her at runtime, `alice` may only use it in conjunction with `pf` of type `B = powm g b p` for hypothetical reasoning. Likewise, `bob` does not gain access to a copy of `alice`'s secret `a` at runtime either.

```
def alice (a p g : int) (c : ch⟨ DH p g ⟩)
: IO 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 ⟩)
: IO 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
  close c
```

Fig. 17. Alice (holder of secret a) and Bob (holder of secret b)

The RSA encryption algorithm [Rivest et al. 1978] can also be encoded as a dependent protocol in a similar manner by using implicit actions to specify the relationship between public keys and private keys. We present a specification of the RSA protocol and a client-server pair implementing the protocol in the appendix.

$$\textsc{Close}_{\mathcal{P}}\ \frac{\Theta; \Gamma; \Delta \vdash m : \Uparrow\mathbf{ch}\langle\bullet\rangle}{\Theta; \Gamma; \Delta \vdash \mathbf{close}\, m : C(\mathbf{unit})} \qquad \textsc{Wait}_{\mathcal{P}}\ \frac{\Theta; \Gamma; \Delta \vdash m : \Downarrow\mathbf{ch}\langle\bullet\rangle}{\Theta; \Gamma; \Delta \vdash \mathbf{wait}\, m : C(\mathbf{unit})}$$

Fig. 18. Termination Primitives

3.3 Data Structures and Algorithms

In this section, we demonstrate how certified concurrent trees equipped with map and fold higher-order functions can be constructed and verified using dependent session types. Each node in the concurrent tree is realized by a process which holds data and channels for communicating with sub-trees. Map and fold operations are triggered by sending request signals between process in the tree. The `signal` type which encodes these requests is presented in Figure 19. Parameter `A` in the definition of `signal` indicates that it is to be sent to concurrent trees with `A` typed data. In the sequential setting, it is well known that `map` which transforms data in a tree can be implemented in terms of `fold` by constructing a new tree with updated nodes. However, such an implementation of map would not be satisfactory for concurrent trees because constructing a new tree is expensive. Instead, concurrent map updates data "in-place" while preserving the original tree structure. This is the reason why `Map` and `Fold` are encoded as different signals as they accomplish different tasks.

```
inductive signal (A : U) : L where
| Free : signal A
| Map  ?{B : U} (f : A → B) : signal A
| Fold ?{B : U} (b : B)
  (f : A → B → B → B) : signal A
```

Fig. 19. Tree Signals

```
#[logical]
def tree_p ?{A : U} (t : tree A) : proto :=
  ⇓(sig : signal A) →
    match sig with
    | Free ⇒ •
    | Map B f ⇒ @tree_p B (map f t)
    | Fold _ b f ⇒
      ⇑(sing (fold b f t)) → tree_p t
```

Fig. 20. Tree Protocol

The `tree_p` protocol governing concurrent trees is presented in Figure 20. Intuitively, it describes a concurrent tree which acts equivalently to sequential tree `t` with regards to map and fold functionality. A process implementing `tree_p` receives `sig` which branches the protocol into performing different actions. In the `Free` case, the protocol terminates and disconnects all processes in the tree from each other. These isolated processes can then be deallocated. For `Map`, the `tree_p` protocol simply recurses with an updated index `map f t`. This means after concurrent map, the resulting concurrent tree is in a state equivalent to sequential `map` applied to `t`. Notice that after `Map`, `tree_p` recurses with type `B` as all `A` typed data in the original tree are transformed by `f` into new data of type `B`. Finally, `Fold` tells the process to send a singleton value equal to `fold b f t` before repeating the protocol.

Figure 21 presents the $\textsc{RecType}_L$ rule for constructing recursive protocols such as `tree_p` and `queue_p`. Recursive protocols are necessary for server-like processes to communicate with their clients indefinitely. In order to ensure that protocols defined through $\mu(x : A).m$ are always productive, a syntactic guard condition is enforced on recursive usages of x where each usage must appear after a protocol action. This essentially makes recursive protocols behave similarly to coinductive data in Coq [The Coq Development Team 2020] or Agda [Agda development team 2023].

$$\textsc{RecType}_L \quad \dfrac{\Gamma, x : A \vdash m : A \qquad x \text{ guarded by protocol action in } m}{\Gamma \vdash \mu(x : A).m : A}$$

Fig. 21. Recursive Type

The `leaf_worker` and `node_worker` functions are presented in Figure 22. The processes which execute these functions are the leaves and nodes of the concurrent tree. For `leaf_worker`, its job is simple as it holds no data. The only interesting behavior of `leaf_worker` is that its type parameter changes from `A` to `B` after `Map` to account for the type change of the overall concurrent tree. The definition of `node_worker` on the other hand is significantly more involved. A `node_worker` function possesses an argument `x` which holds data, two endpoints `l_ch` and `r_ch` for communicating with the left and right sub-trees, and endpoint `c` which connects to the node's parent process. Given the `Free` signal, `node_worker` signals its left and right sub-trees to terminate and deallocates its endpoints. In the case of `Map`, function `f` is of type `A → B` and transforms data held by the node. The function `f` also gets applied to the sub-trees of the current node by forwarded `Map` signals. Finally, `node_worker` responds to `Fold` by requesting the folded results of its left and right sub-trees which are then reduced together with `x` by `f` and sent on `c`[2].

```
#[logical]def ctree_ch ?{A : U} (t : tree A) : L := ch⟨tree_p t⟩
#[logical]def ctree_hc ?{A : U} (t : tree A) : L := hc⟨tree_p t⟩

def leaf_worker ?{A : U} (c : @ctree_ch A Leaf) : IO unit :=
  let* ⟨sig, c⟩ := recv c in
  match sig as sig0, c as _ : ... with
  | Free, c ⇒ close c
  | Map B _, c ⇒ @leaf_worker B c
  | Fold _ b _, c ⇒
    let* c := send c (just b) in leaf_worker c

def node_worker ?{A : U} ?{l r : tree A} (x : A)
: ctree_hc l .→ ctree_hc r .→ ctree_ch (Node x l r) .→ IO unit
| l_ch, r_ch, c ⇒
  let* ⟨sig, c⟩ := recv c in
  match sig as sig0, c as _ : ... with
  | Free, c ⇒
    let* l_ch := send l_ch Free in
    let* r_ch := send r_ch Free in
    close l_ch; close r_ch; close c
  | Map B f, c ⇒
    let* l_ch := send l_ch (Map f) in
    let* r_ch := send r_ch (Map f) in
    node_worker (f x) l_ch r_ch c
  | Fold _ b f, c ⇒
    let* l_ch := send l_ch (Fold b f) in
    let* r_ch := send r_ch (Fold b f) in
    let* ⟨just vl, l_ch⟩ := recv l_ch in
    let* ⟨just vr, r_ch⟩ := recv r_ch in
    let* c := send c (just (f x vl vr)) in
    node_worker x l_ch r_ch c
```

Fig. 22. Leaf and Node Workers

[2] `just` is the unique constructor of `sing`

In order to create processes such as the ones realizing concurrent trees, TLL$_C$ includes a **fork** primitive whose typing rules are presented in Figure 23. Given a program m which requires a computationally relevant variable x of type $\rho\mathbf{ch}\langle A\rangle$, the **fork** primitive spawns a new child-process to evaluate m. A channel facilitating communication between the child process and its parent is also allocated. One endpoint of type $\rho\mathbf{ch}\langle A\rangle$ is held by the child process and substitutes for variable x in m. The opposing endpoint of type $\neg\rho\mathbf{ch}\langle A\rangle$ is held by the parent process. As explained in Section 3.2, role-xor interprets protocol A from the perspective of role ρ as the dual of interpreting from $\neg\rho$'s perspective. Hence, synchronization between parent and child process is achieved.

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

Fig. 23. Process Forking

Concurrent tree "constructors" such as the ones defined in Figure 24 can be applied to allocate and connect leaf and node processes. Analogously to sequential tree constructors which allocate heap memory for storing data, these concurrent constructors allocate processes using `fork` to evaluate their respective worker functions. It is clear from the type signatures of `cleaf` and `cnode` the concurrent trees they produce are structured correctly.

```
#[logical]
def ctree (t : tree) : L := IO (ctree_hc t)

def cleaf ?{A : U} (_ : unit)
: ctree (@Leaf A) :=
  fork (c : ctree_ch Leaf) ⇒ leaf_worker c

def cnode ?{A : U} ?{l r : tree A} (a : A)
: ctree l .→ ctree r .→ ctree (Node a l r)
| l0, r0 ⇒
  let∗ l_ch := l0 in
  let∗ r_ch := r0 in
  fork (c : ctree_ch (Node a l r)) ⇒
    node_worker a l_ch r_ch c
```

Fig. 24. Concurrent Tree Constructors

```
def cmap ?{A B : U} ?{t : tree A}
  (f : A → B) (ct : ctree t)
: ctree (map f t) :=
  let∗ c := ct in
  let∗ c := send c (Map f) in
  return c

def cfold ?{A B : U} ?{t : tree A}
  (b: B) (f: A → B → B → B) (ct: ctree t)
: IO (sing (fold b f t) ⊗ ctree t) :=
  let∗ c := ct in
  let∗ c := send c (Fold b f) in
  let∗ ⟨x, ct⟩ := recv c in
  return ⟨x, return ct⟩
```

Fig. 25. Concurrent Map and Fold

The operational details of `leaf_worker` and `node_worker` can be neatly encapsulated using wrapper functions `cmap` and `cfold` as presented in Figure 25. The arguments expected by these higher-order concurrent functions are almost identical to their sequential counterparts found in popular functional programming languages such as Haskell or OCaml. In terms of functionality, `cmap` and `cfold` operate similarly to the MapReduce model of distributed programming where `cmap` delegates jobs to individual processes in the tree and `cfold` reduces the results. The logical tree index `t` certifies that our higher-order concurrent functions are implemented correctly according to the semantics expected of mapping and folding trees.

In the appendix, we construct a certified concurrent queue whose type signature subsumes the refinement based verification of Das and Pfenning [Das and Pfenning 2020].

4 PROCESS CONFIGURATIONS

In the previous section we presented the language and typing rules for terms which form individual processes. To compose these processes together coherently, we introduce the process level typing judgment $\Theta \Vdash P$ in Figure 26 which formally states that a configuration of processes P is well-typed under endpoint context Θ. Context Θ essentially lists all the active endpoints of P which other processes can connect to.

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

Fig. 26. Process Typing

The rules governing the formation of process configurations are standard in session type literature. The Expr rule asserts that well-typed closed terms of type $C(\mathbf{unit})$ are valid processes, basically giving us the smallest building blocks from which all configurations are constructed. For the Par rule, processes P and Q can be composed together in parallel as $P \mid Q$. The active endpoints held by P and Q are aggregated together as $\Theta_1 \cup \Theta_2$. The final Scope rule connects two endpoints c and d if they are governed by the same protocol A but have opposing roles. This allows processes utilizing c and d to communicate with each other.

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

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

Fig. 27. Structural Congruence

The structural congruence of process configurations is defined as the least congruence relation satisfying the conditions list in Figure 27. Structural congruence specifies that parallel composition is commutative, associative and compatible with channel restriction. Furthermore, processes which terminate with value $()$: **unit** are deallocated. Basically, if there is $P \equiv Q$, then P and Q should be viewed as equivalent processes as far as communication is concerned.

5 META THEORY

The presentation of TLL$_C$'s meta-theory is organized into three main categories: logical, program and process. The logical level and program level theories are mostly the same as TLL. They characterize the properties of a singular TLL$_C$ process. More specifically, the logical level is sound for reasoning about programs and the program level safely tracks resources. Finally, we show communication soundness among processes in a well-typed configuration.

5.1 Logical Level

$$\textsc{StepExplicitBeta}_L\ \frac{m \Rightarrow m' \qquad n \Rightarrow n'}{(\lambda_t(x : A).m)\ n \Rightarrow m'[n'/x]} \qquad \textsc{StepImplicitBeta}_L\ \frac{m \Rightarrow m' \qquad n \Rightarrow n'}{(\lambda_t\{x : A\}.m)\ \{n\} \Rightarrow m'[n'/x]} \qquad \textsc{StepBindElim}_L\ \frac{m \Rightarrow m' \qquad n \Rightarrow n'}{\mathbf{let}\ x \Leftarrow \mathbf{return}\ m\ \mathbf{in}\ n \Rightarrow n'[m'/x]}$$

Fig. 28. Logical Reductions (Excerpt)

The operational semantics of the logical level is defined in terms of a standard parallel reduction relation $m \Rightarrow n$. Figure 28 presents an excerpt of logical reduction. Unlike the program level semantics which is of a call-by-value style, logical reduction is not restricted to any fixed evaluation order. This allows more types to be convertible through logical reduction and leads to a stronger notion of type conversion. Importantly, if a communication primitive such as **recv** is used at the logical level, it cannot be reduced nor can it be executed to communicate with another process. Basically, the logical level is completely pure and is unaffected by the concurrency effect.

The first theorem of the logical level is confluence. As logical reductions do not have a fixed evaluation order, confluence is used to join together different reduction paths. From an implementation perspective, confluence allows one to check the equality of types by first reducing them to their normal forms. Without confluence, different reduction strategies may result in the loss of definitional equalities. We use the standard diamond property technique to prove confluence.

Theorem 5.1 (Confluence of Logical Reduction). *If $m \Rrightarrow^* m_1$ and $m \Rrightarrow^* m_2$, then there exists n such that $m_1 \Rrightarrow^* n$ and $m_2 \Rrightarrow^* n$.*

The logical level of TLL_C is a dependent type theory where types are first class citizens. This means that types themselves have types. The type validity theorem shows that for all types, there exists a sort which it inhabits.

Theorem 5.2 (Type Validity). *For any logical level typing judgment $\Gamma \vdash m : A$, there exists sort s such that $\Gamma \vdash A : s$ holds.*

Sorts (U and L) are of great importance in TLL_C. The modality of any given type is intrinsically determined by the sort it inhabits. By definition, linear types are of sort L and non-linear types are of sort U. Thus, it is important for the sorts of types to be unambiguous as to avoid conflicting situations where a type is both linear and non-linear. The sort uniqueness theorem proves that the sort of each TLL_C type is unique.

Theorem 5.3 (Sort Uniqueness). *If logical typing judgments $\Gamma \vdash A : s$ and $\Gamma \vdash A : t$ are both valid, then sort s must be equal to sort t.*

The subject reduction theorem states that the types of logical level terms are preserved by reductions. It is necessary for types to be preserved by reduction for reduction based type equality checking. This is to prevent the equality checker from altering the types of its subjects.

Theorem 5.4 (Logical Subject Reduction). *If there are logical level typing $\Gamma \vdash m : A$ and reduction $m \Rrightarrow n$, then $\Gamma \vdash n : A$ holds.*

5.2 Program Level

Figure 29 presents an excerpt of the program level reduction relation $m \rightsquigarrow\!\!\!\!\twoheadrightarrow n$. As mentioned in Section 5.1, the reductions at the program level are similar to call-by-value. Notice for the StepExplicitBeta$_P$ and StepBindElim$_P$ rules, the substituted term v is required to be a value. This is in contrast to the logical reduction rules which do not impose any restrictions on the forms of substituted terms. The exception to this call-by-value requirement is StepImplicitBeta$_P$ which immediately substitutes the implicitly applied argument n without first reducing it to a value. Implicitly applied arguments such as n are actually logical level objects as shown in Figure 4 which are used at the program level to instantiate implicit quantifiers. They are computationally irrelevant and do not contain any runtime resources. So StepImplicitBeta$_P$ is allowed to substitute n directly because it will not alter the runtime behavior or the resource consumption of program m.

$$\textsc{StepExplicitBeta}_P\ \frac{v\ \mathit{value}}{(\lambda_s(x : A).m)\ v \rightsquigarrow\!\!\!\!\twoheadrightarrow m[v/x]} \qquad \textsc{StepImplicitBeta}_P\ \frac{}{(\lambda_s\{x : A\}.m)\ \{n\} \rightsquigarrow\!\!\!\!\twoheadrightarrow m[n/x]} \qquad \textsc{StepBindElim}_P\ \frac{v\ \mathit{value}}{\mathbf{let}\ x \Leftarrow \mathbf{return}\ v\ \mathbf{in}\ n \rightsquigarrow\!\!\!\!\twoheadrightarrow m[v/x]}$$

Fig. 29. Program Reductions (Excerpt)

The typing rules of TLL_C designed around call-by-value program semantics are more permissive than call-by-name systems such as QTT [Atkey 2018]. Intuitively, a call-by-value language consumes its resources once to produce a single value which gets used multiple times. On the other hand, call-by-name substitutes a suspended computation which contains references to resources. Each time one of these suspended computations is evaluated, another copy of its resources is required. Formally, the value stability theorem upper bounds the resources held by a TLL_C value with the sort of its type. Notice that both Θ and Δ are bounded by this theorem. For a value inhabiting a non-linear type, value stability implies that it does not contain resources and can be used freely. Values inhabiting linear types are allowed to hold resources because they will be used exactly once.

Theorem 5.5 (Value Stability). *If there is a value v well-typed according to program level typing $\Theta; \Gamma; \Delta \vdash m : A$ and $\Gamma \vdash A : s$, then there is $\Theta \triangleright s$ and $\Delta \triangleright s$.*

To show that program level typing $\Theta; \Gamma; \Delta \vdash m : A$ correctly tracks relevant usages of active endpoints through context Θ, we define the $occurs_c(m)$ meta-function in Figure 30 to count relevant occurrences of endpoint c in m. Appearances of c in type annotations or implicit arguments are not counted by $occurs_c$ because these usages are computationally irrelevant.

$$
\begin{aligned}
occurs_c(\lambda_s\{x : A\}.m) &= occurs_c(m) & occurs_c(m\ \{n\}) &= occurs_c(m)\\
occurs_c(\lambda_s(x : A).m) &= occurs_c(m) & occurs_c(m\ n) &= occurs_c(m) + occurs_c(n)\\
occurs_c(\mathbf{return}\ m) &= occurs_c(m) & occurs_c(\mathbf{let}\ x \Leftarrow m\ \mathbf{in}\ n) &= occurs_c(m) + occurs_c(n)\\
occurs_c(x) &= 0 & occurs_c(d) &= \begin{cases} 1, & \text{if } c = d\\ 0, & \text{otherwise}\end{cases}
\end{aligned}
$$

Fig. 30. Program Level Endpoint Occurrences (Excerpt)

The program endpoint theorem formally relates the active endpoints in context Θ to their number of computationally relevant uses. Basically, all endpoints in Θ are used relevantly exactly once.

Theorem 5.6 (Program Endpoint). *For any valid program typing $\Theta; \Gamma; \Delta \vdash m : A$, if there is $c \in \Theta$, then $occurs_c(m) = 1$. Conversely, if there is $c \notin \Theta$, then $occurs_c(m) = 0$.*

In Martin-Löf style dependent type systems, programs are substituted into types by dependent elimination. This means that program level terms in TLL_C must be able to substitute for logical variables occurring in types. To facilitate dependent elimination, the program reflection theorem allows programs to be reflected into the logical level for free. Furthermore, this theorem greatly reduces code duplication by allowing program level definitions to be shared with the logical level.

Theorem 5.7 (Program Reflection). *For any valid program typing $\Theta; \Gamma; \Delta \vdash m : A$, logical typing $\Gamma \vdash m : A$ is derivable.*

Unlike logical reduction which must work under context for checking type equality, program reductions are only ever applied to well-typed closed programs at runtime. The program subject reduction theorem shows that reductions do not alter the types of closed programs. The presence of context Θ here allows closed programs to still communicate with each other through channels.

Theorem 5.8 (Program Subject Reduction). *If there are program typing $\Theta; \epsilon; \epsilon \vdash m : A$ and reduction $m \rightsquigarrow\!\!\!\!\!\twoheadrightarrow n$, there is $\Theta; \epsilon; \epsilon \vdash n : A$*

5.3 Process Level

The semantics of process interactions is presented in Figure 31. The structural rules ProcPar, ProcScope and ProcCongr are standard in most process calculi. Due to the fact that TLL_C is also a term calculus, the ProcExpr rule is used to lift program reductions to the process level. For the remaining rules concerned with concurrency primitives, their functionality are triggered through monadic bind because concurrency is treated as a monadic effect in TLL_C. Monadic bind essentially becomes a very specific evaluation context where concurrent effects take place.

$$\textsc{ProcPar}\ \frac{P \Rrightarrow Q}{O \mid P \Rrightarrow O \mid Q} \qquad \textsc{ProcScope}\ \frac{P \Rrightarrow Q}{\nu cd.P \Rrightarrow \nu cd.Q} \qquad \textsc{ProcCongr}\ \frac{P \equiv P' \quad P' \Rrightarrow Q' \quad Q' \equiv Q}{P \Rrightarrow Q} \qquad \textsc{ProcExpr}\ \frac{m \rightsquigarrow\!\!\!\!\rightarrow n}{\langle m \rangle \Rrightarrow \langle n \rangle}$$

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

$$\textsc{ProcExplicit}\ \frac{v\ \textit{value}}{\begin{array}{l}\nu cd.(\langle \textbf{let}\ x \Leftarrow \textbf{send}\ c\ v\ \textbf{in}\ n_1 \rangle \mid \langle \textbf{let}\ y \Leftarrow \textbf{recv}\ d\ \textbf{in}\ n_2 \rangle) \Rrightarrow \\ \nu cd.(\langle \textbf{let}\ x \Leftarrow \textbf{return}\ c\ \textbf{in}\ n_1 \rangle \mid \langle \textbf{let}\ y \Leftarrow \textbf{return}\ \langle v, d \rangle_{\text{L}}\ \textbf{in}\ n_2 \rangle)\end{array}}$$

$$\textsc{ProcImplicit}\ \frac{}{\begin{array}{l}\nu cd.(\langle \textbf{let}\ x \Leftarrow \underline{\textbf{send}}\ c\ \{m\}\ \textbf{in}\ n_1 \rangle \mid \langle \textbf{let}\ y \Leftarrow \underline{\textbf{recv}}\ d\ \textbf{in}\ n_2 \rangle) \Rrightarrow \\ \nu cd.(\langle \textbf{let}\ x \Leftarrow \textbf{return}\ c\ \textbf{in}\ n_1 \rangle \mid \langle \textbf{let}\ y \Leftarrow \textbf{return}\ \langle \{m\}, d \rangle_{\text{L}}\ \textbf{in}\ n_2 \rangle)\end{array}}$$

$$\textsc{ProcEnd}\ \frac{}{\begin{array}{l}\nu cd.(\langle \textbf{let}\ x \Leftarrow \textbf{close}\ c\ \textbf{in}\ m \rangle \mid \langle \textbf{let}\ y \Leftarrow \textbf{wait}\ d\ \textbf{in}\ n \rangle) \Rrightarrow \\ \langle \textbf{let}\ x \Leftarrow \textbf{return}\ ()\ \textbf{in}\ m \rangle \mid \langle \textbf{let}\ x \Leftarrow \textbf{return}\ ()\ \textbf{in}\ n \rangle\end{array}}$$

Fig. 31. Process Reductions

For a well-typed process configuration $\Theta \Vdash P$, context Θ is said to track active endpoints in P. To formalize this, the *occurs* meta-function is extended as shown in Figure 32 to count active endpoints in a configuration. Care is taken in the scope case to avoid miscounting clashing endpoint names.

$$\begin{aligned} occurs_c(\langle m \rangle) &= occurs_c(m) \\ occurs_c(P \mid Q) &= occurs_c(P) + occurs_c(Q) \\ occurs_c(\nu c'd.P) &= \begin{cases} occurs_c(P), & \text{if } c \neq c' \text{and } c \neq d \\ 0, & \text{otherwise} \end{cases} \end{aligned}$$

Fig. 32. Process Level Endpoint Occurrences

The process endpoint theorem formally shows that Θ forms a complete list of all active endpoints used in well-type process configurations.

Theorem 5.9 (Process Endpoint). *For any valid process typing $\Theta \Vdash P$, if there is $c \in \Theta$, then $occurs_c(P) = 1$. Conversely, if there is $c \notin \Theta$, then $occurs_c(P) = 0$.*

Similarly to other session type systems [Gay and Vasconcelos 2010; Honda 1993; Thiemann and Vasconcelos 2019] in the tradition of Honda, TLL_C sacrifices deadlock freedom in favor of allowing interleaving communication between processes. We believe TLL_C to be deadlock free in the style of GV [Wadler 2012] if all processes spawn from a single initial process through forking. However, we have not yet formally proven this hypothesis and work is ongoing. The communication soundness of TLL_C is instead shown through Theorem 5.10 (process congruence) and Theorem 5.11 (session fidelity). The process congruence theorem asserts that structural congruence is indeed congruent with regards to process typing. Structurally congruent processes can essentially be identified as an equivalence class as far as process typing and reduction are concerned. The session fidelity theorem states that well-typed process configurations stay well-typed after reduction which means these processes communicate safely and remain synchronized.

Theorem 5.10 (Process Congruence). *If there are process typing $\Theta \Vdash P$ and structural congruence $P \equiv Q$, then there is $\Theta \Vdash Q$.*

Theorem 5.11 (Session Fidelity). *If there are process typing $\Theta \Vdash P$ and process reduction $P \Rrightarrow Q$, then there is $\Theta \Vdash Q$.*

6 IMPLEMENTATION

The high level pipeline of our prototype compiler is illustrated in Figure 33. All components of the compiler are written in OCaml. The emitted C code can be further compiled into an executable binary using popular C compilers such as gcc or clang on platforms that support POSIX threads. We discuss the inference, linearity checking and optimization phases in detail as they constitute the most interesting aspects of the compiler.

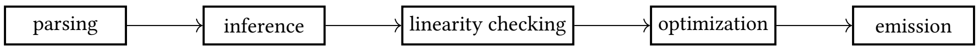

Fig. 33. Compiler Pipeline

6.1 Inference

To reduce the amount of boiler plate code users need to write, we implement two forms of inference in the TLL_C compiler. The first kind of inference deals with instantiating *sort-polymorphic schemes* and the second kind deals with elaborating inferred arguments.

```
def id‹s› ?{A : Type‹s›} (x : A) : A := x
```

Fig. 34. Sort Polymorphic Identity Function

Consider the identity function defined in Figure 34. We refer to `id` as a *sort-polymorphic scheme* as it is parameterized over sort variable `s`. Depending on the modality of the type applied for `A`, sort `s` can be instantiated to either L for a linear type or U for a non-linear type. Basically, `id` works as a generic identity function for both modalities. This eliminates the need to write duplicate versions of `id` to be polymorphic over types of a specific sort. The next parameter `?{A : Type‹s›}` states that `A` is a type with sort `s` and is marked by `?` to be an inferred argument. Inferred arguments do not have to be explicitly written. For example, one could write `id 12` and the TLL_C compiler will automatically infer that `A` = `int`. Additionally, the compiler will infer that `s` = U because `int` is of sort U. The inference phase of the compiler essentially instantiates all sort-polymorphic schemes with proper sorts and elaborates inferred arguments to be explicitly applied. Our `id 12` example is transformed by the inference phase into `(@id‹U›int 12)`.

Our inference algorithm is unification based and uses two kinds of meta-variables for expressing constraints on sorts and terms. Given an expression, the inference algorithm inserts meta-variables in all positions marked as inferable by the user. Next, bidirectional-style type checking is performed to generate equational constraints on the values of meta-variables. The resulting sort equations are quite simple as the language of sorts is very limited. First-order unification suffices to solve for sort meta-variables. On the other hand, the expressiveness of terms greatly increases the difficulty of solving term equations. In fact, it is well known that the higher-order unification required for dependent type inference is undecidable [Goldfarb 1981]. Fortunately, a restricted form of higher-order unification problems known as the *pattern fragment* is decidable [Miller 1991]. Our unification algorithm applies heuristics to transform term equations into the pattern fragment. Equations which cannot be immediately transformed into the pattern fragment are suspended until new equations become available. The idea is that these new equations may provide useful information on how to solve the suspended unification problems.

6.2 Linearity Checking

During the inference phase, the usage of linear variables is not restricted. The type checking algorithm treats TLL_C as a fully structural dependently typed language. Once all sort-polymorphic schemes and inferable arguments have been inferred and made explicit, the linearity checking phase begins. A substructural type checking algorithm is applied to determine if the elaborated syntax tree complies with the actual typing rules of TLL_C. We adopt this two phase approach of type checking to simplify the linearity checking algorithm. Although sort polymorphism greatly reduces code duplication from a user's perspective, it also obfuscates the sorts of types which makes linearity checking much more difficult. The scheme instantiations performed by the inference phase clarifies the modalities of types, allowing linearity to be checked much more easily.

To support dependent pattern matching, we implement a variation of Cockx's algorithm [Cockx and Abel 2018] to type check `match` expressions and elaborate them into well-formed case trees. Cockx's algorithm forms the basis of Agda's [Agda development team 2023] pattern matching mechanism which has been widely praised for its refinement power and user friendliness. Several extensions are made to the original algorithm to account for pattern matching on linear inductive types and matching on computationally irrelevant terms. Our modified algorithm is able to correctly track resource usage in subtle cases such as nested patterns involving linear inductive types.

6.3 Optimization

Once the linearity checking phase has concluded, computationally irrelevant terms are erased in a type directed manner. Although syntax trees obtained from erasure are pruned of logical level terms, special annotations are introduced into the syntax tree to mark the linearity of certain critical expressions. For instance, `match` expressions are annotated with modality information about its discriminee. These annotations facilitate further optimizations for improving runtime efficiency and reducing program memory footprint.

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. Inductive types with one constructor that takes one computationally relevant argument can be immediately unboxed to expose the argument. For example, `just` is the only constructor of the `sing` inductive type and takes exactly one relevant argument. Expressions of the form `just m` are unboxed to reduce the number of indirections needed to access `m` at runtime.

Heap memory allocation is an expensive operation. To improve runtime performance, we utilize in-place updates for linear data to reduce the number of runtime system calls for memory allocation. This optimization is similar to recent works on functional in-place programming [Lorenzen et al. 2023; Reinking* et al. 2020] where allocated heap memory is reused instead of being garbage collected. In contrast to these works which utilize reference counting to dynamically check the viability of an in-place update, our compiler exploits the modality information generated by linearity checking to statically determine where in-place updates can be safely applied. The soundness of our memory optimization is justified by the heap semantics of TLL [Fu and Xi 2023].

7 RELATED WORK

Session types are a class of type systems pioneered by Honda [Honda 1993] for structuring dyadic communication in the π-calculus. Abramsky notices deep connections between the Linear Logic [Girard 1987] of Girard and concurrency, predicting that Linear Logic will play a foundation role in future theories of concurrent computation [Abramsky 1993, 1994]. Caires and Pfenning show an elegant correspondence between session types and Linear Logic. Gay and Vasconcelos integrate session types with λ-calculus [Gay and Vasconcelos 2010] 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 [Wadler 2012].

Toninho together with Caires and Pfenning develops the first dependent session type systems [Pfenning et al. 2011; Toninho et al. 2011]. These works extend the existing logic of Caires and Pfenning [Caires and Pfenning 2010] with universal and existential quantifiers to precisely specify properties of communicated messages.

Toninho and Yoshida present an interesting language [Toninho and Yoshida 2018] 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 [Wu and Xi 2017] implement session types in the ATS programming language [Xi 2010] which supports DML style dependent types [Xi 2007]. This allows them to specify the properties of concurrent programs and verify them using SMT based proof automation. While DML style dependency is well suited automatic reasoning, certain properties can be difficult to encode due to restrictions on the type level language.

Thiemann and Vasconcelos [Thiemann and Vasconcelos 2019] 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 approach to dependent types 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 [Das and Pfenning 2020] 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. Unrestricted Martin-Löf style dependency [Martin-Löf 1984] allows TLL_C to verify concurrent programs with greater precision than DML or refinement based approaches at the cost of decidable proof automation.

Hinrichsen et al. develop the Actris [Hinrichsen et al. 2019] program logic which extends the Iris [Jung et al. 2015] 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 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 CONCLUSIONS

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. Furthermore, the computational irrelevancy mechanism of TLL_C facilitates a novel method of encoding cryptographic protocols as session types which ensures secrecy protection. 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.

An interesting direction of research we intend to explore is the integration of dependency with multi-party session types [Honda et al. 2016]. 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. On the practical side, we plan to investigate applications of dependent session types for zero-knowledge proof systems.

REFERENCES

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

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

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

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

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.

Giovanni Bernardi and Matthew Hennessy. 2016. Using higher-order contracts to model session types. *Logical Methods in Computer Science* 12 (06 2016). https://doi.org/10.2168/LMCS-12(2:10)2016

Giovanni Tito Bernardi, Ornela Dardha, Simon J. Gay, and Dimitrios Kouzapas. 2014. On Duality Relations for Session Types. In *TGC*.

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

Jesper Cockx and Andreas Abel. 2018. Elaborating Dependent (Co)Pattern Matching. *Proc. ACM Program. Lang.* 2, ICFP, Article 75 (jul 2018), 30 pages. https://doi.org/10.1145/3236770

Thierry Coquand and Gérard Huet. 1988. The calculus of constructions. *Information and Computation* 76, 2 (1988), 95–120. https://doi.org/10.1016/0890-5401(88)90005-3

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

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

Jean-Yves Girard. 1995. Linear logic: its syntax and semantics.

Warren D. Goldfarb. 1981. The undecidability of the second-order unification problem. *Theoretical Computer Science* 13, 2 (1981), 225–230. https://doi.org/10.1016/0304-3975(81)90040-2

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. https://doi.org/10.1145/3371074

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

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1, Article 9 (mar 2016), 67 pages. https://doi.org/10.1145/2827695

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. https://doi.org/10.1145/2775051.2676980

Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. 2015. Integrating Linear and Dependent Types. *SIGPLAN Not.* 50, 1 (Jan. 2015), 17–30. https://doi.org/10.1145/2775051.2676969

Sam Lindley and J. Garrett Morris. 2016. Talking Bananas: Structural Recursion for Session Types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 434–447. https://doi.org/10.1145/2951913.2951921

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

Zhaohui Luo. 1994. *Computation and Reasoning: A Type Theory for Computer Science.* Oxford University Press, Inc., USA.

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

Per Martin-Löf. 1984. *Intuitionistic type theory.* Studies in proof theory, Vol. 1. Bibliopolis.

Dale A. Miller. 1991. Unification of Simply Typed Lamda-Terms as Logic Programming. In *International Conference on Logic Programming*. https://api.semanticscholar.org/CorpusID:451294

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.

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..

R. L. Rivest, A. Shamir, and L. Adleman. 1978. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM* 21, 2 (feb 1978), 120–126. https://doi.org/10.1145/359340.359342

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

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. https://doi.org/10.1145/2003476.2003499

Bernardo Toninho, Luis Caires, and Frank Pfenning. 2013. Higher-Order Processes, Functions, and Sessions: A Monadic Integration. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 350–369.

Bernardo Toninho and Nobuko Yoshida. 2018. *Depending on Session-Typed Processes.* 128–145. https://doi.org/10.1007/978-3-319-89366-2_7

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

Hanwen Wu and Hongwei Xi. 2017. Dependent Session Types. *CoRR* abs/1704.07004 (2017). arXiv:1704.07004 http://arxiv.org/abs/1704.07004

Hongwei Xi. 2007. Dependent ML An approach to practical programming with dependent types. *Journal of Functional Programming* 17, 2 (2007), 215–286. https://doi.org/10.1017/S0956796806006216

Hongwei Xi. 2010. *The ATS Programming Language.* http://www.ats-lang.org/