On Expressing Stateful Computation in Linear Logic

Luís Caires
Instituto Superior Técnico (U Lisboa) / INESC-ID
luis.caires@tecnico.ulisboa.pt

In this work, we discuss a notion of linear shared state types, which allow an expressive model of mutable shared state to be parsimoniously represented within linear logic framework. The computational interpretation of linear logic as a session-typed π-calculus [7, 8, 26] has motivated many developments, and may well be accepted as a fairly canonical typed model for stateful concurrent computation with linear resources, pretty much like the lambda calculus is considered a canonical typed model for functional sequential computation with pure values. The interpretation of linear logic as session types [7, 8] promoted the process interpretation of linear logic proofs [1, 2], where proofs correspond to processes interacting via linear session channels, tensor corresponds to session output, co-tensor to session input, and computation progresses by sequences of elementary name-passing interactions, instead of more monolithic beta-reduction style conversion and term substitution as in λ-calculi. In particular, in classical linear logic (CLL) interpretations [8, 26], linear logic negation denotes session type duality. By bringing in additives and exponentials one may then represent full (linear and shared) session types [13, 14, 11]. This basic framework may be further extended with second-order quantifiers [6, 26], useful to express polymorphism and abstract data types and allowing, in particular, (linear) System-F to be fully-abstractly encoded [25], with inductive and co-inductive types [24, 25], and several other mechanisms (e.g., [23]).

Other works [3, 18] demonstrated how notions of shared mutable state may also be accommodated on top of this basic scheme, even if not always remaining faithful to the pure proposition-as-types paradigm. In [20, 21], we have introduced program constructs inspired in the quantitative exponentials of DiLL [10], which allows full stateful shared state computation to be expressed in a system that satisfies the distinguising features of proposition-as-types: programs as proofs, proposition as types, computation as proof simplification. The resulting language CLASS [20, 21], actually a conservative extension of classical linear logic with affine types and state modalities, can express realistic shared state programs which are literally proofs of their own correctness, in the sense that well-typing ensures protocol fidelity, deadlock absence, termination, confluence and memory safety. The key idea from [20] is to explore the meaning of DiLL co-contraction to express sharing of linear objects, namely, of mutable memory cells:

$$\frac{\vdash \Delta', c : \mathsf{U}_\bullet A \quad \vdash \Delta, c : \mathsf{U}_\bullet A}{\vdash \Delta', \Delta, c : \mathsf{U}_\bullet A} \qquad \frac{P \vdash \Delta', c : \mathsf{U}_\bullet A \quad Q \vdash \Delta, c : \mathsf{U}_\bullet A}{\mathsf{share}\ c\ \{P \mid\mid Q\} \vdash \Delta', \Delta, c : \mathsf{U}_\bullet A}$$

On the left, we can find the standard rule expressing co-contraction [10], in this case for the modality $\mathsf{U}_\bullet A$, which represents shared usage of mutable memory. On the right, we show the corresponding typing rule for the program term $\mathsf{share}\ c\ \{P \mid\mid Q\}$ denoting the composition of two independent processes sharing a mutable memory proof object c containing a (linear) value of type A. By iterating use of $\mathsf{share}\ c\ \{P \mid\mid Q\}$ we may express computations where any number of concurrent threads may share a common mutable linear resource. Mutable linear memory cells storing values of type A are represented by program terms of type $\mathsf{S}_\bullet$, the dual type of $\mathsf{U}_\bullet A$, and [21] introduced basic primitives to read from and write to such linear memory cells, which behave pretty much like Haskell MVars [16]. The typing rules for these primitives monolithically integrate multiplicative and monadic principles, and the resulting logic satisfies all the

Submitted to:
TLLA24

$$\frac{}{\mathsf{drop}\ c \vdash c : \mathsf{U}X.A}\ [\text{Tdrop}] \qquad \frac{P \vdash \Delta', c : \mathsf{U}X.A \quad Q \vdash \Delta, c : \{\mathsf{U}X.A/X\}B}{\mathsf{share}\ c\ \{P \parallel Q\} \vdash \Delta', \Delta, c : \{\mathsf{U}X.A/X\}B}\ [\text{Tsh}] \qquad \frac{P \vdash \Delta \quad Q \vdash \Delta}{P + Q \vdash \Delta}\ [\text{Tsum}]$$

$$\mathsf{share}\ c\ \{P \parallel Q\} \equiv \mathsf{share}\ c\ \{Q \parallel P\}$$
$$\mathsf{share}\ c\ \{\mathsf{drop}\ c \parallel Q\} \equiv Q$$
$$\mathsf{share}\ c\ \{\alpha; P \parallel Q\} \equiv \alpha; \mathsf{share}\ c\ \{P \parallel Q\} \quad (\alpha \neq \mathsf{use}\ c)$$
$$\mathsf{share}\ c\ \{\mathsf{use}\ c; Q \parallel \mathsf{use}\ c; P\} \equiv \mathsf{use}\ c; \mathsf{share}\ c\ \{Q \parallel \mathsf{use}\ c; P\} + \mathsf{use}\ c; \mathsf{share}\ c\ \{\mathsf{use}\ c; Q \parallel P\}$$

Figure 1: Selected typing rules and conversions for shared types.

```
type shared Off {
   offer of {
      |#turnOn : On
   }
} and On {
   offer of {
      |#turnOff : Off
   }
}
```

```
proc shared switch (l : Off){
   case l of {
      |#turnOn : switchOn(l)
   }
} and switchOn (l : On){
   case l of {
      |#turnOff : switch(l)
   }
}
```

```
proc main (){
   cut {
      switch(l) |l : Off|
      share l {
         use l;#turnOn l;#turnOff l;drop l
         ||
         use l;#turnOn l;#turnOff l;drop l
         }
      }
}
```

Figure 2: Example: A shared toggle switch.

expected meta-theoretic properties (cut-elimination, confluence, termination). In this work, we propose a reconstruction of these shared state primitives in [21] from more elementary and general operations, while preserving all the fundamental meta-theoretical properties of the resulting type system.

To that end we introduce the shared state type $\mathsf{S}X.A$ and its dual, the shared usage type $\mathsf{U}X.A$. These are essentially adjusted forms of co-recursive and recursive types, that satisfy the key quantitative co-contraction and co-weakening principles of DiLL, and, remarkably, allow us to model the well-known mechanism of serialisation up to a invariant [12, 5], by interpreting unfold segments of shared usage types as critical sections. More concretely, in our setting concurrent clients compete to acquire unique ownership of shared resource via a principal cut reduction between a “use" operation ($\mathsf{use}\ c; P$) and a shared co-recursive object; use operator combines “unfold" with the shared access control of CLASS. This mechanism is closely related to manifest sharing [3], where invariant-based sharing of linear objects within a linear logic interpretation of session types was already achieved, however, in our parsimonious approach based on DiLL principles, computation rules precisely coincide with proof conversions, and confluence and deadlock absence are a natural consequence of cut-elimination, faithfully to the spirit of propositions-as-types. The sharing trees imposed by the “unfold sensitive" co-contraction rule (Figure 1 [Tsh]) disciplines reduction in a way such that each client accessing the shared resource always execute the body of the respective co-recursive definition in mutual exclusion, before allowing other client to take over. Any usage of a shared object may be released using drop (co-weakening, Fig 1 [Tdrop]), as in [21]. The conversion rules expands concurrent unfolds in sums (Fig 1 [Tsum]) of alternative interleavings, thus

ensuring confluence of cut elimination (cf. DiLL [10]), and allowing the system to support equational reasoning about program observational equivalence [19].

We briefly exemplify in Fig. 2 our construction with the definition of a shareable toggle switch (linear) object, where, for readability, we use CLASS syntax. We define the (co)-recursive types *Off* and *On*, where *Off* is shared, and the *switch* process that continuously offers actions *turnOn* and *turnOff*. These actions statefully change the type of the *switch* from *Off* to *On* and from *On* to *Off* respectively. The *main* process composes (using cut) a *switch* object at l with two client threads, which use l concurrently, considering the body of the shared type as a mutual exclusion section with *Off* as invariant type.

In our talk, we motivate and present our basic language and type system, which consists in classical linear logic with inductive / conductive types, affine types, the novel formulation of shared types here, and overview its meta-theoretic properties, which imply strong guarantees of practical interest: programs do not deadlock, always terminate and do not leak resources. We will also demonstrate the expressiveness of the language expressiveness in realistic concurrent session-based shared-state programs written in an extension of the CLASS implementation [22], and briefly compare with recent work on type systems for concurrent programming with resources based on linear logic, which are becoming more and more relevant in computing practice, as witnessed by the widespread adoption of programming languages such as Rust, for general systems programming, Move, for blockchain smart contracts, Linear Haskell, and many others [27, 15, 4, 17, 9].

References

[1] S. Abramsky (1993): *Computational Interpretations of Linear Logic*. *Theor. Comput. Sci.* 111(1), pp. 3–57.

[2] Samson Abramsky (1994): *Proofs as Processes*. *Theor. Comput. Sci.* 135(1), pp. 5–9.

[3] Stephanie Balzer & Frank Pfenning (2017): *Manifest Sharing with Session Types*. *Proc. ACM Program. Lang.* 1(ICFP).

[4] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones & Arnaud Spiwack (2018): *Linear Haskell: practical linearity in a higher-order polymorphic language*. *Proc. ACM Program. Lang.* 2(POPL), pp. 5:1–5:29.

[5] Stephen Brookes & Peter W. O'Hearn (2016): *Concurrent separation logic*. *ACM SIGLOG News* 3(3), pp. 47–65.

[6] Luís Caires, Jorge A. Pérez, Frank Pfenning & Bernardo Toninho (2013): *Behavioral Polymorphism and Parametricity in Session-Based Communication*. In: *ESOP'13*, LNCS, Springer-Verlag, p. 330–349.

[7] Luís Caires & Frank Pfenning (2010): *Session Types as Intuitionistic Linear Propositions*. In Paul Gastin & François Laroussinie, editors: *CONCUR 2010 - Concurrency Theory*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 222–236.

[8] Luís Caires, Frank Pfenning & Bernardo Toninho (2016): *Linear logic propositions as session types*. *Mathematical Structures in Computer Science* 26(3), p. 367–423.

[9] Ankush Das & Frank Pfenning (2022): *Rast: A Language for Resource-Aware Session Types*. *Log. Methods Comput. Sci.* 18(1).

[10] Thomas Ehrhard (2018): *An introduction to differential linear logic: proof-nets, models and antiderivatives*. *Mathematical Structures in Computer Science* 28(7), pp. 995–1060.

[11] S. Gay & M. Hole (2005): *Subtyping for Session Types in the Pi Calculus*. *Acta Inf.* 42(2-3), pp. 191–225.

[12] Charles A. R. Hoare (1972): *Towards a theory of parallel programming*. In: *The origin of concurrent programming*, Springer, pp. 231–244.

[13] Kohei Honda (1993): *Types for dyadic interaction*. In Eike Best, editor: *CONCUR'93*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 509–523.

[14] Kohei Honda, Vasco T. Vasconcelos & Makoto Kubo (1998): *Language primitives and type discipline for structured communication-based programming*. In Chris Hankin, editor: *Programming Languages and Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 122–138.

[15] Jules Jacobs & Stephanie Balzer (2023): *Higher-Order Leak and Deadlock Free Locks*. *Proc. ACM Program. Lang.* 7(POPL), pp. 1027–1057.

[16] Simon Peyton Jones, Andrew Gordon & Sigbjorn Finne (1996): *Concurrent Haskell*. In: *POPL*, 96, Citeseer, pp. 295–308.

[17] Nicholas D. Matsakis & Felix S. Klock (2014): *The Rust Language*. *Ada Lett.* 34(3), p. 103–104.

[18] Zesen Qian, G. A. Kavvos & Lars Birkedal (2021): *Client-Server Sessions in Linear Logic*. *Proc. ACM Program. Lang.* 5(ICFP).

[19] Pedro Rocha (2022): *CLASS: A Logical Foundation for Typeful Programming with Shared State*. Ph.D. thesis, NOVA School of Science and Technology.

[20] Pedro Rocha & Luís Caires (2021): *Propositions-as-types and shared state*. *Proc. ACM Program. Lang.* 5(ICFP), pp. 1–30.

[21] Pedro Rocha & Luís Caires (2023): *Safe Session-Based Concurrency with Shared Linear State*. In Thomas Wies, editor: *ESOP 2023*, *Lecture Notes in Computer Science* 13990, Springer, pp. 421–450.

[22] Pedro Rocha & Luís Caires (2023): *Safe Session-based Concurrency with Shared Linear State (Artifact)*. doi:https://doi.org/10.5281/zenodo.7506064.

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

[24] Bernardo Toninho, Luís Caires & Frank Pfenning (2014): *Corecursion and non-divergence in session-typed processes*. In: *International Symposium on Trustworthy Global Computing*, Springer, pp. 159–175.

[25] Bernardo Toninho & Nobuko Yoshida (2021): *On Polymorphic Sessions and Functions: A Tale of Two (Fully Abstract) Encodings*. *ACM Trans. Program. Lang. Syst.* 43(2).

[26] Philip Wadler (2014): *Propositions as sessions*. *Journal of Functional Programming* 24(2-3), pp. 384–418.

[27] Jingyi Emma Zhong, Kevin Cheang & Shaz Qadeer et. al. (2020): *The Move Prover*. In Shuvendu K. Lahiri & Chao Wang, editors: *CAV 2020*, 12224, Springer, pp. 137–150.