Label-Dependent Session Types

PETER THIEMANN, University of Freiburg, Germany
VASCO T. VASCONCELOS, University of Lisbon, Portugal

Session types have emerged as a typing discipline for communication protocols. Existing calculi with session types come equipped with many different primitives that combine communication with the introduction or elimination of the transmitted value.

We present a foundational session type calculus with a lightweight operational semantics. It fully decouples communication from the introduction and elimination of data and thus features a single communication reduction, which acts as a rendezvous between senders and receivers. We achieve this decoupling by introducing label-dependent session types, a minimalist value-dependent session type system with subtyping. The system is sufficiently powerful to simulate existing functional session type systems. Compared to such systems, label-dependent session types place fewer restrictions on the code. We further introduce primitive recursion over natural numbers at the type level, thus allowing to describe protocols whose behaviour depends on numbers exchanged in messages. An algorithmic type checking system is introduced and proved equivalent to its declarative counterpart. The new calculus showcases a novel lightweight integration of dependent types and linear typing, with has uses beyond session type systems.

CCS Concepts: • **Theory of computation → Type theory**; **Type structures**; • **Software and its engineering → Concurrent programming structures**.

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

ACM Reference Format:
Peter Thiemann and Vasco T. Vasconcelos. 2020. Label-Dependent Session Types. *Proc. ACM Program. Lang.* 4, POPL, Article 67 (January 2020), 29 pages. https://doi.org/10.1145/3371135

1 INTRODUCTION

Session types enable fine-grained static control over communication protocols. They evolved from a structuring device for two-party communication in π-calculus [Honda 1993; Honda et al. 1998; Takeuchi et al. 1994] over calculi embedded in functional languages [Gay and Vasconcelos 2010; Vasconcelos et al. 2006] to a powerful means of describing multi-party orchestration of communication [Honda et al. 2008, 2016]. There are embeddings in object-oriented languages [Dezani-Ciancaglini et al. 2009; Gay et al. 2010] and uses in the context of scripting languages [Honda et al. 2011], just to mention a few. Their logical foundations have been investigated with interpretations in intuitionistic and classical linear logic [Caires and Pfenning 2010; Caires et al. 2016; Wadler 2012].

There is a range of designs for foundational calculi for session types [Caires and Pfenning 2010; Caires et al. 2016; Castagna et al. 2009; Vasconcelos 2012]. They all use a session type to describe a sequence of messages. Its primitive constituents are sending and receiving a typed message

Authors' addresses: Peter Thiemann, Department of Informatics, Faculty of Engineering, University of Freiburg, Germany, thiemann@acm.org; Vasco T. Vasconcelos, LASIGE, Department of Informatics, Faculty of Sciences, University of Lisbon, Portugal, vv@di.fc.ul.pt.

2475-1421/2020/1-ART67
https://doi.org/10.1145/3371135

(!$A.S$ and ?$A.S$), signaling an internal choice ($R \oplus S$), reacting to an external choice ($R \& S$), and marking the end of a conversation (**end**), which is sometimes decomposed into an active and a passive end marker ($\mathbf{end}_!$ and $\mathbf{end}_?$). Types in dependent session calculi [Toninho et al. 2011; Toninho and Yoshida 2018] furthermore contain quantifiers $\forall x : A.S$ and $\exists x : A.S$. This distinction is well-motivated by logical concerns and results in different proof term constructions for each of the session operators. At the operational level, however, the types $!A.S$, $R \oplus S$, $\mathbf{end}_!$, and $\forall x : A.S$ are implemented by sending a message and then acting on it in some manner. It seems wasteful to have many different syntactic forms that fundamentally perform the same operation. Moreover, it would be closer to an actual implementation to have primitive operations for message passing and have the subsequent actions performed using standard types.

Other researchers also strived to reduce the number of primitive communication operations in session calculi. For instance, Lindley and Morris [2016] (following Dardha et al. [2012]; Kobayashi [2002]) elide special expressions for internal and external choice by expressing choice using a standard sum type. In their encoding $\overline{R \oplus S}$ is $!(\overline{R} + \overline{S}).\mathbf{end}_!$, which has the same high-level behavior, but the actual messages that are exchanged are quite different. The standard implementation of $\overline{R \oplus S}$ on the wire sends a single bit to indicate the choice to the receiver and then continues on the same conversation, whereas the implementation of $!(\overline{R} + \overline{S}).\mathbf{end}_!$ sends a representation of the sum value, one bit and then a serialization of a channel for R or a channel S, closes the conversion, and continues the protocol on the other end of the R or S channel. Clearly, the two implementations are not wire compatible with one another. Moreover, the encoding using sum types is more expensive to implement as it involves higher-order channel passing: a new channel must be created, serialized, sent over the existing channel, and deserialized at the other end [Hu et al. 2008].

Padovani [2017a] proposes a different encoding that does not require the creation of new channels or channel passing. The encoding of internal choice sends one of the constructor functions of the sum type and clever typing guarantees that the type of the channel changes appropriately.

Our calculus LDST of Label-Dependent Session Types is yet more economic in that it requires just a single pair of communication operations, send and receive, to implement a binary session-type calculus. Moreover, this implementation does not require higher-order communication, nor transmission of functions, nor clever retyping to achieve type soundness, session fidelity, and communication safety. In particular, the encoding of binary choice only needs to transmit one bit.

Label dependency is a very limited form of dependent types where values can depend on labels drawn from a finite set. The labels play the role of labels in internal and external choices, similar to variant labels in polymorphic variant types [Castagna et al. 2016; Garrigue 1998] or first-class record labels [Nishimura 1998]. Hence, session types in LDST are dependent as in $!(x : A)B$ or $?(x : A)B$, which means to send A (or receive A) and continue as B, which may depend on x.

Labels can further serve as end markers in protocols and thus the label-dependent calculus is "wire compatible" to the standard encoding of functional session types like the LAST calculus [Gay and Vasconcelos 2010] of Linear Asynchronous Session Types. In fact, a synchronous version of LAST can be fully emulated in LDST.

Label dependency does not require a full-blown lambda calculus in the types: a large elimination construct for a finite set of labels—a case expression on labels—suffices. As a more general example, we outline an extension with natural numbers and large elimination with a recursor.

LDST reinforces the connection between session types and linear logic [Caires and Pfenning 2010; Caires et al. 2016; Toninho et al. 2011; Wadler 2012]. The send operation maps a channel of dependent type $!(x : A)B$ into a single-use function $\Pi_{\mathbf{lin}}(x : A)B$ whereas the receive operation takes a channel of dependent type $?(x : A)B$ to a single-use dependent sum $\Sigma(x : A)B$.

Last, but not least, LDST proposes a novel, lightweight approach to integrate linear types with dependent types. The key is an operator $\Gamma \triangleleft x : A$ that conditionally extends a type environment. Roughly speaking, if A is a linear type, then it returns Γ unchanged; if A is unrestricted, then it returns $\Gamma, x : A$. This operator enables a uniform treatment of dependent and non-dependent Pi, Sigma, and other types. For example, type formation for a Pi type like $\Pi_m(x : A)B$ checks the type B by using $\Gamma \triangleleft x : A$. Conditional extension automatically degrades the type $\Pi_m(x : A)B$ to a non-dependent function type if A is linear. On the other hand, B can depend on x, if A is unrestricted. As we will see, LDST can only have meaningful dependencies on label types (and natural numbers in the extended version).

After providing some motivation in Section 2 and reminding the reader of binary session types in Section 3, we claim the following contributions for this work, starting in Section 4.

- A foundational functional session type calculus LDST with a minimal set of communication primitives.
- A type system with label-dependent types, linear session types, and subtyping. Besides the usual Π- and Σ-types, there are label-dependent types for sending and receiving.
- Support for natural numbers and primitive recursion at the type level (Section 5).
- A novel approach to integrating linear types and dependent types using conditional extension.
- Standard metatheoretical results (Section 6).
- Decidable subtyping and type checking that is sound and complete (Section 7).
- A typing- and semantics-preserving embedding of synchronous LAST in LDST (Section 8).
- Implementation of a type checker (Section 9).

Full sets of typing rules, proofs, auxiliary lemmas, and an example type derivation are available in our technical report [Thiemann and Vasconcelos 2019].

2 MOTIVATION

Functional session types extend functional programming languages like Haskell and ML with precise typings for structured communication on bidirectional heterogeneously typed channels. The typing guarantees that communication actions never mismatch (session fidelity) and that only values of the expected type arrive at the receiving end (communication safety).[1]

2.1 Binary Session Types

As an example for a typical system with binary session types [Gay and Vasconcelos 2010; Padovani 2017b], let's consider the code in Listing 1. It describes a compute server, `cServer`, that accepts two commands, `Neg` and `Add`, on a channel, then receives one or two integer arguments depending on the command, performs the respective operation, sends the result, and closes the channel.

The channel type in the argument of `cServer` starts with an external choice & between the two commands. After receiving command `Neg`, the channel has type ?**Int**. !**Int**. **end**$_!$, that is, receive an integer, send an integer, and then close the channel. The case for command `Add` is analogous.

The structure of the code follows the structure of the type. Variable `c` is processed linearly as it changes type according to the state of the channel bound to it. The **rcase** (receiving case) receives a label on the channel and branches accordingly. The `c` following labels `Neg` and `Add` is a binder for the updated channel endpoint in the two branches. The **recv** operation returns a pair of the updated channel and the received value, **send** returns the updated channel, and **close** consumes the channel end and returns unit.

Listing 3 shows a potential client for this server. The type of the channel end `d` is an internal choice ⊕ with a single `Neg`-labeled branch. The code again follows the type structure. It performs a

[1] There are session type systems that guarantee deadlock freedom, but the systems we consider in this paper do not.

```
cServer :
  & { Neg: ?Int. !Int. end!
    , Add: ?Int. ?Int. !Int. end! }
  → Unit
cServer c =
  rcase c of {
    Neg: c. let (x, c) = recv c
                c = send c (-x)
            in  close c,
    Add: c. let (x, c) = recv c
                (y, c) = recv c
                c = send c (x+y)
            in  close c
  }
```

Listing 1. Compute server

```
⊕ { Neg: !Int. ?Int. end?
  , Add: !Int, !Int. ?Int. end? }
```

Listing 2. Dual of cServer's session type

```
negClient :
  ⊕ { Neg: !Int. ?Int. end? }
  → Int → Int
negClient d x =
  let d = select Neg d
      d = send d x
      (r, d) = recv d
      wait d
  in  r
```

Listing 3. Compute client

select operation, which sends the Neg label (an internal choice), then sends an integer operand, receives an integer result, and acknowledges channel closure with the **wait** operation. This single-branch channel type is a supertype of the two-branch type in Listing 2, which is the *dual* of the server's channel type. The dual type has the same structure as the original type, but with sending and receiving types exchanged. All session type systems require that the types of the two endpoints of a channel are duals of one another to guarantee session fidelity. Of course, channel ends can be used at any suitable supertype.

2.2 The Case for Economy

The example demonstrates an issue that makes programming with session types more arcane than necessary. There are three different send operations, **send**, **select**, and **close**, and three matching receive operations, **recv**, **rcase**, and **wait**. They are reasonably easy to use, but they lead to bloated APIs for session types. The multitude of operations also bloats the syntax and semantics of foundational calculi for session types.

Wouldn't it be enticing if there was a session calculus with just a single pair of primitives for sending and receiving messages? The resulting functional session type calculus would be close to an implementation as it would have just one reduction for communication alongside the standard expression reductions.

The problem with the existing calculi is that they entangle the sending/receiving of data with another unrelated operation, which introduces the sent data or eliminates the received data. The **calculus of label-dependent session types** disentangles communication from introducing and eliminating the data values. It comes with a type of first-class labels that plays the role of labels in choice and branch types of traditional session type systems. The calculus features dependent product and sum types where the dependency is limited to labels. The types of the sending and receiving operations can be dependent on the transmitted values if they are labels.

For illustration, we translate the server session type from § 2.1 to a label-dependent session type TServer in Listing 4. One new ingredient is the type $\{l_1, \ldots, l_n\}$, which denotes the non-empty set of labels l_1 through l_n. The other new ingredient is the **case** expression in the type, which dispatches on a label to determine the type of the subsequent communication. This type introduces label dependency.

```
type TServer =
  ?(l:{Neg, Add}).
  case l of
  { Neg: ?Int. !Int. !{EOS}. End
  , Add: ?Int. ?Int. !Int. !{EOS}.
      End }
```

Listing 4. Label-dependent server type

```
type TClient =
  !(l:{Neg, Add}).
  case l of
  { Neg: !Int. ?Int. ?{EOS}. End
  , Add: !Int. !Int. ?Int. ?{EOS}.
      End }
```

Listing 6. Dual type of TServer

```
lServer : TServer → End
lServer c =
  let (l, c) = recv c
      (x, c) = recv c
  in case l of
  { Neg: let c = send c (−x)
         in  send c EOS,
  , Add: let (y, c) = recv c
             c = send c (x+y)
         in  send c EOS
  }
```

Listing 5. Label-dependent compute server

```
lClient : !{Neg}. !Int. ?Int. ?{EOS
    }. End → Int → Int
lClient d x =
  let d = send d 'Neg
      d = send d x
      (r, d) = recv d
      (_, _) = recv d // ((), EOS)
  in  r
```

Listing 7. Compute client

Like the **rcase** operation, a channel of this type first receives a label, on which the rest of the type depends. Type-level reduction of the **case** on the label reveals the type of the rest of the channel. If the label is Neg, then the channel can receive an integer (nothing depends on it), send an integer, and finally send a special end-of-session label (EOS). The case for label Add is analogous.

Listing 5 contains the code for a server of this type. It relies entirely on primitive send and receive operations. The **rcase** operation, which is typical of previous work, decomposes into a standard **recv** operation followed by an ordinary **case** on labels. Moreover, the structure of the code is liberated from the session type. While the standard session type dictates the placement of the **rcase**, the LDST version can examine the tag any time after receiving it. The server code takes advantage of this liberty by pulling the common receive operation for the first argument out of the two branches.

A compatible client (Listing 7) does not have to know about the choice. Its channel argument type is a supertype of the dual of TServer in our calculus (cf. type TClient in Listing 6).

Section 8 shows that any program using binary session types can be expressed with label-dependent session types in a semantics-preserving way. The examples shown in this section give a preview of this embedding.

2.3 Tagged Data and Algebraic Datatypes

Some session calculi support the transmission of tagged data as a primitive [Chen et al. 2017; Scalas and Yoshida 2016; Vasconcelos and Tokoro 1993]. In these works, operations of the form $c!Node(42)$ are used to send a *Node*-tagged message with payload 42 on channel c, effectively combining a **select** operation with the sending some extra data. The corresponding receiving construct dispatches on the tag, as in **rcase**, and also extracts the payload into variables. This construction is akin to packaging tags with data and pattern matching as known from algebraic datatypes in functional programming languages.

```
type NodeC =
  !(tag : {Empty, Node}).
  case tag of { Empty: !Unit. End
              , Node: !Int. End }

sendNode : Node → NodeC → Unit
sendNode n c =
  let (tag, v) = n
      c = send c tag
  in  send c v

recvNode : dualof NodeC → Node
recvNode c =
  let (tag, c) = recv c
      (val, c) = recv c
  in  (tag, val)
```

Listing 8. Sending and receiving nodes

```
type SumServer =
  ?(n:Nat).rec n (!Int.End) [α]?Int
      .α

sum : SumServer → End
sum c =
  let (n,c) = recv c in
  rec n {
    Z: λ(m:Int).λ(c:!Int.End).
       send c m,
    S(_) with [α](y:Int → α → End):
      λ(m:Int).λ(c:?Int.α).
        let (k,c) = recv c in
        y (k+m) c
  } 0 c
```

Listing 9. Summing a given number of integers

Indeed, such algebraic datatypes can be modeled in the functional sublanguage of LDST using a label-dependent Σ-type. As an example, consider the datatype

```
data Node where { Empty: Node
                , Node : Int → Node }
```

and its LDST representation

```
type Node =
    Σ(tag: {Empty, Node})
    case tag of { Empty: Unit, Node: Int }
```

Sending (receiving) a single value of type Node can be performed on a channel of type NodeC (or its dual) as illustrated with sendNode and recvNode in Listing 8.

Recursive datatypes and session type protocols can be supported by extending LDST with recursive types, which we leave to future work.

2.4 Number-Indexed Protocols

One shortcoming of programming-oriented systems for session types is that they do not support families of indexed protocols, a quite common situation in practice. These protocols have variable-length messages where the first item transmitted gives the number of the subsequent items.

To demonstrate how LDST can deal with such protocols, consider the code in Listing 9 which implements a server that first receives a number n and then expects to receive n further numbers, sums them all up and sends them back to the client. The type of this channel is given by

```
type SumServer = ?(n:Nat). rec n (!Int.End) [α]?Int.α
```

The interesting part is the type of the form **rec** n S [α]R, which denotes a type-level recursor on natural numbers. Its first argument is a number n, the second argument S is used if n is zero, and the third argument [α]R is used when n is non-zero and α abstracts over the recursive use. In this case, the type is equivalent to R where α is replaced by the unwinding of the recursor.

In the example, the types evaluate as follows

- **rec** 0 !**Int**.**End** [α]?**Int**.α ≡ !**Int**.**End**,
- **rec** 1 !**Int**.**End** [α]?**Int**.α ≡ ?**Int**.**rec** 0 !**Int**.**End** [α]?**Int**.α ≡ ?**Int** .! **Int**.**End**, and so on.

The implementation of the server has to use the corresponding recursor at the value level. If we write T n for **rec** n (! **Int**.**End**) [α]?**Int**.α, then the recursor returns a function of type **Int** →T n → **End** and is given an expression of type **Int** →!**Int**.**End** →**End** ≡ **Int** →T 0 →**End** for the case zero. In the successor case, the expression has type **Int** →?**Int**.T n →**End** ≡ **Int** →T (S n) →**End** and the variable y is bound to the function returned by the unwinding of the recursor. This code does not use the predecessor as indicated by the underline in S(_). The type annotations for m, c, and y have to be given to enable type checking.

2.5 Assessment

Moving to the dependent calculus LDST has a number of advantages over a traditional calculus like LAST. It liberates the program structure somewhat from the session type structure and increases expressivity as shown in the preceding subsections.

- In LDST, a label-dependent choice in the type can be deferred in the program. Listing 5 does not type check in LAST because it defers the choice compared to the session type.
- LDST supports first class labels. Listing 8 cannot be written in this generic way in LAST because the functions sendNode and recvNode transfer labels without inspecting them. In LAST the receive operation **rcase** also inspects the label: the code would have to be eta-expanded depending on the label set used in the Node type. The LDST code is resilient against such changes. New variants in Node and NodeC can be processed without rewriting the code.
- LDST supports types and protocols defined by recursion on natural numbers. Listing 9 cannot be written in LAST without major changes in the protocol that make it very inefficient. One would have to change the data stream into a list with intervening labels.

3 BINARY SESSION TYPES

The type structure for a functional calculus with binary session types, known as LAST, adds session types S to the types of an underlying lambda calculus with functions and products (cf. Gay and Vasconcelos [2010]). The examples in Section 2.1 are written in LAST with some syntactic sugar.

The calculus LSST (for Linear Synchronous Session Types) introduced in this section is a slight variation of Gay and Vasconcelos LAST calculus. First, we choose a synchronous semantics for the communication primitives. This choice has no impact on the typing, but greatly simplifies the semantics and proofs. Second, we adopt the linear-logic inspired end markers $\mathbf{End}_!$ and $\mathbf{End}_?$ from Wadler's GV calculus [Wadler 2012]. Unlike GV, the LSST calculus is not free of deadlock.

Figure 1 defines the syntax of types and the notion of subtyping. Function types are annotated with multiplicities (also called *kinds*), $m, n \in \{\mathbf{lin}, \mathbf{un}\}$, that restrict the number of eliminations that may be applied to a value of that type: **lin** denotes a linear value that must be eliminated exactly once, **un** denotes an unrestricted value that may be eliminated as many times as needed. Session types are always linear. The subkinding relation $m \preceq n$ relates multiplicities: if a value offers elimination according to m it may also be eliminated according to n. In particular, an unrestricted value may also serve as a linear value. The predicate $\vdash A : m$ determines the multiplicity of a type.

In session types, the branch labels ℓ are drawn from a denumerable set $\mathcal{L}$ of labels. Overlining indexed by some ℓ indicates an iteration over a finite non-empty set of labels $L \subseteq \mathcal{L}$. The type $!A.S$ indicates sending a value of type A and continuing according to S; $?A.S$ indicates receiving a value of type A and continuing according to S; the type $\oplus\{\overline{\ell : S_\ell}\}$ stands for sending a label $\ell \in L$ and then continuing according to S_ℓ; and $\&\{\overline{\ell : S_\ell}\}$ stands for receiving a label $\ell \in L$ and continuing with the

$$\begin{array}{rrcl} \text{Multiplicities} & m, n & ::= & \mathbf{lin} \mid \mathbf{un} \\ \text{Types} & A, B & ::= & S \mid \mathbf{Unit} \mid A \to_m B \mid A \times B \\ \text{Session types} & S, R & ::= & !A.S \mid ?A.S \mid \oplus\{\overline{\ell : S_\ell}^{\ell \in L}\} \mid \&\{\overline{\ell : S_\ell}^{\ell \in L}\} \mid \mathbf{End}_! \mid \mathbf{End}_? \end{array}$$

Subkinding $\boxed{m \le n}$

$$\mathbf{un} \le \mathbf{lin} \qquad \mathbf{lin} \le \mathbf{lin} \qquad \mathbf{un} \le \mathbf{un}$$

Kinding $\boxed{\vdash A : m}$

$$\vdash \mathbf{Unit} : \mathbf{un} \qquad \vdash A \to_m B : m \qquad \frac{\vdash A : m \qquad \vdash B : m}{\vdash A \times B : m} \qquad \vdash S : \mathbf{lin} \qquad \frac{\vdash A : m \qquad m \le n}{\vdash A : n}$$

Subtyping $\boxed{A \le B}$

$$\mathbf{Unit} \le \mathbf{Unit} \qquad \mathbf{End}_! \le \mathbf{End}_! \qquad \mathbf{End}_? \le \mathbf{End}_?$$

$$\frac{A' \le A \qquad B \le B' \qquad m \le n}{A \to_m B \le A' \to_n B'} \qquad \frac{A \le A' \qquad B \le B'}{A \times B \le A' \times B'} \qquad \frac{A' \le A \qquad S \le S'}{!A.S \le !A'.S'} \qquad \frac{A \le A' \qquad S \le S'}{?A.S \le ?A'.S'}$$

$$\frac{L' \subseteq L \qquad (\forall \ell \in L')\ S_\ell \le S'_\ell}{\oplus\{\overline{\ell : S_\ell}^{\ell \in L}\} \le \oplus\{\overline{\ell : S'_\ell}^{\ell \in L'}\}} \qquad \frac{L \subseteq L' \qquad (\forall \ell \in L)\ S_\ell \le S'_\ell}{\&\{\overline{\ell : S_\ell}^{\ell \in L}\} \le \&\{\overline{\ell : S'_\ell}^{\ell \in L'}\}}$$

Fig. 1. Types and subtyping in LSST

chosen S_ℓ. The session types $\mathbf{End}_!$ and $\mathbf{End}_?$ indicate closing the communication and waiting for the other end to close.

LSST's subtyping is driven by multiplicities (linear values may subsume unrestricted ones) and by varying the number of alternatives in branch and choice types (corresponding to width subtyping of records and variants) [Gay and Hole 2005; Gay and Vasconcelos 2010].

Figure 2 describes the syntax of names, expressions, and processes in LSST. Names include variables, x, y, and channel endpoints, c, d. Expressions comprise names, the unit value, pair and function introduction and elimination (in **lin** and **un** versions), and the standard primitives of LSST. Process expressions are either expression processes, parallel processes, or a channel restriction $(\nu cd)P$ that binds the two channel endpoints c and d in the scope provided by process P.

We refrain from giving the full set of LSST typing rules here. As an example, we give the standard typing rules for sending and receiving data and go over rule GV-SEND for illustration. The **send** operation takes a channel endpoint M of type $!A.S$, which is good to write a value of type A. Then **send** M is a function from A to S, which must be used once (because it is closed over the channel endpoint). We flip the arguments for **send** with respect to other presentations in the literature [Gay and Vasconcelos 2010; Igarashi et al. 2017; Lindley and Morris 2016; Wadler 2012], while aligning with those of Padovani [2017b]. The rule GV-NEW for creating channels prescribes that **new** returns a pair of channel endpoints with dual session types. Alternatively, to obtain a deadlock-free calculus, we could couple channel creation with thread creation as in the cut rule of Wadler [2012] or the fork primitive of Lindley and Morris [2014].

Like other type systems with a mix of linear and unrestricted resources [Cervesato and Pfenning 1996; Kobayashi et al. 1996; Walker 2005], LSST relies on an environment splitting relation $\Gamma = \Gamma_1 \circ \Gamma_2$. As a slight abuse of notation, we sometimes write $\Gamma_1 \circ \Gamma_2$ for some Γ such that $\Gamma = \Gamma_1 \circ \Gamma_2$.

$$
\begin{array}{rrcl}
\text{Names} & z & ::= & x \mid c \\
\text{Expressions} & M, N & ::= & z \mid () \mid \lambda_m x.M \mid M\,N \mid \langle M, N\rangle \mid \mathbf{let}\,\langle x, y\rangle = M\,\mathbf{in}\,N \\
& & & \mid \mathbf{fork}\,M \mid \mathbf{new} \mid \mathbf{send}\,M \mid \mathbf{recv}\,M \\
& & & \mid \mathbf{select}\,\ell \mid \mathbf{rcase}\,M\,\mathbf{of}\,\{\overline{\ell : x.N_\ell}^{\ell \in L}\} \mid \mathbf{close}\,M \mid \mathbf{wait}\,M \\
\text{Processes} & O, P, Q & ::= & \langle M\rangle \mid P \mid Q \mid (\nu cd)P \\
\text{Typing environments} & \Gamma, \Delta & ::= & \cdot \mid \Gamma, z : A
\end{array}
$$

Environment formation, environment split $\boxed{\vdash \Gamma : m}$ $\boxed{\Gamma = \Gamma_1 \circ \Gamma_2}$

$$
\vdash \cdot : m \qquad \frac{\vdash \Gamma : m \qquad \vdash A : m}{\vdash (\Gamma, x : A) : m}
$$

$$
\cdot = \cdot \circ \cdot \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad \vdash A : \mathbf{un}}{(\Gamma, z : A) = (\Gamma_1, z : A) \circ (\Gamma_2, z : A)} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad \vdash A : \mathbf{lin}}{(\Gamma, z : A) = (\Gamma_1, z : A) \circ \Gamma_2}
$$

$$
\frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad \vdash A : \mathbf{lin}}{(\Gamma, z : A) = \Gamma_1 \circ (\Gamma_2, z : A)}
$$

Typing expressions (excerpt) $\boxed{\Gamma \vdash M : A}$

GV-SEND
$$
\frac{\Gamma \vdash M : !A.S}{\Gamma \vdash \mathbf{send}\,M : A \to_{\mathbf{lin}} S}
$$

GV-RECV
$$
\frac{\Gamma \vdash M : ?A.S}{\Gamma \vdash \mathbf{recv}\,M : A \times S}
$$

GV-SELECT
$$
\frac{\vdash \Gamma : \mathbf{un} \qquad \ell' \in L}{\Gamma \vdash \mathbf{select}\,\ell' : \oplus\{\overline{\ell : S_\ell}^{\ell \in L}\} \to_{\mathbf{lin}} S_{\ell'}}
$$

GV-RCASE
$$
\frac{\Gamma_1 \vdash M : \&\{\overline{\ell : S_\ell}\}^{\ell \in L} \qquad (\forall \ell \in L)\Gamma_2, x : S_\ell \vdash M_\ell : A}{\Gamma_1 \circ \Gamma_2 \vdash \mathbf{rcase}\,M\,\mathbf{of}\,\{\overline{\ell : x.M_\ell}^{\ell \in L}\} : A}
$$

GV-CLOSE
$$
\frac{\Gamma \vdash M : \mathbf{End}_!}{\Gamma \vdash \mathbf{close}\,M : \mathbf{Unit}}
$$

GV-WAIT
$$
\frac{\Gamma \vdash M : \mathbf{End}_?}{\Gamma \vdash \mathbf{wait}\,M : \mathbf{Unit}}
$$

GV-NEW
$$
\frac{\vdash \Gamma : \mathbf{un}}{\Gamma \vdash \mathbf{new} : S \times (S)^\perp}
$$

Fig. 2. Expressions, processes, and typing in LSST

The reduction relation for the LSST language is in Figure 3. It introduces values V, W, which comprise the usual lambda calculus variety, a communication channel endpoint c, a partially applied send operation $\mathbf{send}\,v$, and a select operation with a label $\mathbf{select}\,\ell$. Evaluation contexts $\mathcal{E}, \mathcal{F}$ formalize a left-to-right call-by-value evaluation order. Unlike in LAST, communication in LSST is synchronous and we add the Rl-Close reduction.

We refrain from defining expression reduction; instead we refer the reader to Gay and Vasconcelos [2010]. But we fully define process reduction. It relies on *structural congruence*, $P \equiv Q$, a relation that specifies that parallel execution is commutative, associative, and compatible with channel restriction and commutation of channel restriction. We assume the variable convention: for example, in the rule sc-swap-r for commuting restrictions it must be that $\{c, d\} \cap \{c', d'\} = \varnothing$. The rule sc-swap-c that swaps the endpoints simplifies the statement of the reduction relation [Igarashi et al. 2017]. Examining the reduction rules, we observe that each communication reduction first performs a rendezvous to transmit information, but the rules (Rl-Branch) and (Rl-Close) do some extra work.

$$\begin{array}{rrcl} \text{Values} & V, W & ::= & c \mid () \mid \lambda_m x.M \mid \langle V, W\rangle \mid \mathbf{send}\, V \mid \mathbf{select}\, \ell \\ \text{Evaluation contexts} & \mathcal{E}, \mathcal{F} & ::= & \Box \mid \mathcal{E}\, M \mid V\, \mathcal{E} \mid \langle \mathcal{E}, f\rangle \mid \langle V, \mathcal{E}\rangle \mid \mathbf{let}\, \langle x, y\rangle = \mathcal{E}\, \mathbf{in}\, M \\ & & & \mid \mathbf{send}\, \mathcal{E} \mid \mathbf{recv}\, \mathcal{E} \mid \mathbf{rcase}\, \mathcal{E}\, \mathbf{of}\, \{\overline{\ell : x.M}\} \mid \mathbf{close}\, \mathcal{E} \mid \mathbf{wait}\, \mathcal{E} \end{array}$$

Structural congruence $\boxed{P \equiv Q}$

$$\textsc{sc-comm}\ P \mid Q \equiv Q \mid P \qquad \textsc{sc-assoc}\ O \mid (P \mid Q) \equiv (O \mid P) \mid Q \qquad \textsc{sc-gc}\ P \mid \langle()\rangle \equiv P$$

$$\textsc{sc-extrusion}\ ((\nu cd)P) \mid Q \equiv (\nu cd)(P \mid Q) \qquad \textsc{sc-swap-c}\ (\nu cd)P \equiv (\nu dc)P \qquad \textsc{sc-swap-r}\ (\nu cd)(\nu c'd')P \equiv (\nu c'd')(\nu cd)P$$

Process reduction $\boxed{P \longrightarrow Q}$

$$\langle \mathcal{E}[\mathbf{fork}\, M]\rangle \longrightarrow \langle \mathcal{E}[()]\rangle \mid \langle M\rangle \quad (\textsc{Rl-Fork})$$

$$\langle \mathcal{E}[\mathbf{new}]\rangle \longrightarrow (\nu cd)\, \langle \mathcal{E}[\langle c, d\rangle]\rangle \quad (\textsc{Rl-New})$$

$$(\nu cd)\, \langle \mathcal{E}[\mathbf{send}\, c\, V]\rangle \mid \langle \mathcal{F}[\mathbf{recv}\, d]\rangle \longrightarrow (\nu cd)\langle \mathcal{E}[c]\rangle \mid \langle \mathcal{F}[\langle V, d\rangle]\rangle \quad (\textsc{Rl-Com})$$

$$(\nu cd)\langle \mathcal{E}[\mathbf{select}\, \ell' c]\rangle \mid \langle \mathcal{F}[\mathbf{rcase}\, d\, \mathbf{of}\, \{\overline{\ell : x.M_\ell}\}]\rangle \longrightarrow (\nu cd)\langle \mathcal{E}[c]\rangle \mid \langle \mathcal{F}[M_{\ell'}[d/x]]\rangle \quad (\textsc{Rl-Branch})$$

$$(\nu cd)\, \langle \mathcal{E}[\mathbf{close}\, c]\rangle \mid \langle \mathcal{F}[\mathbf{wait}\, d]\rangle \longrightarrow \langle \mathcal{E}[()]\rangle \mid \langle \mathcal{F}[()]\rangle \quad (\textsc{Rl-Close})$$

$$\textsc{Rl-Ctx-Exp}\ \frac{M \longrightarrow N}{\langle M\rangle \longrightarrow \langle N\rangle} \qquad \textsc{Rl-Ctx-Par}\ \frac{P \longrightarrow Q}{O \mid P \longrightarrow O \mid Q} \qquad \textsc{Rl-Ctx-Res}\ \frac{P \longrightarrow Q}{(\nu cd)P \longrightarrow (\nu cd)Q} \qquad \textsc{Rl-Cong}\ \frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q' \equiv Q}{P \longrightarrow Q}$$

Fig. 3. Reduction in LSST

Part of the motivation for this work comes from trying to disentangle the extra work from the pure communication.

4 THE LABEL-DEPENDENT SESSION CALCULUS

We propose LDST, a new calculus for functional sessions. Compared to LSST, LDST introduces types that depend on labels and restricts the communication instructions to the fundamental send and receive operations. The example in Section 2.2 hints that every LSST program can be expressed in LDST, a claim formally stated and proved in Section 8.

The dynamics of LDST are simpler than LSST's, but its statics are more involved. They build on a range of earlier work, most notably trellys [Casinghino et al. 2014; Sjöberg et al. 2012] and F* [Swamy et al. 2013], to formalize a flexible dependently-typed system based on call-by-value execution augmented with linear types.

Figure 4 describes LDST's values, types, and some auxiliary operations on type environments. Kinds are as in LSST: **lin** for linear (single use) types and **un** for unrestricted types; and unrestricted values can also be used linearly.

Values comprise the usual lambda calculus values and **send** V as in Section 3. Recall from Figure 2 that z stands for a variable x or a channel end c. Variables are included in the set of values as they can only be bound to values as customary when reasoning with open expressions.

Types of the calculus comprise session types; the unit type; the label type $\{\ell_1, \ldots, \ell_n\}$, for $n > 0$, inhabited by the labels $\ell_1, \ldots, \ell_n$—for brevity, we let L range over finite non-empty sets of labels; the equality type $V = W$ inhabited by evidence that the value V is equal to value W; the dependent function and product types $\Pi_m(x : A)B$ and $\Sigma(x : A)B$ of multiplicity m. Session types comprise

$$\text{Values } V,W ::= z \mid \ell \mid () \mid \lambda_m(x:A).M \mid \langle x:A = V, W\rangle \mid \mathbf{send}\, V$$
$$\text{Types } A,B ::= S \mid \mathbf{Unit} \mid L \mid V = W \mid \mathbf{case}\, V\, \mathbf{of}\, \{\overline{\ell : A_\ell}\} \mid \Pi_m(x:A)B \mid \Sigma(x:A)B$$
$$\text{Session Types } S,R ::= \mathbf{End} \mid \mathbf{case}\, V\, \mathbf{of}\, \{\overline{\ell : S_\ell}\} \mid !(x:A)S \mid ?(x:A)S$$

Environment split $\boxed{\Gamma = \Gamma_1 \circ \Gamma_2}$

$$\cdot = \cdot \circ \cdot \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Gamma_1 \vdash A : \mathbf{un} \qquad \Gamma_2 \vdash A : \mathbf{un}}{(\Gamma, z:A) = (\Gamma_1, z:A) \circ (\Gamma_2, z:A)} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Gamma_1 \vdash A : \mathbf{lin}}{(\Gamma, z:A) = (\Gamma_1, z:A) \circ \Gamma_2}$$
$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Gamma_2 \vdash A : \mathbf{lin}}{(\Gamma, z:A) = \Gamma_1 \circ (\Gamma_2, z:A)}$$

Conditional extension, the unrestricted part of an env. $\boxed{\Gamma \triangleleft x:A = \Delta}$ $\boxed{\Gamma^{\mathbf{un}} = \Delta}$

$$\frac{\Gamma^{\mathbf{un}} \vdash A : \mathbf{lin}}{\Gamma \triangleleft x:A = \Gamma} \qquad \frac{\Gamma^{\mathbf{un}} \vdash A : \mathbf{un}}{\Gamma \triangleleft x:A = \Gamma, x:A} \qquad \cdot^{\mathbf{un}} = \cdot \qquad (\Gamma, z:A)^{\mathbf{un}} = (\Gamma^{\mathbf{un}}) \triangleleft z:A$$

Session type duality $\boxed{S^\perp = R}$

$$(!(x:A)S)^\perp = ?(x:A)S^\perp \qquad (?(x:A)S)^\perp = !(x:A)S^\perp$$
$$(\mathbf{case}\, V\, \mathbf{of}\, \{\overline{\ell : S_\ell}\})^\perp = \mathbf{case}\, V\, \mathbf{of}\, \{\overline{\ell : S_\ell{}^\perp}\} \qquad \mathbf{End}^\perp = \mathbf{End}$$

Fig. 4. Values and types in LDST

End to signify the end of a session; the dependent session types $!(x:A)S$ and $?(x:A)S$ for endpoints that send or receive a value of type A and continue as session type S, which may depend on x. The type $\mathbf{case}\, V\, \mathbf{of}\, \{\overline{\ell : A_\ell}\}$ indicates large elimination for labels and it may occur in both types and session types.

The basic operations on type environments are inherited from LSST: environment formation $\vdash \Gamma : n$ and splitting $\Gamma = \Gamma_1 \circ \Gamma_2$. Both rely on kinding (they are mutually recursive as expected in a dependently typed calculus) and we present a revised definition of kinding (type formation) shortly. Environment formation and environment split for LDST are both adapted from LAST (Figure 2). In the case of formation, premise $\vdash A : m$ becomes $\Gamma \vdash A : m$ to reflect the new type formation rules (in Figure 5). For environment split we require type A to be well formed in the relevant contexts, so that $\Gamma_1, z:A$ and $\Gamma_2, z:A$ both become well formed contexts.

The new operations are conditional extension $\Gamma \triangleleft x:A = \Delta$ and projecting the unrestricted part of an environment $\Gamma^{\mathbf{un}} = \Delta$. The conditional extension $\Gamma \triangleleft x:A$ only includes the binding $x:A$ in the resulting environment if A is unrestricted. In the upcoming type formation rules, this mechanism is used to keep linear values out of the environment so that any dependency on linear objects is ruled out.

Conditional extension is used in the formation rule for all dependent types. As an example, take the function type $\Gamma \vdash \Pi_m(x:A)B : m$. Here, we do not wish to force type A to be unrestricted. Rather, we wish to express that B can depend on $x:A$ iff A is unrestricted. To this end, the premise uses the conditional extension to check B as in $\Gamma \triangleleft x:A \vdash B : n$. Right now, this setup is more general than strictly needed because we can only compute with labels in types, that is, we need $A = L$ and B can at most contain a **case** on x.

Type formation $\boxed{\Gamma \vdash A : m}$

$$\textsc{Equality-F}\ \frac{\Gamma \vdash V : A \qquad \Gamma \vdash W : A \qquad \Gamma \vdash A : m}{\Gamma \vdash V = W : m} \qquad \textsc{Unit-F}\ \frac{\vdash \Gamma : \mathbf{un}}{\Gamma \vdash \mathbf{Unit} : \mathbf{un}} \qquad \textsc{End-F}\ \frac{\vdash \Gamma : \mathbf{un}}{\Gamma \vdash \mathbf{End} : \mathbf{un}} \qquad \textsc{Lab-F}\ \frac{\vdash \Gamma : \mathbf{un}}{\Gamma \vdash L : \mathbf{un}}$$

$$\textsc{Lab-E'}\ \frac{\Gamma^{\mathbf{un}} \vdash V : L \qquad (\forall \ell \in L)\ \Gamma \lhd x : V = \ell \vdash A_\ell : m}{\Gamma \vdash \mathbf{case}\ V\ \mathbf{of}\ \{\overline{\ell : A_\ell}^{\ell \in L}\} : m} \qquad \textsc{Pi-F}\ \frac{\Gamma \lhd x : A \vdash B : n}{\Gamma \vdash \Pi_m(x : A)B : m}$$

$$\textsc{Sigma-F}\ \frac{\Gamma \vdash A : m \qquad \Gamma \lhd x : A \vdash B : m}{\Gamma \vdash \Sigma(x : A)B : m} \qquad \textsc{Ssn-Out-F}\ \frac{\Gamma \lhd x : A \vdash S : \mathbf{lin}}{\Gamma \vdash !(x : A)S : \mathbf{lin}} \qquad \textsc{Ssn-In-F}\ \frac{\Gamma \lhd x : A \vdash S : \mathbf{lin}}{\Gamma \vdash ?(x : A)S : \mathbf{lin}}$$

$$\textsc{Sub-Kind}\ \frac{\Gamma \vdash A : m \qquad m \le n}{\Gamma \vdash A : n}$$

Fig. 5. Type formation in LDST

The unrestricted part of an environment is used when switching from expression formation to type formation or subtyping. As $\Gamma = \Gamma \circ \Gamma^{\mathbf{un}}$, it is ok to use an environment and its unrestricted part side by side.

The *dual of a session type*, $S^\perp$, is also defined in Figure 4 and has the same structure as the original type S, but swaps the direction of communication. Duality is an inductive metafunction on session types and is involutory: $(S^\perp)^\perp = S$.

Type formation is defined in Figure 5. As types do not depend on linear values, all type environments involved in the type formation judgment $\Gamma \vdash A : m$ are unrestricted (i.e., $\vdash \Gamma : \mathbf{un}$). Equality types are unrestricted types constructed from a value of label type and a concrete label (Equality-F). This rule refers to the upcoming typing judgment. The **Unit** type and the **End** type both have kind **un** (Unit-F, End-F). The label type is an index type for any non-empty, finite set of labels (Lab-F). Label elimination Lab-E' for value V constructs a witness for the equality type $V = \ell$ in the branch for label ℓ to model dependent matching [Casinghino et al. 2014]. Formation of the type $\Pi_m(x : A)B$ showcases conditional extension (Pi-F). The type A may be linear or unrestricted. In the former case, the binding for x must not be used in B, in the latter case, it may. The conditional extension expresses this desire precisely. The kind of the Π-type is determined by its annotation m. The same rationale applies to the formation rule Sigma-F of the type $\Sigma(x : A)B$, but we need to check the kind of A explicitly to make sure it matches the kind of B. Kind subsumption Sub-Kind enables products with components that have different kinds. Rules Ssn-Out-F and Ssn-In-F manage dependency just like functions and products.

Figure 6 describes type conversion and subtyping. Type conversion $\Gamma \vdash A \equiv B : m$ specifies that types A and B of kind m are equal up to substitutions that can be justified by equations in Γ, beta and eta conversion of cases. Eta conversion enables commuting conversions that move common (session) type prefixes in and out of **case** types. Conversion is closed under reflexivity, symmetry, and transitivity.

As an example for type conversion in action, consider typing a function that returns values of different primitive types **Int** and **String** depending on its input.

```
λ (b : {True, False}) case b of { True: 0, False: "foo" }
```

Type conversion $\boxed{\Gamma \vdash A \equiv B : m}$

$$\textsc{Conv-Subst}\ \frac{\Gamma \vdash y : V = W \qquad \Gamma \vdash V : L \qquad (\forall \ell \in L)\ \Gamma \triangleleft z : V = \ell \vdash A_\ell : m}{\Gamma \vdash \mathbf{case}\, V\, \mathbf{of}\, \{\overline{\ell : A_\ell}^{\ell \in L}\} \equiv \mathbf{case}\, W\, \mathbf{of}\, \{\overline{\ell : A_\ell}^{\ell \in L}\} : m}$$

$$\textsc{Conv-Beta}\ \frac{(\forall \ell \in L)\ \Gamma \vdash A_\ell : m \qquad \ell' \in L}{\Gamma \vdash \mathbf{case}\, \ell'\, \mathbf{of}\, \{\overline{\ell : A_\ell}^{\ell \in L}\} \equiv A_{\ell'} : m} \qquad \textsc{Conv-Eta}\ \frac{\Gamma \vdash A : m \qquad \Gamma \vdash x : L}{\Gamma \vdash A \equiv \mathbf{case}\, x\, \mathbf{of}\, \{\overline{\ell : A}^{\ell \in L}\} : m} \qquad \textsc{Conv-Refl}\ \frac{\Gamma \vdash A : m}{\Gamma \vdash A \equiv A : m}$$

$$\textsc{Conv-Sym}\ \frac{\Gamma \vdash A \equiv B : m}{\Gamma \vdash B \equiv A : m} \qquad \textsc{Conv-Trans}\ \frac{\Gamma \vdash A \equiv B : m \qquad \Gamma \vdash B \equiv C}{\Gamma \vdash A \equiv C : m}$$

Subtyping $\boxed{\Gamma \vdash A \leq B : m}$

$$\textsc{Sub-Conv}\ \frac{\Gamma \vdash A \equiv B : m}{\Gamma \vdash A \leq B : m} \qquad \textsc{Sub-Lab}\ \frac{\vdash \Gamma : \mathbf{un} \qquad L \subseteq L'}{\Gamma \vdash L \leq L' : \mathbf{un}} \qquad \textsc{Sub-Trans}\ \frac{\Gamma \vdash A \leq B : m \qquad \Gamma \vdash B \leq C : m}{\Gamma \vdash A \leq C : m}$$

$$\textsc{Sub-Sub}\ \frac{\Gamma \vdash A \leq B : m \qquad m \preceq n}{\Gamma \vdash A \leq B : n} \qquad \textsc{Sub-Pi}\ \frac{\Gamma \vdash A' \leq A : m_A \qquad \Gamma \triangleleft x : A' \vdash B \leq B' : m_B \qquad m \preceq n}{\Gamma \vdash \Pi_m (x : A) B \leq \Pi_n (x : A') B' : n}$$

$$\textsc{Sub-Sigma}\ \frac{\Gamma \vdash A \leq A' : m \qquad \Gamma \triangleleft x : A \vdash B \leq B' : m}{\Gamma \vdash \Sigma (x : A) B \leq \Sigma (x : A') B' : m} \qquad \textsc{Sub-Send}\ \frac{\Gamma \vdash A' \leq A : m \qquad \Gamma \triangleleft x : A' \vdash S \leq S' : \mathbf{lin}}{\Gamma \vdash !(x : A) S \leq !(x : A') S' : \mathbf{lin}}$$

$$\textsc{Sub-Recv}\ \frac{\Gamma \vdash A \leq A' : m \qquad \Gamma \triangleleft x : A \vdash S \leq S' : \mathbf{lin}}{\Gamma \vdash ?(x : A) S \leq ?(x : A') S' : \mathbf{lin}}$$

$$\textsc{Sub-Case}\ \frac{\begin{array}{l}(\forall \ell \in L \setminus L')\ \Gamma \setminus x \triangleleft x : (L \setminus L') \triangleleft y : x = \ell \vdash A_\ell : m \\ (\forall \ell \in L' \setminus L)\ \Gamma \setminus x \triangleleft x : (L' \setminus L) \triangleleft y : x = \ell \vdash A'_\ell : m \\ \Gamma^{\mathbf{un}} \vdash x : L \cap L' \qquad (\forall \ell \in L \cap L')\ \Gamma \triangleleft y : x = \ell \vdash A_\ell \leq A'_\ell : m\end{array}}{\Gamma \vdash \mathbf{case}\, x\, \mathbf{of}\, \{\overline{\ell : A_\ell}^{\ell \in L}\} \leq \mathbf{case}\, x\, \mathbf{of}\, \{\overline{\ell : A'_\ell}^{\ell \in L'}\} : m}$$

Fig. 6. Type conversion and subtyping in LDST

The **True** branch typechecks with `0 : Int` whereas the **False** branch typechecks with `"foo" : String`. Thanks to the Conv-Beta rule, we can expand the type of the **True** branch to `0 : case True of { True: Int, False: String}` and in the **False** branch to `"foo" : case False of { True: Int, False: String}`. According to the upcoming **case** elimination rule Lab-E, each branch for the **case** b adopts the equation of the respective branch as in `b = True` or `b = False`. Hence, the substitution rule Conv-Subst applies to obtain the type `case b of { True: Int, False: String}` for both branches and thus for the entire case expression.

The rule Conv-Eta is needed for typechecking examples like the code in Listing 5. After the first **recv** operation in line 3, the type of `c` is `case l of { Neg: ?Int.NegType, Add: ?Int.AddType }`, but the next operation is **recv** `c`. The trick is to first beta-expand the continuation types NegType and AddType to `CType = case l of { Neg: NegType, Add: Addtype }` in both branches using Conv-Beta

as in the preceding example. The resulting converted type of c now reads **case** ! **of** { Neg: ?**Int**. CType, Add: ?**Int**.CType}, which is clearly convertible to ?**Int**.CType using Conv-Eta. Hence, **recv** c typechecks and returns a channel end of type CType!

Subtyping, also in Figure 6, is generated by conversion (rule Sub-Conv), subsetting of label types (rule Sub-Lab), and closed under transitivity (Sub-Trans), subkinding (Sub-Sub), function and product types (Sub-Pi, Sub-Sigma), as well as session send and receive types (Sub-Send, Sub-Recv).

Subtyping of Π- and Σ-types extends the definitions of Aspinall and Compagnoni [2001]. The novel parts are the conditional binding for the $(x : A)$ part as discussed for the formation rules and the additional constraints on the multiplicities. Sub-Send (Sub-Recv) is a simplified variant of Sub-Pi (Sub-Sigma, respectively).

The rule Sub-Case deserves special attention. Intended to derive the premises for $B \leq B'$ in the rules Sub-Pi, Sub-Sigma, Sub-Send, and Sub-Recv, it deals with the typical case that a function has type $\Pi(x : L)\mathbf{case}\, x\, \mathbf{of}\, \{\overline{\ell : B_\ell}^{\ell \in L}\}$ and we need to determine whether this type is a subtype of $\Pi(x : L')\mathbf{case}\, x\, \mathbf{of}\, \{\overline{\ell : B'_\ell}^{\ell \in L'}\}$. In this case, $L' \subseteq L$ is required and we adopt the assumption $x : L'$ to prove the subtyping judgment on the case types. For the corresponding product types, however, $L \subseteq L'$ is required and the assumption for the case expression reads $x : L$. Both cases are covered by the assumption $x : L \cap L'$ which is the premise in the Sub-Case rule.

In principle, subtyping is not required for LDST to work. However, it is included for two reasons. First, it enables us to establish a tight correspondence with the LSST calculus which features subtyping (cf. Section 8). Second, if we elided subtyping it would be necessary to define a type equivalence relation, say, $\approx$ by a ruleset analogous to the one in Figure 6, where all occurrences of $\leq$ would be replaced by $\approx$ and the comparisons between label sets would change from $\subseteq$ to $=$ (and the same holds for algorithmic subtyping vs. algorithmic type equivalence in Section 7). Hence, the system without subtyping would not be simpler than the one presented.

Figure 7 defines the expressions of LDST, most of which are taken from LSST. In a dependent pair $\langle x : A = M, N\rangle$, the first component M is bound to a variable x which may be used in the second component. The expression **new** creates a new channel of type S and returns a pair of channel endpoints, one of type S and the other of type $S^\perp$. The expressions **send** M and **recv** M have the same operational behavior as in LSST.

Figure 7 also contains the inference rules for expression typing. Most rules are standard, so we only highlight a few specific rules. Rule Lab-E is the expression-level counterpart of the same-named rule at the typing level (Figure 5). It characterizes a dependent case elimination on a label type. In each branch it pushes an equation, $V = \ell$, on the typing environment, which can be exploited in the type derivation for the branch.

Manipulation of Π-types is largely standard (Pi-I). Well-formedness of the Π-type follows from the agreement lemma, as for all other type constructors. Elimination for Π-types is limited to well-formed return types: if the function type depends on x, then the argument must be a value.

Manipulation of Σ-types is similarly restricted to dependency on unrestricted values. Sigma-I introduces a pair, which binds the first component to a variable that can be used in the second component. It behaves like a dependent record. If the first component V is linear, then x can be used in N, but it cannot influence its type due to the well-formedness assumption of the Σ type.

The rule Sigma-G is a refined elimination rule that enables checking the second component of a product repeatedly with all possible assumptions about the label in the first component. It performs a local eta expansion to increase the precision of typing.

As an example for a use of Sigma-G consider the code in Listing 8. If we naively typecheck the product elimination **let** (tag, v) = n in the definition of sendNode, then tag : {Empty, Node} and v : **case** tag **of** {Empty: **Unit**, Node: **Int**}. Sending the tag in the next line updates the type of the

Expressions

$$M, N ::= V \mid \mathbf{case}\, V\, \mathbf{of}\, \{\overline{\ell : N_\ell}\} \mid M\,N \mid \langle x : A = V, N\rangle \mid \mathbf{let}\,\langle x, y\rangle = M\,\mathbf{in}\,N \mid \mathbf{new} \mid \mathbf{fork}\,M \mid \mathbf{send}\,M \mid \mathbf{recv}\,M$$

Expression formation $\boxed{\Gamma \vdash M : A}$

$$\textsc{Sub-Type}\ \frac{\Gamma \vdash M : A \qquad \Gamma^{\mathbf{un}} \vdash A \leq B : m}{\Gamma \vdash M : B} \qquad \textsc{Name}\ \frac{\vdash \Gamma_1 \lhd z : A, \Gamma_2 : \mathbf{un}}{\Gamma_1, z : A, \Gamma_2 \vdash z : A} \qquad \textsc{Unit-I}\ \frac{\vdash \Gamma : \mathbf{un}}{\Gamma \vdash () : \mathbf{Unit}} \qquad \textsc{Lab-I}\ \frac{\vdash \Gamma : \mathbf{un}}{\Gamma \vdash \ell : \{\ell\}}$$

$$\textsc{Lab-E}\ \frac{\Gamma^{\mathbf{un}} \vdash V : L \qquad (\forall \ell \in L)\ \Gamma \lhd y : V = \ell \vdash N_\ell : B}{\Gamma \vdash \mathbf{case}\, V\, \mathbf{of}\, \{\overline{\ell : N_\ell}^{\ell \in L}\} : B} \qquad \textsc{Pi-I}\ \frac{\vdash \Gamma : m \qquad \Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda_m(x : A).M : \Pi_m(x : A)B}$$

$$\textsc{Pi-E}\ \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Gamma_1 \vdash M : \Pi_m(x : A)B \qquad \Gamma_2 \vdash N : A}{\Gamma \vdash M\,N : B[N/x]}$$

$$\textsc{Sigma-I}\ \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Gamma_1 \vdash V : A \qquad \Gamma_2, x : A \lhd z : x = V \vdash N : B}{\Gamma \vdash \langle x = V, N\rangle : \Sigma(x : A)B}$$

$$\textsc{Sigma-E}\ \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Gamma_1 \vdash M : \Sigma(x : A)B \qquad \Gamma_2, x : A, y : B \vdash N : C \qquad x, y \notin fv(C)}{\Gamma \vdash \mathbf{let}\,\langle x, y\rangle = M\,\mathbf{in}\,N : C}$$

$$\textsc{Sigma-G}\ \frac{\begin{array}{c}\Gamma = \Gamma_1 \circ \Gamma_2 \\ \Gamma_1 \vdash M : \Sigma(x : L)B \qquad x \in fv(B) \qquad \Gamma_2, x : L, y : B \vdash \mathbf{case}\, x\, \mathbf{of}\, \{\overline{\ell : N}^{\ell \in L}\} : C \qquad x, y \notin fv(C)\end{array}}{\Gamma \vdash \mathbf{let}\,\langle x, y\rangle = M\,\mathbf{in}\,N : C}$$

$$\textsc{Fork}\ \frac{\Gamma \vdash M : \mathbf{Unit}}{\Gamma \vdash \mathbf{fork}\,M : \mathbf{Unit}} \qquad \textsc{Ssn-I}\ \frac{\Gamma^{\mathbf{un}} \vdash S : \mathbf{lin}}{\Gamma \vdash \mathbf{new} : \Sigma(x : S)S^{\perp}} \qquad \textsc{Ssn-Send-E}\ \frac{\Gamma \vdash M : !(x : A)S}{\Gamma \vdash \mathbf{send}\,M : \Pi_{\mathbf{lin}}(x : A)S}$$

$$\textsc{Ssn-Recv-E}\ \frac{\Gamma^{\mathbf{un}} \vdash A : m \qquad \Gamma \vdash M : ?(x : A)S}{\Gamma \vdash \mathbf{recv}\,M : \Sigma(x : A)S}$$

Fig. 7. Expression formation in LDST

channel end to `c : case tag of {Empty: !Unit, Node: !Int}`. But now the typecheck for the final **send** operation fails because the value of the tag is unknown.

The Sigma-G rule prevents this issue. When eliminating a product on a label type as in `let (tag, v) = n`, the rule checks the body of the **let** for each possible value of tag. The rule expresses this repeated check by a premise that checks a case expression on the first component, tag, which replicates the body of the **let** in all branches (i.e., the body is eta-expanded). As the Lab-E rule for the case adopts a different equation `tag = ...` for each branch, all ramifications are typechecked exhaustively. In the above example, the types for `v` and `c` could both beta-reduce on the known tag and thus unblock the typechecking for the **send** operation.

Evaluation contexts $\quad \mathcal{E}, \mathcal{F} ::= \Box \mid \mathbf{case}\,\mathcal{E}\,\mathbf{of}\,\{\overline{\ell : N_\ell}\} \mid \mathcal{E}\,N \mid V\,\mathcal{E}$
$\quad \mid \langle x : A = V, \mathcal{E}\rangle \mid \mathbf{let}\,\langle x, y\rangle = \mathcal{E}\,\mathbf{in}\,N \mid \mathbf{send}\,\mathcal{E} \mid \mathbf{recv}\,\mathcal{E}$

Expression reduction $\boxed{M \longrightarrow N}$

Rl-Case
$$\frac{\ell' \in L}{\mathbf{case}\,\ell'\,\mathbf{of}\,\{\overline{\ell : M_\ell}^{\ell \in L}\} \longrightarrow M_{\ell'}}$$

Rl-Betav
$$\frac{}{(\lambda_m(x : A).M)\,V \longrightarrow M[V/x]}$$

Rl-Prod-Elim
$$\frac{}{\mathbf{let}\,\langle x, y\rangle = \langle x : A = V, W\rangle\,\mathbf{in}\,M \longrightarrow M[W/y][V/x]}$$

Rl-ctx-exp
$$\frac{M \longrightarrow N}{\mathcal{E}[M] \longrightarrow \mathcal{E}[N]}$$

Process reduction $\boxed{P \longrightarrow Q}$

$$\langle \mathcal{E}[\mathbf{new}]\rangle \longrightarrow (\nu cd)\,\langle \mathcal{E}[\langle x = c, d\rangle]\rangle \qquad \text{(Rl-New)}$$

$$(\nu cd)\,\langle \mathcal{E}[\mathbf{send}\,c\,V]\rangle \mid \langle \mathcal{F}[\mathbf{recv}\,d]\rangle \longrightarrow (\nu cd)\langle \mathcal{E}[c]\rangle \mid \langle \mathcal{F}[\langle x = V, d\rangle]\rangle \qquad \text{(Rl-Com)}$$

(Plus rule (Rl-Fork), the context and the structural congruence rules from Figure 3)

Fig. 8. Reduction in LDST

The last block of rules in Figure 7 governs the typing of the session operations. The **new** expression returns a linear pair of session endpoints where the types are duals of one another. The send operation turns a channel which is ready to send into a single-use dependent function that returns the depleted channel (Ssn-Send-E). The receive operation turns a channel which is ready to receive into a linear dependent pair of the received value and the depleted channel (Ssn-Recv-E).

Process typing is standard (cf. [Gay and Vasconcelos 2010; Vasconcelos 2012]).

Figure 8 defines call-by-value reduction in LDST. Evaluation contexts are standard. Given all that, the dynamics of LDST is pleasingly simple: it is roughly the dynamics of LSST with a few rules removed. Expression reduction comprises a case rule for labels, beta-value reduction, decomposition of products, and lifting over evaluation contexts.

Process reduction gets simplified to a subset of three base cases from five in related work [Gay and Vasconcelos 2010; Igarashi et al. 2017]. From Figure 3, only one (out of three) communication rule remains (rules Rl-Branch and Rl-Close are not part of LDST). Rules (Rl-New) and (Rl-Com) behave as before, but on dependent pairs.

5 NATURAL NUMBERS AND THE RECURSOR

The infrastructure developed in the previous sections is easily amenable to extensions. In this section we report on the support for natural numbers and a type recursor inspired by Gödel's system **T** (cf. Harper [2016]). The required extensions are in Figure 9.

Newly introduced expressions comprise the natural number constructors (Z and $S(V)$) and a *recursor*. A natural number n is encoded as $\overline{n} = S(\ldots S(Z))$, where the successor constructor is applied $n \geq 0$ times to the zero constructor. An expression of the form $\mathbf{rec}\,V\,M\,x.y.N$ represents the V-iteration of the transformation $\lambda x.\lambda y.N$ starting from M. The bound variable x represents the predecessor and the bound variable y the result of x-iteration. Its behaviour is clearly captured by the expression reduction rules in the figure: the recursor evaluates to M when V is zero, and to N with the appropriate substitutions for x and y, otherwise.

$$
\begin{array}{rrcl}
\text{Polarities} & p & ::= & \ldots \mid \oplus \mid \ominus \\
\text{Values} & V, W & ::= & \ldots \mid Z \mid S(V) \\
\text{Types} & A, B & ::= & \ldots \mid \alpha_p \mid \mathbf{rec}\, V\, A\, [\alpha]\, B \mid \mathbf{Nat} \\
\text{Session Types} & S, R & ::= & \ldots \mid \alpha_p \mid \mathbf{rec}\, V\, S\, [\alpha]\, R \\
\text{Expressions} & M, N & ::= & \ldots \mid Z \mid S(M) \mid \mathbf{rec}\, V\, M\, x.y.N \\
\text{Typing environment} & \Gamma, \Delta & ::= & \ldots \mid \Gamma, \alpha \\
\text{Evaluation contexts} & \mathcal{E}, \mathcal{F} & ::= & \ldots \mid \mathbf{rec}\, \mathcal{E}\, M\, x.y.N
\end{array}
$$

Environment formation $\boxed{\vdash \Gamma : m}$

$$\frac{\vdash \Gamma : m}{\vdash (\Gamma, \alpha) : m}$$

Type formation $\boxed{\Gamma \vdash A : m}$

$$\textsc{Nat-F}\ \frac{\vdash \Gamma : \mathbf{un}}{\Gamma \vdash \mathbf{Nat} : \mathbf{un}} \qquad \textsc{TVar-F}\ \frac{\vdash \Gamma, \alpha, \Gamma' : \mathbf{un}}{\Gamma, \alpha, \Gamma' \vdash \alpha_p : \mathbf{un}} \qquad \textsc{Rec-F}\ \frac{\Gamma \vdash V : \mathbf{Nat} \qquad \Gamma \vdash A : m \qquad \Gamma, \alpha \vdash B : m}{\Gamma \vdash \mathbf{rec}\, V\, A\, [\alpha]\, B : m}$$

Type conversion $\boxed{\Gamma \vdash A \equiv B : m}$

$$\textsc{Conv-Z}\ \frac{\Gamma \vdash A : m \qquad \Gamma, \alpha \vdash B : m}{\Gamma \vdash \mathbf{rec}\, Z\, A\, [\alpha]\, B \equiv A : m} \qquad \textsc{Conv-S}\ \frac{\Gamma \vdash V : \mathbf{Nat} \qquad \Gamma \vdash A : m \qquad \Gamma, \alpha \vdash B : m}{\Gamma \vdash \mathbf{rec}\, S(V)\, A\, [\alpha]\, B \equiv B[\mathbf{rec}\, V\, A\, [\alpha]\, B/\alpha] : m}$$

Subtyping $\boxed{\Gamma \vdash A \leq B : m}$

$$\textsc{Sub-Rec}\ \frac{\Gamma \vdash V : \mathbf{Nat} \qquad \Gamma \vdash A \leq A' : m \qquad \Gamma, \alpha \vdash B \leq B' : m}{\Gamma \vdash \mathbf{rec}\, V\, A\, [\alpha]\, B \leq \mathbf{rec}\, V\, A'\, [\alpha]\, B' : m} \qquad \textsc{Sub-TVar}\ \frac{\Gamma \vdash \alpha_p : m}{\Gamma \vdash \alpha_p \leq \alpha_p : m}$$

Session type duality $\boxed{S^\perp = R}$

$$(\mathbf{rec}\, V\, S\, [\alpha]\, R)^\perp = \mathbf{rec}\, V\, S^\perp\, [\alpha]\, R^\perp[\alpha_\ominus/\alpha_\oplus, \alpha_\oplus/\alpha_\ominus] \qquad {\alpha_\oplus}^\perp = \alpha_\ominus \qquad {\alpha_\ominus}^\perp = \alpha_\oplus$$

Expression formation $\boxed{\Gamma \vdash M : A}$

$$\textsc{Nat-I-Z}\ \frac{\vdash \Gamma : \mathbf{un}}{\Gamma \vdash Z : \mathbf{Nat}} \qquad \textsc{Nat-I-S}\ \frac{\Gamma \vdash M : \mathbf{Nat}}{\Gamma \vdash S(M) : \mathbf{Nat}}$$

$$\textsc{Nat-E}\ \frac{\Gamma^{\mathbf{un}} \vdash V : \mathbf{Nat} \qquad \Gamma \lhd z : V = Z \vdash M : A[V/x] \qquad \Gamma^{\mathbf{un}}, x : \mathbf{Nat}, y{:}\, A \lhd z : V = S(x) \vdash N : A[V/x]}{\Gamma \vdash \mathbf{rec}\, V\, M\, x.y.N : A[V/x]}$$

Expression reduction $\boxed{M \longrightarrow N}$

$$\textsc{RL-Z}\ \mathbf{rec}\, Z\, M\, x.y.N \longrightarrow M \qquad \textsc{RL-S}\ \mathbf{rec}\, S(V)\, M\, x.y.N \longrightarrow N[V/x][\mathbf{rec}\, V\, M\, x.y.N/y]$$

Fig. 9. Extensions for natural numbers and recursor

Types now incorporate type variables α, β of kind **un**, the type **Nat** of natural numbers, and a *type recursor*. The type formation rules for type variables and natural numbers should be self-explanatory. The rule for the type recursor, $\mathbf{rec}\, V\, A\, [\alpha]\, B$, requires V to be a natural number and types A and B to be of the same kind m. The recursor variable α may appear free in B, thus accounting for the recursive behaviour of the recursor. For example, if n is a natural number, then type $\mathbf{rec}\, \overline{n}\, (!\mathbf{Int})\mathbf{End}\, [\alpha]\, (?\mathbf{Int})\alpha$ intuitively represents the type $(?\mathbf{Int}) \ldots (?\mathbf{Int})(!\mathbf{Int})\mathbf{End}$ composed of n copies of $?\mathbf{Int}$ and terminated by $(!\mathbf{Int})\mathbf{End}$. As before, we introduce a type recursor for types and for session types. For natural numbers, we need two new instances of the equality type, $V = Z$ and $V = S(W)$, which fit in with the previously defined rule Equality-F in Figure 5.

The rules for type conversion should be easy to understand based on those for expressions: a type $\mathbf{rec}\, V\, A\, [\alpha]\, B$ may be converted to A when V is zero and to B (with the appropriate substitution), otherwise. A third rule (not shown) allows replacing an expression-variable x by a natural number V when an entry $x = V$ can be found in the context (analogous to rule Conv-Subst in Figure 6).

Now for duality and subtyping. Defining the dual of the recursor is subtle and we adopt an approach inspired by Lindley and Morris [2016]'s treatment of general recursive types. Type variables are adorned with a polarity $p \in \{\oplus, \ominus\}$. The polarity "remembers" whether the variable α_p stands for the unrolled recursion ($p = \oplus$) or for its dual ($p = \ominus$). To dualize the recursor $\mathbf{rec}\, V\, S\, [\alpha]\, R$ we first apply the usual dual to S and R. When the transformation reaches a variable α_p in R, it flips its polarity. Next, we swap the polarities of all occurrences of the recursion variable in $R^\perp$. With this definition, duality is an involution on session types. One caveat is that unrolling the recursion into a negative variable (cf. Conv-S) will substitute the *dual* of the type for the variable.

The definition of subtyping for the recursor is fairly restrictive to avoid a coinductive definition. Rule Sub-Rec essentially forces recursive types to synchronize and rule Sub-TVar enforces an invariant treatment of the recursion variables. This choice avoids additional complication with the interplay of variance and the polarity of type variables, while ensuring the basic relation between subtyping and duality ($\Gamma \vdash S^\perp \leq R^\perp : m$ when $\Gamma \vdash R \leq S : m$). A more flexible approach would proceed coinductively; we expect that the solution of Gay and Hole [2005] adaptable to our setting.

Finally, a word on the formation rules for the new expressions. Those for natural numbers Z and $S(V)$ are standard. That for the recursor $\mathbf{rec}\, V\, M\, x.y.N$ requires V to be a natural number and expressions M and N to have the same type $A[V/x]$. The type for M is extracted from a context containing an extra entry stating that V is zero ($z : V = Z$). For N we add bindings for the bound variables x and y, as well as an extra entry stating that V is the successor of x ($z : V = S(x)$). Moreover, whereas M is certainly used once, N may be used arbitrarily often. Hence, we must typecheck N in an unrestricted environment $\Gamma^{\mathbf{un}}$!

6 METATHEORY

The main metatheoretical results for LDST are subject reduction for expressions, typing preservation for processes, and absence of run-time errors. All proofs and auxiliary results may be found in our technical report [Thiemann and Vasconcelos 2019].

Theorem 6.1 (Typing preservation for expressions). *If* $\Gamma \vdash M : A$ *and* $M \longrightarrow N$, *then* $\Gamma \vdash N : A$.

Its proof requires the usual substitution and weakening lemmas along with lemmas about environment splitting.

Theorem 6.2 (Typing preservation for processes). *If* $\Gamma \vdash P$ *and* $P \longrightarrow Q$, *then* $\Gamma \vdash Q$.

Its proof relies on Theorem 6.1 and the adaptation of two results about the manipulation of subderivations by Gay and Vasconcelos [2010].

Algorithmic value conversion $\boxed{\Gamma \vdash V \Rightarrow \ell}$

$$\textsc{AC-Refl}\ \frac{}{\Gamma \vdash \ell \Rightarrow \ell} \qquad \textsc{AC-Assoc}\ \frac{}{\Gamma, y : x = \ell, \Delta \vdash x \Rightarrow \ell}$$

Algorithmic value unfolding $\boxed{\Gamma \vdash A \Downarrow B}$

$$\textsc{A-Unfold-Refl}\ \frac{A \text{ not a case}}{\Gamma \vdash A \Downarrow A} \qquad \textsc{A-Unfold-Case}\ \frac{\Gamma^{\mathbf{un}} \vdash V \Rightarrow \ell' \qquad \Gamma \vdash B_{\ell'} \Downarrow A}{\Gamma \vdash \mathbf{case}\, V\, \mathbf{of}\, \{\overline{\ell : B_\ell}^{\ell \in L}\} \Downarrow A}$$

$$\textsc{A-Unfold-Case1}\ \frac{(\nexists\, \ell)\ \Gamma^{\mathbf{un}} \vdash x \Rightarrow \ell \qquad (\forall \ell \in L)\ \Gamma \vdash A_\ell \Downarrow L'}{\Gamma \vdash \mathbf{case}\, x\, \mathbf{of}\, \{\overline{\ell : A_\ell}^{\ell \in L}\} \Downarrow L'}$$

$$\textsc{A-Unfold-Case2}\ \frac{\begin{array}{c}(\nexists\, \ell)\ \Gamma^{\mathbf{un}} \vdash x \Rightarrow \ell \\ (\forall \ell \in L)\ \Gamma \vdash A_\ell \Downarrow \mathcal{P}[B_\ell] \qquad \mathcal{P} \in \{\Pi_m(y : A)\Box, \Sigma(y : A)\Box, !(y : A)\Box, ?(y : A)\Box\}\end{array}}{\Gamma \vdash \mathbf{case}\, x\, \mathbf{of}\, \{\overline{\ell : A_\ell}^{\ell \in L}\} \Downarrow \mathcal{P}[\mathbf{case}\, x\, \mathbf{of}\, \{\overline{\ell : B_\ell}^{\ell \in L}\}]}$$

Fig. 10. Algorithmic value conversion and unfolding

An absence of runtime errors result for LDST is based on Gay and Vasconcelos [2010]; Honda et al. [1998]; Igarashi et al. [2017]; Vasconcelos [2012]. We start by defining what it means for a process to be an *error*: a) an attempt to match against a non-value label or a label that is not in the expected set (rule Rl-Rec, Figure 8), eliminate a function, a **fix**, a pair or a natural number against the wrong value (rules Rl-Betav, Rl-RecBetav, and Rl-Prod-Elim in Figure 8; rules Rl-Z and RL-S in Figure 9), and b) two processes trying to access the same channel endpoint, or accessing the different endpoints both for reading or for writing (rule Rl-Com, Figure 8).

Theorem 6.3 (Absence of run-time errors). *If* $\vdash P$*, then* P *is not an error.*

7 ALGORITHMIC TYPE CHECKING

Section 4 presents a declarative type system for LDST. In this section, we prove that type checking is decidable. Our algorithm for type checking is based on bidirectional typing [Dunfield and Krishnaswami 2013; Ferreira and Pientka 2014; Pierce and Turner 2000] and comprises several syntax-directed judgments collected in the table below.

$\Gamma \vdash V \Rightarrow W$	Given Γ and V, compute a convertible value W
$\Gamma \vdash A \Downarrow B$	Given Γ and A, compute a type B convertible to A which is not a case
$\Gamma \vdash A \leq B \Rightarrow m$	Given Γ, A, and B, check that A is a subtype of B and synthesize its kind m
$\Gamma \vdash A \leq B \Leftarrow m$	Given Γ, A, B, and m, check that A is a subtype of B at kind m
$\Gamma \vdash A \Rightarrow m$	Given Γ and type A, synthesize its kind m
$\Gamma \vdash A \Leftarrow m$	Given Γ, A, and m, check that A has kind m
$\Gamma \vdash M \Rightarrow A; \Delta$	Given Γ and expression M, synthesize its type A and the environment after Δ
$\Gamma \vdash M \Leftarrow A; \Delta$	Given Γ, M, and type A, check that M has type A and synthesize Δ

The first building block is value conversion and unfolding, two partial functions presented in Figure 10. Value conversion $\Gamma \vdash V \Rightarrow W$ outputs W if V can be converted to some W given the assumptions Γ. There are two rules. AC-Refl applies if V is already a label. AC-Assoc locates an assumption $x = W$ in Γ and returns W. In our system, all equations have the form $x = W$ so that no further rules are needed.

The unfolding judgment $\Gamma \vdash A \Downarrow B$ is needed in the elimination rules for expression typing. Unfolding exposes the top-level type constructor by commuting case types. The exposed type B is

Algorithmic subtyping (synthesis) $\boxed{\Gamma \vdash A \le B \Rightarrow m}$

$$\text{AS-Case-Left1}\quad \frac{\Gamma \vdash V \Rightarrow \ell' \qquad \ell' \in L \qquad \Gamma \vdash A_{\ell'} \le B \Rightarrow m}{\Gamma \vdash \mathbf{case}\, V\, \mathbf{of}\, \{\overline{\ell : A_\ell}^{\ell \in L}\} \le B \Rightarrow m}$$

$$\text{AS-Case-Left2}\quad \frac{\Gamma \vdash x \Downarrow L \qquad L \subseteq L' \qquad (\nexists\, \ell')\ \Gamma \vdash x \Rightarrow \ell' \qquad (\forall \ell \in L)\ \Gamma, y : x = \ell \vdash A_\ell \le B \Rightarrow m_\ell}{\Gamma \vdash \mathbf{case}\, x\, \mathbf{of}\, \{\overline{\ell : A_\ell}^{\ell \in L'}\} \le B \Rightarrow \bigsqcup_{\ell \in L} m_\ell}$$

Fig. 11. Algorithmic subtyping in LDST (excerpt)

convertible to A. If A is not a case type, then no unfolding happens (A-Unfold-Refl). If the left type is a case on a known value V, then recurse on the selected branch (A-Unfold-Case). Otherwise, we try to expose the same top-level type constructor in all branches of the case and commute it on top of the case (A-Unfold-Case2). Rule A-Unfold-Case1 deals with the special case where the branches have label type L'. Unfolding of a case fails if no common top-level constructor exists.

The rules for the algorithmic subtyping judgment $\Gamma \vdash A \le B \Rightarrow m$ mostly follow the declarative subtyping rules in Figure 6. If A is a subtype of B given the assumptions Γ, then the judgment produces the minimal kind m for B. The full set of rules is shown in our technical report [Thiemann and Vasconcelos 2019]. Here, we only discuss the rules AS-Case-Left1 and AS-Case-Left2 (in Figure 11) that deal with case types when they occur on the left (the rules for case on the right mirror the left rules). Rule AS-Case-Left1 invokes algorithmic conversion to find out if V is convertible to a label ℓ under Γ. In that case, the left hand side (case-) beta reduces to A_ℓ so that we synthesize $A_\ell \le B$ recursively. If the attempt to convert the case header to a label fails, then the header must be a variable and its type must unfold to a label type (AS-Case-Left2). Hence, we recursively check that each case branch A_ℓ is a subtype of the right hand type B under the assumption that $x = \ell$.

The algorithmic kinding rules for judgment $\Gamma \vdash A \Rightarrow m$ are straightforward as the type language is a simply-kinded first-order language with subkinding. They may be found in our technical report [Thiemann and Vasconcelos 2019].

The rules for synthesizing a type (Figure 12) define the judgment $\Gamma \vdash M \Rightarrow A; \Delta$. From environment Γ and expression M, the judgment computes M's least type and the remaining type environment Δ. The difference between Γ and Δ indicates which linear resources are used by M: if the binding $x : A \in \Gamma$ with $A : \mathbf{lin}$ is used in M, then Δ does not contain a binding for x. No other changes are possible. Most of the rules are adaptations of the declarative typing rules from Figure 7 to the bidirectional setting. We explain the most relevant.

In rule A-Pi-I we synthesize the kind n of the argument type A. After synthesizing the type of the body with the environment $\Gamma_1, x : A$, the returned environment must have the form $\Gamma_2 \triangleleft x : A$. Thus, we expect that x is used in the body if A is linear. Moreover, if the function's multiplicity m is unrestricted, then no resources in Γ_1 must be used. This constraint is imposed by checking $\Gamma_1 = \Gamma_2$.

Typing an application $M\,N$ (rule A-Pi-E) first synthesizes the type of M. We cannot expect the resulting type C to be a Π type; it may just as well be a case type! Unfolding exposes the top-level non-case type constructor, which we can check to be a Π type and then extract domain and range types A and B. Next, we check that N's type is a subtype of A, and finally that $B[N/x]$ is well-formed.

Rule A-Lab-E1 applies if the conversion judgment $\Gamma_1^{\mathbf{un}} \vdash V \Rightarrow \ell$ figures out that V is convertible to label ℓ. In this case, we only synthesize the type for the branch N_ℓ and return that type.

Algorithmic type checking for expressions (synthesize) $\boxed{\Gamma \vdash M \Rightarrow A; \Delta}$

$$\textsc{A-Name}\ \ \Gamma_1, z : A, \Gamma_2 \vdash z \Rightarrow A; \Gamma_1 \lhd z : A, \Gamma_2 \qquad \textsc{A-Unit-I}\ \ \Gamma \vdash () \Rightarrow \mathbf{Unit}; \Gamma \qquad \textsc{A-Lab-I}\ \ \Gamma \vdash \ell \Rightarrow \{\ell\}; \Gamma$$

$$\textsc{A-Lab-E1}\ \ \frac{\Gamma^{\mathbf{un}} \vdash V \Rightarrow \ell' \qquad \ell' \in L \qquad \Gamma \vdash M_{\ell'} \Rightarrow A; \Delta}{\Gamma \vdash \mathbf{case}\, V\, \mathbf{of}\, \{\overline{\ell : M_\ell}^{\ell \in L}\} \Rightarrow A; \Delta}$$

$$\textsc{A-Lab-E2}\ \ \frac{(\nexists\, \ell')\ \Gamma^{\mathbf{un}} \vdash x \Rightarrow \ell' \qquad \Gamma^{\mathbf{un}} \vdash x \Downarrow L \qquad L \subseteq L' \qquad (\forall \ell \in L)\ \Gamma, y : x = \ell \vdash M_\ell \Rightarrow A_\ell; \Delta, y : x = \ell}{\Gamma \vdash \mathbf{case}\, x\, \mathbf{of}\, \{\overline{\ell : M_\ell}^{\ell \in L'}\} \Rightarrow \mathbf{case}\, x\, \mathbf{of}\, \{\overline{\ell : A_\ell}^{\ell \in L}\}; \Delta}$$

$$\textsc{A-Pi-I}\ \ \frac{\Gamma^{\mathbf{un}} \vdash A \Rightarrow n \qquad \Gamma, x : A \vdash M \Rightarrow B; \Delta \lhd x : A \qquad m = \mathbf{un} \text{ implies } \Gamma = \Delta}{\Gamma \vdash \lambda_m(x : A).M \Rightarrow \Pi_m(x : A)B; \Delta}$$

$$\textsc{A-Pi-E}\ \ \frac{\Gamma_1 \vdash M \Rightarrow C; \Gamma_2 \qquad \Gamma_2^{\mathbf{un}} \vdash C \Downarrow \Pi_m(x : A)B \qquad \Gamma_2 \vdash N \Leftarrow A; \Gamma_3 \qquad \Gamma_1^{\mathbf{un}} \vdash B[N/x] \Rightarrow n}{\Gamma_1 \vdash M\, N \Rightarrow B[N/x]; \Gamma_3}$$

$$\textsc{A-Sigma-I}\ \ \frac{\Gamma_1 \vdash V \Rightarrow A; \Gamma_2 \qquad \Gamma_2, x : A \lhd z : x = V \vdash N \Rightarrow B; \Gamma_3 \lhd x : A \lhd z : x = V}{\Gamma_1 \vdash \langle x = V, N \rangle \Rightarrow \Sigma(x : A)B; \Gamma_3}$$

$$\textsc{A-Sigma-E}\ \ \frac{\begin{array}{c}\Gamma_1 \vdash M \Rightarrow D; \Gamma_2 \qquad \Gamma_2^{\mathbf{un}} \vdash D \Downarrow \Sigma(x : A)B \\ \Gamma_2, x : A, y : B \vdash N \Rightarrow C; \Gamma_3 \lhd x : A \lhd y : B \qquad x, y \notin fv(C)\end{array}}{\Gamma_1 \vdash \mathbf{let}\, \langle x, y \rangle = M\, \mathbf{in}\, N \Rightarrow C; \Gamma_3}$$

$$\textsc{A-Sigma-G}\ \ \frac{\begin{array}{c}\Gamma_1 \vdash M \Rightarrow D; \Gamma_2 \qquad \Gamma_2^{\mathbf{un}} \vdash D \Downarrow \Sigma(x : L)B \\ \Gamma_2, x : L, y : B \vdash \mathbf{case}\, x\, \mathbf{of}\, \{\overline{\ell : N}^{\ell \in L}\} \Rightarrow C; \Gamma_3, x : L \lhd y : B \qquad x, y \notin fv(C)\end{array}}{\Gamma_1 \vdash \mathbf{let}\, \langle x, y \rangle = M\, \mathbf{in}\, N \Rightarrow C; \Gamma_3}$$

$$\textsc{A-Fork}\ \ \frac{\Gamma_1 \vdash M \Leftarrow \mathbf{Unit}; \Gamma_2}{\Gamma_1 \vdash \mathbf{fork}\, M \Rightarrow \mathbf{Unit}; \Gamma_2} \qquad \textsc{A-Ssn-I}\ \ \frac{\Gamma^{\mathbf{un}} \vdash S \Leftarrow \mathbf{lin}}{\Gamma \vdash \mathbf{new} \Rightarrow \Sigma(x : S)S^{\perp}; \Gamma}$$

$$\textsc{A-Ssn-Send-E}\ \ \frac{\Gamma_1 \vdash M \Rightarrow C; \Gamma_2 \qquad \Gamma_2^{\mathbf{un}} \vdash C \Downarrow\, !(x : A)S}{\Gamma_1 \vdash \mathbf{send}\, M \Rightarrow \Pi_{\mathbf{lin}}(x : A)S; \Gamma_2} \qquad \textsc{A-Ssn-Recv-E}\ \ \frac{\Gamma_1 \vdash M \Rightarrow C; \Gamma_2 \qquad \Gamma_2^{\mathbf{un}} \vdash C \Downarrow\, ?(x : A)S}{\Gamma_1 \vdash \mathbf{recv}\, M \Rightarrow \Sigma(x : A)S; \Gamma_2}$$

Algorithmic type checking for expressions (check against) $\boxed{\Gamma \vdash M \Leftarrow A; \Delta}$

$$\textsc{A-Sub-Type}\ \ \frac{\Gamma \vdash M \Rightarrow B; \Delta \qquad \Gamma^{\mathbf{un}} \vdash B \leq A \Rightarrow m}{\Gamma \vdash M \Leftarrow A; \Delta}$$

Fig. 12. Algorithmic typing for LDST

In rule A-Lab-E2, if the variable x is not convertible to a label, then we must synthesize the types for all branches. For each branch, we adopt the equation $x = \ell$ and remove it from the returned environment. As all branches must use resources in the same way, the rule checks that the outgoing environments Δ_ℓ are equal for all branches.

Rule A-Sigma-G, together with its counterpart Sigma-G in Figure 7, is a significant innovation of our system. It governs the elimination of a sigma type where the first component of the pair is a variable of label type L. Instead of type checking the body of the eliminating let once, the rule checks it multiple times, once for each $\ell \in L$. This eta-expansion of the label type enables us to accurately check this construct and enable examples such as those in Section 2.3.

We now address the metatheory for algorithmic type checking. As usual, soundness results rely on strengthening and completeness on weakening, two results that we study below. Below we write $\Gamma_1 \circ \Gamma_2$ to denote the type environment Γ such that $\Gamma = \Gamma_1 \circ \Gamma_2$, when the environment splitting operation is defined.

Lemma 7.1 (Algorithmic Weakening).
(1) If $\Gamma_1 \vdash V \Rightarrow W$, *then* $(\Gamma_1 \circ \Gamma_2) \vdash V \Rightarrow W$.
(2) If $\Gamma_1 \vdash A \Downarrow B$, *then* $(\Gamma_1 \circ \Gamma_2) \vdash A \Downarrow B$.
(3) If $\Gamma_1 \vdash A \le B \Rightarrow m$, *then* $(\Gamma_1 \circ \Gamma_2) \vdash A \le B \Rightarrow m$.
(4) If $\Gamma_1 \vdash A \le B \Leftarrow m$, *then* $(\Gamma_1 \circ \Gamma_2) \vdash A \le B \Leftarrow m$.
(5) If $\Gamma_1 \vdash A \Rightarrow m$, *then* $(\Gamma_1 \circ \Gamma_2) \vdash A \Rightarrow m$.
(6) If $\Gamma_1 \vdash A \Leftarrow m$, *then* $(\Gamma_1 \circ \Gamma_2) \vdash A \Leftarrow m$.
(7) If $\Gamma_1 \vdash M \Rightarrow A; \Gamma_2$, *then* $(\Gamma_1 \circ \Gamma_3) \vdash M \Rightarrow A; (\Gamma_2 \circ \Gamma_3)$.
(8) If $\Gamma_1 \vdash M \Leftarrow A; \Gamma_2$, *then* $(\Gamma_2 \circ \Gamma_3) \vdash M \Leftarrow A; (\Gamma_2 \circ \Gamma_3)$.

Lemma 7.2 (Algorithmic Linear Strengthening). *Suppose that* $\Gamma_1^{\mathbf{un}} \vdash A : \mathbf{lin}$.
(1) If $\Gamma_1, x : A \vdash M \Rightarrow \Gamma_2, x : A; ,$ *then* $\Gamma_1 \vdash M \Rightarrow B; \Gamma_2$.
(2) If $\Gamma_1, x : A \vdash M \Leftarrow \Gamma_2, x : A; ,$ *then* $\Gamma_1 \vdash M \Leftarrow B; \Gamma_2$.

The rest of this section is dedicated to the soundness and completeness results for the various relations in algorithmic type checking. Proofs are by mutual rule induction, even if we present the results separately, for ease of understanding. Proofs can be found in our technical report [Thiemann and Vasconcelos 2019].

Lemma 7.3 (Soundness of Unfolding). *Suppose that* $\Gamma \vdash A : m$ *and* $\Gamma \vdash A \Downarrow B$. *Then* B *is not a case and* $\Gamma \vdash A \equiv B : m$.

Lemma 7.4 (Completeness of Unfolding). *Suppose that* $\Gamma \vdash A : m$ *and there exists some* $\mathcal{P} \in \{L, \Pi_m(y : A)\Box, \Sigma_m(y : A)\Box, !(y : A)\Box, ?(y : A)\Box\}$ *such that* $\Gamma \vdash A \equiv \mathcal{P}[B] : m$. *Then* $\Gamma \vdash A \Downarrow \mathcal{P}[B']$ *where* $\Gamma \vdash B \equiv B' : m$.

Lemma 7.5 (Algorithmic Subtyping Soundness).
(1) If $\Gamma \vdash B \le A \Rightarrow m$, *then* $\Gamma \vdash B \le A : m$.
(2) If $\Gamma \vdash B \le A \Leftarrow m$, *then* $\Gamma \vdash B \le A : m$.

Lemma 7.6 (Algorithmic Subtyping Completeness). *Let* $\Gamma \vdash B \le A : m$. *Then,*
(1) $\Gamma \vdash A \le B \Rightarrow m'$ *with* $m \preceq m'$.
(2) $\Gamma \vdash A \le B \Leftarrow m$.

Lemma 7.7 (Algorithmic Kinding Soundness).

Type translation $\boxed{\langle\!\langle A \rangle\!\rangle = B}$

$$\langle\!\langle !A.S \rangle\!\rangle = !(x : \langle\!\langle A \rangle\!\rangle)\langle\!\langle S \rangle\!\rangle \qquad \langle\!\langle ?A.S \rangle\!\rangle = ?(x : \langle\!\langle A \rangle\!\rangle)\langle\!\langle S \rangle\!\rangle$$
$$\langle\!\langle \oplus\{\overline{\ell : S_\ell}\} \rangle\!\rangle = !(x : L)\mathbf{case}\, x\, \mathbf{of}\, \{\overline{\ell : \langle\!\langle S_\ell \rangle\!\rangle}\} \qquad \langle\!\langle \&\{\overline{\ell : S_\ell}\} \rangle\!\rangle = ?(x : L)\mathbf{case}\, x\, \mathbf{of}\, \{\overline{\ell : \langle\!\langle S_\ell \rangle\!\rangle}\}$$
$$\langle\!\langle \mathbf{End}_! \rangle\!\rangle = !(x : \{\text{EOS}\})\mathbf{End} \qquad \langle\!\langle \mathbf{End}_? \rangle\!\rangle = ?(x : \{\text{EOS}\})\mathbf{End}$$

Environment translation $\boxed{\langle\!\langle \Gamma \rangle\!\rangle = \Delta}$

$$\langle\!\langle \cdot \rangle\!\rangle = \cdot \qquad \langle\!\langle \Gamma, x : A \rangle\!\rangle = \langle\!\langle \Gamma \rangle\!\rangle, x : \langle\!\langle A \rangle\!\rangle$$

Expression translation $\boxed{\langle\!\langle \Gamma \vdash_{\text{LSST}} M : A \rangle\!\rangle = N}$

$$\langle\!\langle \Gamma \vdash_{\text{LSST}} \mathbf{send}\, M : A \to_{\mathbf{lin}} S \rangle\!\rangle = \mathbf{send}\, \langle\!\langle \Gamma \vdash_{\text{LSST}} M : !A.S \rangle\!\rangle$$
$$\langle\!\langle \Gamma \vdash_{\text{LSST}} \mathbf{recv}\, M : A \times S \rangle\!\rangle = \mathbf{recv}\, \langle\!\langle \Gamma \vdash_{\text{LSST}} M : ?A.S \rangle\!\rangle$$
$$\langle\!\langle \Gamma \vdash_{\text{LSST}} \mathbf{select}\, \ell' : \oplus\{\overline{\ell : S_\ell}^{\ell \in L}\} \rangle\!\rangle = \lambda_{\mathbf{lin}}(x : !(y : L)\mathbf{case}\, y\, \mathbf{of}\, \{\overline{\ell : \langle\!\langle S_\ell \rangle\!\rangle}\}).\mathbf{send}\, x\, \ell'$$
$$\langle\!\langle \Gamma_1 \circ \Gamma_2 \vdash_{\text{LSST}} \mathbf{rcase}\, M\, \mathbf{of}\, \{\overline{\ell : y.N_\ell}\} : A \rangle\!\rangle = \mathbf{let}\, \langle x, y \rangle = \mathbf{recv}\, \langle\!\langle \Gamma_1 \vdash_{\text{LSST}} M : \&\{\overline{\ell : S_\ell}\} \rangle\!\rangle\, \mathbf{in}$$
$$\mathbf{case}\, x\, \mathbf{of}\, \{\overline{\ell : \langle\!\langle \Gamma_2, y : S_\ell \vdash_{\text{LSST}} N_\ell : A \rangle\!\rangle}\} \quad \text{where } \Gamma_1 \vdash M : \&\{\overline{\ell : S_\ell}\}$$
$$\langle\!\langle \Gamma \vdash_{\text{LSST}} \mathbf{close}\, M : \mathbf{Unit} \rangle\!\rangle = \mathbf{send}\, \langle\!\langle \Gamma \vdash_{\text{LSST}} M : \mathbf{End}_! \rangle\!\rangle\, \text{EOS}$$
$$\langle\!\langle \Gamma \vdash_{\text{LSST}} \mathbf{wait}\, M : \mathbf{Unit} \rangle\!\rangle = \mathbf{let}\, \langle x, y \rangle = \mathbf{recv}\, \langle\!\langle \Gamma \vdash_{\text{LSST}} M : \mathbf{End}_? \rangle\!\rangle\, \mathbf{in}\, ()$$

Fig. 13. Translation from LSST to LDST

(1) If $\Gamma \vdash M \Rightarrow m$, *then* $\Gamma \vdash M : m$.
(2) If $\Gamma \vdash M \Leftarrow m$, *then* $\Gamma \vdash M : m$.

Lemma 7.8 (Algorithmic Kinding Completeness). *If* $\Gamma \vdash A : m$, *then*
(1) $\Gamma \vdash A \Rightarrow m'$ *with* $m' \leq m$ *and*
(2) $\Gamma \vdash A \Leftarrow m$.

Theorem 7.9 (Algorithmic soundness). *Suppose that* $\Delta \Rightarrow \mathbf{un}$.
(1) If $\Gamma \vdash M \Rightarrow A; \Delta$, *then* $\Gamma \vdash M : A$.
(2) If $\Gamma \vdash M \Leftarrow A; \Delta$, *then* $\Gamma \vdash M : A$.

Theorem 7.10 (Algorithmic Completeness). *If* $\Gamma \vdash M : A$, *then*
(1) $\Gamma \vdash M \Rightarrow B; \Gamma^{\mathbf{un}}$ *with* $\Gamma^{\mathbf{un}} \vdash B \leq A : m$.
(2) $\Gamma \vdash M \Leftarrow A; \Gamma^{\mathbf{un}}$.

The development in this section does not cover the extension to natural numbers from Section 5. However, we present the necessary rules in our technical report [Thiemann and Vasconcelos 2019] and we believe the technical results extend straightforwardly.

8 EMBEDDING LSST INTO LDST

The translation in Figure 13 maps LSST's types, environments, and typing derivations to LDST. It extends homomorphically over all types, expressions, and processes that are not mentioned explicitly. The translation of typing environments annotates each binding with the multiplicity derived from the type. As expected, internal (external) choice maps to sending (receiving) a label

followed by a case distinction on that label. Actively (passively) ending a connection maps to sending (receiving) a distinguished EOS token and dropping the channel.

The translation is a conservative embedding as it preserves subtyping and typing. We establish a simulation and a co-simulation between the original LSST expression and its image in LDST. In the simulation, each step gives rise to one or more steps in the image of the translation. In co-simulation, one step in the image may yield an expression that is still related to the same preimage.

Theorem 8.1 (Typing Preservation).

(1) If $\Gamma \vdash_{LSST} M : A$, *then* $\vdash_{LDST} \langle\!\langle \Gamma \rangle\!\rangle : \mathbf{lin}$ *and* $\langle\!\langle \Gamma \rangle\!\rangle \vdash_{LDST} \langle\!\langle \Gamma \vdash_{LSST} M : A \rangle\!\rangle : \langle\!\langle A \rangle\!\rangle$.
(2) If $\Gamma \vdash_{LSST} P$, *then* $\langle\!\langle \Gamma \rangle\!\rangle \vdash_{LDST} \langle\!\langle \Gamma \vdash_{LSST} P \rangle\!\rangle$.

Theorem 8.2 (Simulation).

(1) If $\Gamma \vdash_{LSST} M : A$ *and* $M \longrightarrow_{LSST} N$, *then* $\langle\!\langle \Gamma \vdash_{LSST} M : A \rangle\!\rangle \longrightarrow^{+}_{LDST} \langle\!\langle \Gamma \vdash_{LSST} N : A \rangle\!\rangle$.
(2) If $\Gamma \vdash_{LSST} P$ *and* $P \longrightarrow_{LSST} Q$, *then* $\langle\!\langle \Gamma \vdash_{LSST} P \rangle\!\rangle \longrightarrow^{+}_{LDST} \langle\!\langle \Gamma \vdash_{LSST} Q \rangle\!\rangle$.

Theorem 8.3 (Co-Simulation).

(1) If $\Gamma \vdash_{LSST} M : A$ *and* $\langle\!\langle \Gamma \vdash_{LSST} M : A \rangle\!\rangle \longrightarrow_{LDST} N$, *then* $M \longrightarrow_{LSST} M'$ *and* $N \longrightarrow^{*}_{LDST} \langle\!\langle \Gamma \vdash_{LSST} M' : A \rangle\!\rangle$.
(2) If $\Gamma \vdash_{LSST} P$ *and* $\langle\!\langle \Gamma \vdash_{LSST} P \rangle\!\rangle \longrightarrow_{LDST} Q$, *then* $P \longrightarrow_{LSST} P'$ *and* $Q \longrightarrow^{*}_{LDST} \langle\!\langle \Gamma \vdash_{LSST} P' \rangle\!\rangle$.

9 IMPLEMENTATION

We implemented a frontend consisting of a parser and a type checker for the LDST calculus, which is available in a GitHub repository[2]. The parser implements an OCaml-inspired syntax which deviates slightly from the Haskell-inspired syntax used in Section 2.

The type checker implements exactly the algorithmic rules from Section 7 including subtyping as well as additional algorithmic unfolding, subtyping, and synthesis rules dealing with natural numbers and their recursor. The type checker supports a coinductive reading of the typing and subtyping rules so that types and session types can be equirecursive. The implementation requires caching of the weakened judgments modulo alpha conversion. This complication arises because types may contain free variables of label type. A weakened judgment contains just the bindings for these free variables; comparing modulo alpha conversion means that the names of the free variables do not matter: $x : L \vdash \mathbf{case}\, x\, \mathbf{of}\, \{\dots\}$ is equal to $y : L \vdash \mathbf{case}\, y\, \mathbf{of}\, \{\dots\}$ (if the $\dots$ match). Caching modulo alpha conversion is needed to make the type checker terminate.

10 RELATED WORK

Linear and Dependent Types. Cervesato and Pfenning [1996] developed the first logical framework supporting linear type theory and dependent types. Shi and Xi [2013] propose using linear types on top of ATS, their dependently typed language for developing provably correct code.

F* [Swamy et al. 2013] is a language that includes linear types and value dependent types. The authors use affine environments to control the use of linear values and distinguish between value application and standard application to properly deal with dependency. F* has further developed into a verification system with full-fledged dependent types [Ahman et al. 2018].

Trellys [Casinghino et al. 2014] combines a general computation language with a specification language via dependent types. While Trellys has no support for linear types, it has been inspiring in finding a replacement for value dependency and in its treatment of equations.

[2] Available at https://github.com/proglang/ldgv

Idris [Brady 2013] is a dependently typed language with uniqueness types. While linear types avoid duplication and dropping of values, a value with unique type is referenced at most once at run time. Brady [2017] shows how to use this feature combination to develop concurrent systems.

Dal Lago and Gaboardi [2011] introduce a lambda calculus with linear dependent types and full higher-order recursion. It relies on a decoration of PCF with first-order index expressions. Under certain assumptions, their type system is complete, i.e., all operational behavior can be captured by typing. Dal Lago and Petit [2012] also consider a sound and complete linear dependent type system. Their emphasis is on complexity analysis for higher-order functional programs.

Krishnaswami et al. [2015] propose a full-spectrum language that integrates linear and dependent types. It is based on the observation that intuitionistic linear logic can be modeled with an adjunction. The resulting syntactic theory consists of an intuitionistic and a linear lambda calculus combined via two modal operators corresponding to the adjunction.

Our work stays in the tradition that keeps linear and unrestricted resources apart. Computations and processes are allowed to depend on unrestricted index values, but dependencies on linear resources are ruled out. Unlike the cited work, our calculus supports dependent subtyping [Aspinall and Compagnoni 2001].

McBride's and Atkey's works combine linear and dependent types in Quantitative Type Theory (QTT) [Atkey 2018; McBride 2016]. In QTT types may depend on linear resources, whereas types in our system can only depend on unrestricted values.

Linear Haskell [Bernardy et al. 2018] is a proposal to integrate linear types with stock functional programming. It does not have dependent types and it manages bindings using a semiring.

Session Types. Caires and Pfenning [2010] developed logical foundations for session types building on intuitionistic linear logic. Their approach enables viewing π-calculus reductions as proof transformations in the logic. Wadler [2012] proposed a foundation based on classical linear logic.

Dependent session types have been proposed first by Toninho et al. [2011] for the π-calculus with value passing. The calculus is aimed at specification and verification, and features a rich logic structure with correspondingly rich proof terms. Wu and Xi [2017] encode session types in their wide spectrum language ATS, which includes DML-style type dependency. Indexed types with unpolarized quantification are used to represent channel types. Types for channel ends are obtained by interpreting the quantifiers. While ATS provides all features for verification, LDST is a minimalist dependent calculus geared towards practical applications. Toninho and Yoshida [2018] develop a language with dependent session types that integrates processes and functional computation via a monadic embedding. Processes may thus depend on expressions as well as expressions may depend on monadic process values.

Compared to our work, their theory encompasses type-level functions with type and value dependent kinds and monads, whereas type-level computation in LDST is restricted to label introduction and elimination. Their work strictly separates linear and unrestricted assumptions, which leads to further duplication, and it has no notion of subtyping. Our setup formalizes large elimination for labels, which is needed in practical applications, but not considered in their work. Moreover, the point of our calculus is to showcase an economic operational semantics with just one communication reduction at the process level. We expect that LDST can be extended with further index types and type-level computation without complicating the operational semantics.

Lolliproc [Mazurak and Zdancewic 2010] is a core calculus for concurrent functional programming. Its primitives are derived from a Curry-Howard interpretation of classical linear logic. Some form of session types can be expressed in Lolliproc, but it does not support unrestricted values nor dependency.

Baltazar et al. [2012] introduce a notion of session types with refinements over linear resources specified by uninterpreted predicates. Even if linear, the dependency is not on expressions of the programming language, thus greatly simplifying the underlying theory. Bonelli et al. [2004] study a simpler extension for session types whereby assume/assert labels present in expressions make their way into types to represent starting and ending points in protocols.

Goto et al. [2016] consider a polymorphic session typing system for a π-calculus which replaces branching and choice by matching and mismatching tests. These tests compare tokens, akin to our labels, and introduce (in)equational constraints in the type system.

Others. Nishimura [1998] considers a calculus for objects where messages (a method name and parameters) are first-class constructs. Each such message is typed as the set of method names that may be invoked by the message, formalized in a second order polymorphic type system. Vasconcelos and Tokoro [1993] and Sangiorgi [1998] pursue a similar idea in the context of the π-calculus that allow the transmission of variant values, say a label, together with an integer value. We follow a different approach, by exchanging values only, while labels appear as a particular case. Neither these works use (label) dependent types to classify messages.

ROSE [Morris and McKinna 2019] is a versatile theory of row typing that could be applied to session types among other applications. Strikingly, a row is a mapping from labels to types. Hence, a row type could be expressed by a Π type in our system using a case for the label dispatch. ROSE has a fixed set of constraints for combining rows and requires labels to be compile-time constants. LDST labels are first-class objects and combinations are expressed with user-defined functions.

11 CONCLUSIONS

LDST is a minimalist calculus that combines dependent types and session types from the point of economy of expression: a single pair of communication primitives is sufficient. It faithfully extends existing systems while retaining wire compatibility with them. Building the calculus on dependent types liberates the structure of session-typed programs from mimicking the type structure.

LDST supports encodings of algebraic datatypes with subtyping by modeling tagged data with Σ-types. The same approach may be used to simulate session calculi based on sending and receiving tagged data. It further incorporates natural numbers and primitive recursion at the type level.

We are currently working on a few extensions for LDST.

(1) Our implementation already supports recursive session types and we expect that the properties of algorithmic typing also extend to this setting.
(2) We plan to address subtyping for the type recursor in a coinductive manner.
(3) It would be interesting to add further kinds of predicates beyond equality as well as type dependency (as supported by previous work [Toninho et al. 2011; Toninho and Yoshida 2018]).

ACKNOWLEDGMENTS

This work was supported by FCT through the LASIGE Research Unit, ref. UID/CEC/00408/2019, and by Cost Action CA15123 EUTypes.

REFERENCES

Danel Ahman, Cédric Fournet, Catalin Hritcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. 2018. Recalling a Witness: Foundations and Applications of Monotonic State. *PACMPL* 2, POPL (2018), 65:1–65:30.

David Aspinall and Adriana B. Compagnoni. 2001. Subtyping Dependent Types. *Theoretical Computer Science* 266, 1-2 (2001), 273–309. https://doi.org/10.1016/S0304-3975(00)00175-4

Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *LICS*. ACM, 56–65.

Pedro Baltazar, Dimitris Mostrous, and Vasco Thudichum Vasconcelos. 2012. Linearly Refined Session Types. In *LINEARITY (EPTCS)*, Vol. 101. 38–49.

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language. *PACMPL* 2, POPL (2018), 5:1–5:29.

Eduardo Bonelli, Adriana B. Compagnoni, and Elsa L. Gunter. 2004. Correspondence Assertions for Process Synchronization in Concurrent Communications. *Electr. Notes Theor. Comput. Sci.* 97 (2004), 175–195. https://doi.org/10.1016/j.entcs.2004.04.036

Edwin Brady. 2013. Idris, A General-Purpose Dependently Typed Programming Language: Design and Implementation. *J. Funct. Program.* 23, 5 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

Edwin Brady. 2017. Type-driven Development of Concurrent Communicating Systems. *Computer Science (AGH)* 18, 3 (2017). https://doi.org/10.7494/csci.2017.18.3.1413

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR (LNCS)*, Vol. 6269. Springer, Paris, France, 222–236.

Luís Caires, Frank Pfenning, and Bernardo Toninho. 2016. Linear logic propositions as session types. *Mathematical Structures in Computer Science* 26, 3 (2016), 367–423. https://doi.org/10.1017/S0960129514000218

Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. 2014. Combining Proofs and Programs in a Dependently Typed Language. In *POPL*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 33–46. https://doi.org/10.1145/2535838.2535883

Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Luca Padovani. 2009. Foundations of Session Types. In *Principles and Practice of Declarative Programming, PPDP 2009*, António Porto and Francisco J. López-Fraguas (Eds.). ACM, Coimbra, Portugal, 219–230.

Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyen. 2016. Set-Theoretic Types for Polymorphic Variants. In *ICFP*. ACM, 378–391.

Iliano Cervesato and Frank Pfenning. 1996. A Linear Logical Framework. In *LICS*. IEEE Computer Society, 264–275.

Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. 2017. On the Preciseness of Subtyping in Session Types. *Logical Methods in Computer Science* 13, 2 (2017).

Ugo Dal Lago and Marco Gaboardi. 2011. Linear Dependent Types and Relative Completeness. *Logical Methods in Computer Science* 8, 4 (2011).

Ugo Dal Lago and Barbara Petit. 2012. Linear Dependent Types in a Call-By-Value Scenario. In *PPDP*. ACM, 115–126.

Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2012. Session Types Revisited. In *PPDP*. ACM, 139–150.

Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Dimitris Mostrous, and Nobuko Yoshida. 2009. Objects and Session Types. *Information and Computation* 207, 5 (2009), 595–641.

Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ICFP*. ACM, 429–442.

Francisco Ferreira and Brigitte Pientka. 2014. Bidirectional Elaboration of Dependently Typed Programs. In *PPDP*. ACM, 161–174.

Jacques Garrigue. 1998. Programming with Polymorphic Variants. In *In ACM Workshop on ML*.

Simon J. Gay and Malcolm Hole. 2005. Subtyping for Session Types in the Pi Calculus. *Acta Informatica* 42, 2-3 (2005), 191–225.

Simon J. Gay and Vasco Thudichum Vasconcelos. 2010. Linear Type Theory for Asynchronous Session Types. *J. Funct. Program.* 20, 1 (2010), 19–50.

Simon J. Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. 2010. Modular Session Types for Distributed Object-Oriented Programming, See [POPL 2010 2010], 299–312. https://doi.org/10.1145/1706299.1706335

Matthew A. Goto, Radha Jagadeesan, Alan Jeffrey, Corin Pitcher, and James Riely. 2016. An Extensible Approach to Session Polymorphism. *Mathematical Structures in Computer Science* 26, 3 (2016), 465–509.

Robert Harper. 2016. *Practical Foundations for Programming Languages* (second ed.). Cambridge University Press.

Kohei Honda. 1993. Types for Dyadic Interaction. In *Proceedings of 4th International Conference on Concurrency Theory (LNCS)*, Eike Best (Ed.). Springer, 509–523.

Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. 2011. Scribbling Interactions with a Formal Foundation. In *ICDCIT 2011 (LNCS)*, Vol. 6536. Springer, Bhubaneshwar, India, 55–75.

Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Proc. 7th ESOP (LNCS)*, Chris Hankin (Ed.), Vol. 1381. Springer, Lisbon, Portugal, 122–138.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *Proc. 35th ACM Symp. POPL*, Phil Wadler (Ed.). ACM Press, San Francisco, CA, USA, 273–284.

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

Raymond Hu, Nobuko Yoshida, and Kohei Honda. 2008. Session-Based Distributed Programming in Java. In *22nd ECOOP (LNCS)*, Jan Vitek (Ed.), Vol. 5142. Springer, Paphos, Cyprus, 516–541.

Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, and Philip Wadler. 2017. Gradual Session Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 38 (Sept. 2017), 28 pages. https://doi.org/10.1145/3110282

Naoki Kobayashi. 2002. Type Systems for Concurrent Programs. In *10th Anniversary Colloquium of UNU/IIST (Lecture Notes in Computer Science)*, Vol. 2757. Springer, 439–453.

Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. 1996. Linearity and the pi-calculus. In *Proc. 1996 ACM Symp. POPL*. ACM Press, St. Petersburg Beach, FL, USA, 358–371.

Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. 2015. Integrating Linear and Dependent Types. In *POPL*. ACM, 17–30.

Sam Lindley and J. Garrett Morris. 2014. Sessions as Propositions. In *Proceedings 7th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2014, Grenoble, France, 12 April 2014. (EPTCS)*, Alastair F. Donaldson and Vasco T. Vasconcelos (Eds.), Vol. 155. 9–16. https://doi.org/10.4204/EPTCS.155.2

Sam Lindley and J. Garrett Morris. 2016. Talking Bananas: Structural Recursion for Session Types. In *ICFP*. ACM, 434–447.

Karl Mazurak and Steve Zdancewic. 2010. Lolliproc: to concurrency from classical linear logic via curry-howard and control. In *ICFP*. ACM, 39–50.

Conor McBride. 2016. I Got Plenty o' Nuttin'. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (LNCS)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.), Vol. 9600. Springer, 207–233. https://doi.org/10.1007/978-3-319-30936-1_12

J. Garrett Morris and James McKinna. 2019. Abstracting Extensible Data Types: or, Rows by Any Other Name. *PACMPL* 3, POPL (2019), 12:1–12:28. https://dl.acm.org/citation.cfm?id=3290325

Susumu Nishimura. 1998. Static Typing for Dynamic Messages. In *Proc. 25th ACM Symp. POPL*, Luca Cardelli (Ed.). ACM Press, San Diego, CA, USA, 266–278. https://doi.org/10.1145/268946.268968

Luca Padovani. 2017a. Context-Free Session Type Inference. In *ESOP (Lecture Notes in Computer Science)*, Vol. 10201. Springer, 804–830.

Luca Padovani. 2017b. A Simple Library Implementation of Binary Sessions. *J. Funct. Program.* 27 (2017), e4. https://doi.org/10.1017/S0956796816000289

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM TOPLAS* 22, 1 (2000), 1–44. https://doi.org/10.1145/345099.345100

POPL 2010 2010. *Proc. 37th ACM Symp. POPL*. ACM Press, Madrid, Spain.

Davide Sangiorgi. 1998. An Interpretation of Typed Objects into Typed pi-Calculus. *Inf. Comput.* 143, 1 (1998), 34–73.

Alceste Scalas and Nobuko Yoshida. 2016. Lightweight Session Programming in Scala. In *ECOOP (LIPIcs)*, Vol. 56. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 21:1–21:28.

Rui Shi and Hongwei Xi. 2013. A Linear Type System for Multicore Programming in ATS. *Science of Computer Programming* 78, 8 (2013), 1176–1192. https://doi.org/10.1016/j.scico.2012.09.005

Vilhelm Sjöberg, Chris Casinghino, Ki Yung Ahn, Nathan Collins, Harley D. Eades III, Peng Fu, Garrin Kimmell, Tim Sheard, Aaron Stump, and Stephanie Weirich. 2012. Irrelevance, Heterogeneous Equality, and Call-by-value Dependent Type Systems. In *Proceedings Fourth Workshop on Mathematically Structured Functional Programming, MSFP 2012, Tallinn, Estonia, 25 March 2012. (EPTCS)*, James Chapman and Paul Blain Levy (Eds.), Vol. 76. 112–162. https://doi.org/10.4204/EPTCS.76.9

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2013. Secure Distributed Programming With Value-Dependent Types. *J. Funct. Program.* 23, 4 (2013), 402–451. https://doi.org/10.1017/S0956796813000142

Kaku Takeuchi, Kohei Honda, and Makoto Kubo. 1994. An Interaction-Based Language and its Typing System. In *6th International PARLE Conference*, C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis (Eds.). LNCS, Vol. 817. Springer, Athens, Greece, 398–413.

Peter Thiemann and Vasco T. Vasconcelos. 2019. Label-Dependent Session Types. *CoRR* abs/1911.00705 (2019). http://arxiv.org/abs/1911.00705 extended version.

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2011. Dependent Session Types via Intuitionistic Linear Type Theory. In *PPDP*, Peter Schneider-Kamp and Michael Hanus (Eds.). ACM, Odense, Denmark, 161–172.

Bernardo Toninho and Nobuko Yoshida. 2018. Depending on Session-Typed Processes. In *FoSSaCS (Lecture Notes in Computer Science)*, Vol. 10803. Springer, 128–145.

Vasco T. Vasconcelos. 2012. Fundamentals of Session Types. *Information and Control* 217 (2012), 52–70.

Vasco T. Vasconcelos, António Ravara, and Simon J. Gay. 2006. Type Checking a Multithreaded Functional Language with Session Types. *Theoretical Computer Science* 368, 1-2 (2006), 64–87.

Vasco Thudichum Vasconcelos and Mario Tokoro. 1993. A Typing System for a Calculus of Objects. In *ISOTAS (Lecture Notes in Computer Science)*, Vol. 742. Springer, 460–474.

Philip Wadler. 2012. Propositions as Sessions. In *ICFP'12*, Robby Bruce Findler (Ed.). ACM, Copenhagen, Denmark, 273–286.

David Walker. 2005. Substructural Type Systems. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). MIT Press, Chapter 1.
Hanwen Wu and Hongwei Xi. 2017. Dependent Session Types. http://arxiv.org/abs/1704.07004. (2017). arXiv CoRR.