Intrinsically-Typed Definitional Interpreters for Linear, Session-Typed Languages

Arjen Rouvoet
Delft University of Technology
The Netherlands

Casper Bach Poulsen
Delft University of Technology
The Netherlands

Robbert Krebbers
Delft University of Technology
The Netherlands

Eelco Visser
Delft University of Technology
The Netherlands

Abstract

An intrinsically-typed definitional interpreter is a concise specification of dynamic semantics, that is executable and type safe by construction. Unfortunately, scaling intrinsically-typed definitional interpreters to more complicated object languages often results in definitions that are cluttered with manual proof work. For linearly-typed languages (including session-typed languages) one has to prove that the interpreter, as well as all the operations on semantic components, treat values linearly. We present new methods and tools that make it possible to implement intrinsically-typed definitional interpreters for linearly-typed languages in a way that hides the majority of the manual proof work. Inspired by separation logic, we develop reusable and composable abstractions for programming with linear operations using dependent types. Using these abstractions, we define interpreters for linear lambda calculi with strong references, concurrency, and session-typed communication in Agda.

CCS Concepts • **Theory of computation → Separation logic**; • **Software and its engineering → Semantics**; *Concurrent programming structures.*

Keywords definitional interpreters, dependent types, mechanized semantics, Agda, type safety, linear types, session types, separation logic

ACM Reference Format:
Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. 2020. Intrinsically-Typed Definitional Interpreters for Linear, Session-Typed Languages. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '20), January 20–21, 2020, New Orleans, LA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3372885.3373818

CPP '20, January 20–21, 2020, New Orleans, LA, USA

ACM ISBN 978-1-4503-7097-4/20/01.
https://doi.org/10.1145/3372885.3373818

1 Introduction

An intrinsically-typed interpreter is an appealing, and practical specification of the dynamic semantics of a programming language, that ensures both an executable and a type-safe semantics. Intrinsically-typed interpreters for simple languages, such as the simply-typed lambda calculus (STLC), demonstrate the appeal well (Fig. 1). Because it is an executable specification, one can test that the specified behavior is correct. At the same time, the type of the interpreter expresses a type safety theorem that is enforced *interactively* during the development of the specification: if the interpreter passes the type checker of the host language, then the theorem holds. The integration of the proof in the definition of the interpreter ensures that the proof can never lag behind changes of the object language. Moreover, the integration allows the types of the object language to inform the implementation. Specifically, *dependent pattern matching* [10, 12] rules out the bad cases that cause partiality in an untyped STLC interpreter. An untyped interpreter cannot guarantee that lookup of lexical variables succeeds, nor that the evaluation of the function expression f in app f e results in a

```
data Exp : Ty → List Ty → Set where
  tt   : Exp unit Γ
  var  : a ∈ Γ → Exp a Γ
  lam  : Exp b (a :: Γ) → Exp (fun a b) Γ
  app  : Exp (fun a b) Γ → Exp a Γ → Exp b Γ
data Val : Ty → Set where
  tt   : Val unit
  clos : Env Γ → Exp b (a :: Γ) → Val (fun a b)
eval : Exp a Γ → Env Γ → Val a
eval tt         env  = tt
eval (var x)    env  = lookup env x
eval (lam e)    env  = clos env e
eval (app f e)  env with eval f env
... | clos env′ body = eval body (eval e env :: env′)
```

Figure 1. Intrinsically-typed interpreter for STLC in Agda

closure value. Because the integration of the proof into the interpreter requires no extra proof work and avoids cases that cannot occur when evaluating typed terms, integrating the type safety proof into the interpreter improves the clarity of the specification.

Unfortunately, extending the object language with new features, such as computational effects, may require additional proof work. For example, adding state to a functional language complicates the type safety invariant of the language with monotonicity of store extension [32]. A straightforward extension of the STLC interpreter to STLC with ML references requires manual proof work for weakening references using store extension witnesses. These proof terms in the interpreter obscure the semantics of the object language [6]. Similarly, an interpreter for a linearly-typed language with threads and session-typed communication needs to do work to prove that linearity is maintained throughout all the operations. This proof work quickly exceeds the computational work of the interpreter (as we will demonstrate in §2). This undermines the quality of the specification, as it no longer clearly communicates the dynamic semantics of the object language.

This tension between clarity and the features of the object languages that we want to specify is painful. We would like to avoid the partiality of an untyped interpreter, and avoid maintaining a separate type safety proof, but losing the clarity of the semantics is a high price to pay. The aim of our work is to resolve this tension, by developing reusable abstractions that not only implement the *functional* aspects of a language feature, but also cooperate in *proving* type safety. In this paper we develop such dependently-typed, reusable abstractions for intrinsically-typed interpretation of linearly-typed languages, with concurrency and session-typed communication. We show that despite the pervasiveness of linearity, we can use these abstractions to safely interpret languages with linear references in a non-linear host language like Agda with minimal proof work. We accomplish this by extending the solution of Bach Poulsen et al. [6] for languages with monotone state, proposing the novel abstraction of monads on predicates over *proof-relevant separation algebras*. To that end we adapt ideas from separation logic [29, 33] to the proof-relevant setting. The resulting monadic interpreters are almost free of proof terms. At the mere cost of the few (and trivial) proof terms that remain, we avoid partiality due to ill-typed cases, and obtain type safety by construction.

Notation Our abstractions and interpreters (like those in Fig. 1) are constructed in the dependently-typed language Agda [28]. We present them in this paper in almost valid Agda code, omitting universe levels and implicit universal quantifications for brevity. Sometimes we omit constructors from syntax definitions, in which case we write an ellipsis in the signature: **data** A : Set **where** (...). We also gloss over termination issues of the interpreters that we present. In the artifact that accompanies this paper [34], we make universe levels and quantifications explicit, and work in a (sized) delay monad [1, 9, 13] or with fueled interpreters [5, 30, 35].

Contributions We present a new method that makes it possible to implement intrinsically-typed definitional interpreters for linearly-typed languages in a concise way that hides the majority of the manual proof work. Concretely, we make the following technical contributions:

- We develop a proof-relevant version of separation logic using which we can concisely specify co-de-Bruijn binding in syntax. We use it to specify not just (linear) lexical binding, but also linear references, without explicitly mentioning the separation of typing contexts or store types (§3.1).
- We show that the separation logic connectives, and the magic wand in particular, are useful to specify operations on these syntaxes, by defining a linear reader monad transformer using the separation logic connectives and interpreting a linearly-typed lambda calculus in a type-safe fashion (§3.2).
- We define *proof-relevant separation algebras* (PRSAs) as a generalization of traditional separation algebras, and show that our co-de-Bruijn-style separation logic is an instance of it (§4.1).
- We define a PRSA called Market that takes care of the accounting of *supply* and *demand* in the presence of a store, and references into it (§4.2).
- We construct a linear state monad using the Market PRSA, which can be used to interpret linearly-typed stateful languages in an intrinsically type-safe manner, without manual accounting of store separation (§4.3).
- We present two case studies that apply these methods to define monadic and type-safe interpreters that are almost entirely free of proof terms, for:
  - LTLC_{ref}—a linearly-typed lambda calculus with linear references and strong update (§4.4).
  - LTLC_{ses}—a linearly-typed lambda calculus with concurrency and session-typed communication (§5). We develop a linear free monad, ensuring that the expression semantics is independent of the implementation of scheduling and communication.
- We implement the library for proof-relevant separation logic and the case studies in Agda [34].

2 Linear Languages Typed Intrinsically

Before we present the technical contributions of this paper, we briefly analyze the operations and typing of a functional, linear language with session-typed communication (§2.1). We also summarize how we would type the syntax of a linear language, its semantic components, and its interpreter in Agda using the standard type formers of Set (§2.2), and show how this yields interpreters with a lot of proof terms (§2.3).

2.1 Linear, Session-Typed Languages

As a simple session-typed language [17, 38] we consider a linearly-typed lambda calculus, extended with primitives for concurrency (*fork*) and communication (*mkchan*, *send*, *recv*, and *close*). The following sample program (in pseudo concrete syntax for such a language) exchanges a constant number between two threads:

$$\begin{array}{lll}\textbf{let } \varphi_l\ ,\ \varphi_r & = & \mathit{mkchan}\ (?\mathit{nat}\ .\ \mathit{end}) \\ \textbf{in let } \_ & = & \mathit{fork}\ (\lambda\ \_\ \rightarrow \\ & & \qquad \textbf{let } x, \varphi_{l2} \ = \ \mathit{recv}\ \varphi_l\ \textbf{in}\ \mathit{close}\ \varphi_{l2}) \\ \textbf{in let } \varphi_{r2} & = & \mathit{send}\ 42\ \varphi_r \\ \textbf{in let } \_ & = & \mathit{close}\ \varphi_{r2}\end{array}$$

The *mkchan* construct spawns a new channel, over which communication occurs according to the given *session type*. This produces two endpoints of dual types: φ_l and φ_r with session types $nat\ ?\ end$ and $nat\ !\ end$, respectively. The first type denotes that the endpoint expects to receive a single *nat*. The second (dual) type expresses that a single *nat* should be sent on it. Communication can occur because one endpoint is captured by the closure that is executed in a new thread.

When communication occurs on an endpoint, the protocol described by the session type progresses. This is reflected in our small language by the fact that communication primitives return endpoint references with an updated type. The following typing rule for *send* and *recv* make this precise.

$$\frac{\Gamma\ \vdash\ \mathit{ch} : ?\ a\ .\ \beta}{\Gamma\ \vdash\ \mathit{recv}\ \mathit{ch}\ :\ a\ \times\ \beta}$$

$$\frac{\Gamma_1\ \uplus\ \Gamma_2\ \simeq\ \Gamma \qquad \Gamma_1\ \vdash\ \mathit{ch} : !\ a\ .\ \beta \qquad \Gamma_2\ \vdash\ e\ :\ a}{\Gamma\ \vdash\ \mathit{send}\ \mathit{ch}\ e\ :\ \beta}$$

To ensure type safety, subsequent communication *must* use the updated channel reference. Reusing the channel endpoint φ_r of type $!\ nat\ .\ end$ (after sending the number on it) violates *session fidelity*—i.e., the communication was not according to the protocol described by the channel's session type—and breaks type safety. Alternatively, discarding the channel endpoint before sending a value on it leaves the other thread waiting, thus causing the program to be stuck. Hence, to attain type safety, one must ensure that channel references, or values in general, are used linearly—i.e., exactly once. Linearity is visible in the separation of the lexical context Γ into two disjoint parts Γ_1 and Γ_2 in the rule T-SEND (written $\Gamma_1 \uplus \Gamma_2 \simeq \Gamma$), and is further enforced by limiting the shape of the context in the rules for literals and variables:

$$\frac{}{\epsilon\ \vdash\ n\ :\ \mathit{nat}} \qquad \frac{}{(x\ :\ a)\ \vdash\ x\ :\ a}$$

2.2 The Language LTLC_{ref}

A channel endpoint is an example of a *reference that admits strong update*: a reference that can change type when operated upon. It is well-known that strong update is incompatible with sharing, and consequently we find linear type systems in other languages with strong update, such as a linear lambda calculus with strong references (LTLC_{ref}). We will use LTLC_{ref} as a case study, to demonstrate the negative impact of a linear type system and of linear references on the clarity of an intrinsically-typed interpreter.[1]

We formalize the typed syntax of LTLC_{ref} as an inductive family Exp in Agda. The type Exp of expressions is indexed by a syntactic type Ty[2] and a typing context Ctx:

```
data Ty where
  unit  : Ty
  ref   : Ty → Ty
  prod  : Ty → Ty → Ty
  _—∘_  : Ty → Ty → Ty
Ctx = List Ty

data Exp : Ty → Ctx → Set where (...)
  var   : Exp a [ a ]
  lam   : Exp b (a :: Γ) → Exp (a —∘ b) Γ
  app   : Exp (a —∘ b) Γ₁ → (Γ₁ ⊎ Γ₂ ≃ Γ) →
          Exp a Γ₂ → Exp b Γ
  pair  : Exp a Γ₁ → (Γ₁ ⊎ Γ₂ ≃ Γ) → Exp b Γ₂ →
          Exp (prod a b) Γ
  ref   : Exp a Γ → Exp (ref a) Γ
  swap  : Exp (ref a) Γ₁ → (Γ₁ ⊎ Γ₂ ≃ Γ) → Exp b Γ₂ →
          Exp (prod a (ref b))
  del   : Exp (ref unit) Γ → Exp unit Γ
```

The type family combines the grammar of the language with its typing rules. Note that some expressions (the introduction and elimination of **unit**, and the elimination of pairs) are omitted for brevity. Our representation of the syntax is informed by the typing rules of the language. Particularly, we followed the linear treatment of binders and contexts in the typing rules. The *context separation* $\Gamma_1 \uplus \Gamma_2 \simeq \Gamma$, a ternary relation, is inductively defined as an order preserving interleaving of lists. The representation of lexical variables **var** is nothing more than the observation that the context is a singleton. This nameless representation of binders is known as the *co-de-Bruijn* representation [2, 26]. Whereas in a de-Bruijn representation of binding, the choice between variables in scope is delayed until the leaves of the syntax tree, in a co-de-Bruijn representation, the choice is made at the earliest opportunity. That is, variables are only kept in the context of a subtree if they are used there. Hence, all of the information about how to dereference a name is captured in the context separation witnesses $\Gamma_1 \uplus \Gamma_2 \simeq \Gamma$.

At run time we have a *store* of values, typed by a store type ST. The values can be referenced and updated, even if that changes their type (i.e., strong update). We follow the

[1] Although LTLC_{ref}'s linear references are too strict for practical purposes, it is useful for demonstrating the problem and our solution in a simple setting.
[2] The underscores in a name defined in Agda denote where the arguments go. For example, the type for linear functions _—∘_ is used as $a \multimap b$.

specification of static variables and also use a co-de-Bruijn representation for references, thus only revealing to a value the part of the store that it refers to:

```
ST = List Ty

data Val : Ty → ST → Set where (...)
  ref  : Val a [ a ]
  pair : Val a Φ₁ → (Φ₁ ⊎ Φ₂ ≃ Φ) → Val b Φ₂ →
         Val (prod a b) Φ
  clos : Exp b (a :: Γ) → Env Γ Φ → Val (a ⊸ b) Φ
```

Here, Env is the *environment* captured by the closure value. Both environments and stores are essentially typed, heterogeneous lists of values. However, the values in these lists can themselves be references, such that these lists (1) must maintain internal separation, and (2) must be separated from any values outside of it. We index these list by the typing of their elements (Ψs), and the part of the store that they consume (Φs):

```
data All : List Ty → ST → Set where
  nil  : All [] []
  cons : Val a Φ₁ → (Φ₁ ⊎ Φ₂ ≃ Φ) → All Ψ Φ₂ →
         All (a :: Ψ) Φ
Env   = All
Store = All
```

2.3 The Type of a Linear Interpreter

By typing the source language, the target languages, and the semantic components, we can specify the type of an intrinsically-typed interpreter:

```
eval : Exp a Γ →
       Env Γ Φ₁ → (Φ₁ ⊎ Φ₂ ≃ Ψ) → Store Ψ Φ₂ →
       ∃ λ Ψ₂ Φ₃ Φ₄ →
         Store Ψ₂ Φ₃ × (Φ₃ ⊎ Φ₄ ≃ Ψ₂) × Val a Φ₄
```

This type expresses: (1) type preservation: the value type matches the expression type, (2) disjoint store consumption: the Φs that denote consumption of the inputs and outputs of eval are separated, and (3) absence of dangling references: the usages sum up to the entire content of the store. The existential quantification on the right-hand side is due to the fact that evaluation may add, remove or change cells in the store in statically unknown ways.

Unfortunately, while correct as a top-level specification, the type of eval is not strong enough to evaluate effectful expressions inside a store that contains more than what the expression itself refers to. This occurs for example when we attempt to implement function application app f e: we want to evaluate the function expression f in the part of the environment that belongs to it. The remainder of the environment (for e) becomes a *frame* of the computation that is not passed to eval, but must remain separated from the values that the recursive call does manipulate. Consequently, the consumption of the environment Φ_1 and the store Φ_2 do not necessarily add up to the complete store type Ψ. We can generalize the type to additionally include a disjoint frame Φ_f. Crucial is that the frame is *preserved* by the computation, such that pointers in the frame are not invalidated by eval:

```
eval : Exp a Γ →
       Env Γ Φ₁ → (Φ₁ ⊎ Φ₂ ≃ Φ) → Store Ψ Φ₂ →
       (Φ ⊎ Φ_f ≃ Ψ) →
       ∃ λ Ψ₂ Φ₃ Φ₄ Φ₅ →
         Store Ψ₂ Φ₃ × (Φ₃ ⊎ Φ₄ ≃ Φ₅) × Val a Φ₄
         × (Φ₅ ⊎ Φ_f ≃ Ψ₂)
```

The appearance of separation of the store type using _⊎_≃_ in the interpreter type already obscures intention. Implementations of this type are not any better: even if we look past pattern matching on an 8-tuple, the separation witnesses of the left- and right-hand side are not merely being passed around. For recursive calls to eval the separation witnesses have to be reassociated. Recursive evaluation may update the store, resulting in more separation witnesses and more proof obligations. The overhead of these proof terms obscures the computational content of the interpreter and makes writing them a tedious exercise. This manipulation of separation proofs in intrinsically-typed semantics of linear languages has previously been identified as a key issue of the approach [37].

2.4 Key Idea

The presence of an excessive amount of proof terms comprises a big gap in clarity between the typed interpreter of STLC from Fig. 1 and an interpreter for a linearly-typed language. This paper bridges that gap. The key idea is to use separation logic to build monadic abstractions with which we hide the explicit separation of the store typing from the type and the implementation of the above evaluation function:

```
eval : Exp a Γ → ϵ[ ReaderT State Γ [] (Val a) ]
eval (app (f ⟨ σ₁ ⟩ e)) = do
  clos b env   ← frame σ₁ (eval f)
  v ⟨ σ₂ ⟩ env ← eval e &⟨ ⊎-id^l ⟩ env
  empty        ← append (cons (v ⟨ σ₂ ⟩ env))
  eval b
```

We have yet to define many of the types and operations in this snippet, but it gives an impression of how our results contrast with the straightforward approach represented by the completely explicated evaluation signature above. The type and implementation resemble untyped monadic interpreters of functional languages. Monadic operations like append and frame manipulate the semantic components (such as the evaluation environment) in ways that preserve linearity (§3). The strong guarantees of these operations can be made precise in their types without drowning the computational content in specifications of separation.

The proof work of ensuring linear usage of values is mostly done by the implementation of the semantic operations, and by the monad that is being employed underneath the do-notation. To witness that values (such as in *env*) are separated from values returned from subsequent evaluation we can use *monadic strength* _&⟨_⟩_ (§3.2). By explicitly carrying the *env* across the bound operation using monadic strength, we get the additional separation witness σ_2, which we need to prove that the value v is separated from the closure environment. The only remaining explicit proof term is the use of the left identity law of separation ($\uplus$-id$^{\text{l}}$). In the next sections we will build the abstractions that make this possible.

3 Proof Relevant Separation Logic

In §2 we saw that explicitly writing out all typing contexts and the separation between them quickly becomes a tedious exercise, which obscures the clarity of the specification and implementation. In the last shown signature of eval (§2.4), we addressed this problem by using abstractions based on separation logic, which allowed us to omit all separation entirely. In this section we first recall the basic connectives of *separation logic* (§3.1). We then show that separation logic can be used to consisely program an intrinsically-typed interpreter for LTLC—the linearly-typed lambda calculus (§3.2). In §4 we will extend this to the state operations of LTLC$_{\text{ref}}$, and in §5 we will extend this to the operations for concurrency and communication of LTLC$_{\text{ses}}$.

3.1 Separation Logic in Agda

To specify separation of contexts (Ctx) and store types (ST), we define a classic model of separation logic using *predicates*:

```
Pred A = A → Set

_⇒_ : Pred A → Pred A → Pred A
P ⇒ Q = λ a → P a → Q a

∀[_] : Pred A → Set
∀[ P ] = ∀ {a : A} → P a

data _∪_ (P Q : Pred A) : Pred A where
  inj₁ : ∀[ P ⇒ (P ∪ Q) ]
  inj₂ : ∀[ Q ⇒ (P ∪ Q) ]
```

On predicates over lists of types, we define the standard connectives from separation logic [29]: *separating conjunction* $*$, its unit Emp, and *magic wand* (or separating implication) $-\!\!*$:[3]

```
record _*_ (P Q : Pred (List Ty))
           (Φ : List Ty) : Set where
  constructor _⟨_⟩_
  field
    {Φl Φr} : List Ty
    px      : P Φl
    split   : Φl ⊎ Φr ≃ Φ
    qx      : Q Φr

data Emp : Pred (List Ty) where
  empty : Emp []

_-*_ : (P Q : Pred (List Ty)) → Pred (List Ty)
(P -* Q) Φ₁ = ∀ {Φ₂ Φ₃}
  → (Φ₁ ⊎ Φ₂ ≃ Φ₃) → P Φ₂ → Q Φ₃
```

We use List Ty so we can use these connectives on contexts (Ctx) and store types (ST), which are both defined as such. The separating conjunction $*$ allows us to express concisely that two predicates are conjoined disjointly. The $*$ associates to the left, and binds more tightly than both the magic wand $-\!\!*$ and $\Rightarrow$. Note that the quantification in the definition of magic wand $-\!\!*$ is important: a magic wand is a function, closing over some part of the resource Φ_1. The wand accepts an element of P that consumes a *disjoint* part Φ_2. The result of the function, in Q, must consume the sum Φ_3 of those parts, such that nothing is lost. The magic wand is the adjoint of the separating conjunction:

```
uncurry : ∀[ (P * Q) ⇒ R ]   → ∀[ P ⇒ (Q -* R) ]
curry   : ∀[ P ⇒ (Q -* R) ] → ∀[ (P * Q) ⇒ R ]
```

A universally closed function (∀[$P \Rightarrow Q$]) is the same as a wand that does not close over any resources (ϵ[P $-\!\!*$ Q]), where:

```
ϵ[_] : Pred (List Ty) → Set
ϵ[ P ] = P []
```

We often prefer to use the former, because its use and implementation involve one less separation witness.

Finally, we also make use of the predicate One a that asserts that the context is a singleton containing a:

```
data One : Ty → Pred (List Ty) where
  one : One a [ a ]
```

3.2 Typing Linear Syntax and Functions

We now combine separation logic with the technique of Allais et al. [3] and McBride [26] to specify the typed syntax of a linearly-typed lambda calculus (LTLC) concisely:

```
data Exp : Ty → Pred Ctx where (...)
  var  : ∀[ One a ⇒ Exp a ]
  lam  : ∀[ ((cons a) ⊢ Exp b) ⇒ Exp (a ⊸ b) ]
  app  : ∀[ Exp (a ⊸ b) * Exp a ⇒ Exp b ]
  pair : ∀[ Exp a * Exp b ⇒ Exp (prod a b) ]
```

[3] The definition of $*$ uses a record to hide the existential quantification over the resources Φ_l and Φ_r. The ternary constructor of the record has the left projection, the separation witness, and the right projection as arguments

Other constructors are omitted, but they follow the same pattern. We use the operator $\vdash$ on predicates [3] to extend the implicit context in the **lam** constructor:

$$\_ \vdash \_ : (A \to B) \to \mathsf{Pred}\ B \to \mathsf{Pred}\ A$$
$$f \vdash P = P \circ f$$

Although technically the values of LTLC are pure and do not consume any runtime resource (like a store), we will define values (and thus also environments) as predicates over store types ST. This prepares us for adding references to the language, which we will do in the next section.[4] Because references, like variables, can also be encoded using the co-de-Bruijn representation, we can use the exact same approach to specify the run time objects—i.e., values and environments. For example:

$$\textbf{data}\ \mathsf{Val} : \mathsf{Ty} \to \mathsf{Pred\ ST}\ \textbf{where}\ (\ldots)$$
$$\quad \textbf{clos} : \mathsf{Exp}\ b\ (a :: \Gamma) \to \forall[\ \mathsf{Env}\ \Gamma \Rightarrow \mathsf{Val}\ (a \multimap b)\ ]$$
$$\quad \textbf{pair} : \forall[\ \mathsf{Val}\ a * \mathsf{Val}\ b \Rightarrow \mathsf{Val}\ (\textbf{prod}\ a\ b)\ ]$$

The only effect in this linear language is *reading* a variable from the environment. We specify the semantics of this effect by means of a *linear* Reader *predicate transformer*. Because of linearity, reading a value removes that value from the environment. Reader computations return the part of the environment that is unread, as it cannot be discarded:

$$\mathsf{Reader} : \mathsf{Ctx} \to \mathsf{Ctx} \to \mathsf{Pred\ ST} \to \mathsf{Pred\ ST}$$
$$\mathsf{Reader}\ \Gamma_1\ \Gamma_2\ P = \mathsf{Env}\ \Gamma_1 \mathrel{-\!\!*} \mathsf{Env}\ \Gamma_2 * P$$

The indices Γ_1 and Γ_2 of Reader denote the shape of the environment before and after running the computation, respectively. The magic wand $-\!\!*$ denotes that the environment that the reader computation expects, must be separated from any value that the computation closes over, while the separating conjunction $*$ means that the returned environment and value in P are separated.

Using Reader we can type the interpreter for LTLC expressions analogous to a monadic interpreter for STLC:

$$\mathsf{eval} : \mathsf{Exp}\ a\ \Gamma \to \epsilon[\ \mathsf{Reader}\ \Gamma\ []\ (\mathsf{Val}\ a)\ ]$$

The indices Γ and ϵ of Reader denote that the computation consumes an environment of shape Γ entirely. The fact that the type is closed with $\epsilon[\_]$ denotes that it does not depend on any resources outside of the reader computation. Similarly we can define the operation that allows reading the entire environment:

$$\mathsf{ask} : \epsilon[\ \mathsf{Reader}\ \Gamma\ []\ (\mathsf{Env}\ \Gamma)\ ]$$

By indexing reader computations with a pre and post environment, we allow for computations that do not consume the entire environment. Consequently, we can also specify operations that extend the environment:

$$\mathsf{prepend} : \forall[\ \mathsf{Env}\ \Gamma_1 \Rightarrow \mathsf{Reader}\ \Gamma_2\ (\Gamma_1 +\!\!+ \Gamma_2)\ \mathsf{Emp}\ ]$$
$$\mathsf{append} : \forall[\ \mathsf{Env}\ \Gamma_1 \Rightarrow \mathsf{Reader}\ \Gamma_2\ (\Gamma_2 +\!\!+ \Gamma_1)\ \mathsf{Emp}\ ]$$

Furthermore, we can use the fact that context separation implies environment separation to frame reader computations inside larger environments:

$$\mathsf{frame} : \Gamma_1 \uplus \Gamma_3 \simeq \Gamma_2 \to$$
$$\quad \forall[\ \mathsf{Reader}\ \Gamma_1\ []\ P \Rightarrow \mathsf{Reader}\ \Gamma_2\ \Gamma_3\ P\ ]$$

In an untyped setting we would use the fact that Reader is a monad in Set to compose these operations into larger computations. However, it is a priori unclear in what sense Reader is a monad. A first attempt is to implement the interface of a monad on predicates:

$$\mathsf{return} : \forall[\ P \Rightarrow \mathsf{Reader}\ \Gamma\ \Gamma\ P\ ]$$
$$\mathsf{bind} : \forall[\ P \Rightarrow \mathsf{Reader}\ \Gamma_2\ \Gamma_3\ Q\ ] \to$$
$$\quad \forall[\ \mathsf{Reader}\ \Gamma_1\ \Gamma_2\ P \Rightarrow \mathsf{Reader}\ \Gamma_1\ \Gamma_3\ Q\ ]$$

Unfortunately, this bind is not strong enough by itself to implement an interpreter for LTLC. This can be seen more easily if we recall the equivalence between pointwise-lifted functions and magic wands, and rewrite the bind as:

$$\mathsf{bind} : \epsilon[\ P \mathrel{-\!\!*} \mathsf{Reader}\ \Gamma_2\ \Gamma_3\ Q\ ] \to$$
$$\quad \epsilon[\ \mathsf{Reader}\ \Gamma_1\ \Gamma_2\ P \mathrel{-\!\!*} \mathsf{Reader}\ \Gamma_1\ \Gamma_3\ Q\ ]$$

That is: we can only bind functions that do not close over any resources. This is insufficient, for example, for interpreting binary expressions (e.g., function application), where bound continuations close over previously computed values. To remedy this, we can *internalize* the bind:

$$\mathsf{bind} : \forall[\ (P \mathrel{-\!\!*} \mathsf{Reader}\ \Gamma_2\ \Gamma_3\ Q) \Rightarrow$$
$$\quad (\mathsf{Reader}\ \Gamma_1\ \Gamma_2\ P \mathrel{-\!\!*} \mathsf{Reader}\ \Gamma_1\ \Gamma_3\ Q)\ ]$$

This *internal bind* is strong enough to implement the interpreter, but the use of magic wands also has a downside: the fact that a magic wand takes a separation witness as an additional argument, means that every step in the interpreter will receive a proof term that needs to be passed around. As a side effect, we can also not use Agda's builtin do-notation, which expects a bind with the usual arity.

Fortunately, there is third option: a formulation of the monadic structure with the expressiveness of the internal bind, but the convenience of the external bind. This formulation, which has been used before by Bach Poulsen et al. [6] to program with monads in a category of monotone predicates in Agda, uses the fact that a monad with an internal bind is equivalent to a strong monad [23, 27]. That is, a monad with an external bind and *monadic strength*:

$$\mathsf{str} : \forall[\ \mathsf{Reader}\ \Gamma_1\ \Gamma_2\ P * Q \Rightarrow \mathsf{Reader}\ \Gamma_1\ \Gamma_2\ (P * Q)\ ]$$

We abbreviate $\mathsf{str}\ (mp \langle\ \sigma\ \rangle\ qx)$ as $mp\ \&\langle\ \sigma\ \rangle\ qx$. Using monadic strength, we do not have to close the argument of a bind over any outside resources. Instead, we pass them to the bound function *through* bind. The result is that despite the rich and complicated underlying structure of separation

[4] In practice we would make it parametric in the runtime resource using the generalized separation logic described in §4.1.

logic, we can program with these monads as regular monads in Agda.[5] The following excerpt of the LTLC interpreter from the accompanying code [34] of this paper shows how we combine reader operations to interpret linear function application:

```
eval (app (f ⟨ σ₁ ⟩ e)) = do
  clos b env   ← frame σ₁ (eval f)
  v ⟨ σ₂ ⟩ env ← eval e &⟨ ⊎-id^l ⟩ env
  empty        ← append (cons (v ⟨ σ₂ ⟩ env))
  eval b
```

Using frame we first evaluate the function expression in the part of the environment prescribed by the context separation σ_1. By dependent pattern matching we obtain the corresponding closure clos b env. This leaves exactly the part of the environment that is required to evaluate the argument e. To get evidence that the previously obtained closure environment is separated from the resulting value v, we use monadic strength. This evidence is required to construct the environment for the body b. The indices of the linear Reader monad ensure that we have constructed an environment that matches the context for the body. The proof term uses the fact that separation of lists has [] as a left identity ($\uplus$-id^l). We discuss this and other laws of separation in detail in §4.1.

With minimal proof overhead we have implemented a linear function application and proven it type safe. Moreover, any violation of any of the properties that we want to hold, is caught by Agda during the development of the interpreter. For example, appending v and env in the wrong order would be caught because the shape of the environment would not match the context of b. Or, if we were to forget v and put any other value in its place, Agda would not accept the definition, because some resources were lost.

In the process we have developed a reusable abstraction for linear reader effects. The actual implementation of the Reader monad is parameterized over an abstract value predicate. Moreover, as its counterpart in Set, Reader be generalized to a strong monad transformer ReaderT $M\ \Gamma_1\ \Gamma_2\ P$ = Env Γ_1 $-\!\!*$ M (Env Γ_2 $*$ P) for any other strong monad M, which we will use in the next section to extend the interpreter to the state operations of LTLC$_{\text{ref}}$.

4 Intrinsically-Safe Memory

In §2.3 we noted that the explicit signature of a definitional interpreter expresses not only type preservation and disjoint consumption of resources, but also memory safety (i.e., there are no dangling references). These last two properties were encoded together into a single observation that must hold on both sides of the evaluation signature: the total amount consumed, including the frame and the consumption of the store itself, must combine exactly to what the store provides.

[5] **do** $v \leftarrow m_1; m_2$ desugars to $m_1 \gg\!= (\lambda\ v\ \rightarrow\ m_2)$ where v can be a pattern.

The separation logic that we presented is by itself insufficient to express this notion of memory safety. In particular, while the $*$ can express the disjoint distribution of *demand* for store cells, it does *not* provide a means to equate this with the *supply* of the actual store. In this section we develop the abstractions to *balance* supply and demand.

We will do this by using predicates not over store types ST, but over a novel resource Market ST. To that end, we first generalize the separation logic connectives that we defined in the previous section to PRSAs—*proof-relevant separation algebras* (§4.1). We then define the Market PRSA (§4.2), and show how it can be used for dependently-typed programming (§4.3), and finally interpret the linear state operations of LTLC$_{\text{ref}}$ (§4.4).

4.1 Proof-Relevant Separation Algebras (PRSAs)

In §3 we constructed a simple separation logic for a model of lists of types and their interleavings. To generalize the separation logic we consider whether our model is an instance of a *separation algebra* [8, 14]: a class of models that yield well-behaved separation logics. Dockins et al. [14] give an axiomatization of a separation algebra (SA) based on a ternary *join* relation. A carrier with a join relation is a separation algebra if it is functional, cancellative, commutative, associative, and has a unit ϵ.

Although list interleavings are commutative, associative, and have a unit ϵ = [], they are *neither* cancellative, nor functional. Functionality means that $x \uplus y \simeq z_1$ and $x \uplus y \simeq z_2$ imply that z_1 and z_2 are equal, and cancellativity means that $x_1 \uplus y \simeq z$ and $x_2 \uplus y \simeq z$ imply that x_1 and x_2 are equal. Lacking functionality and cancellativity is ultimately due to the application. Dockins et al. [14] use ternary relations as a substitute for partial functions: given an arbitrary x and y, there is no guarantee that a join z exists such that $x \uplus y = z$. For context interleavings the opposite is the case: there is not just one, but many possible interleavings to choose from. For their use in code-Bruijn encodings of syntax, the choice is also relevant: it determines the binding in an expression. Consider for example the following expression:

lam (lam (pair (var one ⟨ σ ⟩ var one)))

where _⟨_⟩_ is the constructor of separating conjunction $*$. This term can denote either $\lambda\ (x : a)\ .\ \lambda\ (y : b)\ .\ (x, y)$ or $\lambda\ (x : a)\ .\ (\lambda\ (y : b)\ .\ (y, x))$, depending on the two choices for the separation witness σ that distributes the context:

1. σ : [a] $\uplus$ [b] $\simeq$ $a :: b ::$ [], or,
2. σ : [b] $\uplus$ [a] $\simeq$ $a :: b ::$ [].

If the types a and b are the same, then one really has to look at the proof term to find out how the variables are bound. To accommodate for models of separation logic with a proof-relevant notion of separation, we propose a generalization of SAs that we coin *proof-relevant separation algebras* (PRSAs).

A PRSA consists of a carrier C, a ternary separation relation $(\_ \uplus \_ \simeq \_)$ on C, and a unit $\epsilon : C$, satisfying:

$$\uplus\text{-comm} : (a \uplus b \simeq c) \to (b \uplus a \simeq c)$$
$$\uplus\text{-assoc} : (a \uplus b \simeq ab) \to (ab \uplus c \simeq abc) \to \exists\, \lambda\, bc \to (a \uplus bc \simeq abc) \times (b \uplus c \simeq bc)$$
$$\uplus\text{-id}^{l} : \epsilon \uplus a \simeq a$$
$$\uplus\text{-id}^{-l} : (\epsilon \uplus a \simeq b) \to a \equiv b$$

A PRSA is said to be total if it has an append operation $(\_ \bullet \_ : C \to C \to C)$ that forms a monoid w.r.t. ϵ, and in addition satisfies:

$$\uplus\text{-}\bullet_{l} : (b \uplus c \simeq bc) \to ((a \bullet b) \uplus c \simeq (a \bullet bc))$$

For monoidal PRSAs we get the fact that at least one join exists for every pair of elements:

$$\uplus\text{-}\bullet : a \uplus b \simeq (a \bullet b)$$

Instead of dropping the functionality axiom from the notion of SAs by Dockins et al. [14], one could use a weaker notion of equality (i.e., a setoid that expresses equality modulo reordering). However, there is little hope that we could write proof-obligation free interpreters that way: state-of-the-art dependently-typed programming languages like Agda do not have native support for programming with such weaker notions of equality. Moreover, while the previously presented notion of separation has a clear candidate for an appropriate notion of equality, we will also see examples of PRSAs where this is not the case (§4.2 and §5.1)

We use several instances of PRSAs in this paper. The separation logic in §3.1, and the Reader monad transformer discussed in §3.2 generalize easily to arbitrary PRSAs [34], since the definition does not make use of anything specific to list separation.

4.2 The Market PRSA

Recall (from §2.2) that Store $\Psi\ \Phi$ is a store with cells typed by Ψ, and a combined consumption Φ. We will call Ψ the *supply* of the store, and Φ its *demand*. The runtime invariant for linearity is that the total demand of all *consumers* of the store, must add up to the supply. For example, looking at the explicit type of the semantic action newref to implement the semantics of **ref**, we have three consumers: a frame, a value and the input store:

$$\text{newref} : (\Phi_1 \uplus \Phi_2 \simeq \Phi) \to (\Phi \uplus \Phi_f \simeq \Psi_1) \to \text{Val}\ a\ \Phi_1 \to \text{Store}\ \Psi_1\ \Phi_2 \to \exists\, \lambda\, \Psi_2\ \Phi_3 \to \text{Store}\ \Psi_2\ \Phi_3 \times \ldots$$

Their demand–Φ_f, Φ_1 and Φ_2 respectively—add up to the total left-hand side supply Ψ_1. We want to hide all of the separation accounting. To that end, we need a PRSA that takes care of both supply and demand. Although we can construct a product PRSA, this will not suffice, because it will account for supply and demand in isolation of each other, and not enforce the top-level equation between them.

Additionally, if we account the supply and demand separately, the left- and right-hand side supply for an operation like newref is not going to balance, because both supply and demand increase.

The key to balancing the equation is not to look at supply and demand separately, but at their sum. The operation newref is memory-safe, because it increases supply and demand *equally* when it returns a pointer to the freshly allocated cell.

To achieve this we construct a PRSA Market A that tracks the *net supply* of the PRSA A, in the presence of multiple consumers ($\downarrow$) and *at most one* supplier ($\uparrow$) for A:

data Market $(A : \text{Set}) : \text{Set}$ **where**
$\uparrow : (s : A) \to \text{Market}$
$\downarrow : (d : A) \to \text{Market}$

By restricting to a single supplier we can apply a simple accounting method operating in two modes, formalized as a separation relation $\_ \uplus \_ \simeq_M \_$ on Market A with three constructors. If no supplier is present, then we are simply adding up demand (let $\_ \uplus \_ \simeq_A \_$ denote the separation on A):

$$\downarrow_{lr} : (d_1 \uplus d_2 \simeq_A d) \to (\downarrow d_1) \uplus (\downarrow d_2) \simeq_M (\downarrow d)$$

Or, if a supplier is present, we subtract the demand from it, tracking how much supply is left over:

$$\uparrow_{l} : (s_2 \uplus r \simeq_A s_1) \to (\uparrow s_1) \uplus (\downarrow r) \simeq_M (\uparrow s_2)$$
$$\uparrow_{r} : (r \uplus s_2 \simeq_A s_1) \to (\downarrow r) \uplus (\uparrow s_1) \simeq_M (\uparrow s_2)$$

Importantly, there is no constructor that permits a supplier to be present on *both* sides. This ensures that if a single supplier is present, then every reference is bound in that supplier.

The type Market A, together with this definition of separation, is a PRSA for every PRSA A, and has a unit $\downarrow \epsilon$.

4.3 Programming with the Market PRSA

The predicates Store, Val, and One can be lifted into Market ST, provided that we clarify their role as suppliers or consumers. Suppliers, like Store, that are indexed by both supply and demand, can be lifted under the side-condition that their internal demand does not exceed supply:

data $\bullet\ (P : A \to \text{Pred}\ A) : \text{Pred}\ (\text{Market}\ A)$ **where**
supplier $: P\ s_1\ d \to d \uplus s_2 \simeq_A s_1 \to \bullet\ P\ (\uparrow s_2)$

Consumers, like Val and One, can be lifted as follows:

data $\circ\ (P : \text{Pred}\ A) : \text{Pred}\ (\text{Market}\ A)$ **where**
consumer $: P\ d \to \circ\ P\ (\downarrow d)$

Both $\bullet$ and $\circ$ associate more tightly than the arrow and wand. Using these type formers, we can now consisely express the type- and memory-safe signature of an operation newref that allocates a new memory cell:

$$\text{newref} : \forall[\ \circ\ (\text{Val}\ a) \Rightarrow \bullet\ \text{Store} \multimap \circ\ (\text{One}\ a) * \bullet\ \text{Store}\ ]$$

The supply and demand equation balances: the fact that the store is extended by a single new cell of type a, is offset by the returned pointer $\bigcirc$ (One a).

We now define a State monad, taking special care to define it as an endofunctor in Pred A rather than Pred (Market A).[6] We accomplish this by making explicit that any wand that takes a *supplier* as an argument, can itself only be a *client*. That is, for any f : ($\bullet$ P $-\!\!*$ Q) Φ_1 we must have that $\Phi_1 \equiv \downarrow \Phi_2$ for some Φ_2. This holds, because the resource that the wand uses must be separated from the resource used by the argument, which is fixed to be a supplier. Without loss of generality, we can thus define the State monad as an endofunctor as follows:

```
State     : Pred A → Pred A
State P a = (● Store −* ○ P * ● Store) (↓ a)
```

This predicate transformer is a strong monad for any notion of state indexed by both supply and demand, and, as before, we can generalize it to a monad transformer StateT. The operations newref, read, and update can be implemented if the state is instantiated to Store, but are parametric in the type of values, and the notion of separation between them:

```
newref : ∀[ Val a ⇒ State (One a) ]
read   : ∀[ One a ⇒ State (Val a) ]
write  : ∀[ One a ⇒ Val b −* State (One b * Val a) ]
update : ∀[ One a ⇒ (Val a −* State (Val b)) −*
            State (One b) ]
```

Note that the signature of read tells us that the cell pointed to by the passed reference is destroyed, as the reference is not returned from this operation. In contrast, the write operation keeps the cell, returning a pointer with its new type, and also the value that used to be in it. The update operation is an example of a higher-order operation, for which the use of a $-\!\!*$ is a necessity.

This strong monad can be programmed with as with any other monad. Because we defined State as an endofunctor in Pred A, neither the operations of this monad, nor the monadic interface, mention $\bullet$ or $\bigcirc$. As a consequence, the user of this interface does not need to be aware of the Market structure that is used for the accounting between the consumers and the supplier.

4.4 An Interpreter for LTLC$_{\text{ref}}$

To interpret the entirety of LTLC$_{\text{ref}}$, we first extend the values of LTLC with a constructor for references:

```
data Val : Ty → Pred ST where (...)
  ref : ∀[ One a ⇒ Val (ref a) ]
```

[6] Alternatively we may define it as a monad *relative* [4] to the separation preserving functor $\bigcirc$. One still has to take special care to avoid the use of $\bigcirc$ in the interface, such that the user of the monad does not need to wrap/unwrap values passed to/returned from the monad.

We instantiate the linear State monad for stores over these values, and nest it inside the linear Reader monad transformer. The state operations of LTLC$_{\text{ref}}$ are then implemented as follows:

```
eval : Exp a Γ → ϵ[ ReaderT State Γ [] (Val a) ]
eval (ref e) = do
  v ← eval e
  r ← liftM (newref v)
  return (ref r)

eval (swap (e₁ ⟨ σ ⟩ e₂)) = do
  ref ra       ← frame σ (eval e₁)
  vb ⟨ σ₁ ⟩ ra ← eval e₂ &⟨ ⊎−id¹ ⟩ ra
  rb ⟨ σ₂ ⟩ va ← liftM (write ra (⊎−comm σ₁) vb)
  return (pair (va ⟨ ⊎−comm σ₂ ⟩ ref rb))

eval (del e) = do
  ref r ← eval e
  liftM (read r)
```

The implementation of swap first interprets the left- and right-hand side sub-expressions, from which we obtain a reference and a value. We also get a witness that these are separated by using monadic strength in the second step. This witness is needed in the subsequent evaluation step. Using liftM, we lift the write operation of the State monad into the reader transformer. From the write, we again obtain a reference and a value, separated according to σ_2. All that remains is to construct the pair value, which we return.

After context separation, we have now constructed a second proof-relevant separation algebra: Market A, formalizing the accounting of supply and demand of some underlying resource A. We have used this to construct a monad State with the familiar semantic operations on typed heaps, transported to the linear setting. Remarkably, the complexities of the underlying accounting is hidden from the user entirely.

5 Intrinsically-Safe Session Types

We now turn to the session-typed language LTLC$_{\text{ses}}$, and its operations for spawning threads and conducting communication. We stage the interpretation into two layers. The first layer interprets the expression language into *command trees* [18], interleaving the communication and threading commands with thunked evaluation of the expression language (§5.1). The second layer interprets these command trees, thus implementing the scheduling and communication semantics (§5.2).

To implement these stages, we apply the abstractions that we developed so far. The syntax of the language, being an extended linearly-typed lambda calculus, again uses the list PRSA to deal with co-de-Bruijn variables. The monad for the expression language interpreter nests a novel free monad inside the reader transformer. For the command semantics, we reuse the Market PRSA and the state monad transformer

StateT. We combine those with an Error monad to handle the partiality of receiving messages, and we use a new PRSA within Market to split channels into their endpoints.

The benefit of staging the interpretation, is that we clearly separate the runtime (i.e., the communication and concurrency model) from the language. The first and second stage are unaware of each other, and are thus independent and reusable. We will only implement round-robin scheduling and asynchronous communication, but one could swap this semantics of command trees for other schedulers and/or a synchronous communication model, without adjusting the expression language interpreter.

5.1 Stage I: Interpreting the Expression Language

We embed the syntax of the concurrency and communication primitives in Agda, starting by extending the types of LTLC with a type for *channel endpoint references* $\triangleright\ \alpha$:

```
mutual
  data STy : Set where
    end : STy
    _?_ : Ty → STy → STy
    _!_ : Ty → STy → STy
  data Ty : Set where (...)
    ▷ : (α : STy) → Ty
```

Let $\triangleright$ bind weaker than the session type constructors. Session type constructors for sending and receiving are written in an infix style, for example $a\ !\ \beta$ for the protocol that sends an a and continues as β. We again use greek variables to denote session types, and we write α^{-1} for the dual of a session type.

We then extend LTLC with the five primitive operations for spawning threads, and conducting communication. The communication primitives all act on *channel endpoint references* and require the right protocol shape:

```
data Exp : Ty → Pred Ctx where (...)
  fork  : ∀[ Exp (unit ⊸ unit) ⇒ Exp unit ]
  mkch  : ∀ α → ∀[ Exp (prod (▷ α) (▷ α ⁻¹)) ]
  recv  : ∀[ Exp (▷ a ? β) ⇒ Exp (prod (▷ β) a) ]
  send  : ∀[ Exp a ∗ Exp (▷ a ! β) ⇒ Exp (▷ β) ]
  close : ∀[ Exp (▷ end) ⇒ Exp unit ]
```

We will implement a semantics for these primitive operations in §5.2. The runtime will maintain a collection of channels, similar to how the runtime of LTLC_{ref} maintained a collection of cells. Unlike cells however, open channels have two handles which can be referenced independently: one handle for each endpoint. Hence, we model the session (or runtime) context SCtx as a list of *runtime types* [16], which can be either a single endpoint, or an entire channel consisting of two typed endpoints:

```
data RTy : Set where
  ▷ᵣ    : STy → RTy
  chanᵣ : STy → STy → RTy
SCtx = List RTy
```

Although conceptually channel endpoints have dual types, in practice, for buffered communication, this may not be the case [16]. We explain this in more detail in §5.2.

Mere interleavings of SCtx do not describe all the ways that session contexts can be split. In particular, we have to account for the separation of channels into their respective endpoints. We model this using a PRSA Ends:[7]

```
data Ends : RTy → RTy → RTy → Set where
  lr : Ends (▷ᵣ a) (▷ᵣ b) (chanᵣ a b)
  rl : Ends (▷ᵣ b) (▷ᵣ a) (chanᵣ a b)
```

We then define a PRSA on lists of a type that can itself be split—i.e., interleavings with an additional constructor divide for making ends meet. For SCtx, we obtain:

```
data _⊎_≃_ : (xs ys zs : SCtx) → Set a where
  divide : Ends xˡ xʳ x → (xs ⊎ ys ≃ zs) →
           (xˡ :: xs) ⊎ (xʳ :: ys) ≃ (x :: zs)
  left   : Split xs ys zs → Split (z :: xs) ys (z :: zs)
  right  : Split xs ys zs → Split xs (z :: ys) (z :: zs)
  nil    : Split [] [] []
```

In the same way that values of LTLC_{ref} are predicates over a store type ST, values and other runtime objects of the session-typed language are predicates over SCtx. As before, references are typed in a co-de-Bruijn style:

```
data Val : Ty → Pred SCtx where (...)
  cref : ∀[ One (▷ᵣ α) ⇒ Val (▷ α) ]
```

Expressions are not interpreted to plain values, but to command trees [18, 36] with values at the leaves. Outgoing commands may contain channel endpoint references, and must therefore be separated from their continuation. This yields the following strong monad F P of command trees that will eventually return an instance of P:

```
data F (P : Pred A) : Pred A where
  pure   : ∀[ P ⇒ F P ]
  impure : ∀[ ∃[ Cmd ]∗ (λ c → δ c -∗ F P) ⇒ F P ]
```

Both (Cmd : Pred A) and (δ : Cmd Φ → Pred A) are parameters of this construction. The argument $\delta\ c$ of the continuation denotes the (dependently typed) response to the command c. The impure constructor of F uses a *dependent separating conjunction* to pair a command with its continuation, generalizing the separating conjunction:

[7] Technically Ends does not have an identity, and hence does not satisfy our definition of a PRSA. In the Agda library we distinguish PRSAs with and without an identity.

```
record ∃[_]∗_
  (P : Pred A) (Q : ∀ {Φ} → P Φ → Pred A)
  (Φ : A) : Set where
  field
    {Φl Φr} : A
    px      : P Φl
    sep     : Φl ⊎ Φr ≃ Φ
    qx      : Q px Φr
```

The type of impure captures the exchange of resources that occurs: when a command is issued, a thread gives away some resources via the command, and keeps the remainder enclosed in the continuation. The magic wand in the type of the continuation denotes that the thread may receive in response new resources, separated from what it already owned. Using impure, every command can be lifted to an F-computation:

```
⟪_⟫ : ∀ (c : Cmd Φ) → F (δ c) Φ
```

The commands that may appear in command trees are the effectful operations that are left abstract by the first stage of interpretation—i.e., the primitive operations for concurrency and communication:

```
data Cmd : Pred SCtx where
  forkc  :          ∀[ F (Val unit) ⇒ Cmd ]
  mkchc  : ∀ β →    ϵ[ Cmd ]
  sendc  : ∀ a β →  ∀[ One (▷r a ! β) ∗ Val a ⇒ Cmd ]
  recvc  : ∀ a β →  ∀[ One (▷r a ? β) ⇒ Cmd ]
  closec :          ∀[ One (▷r end) ⇒ Cmd ]

δ : Cmd Φ → Pred SCtx
δ (forkc _)      = Emp
δ (mkchc β)      = One (▷r β) ∗ One (▷r β ⁻¹)
δ (sendc _ β _)  = One (▷r β)
δ (recvc a β _)  = Val a ∗ One (▷r β)
δ (closec _)     = Emp
```

Importantly, fork takes a computation in F, rather than an Exp to represent the computation. This allows the first and second stages to be truly independent. By nesting F inside the reader transformer, we can interpret all the effects of LTLC_{ses}. For example:

```
eval : Exp a Γ → ϵ[ ReaderT F Γ [] (Val a) ]
eval (recv e) = do
  cref φ1          ← eval e
  φ2 ⟨ σ ⟩ v       ← liftM ⟪ recvc _ _ φ1 ⟫
  return (pair (cref φ2 ⟨ σ ⟩ v))
eval (send (e1 ⟨ σ ⟩ e2)) = do
  v1               ← frame σ (eval e1)
  cref φ1 ⟨ σ ⟩ v1 ← eval e2 &⟨ ⊎-id' ⟩ v1
  φ2               ← liftM ⟪ sendc _ _ (φ1 ⟨ σ ⟩ v1) ⟫
  return (cref φ2)
```

Again, the hard work of maintaining separation is hidden. Additionally, using the free monad construction, the concurrency is hidden. Finally, the fact that receiving a message on a channel is a blocking operation and may have to wait for the corresponding sent is completely opaque. The implementation of concurrency and communication is completely up to the second stage interpreter.[8]

5.2 Stage II: Interpreting Command Trees

By interpreting the expression language to command trees, we have given an operational semantics for everything except the five primitive operations for concurrency and communication. These are the operations that operate on the runtime state of the language: the collection of channels and the threadpool. In this section we first explain how threads are represented, and how one computes a single step in a thread. Then, we look closely at the channel state and the communication operations on it. Finally we wrap it all up into a function handle that interprets a single command of our language, and we give the top-level scheduler run that selects threads from the threadpool to compute in.

Threads are simply suspended computations—represented using the F monad—that return values. We distinguish the main thread which returns a value, from forked threads returning a unit value [16]:

```
data Thread : Pred SCtx where
  forked : ∀[ F (Val unit) ⇒ Thread ]
  main   : ∀[ F (Val a)    ⇒ Thread ]
```

To take a step in a thread is to unfold one layer of an impure computation in F, which requires a *handler* that is able to respond appropriately to all of the commands:[9]

```
stepf : (∀ {Φ} → (c : Cmd Φ) → M (δ c) Φ →
          ∀[ F P ⇒ M (F P) ]
stepf handler (pure px)              = return (pure px)
stepf handler (impure (c ⟨ σ ⟩ κ)) = do
  r ⟨ σ ⟩ κ ← handler c &⟨ σ ⟩ κ
  return (κ (⊎-comm σ) r)
```

This operation can easily be lifted from F to Threads:

```
step : (∀ {Φ} → (c : Cmd Φ) → M (δ c) Φ →
         ∀[ Thread ⇒ M Thread ]
```

Finally, we represent the thread pool as a big separating conjunction over Threads. The implementation of fork *thr* is simply to enqueue the thread *thr*, by appending it to the pool. Conversely, dequeue returns the frontmost thread that is not done from the threadpool if there is at least one, or

[8] In this interpreter, threads only yield control when they send a command. More fine-grained concurrency can be achieved by adding a *yield* command and manually inserting it where desired, or incorporating it into the bind of the stage 1 interpreter.

[9] The argument *handler* of step is essentially an empty *dependent wand*, from a command to its response.

returns Emp otherwise. Although the scheduler is unaware of this, the runtime for a session-typed language will always have at least the main thread in the threadpool.

```
enqueue : ∀[ Thread ⇒ M Emp ]
dequeue : ϵ[ M (Emp ∪ Thread) ]
```

We implement an asynchronous communication model with buffered channels, inspired by the buffer threads used by Fowler et al. [16]. The asynchronous, buffered model is a good fit for executable semantics, as it is close to practical implementations. At the same time it avoids the need to organize a rendezvous between two communicate threads, as all communication is mediated by the buffer. Type safety of session-typed languages with asynchronous communication relies on a number of invariants related to channels:

1. If a channel endpoint is sending (i.e., has a type $a\ !\ \beta$), then its buffer must be empty.
2. If a channel endpoint is sending then the buffer of the communicating endpoint must accept those values.
3. If one side of the channel has been closed, then the other end *cannot* be sending.

We will represent the state in a way that makes these observations evident from dependent pattern matching.

Buffers are lists of values waiting to be received at an endpoint. We write $\alpha \rightsquigarrow \beta$ (or $\beta \lightsquigarrow \alpha$) for a typed buffer with two ends: the *external* endpoint β corresponding to the endpoint of a channel, and an *internal* endpoint α. The type of the internal endpoint denotes the type of the channel endpoint after it will have caught up with the buffered values:

```
data _⇜_ : STy → STy → Pred SCtx where
  emp  : (α ⇜ α) ϵ
  cons : ∀[ Val a ∗ (β ⇜ γ) ⇒ ((a ? β) ⇜ γ) ]
```

Channels can then be represented as two linked buffers, or, in case either endpoint has already been closed, a single buffer. The duality between endpoints of a channel holds when both sides are completely caught up with any communication. Consequently, the duality is enforced at the *internal* endpoints of the buffers:

```
data Chan : RTy → Pred SCtx where
  both   : ∀[ α ⇜ β ∗ β ⁻¹ ⇝ γ ⇒ Chan (chan_r α γ) ]
  single : ∀[ end ⇝ β          ⇒ Chan (▷_r β) ]
```

The typing of buffers makes the first invariant holds: we can only have values waiting if the external end is a type $a\ ?\ \beta$. The duality of the internal endpoints of linked buffers ensures that the second invariant is satisfied. The third invariants holds, because the available constructors for buffers of type $\mathbf{end} \rightsquigarrow \beta$ restrict β to be either also **end**, or $a\ ?\ \gamma$.

We then proceed to instantiate the State monad with a heterogeneous list of Chans, rather than the list of plain values used for LTLC_{ref}. This yields a monad Cm which can implement the operations for manipulating the channels:

```
newch  : ϵ[ Cm (One (▷_r α) ∗ One (▷_r (α ⁻¹))) ]
recv?  : ∀[ One (▷_r (a ? β)) ⇒
            Cm (Val a ∗ One (▷_r β)) ]
send!  : ∀[ One (▷_r (a ! β)) ⇒ Val a ⊸
            Cm (One (▷_r β)) ]
closech : ∀[ One (▷_r end) ⇒ Cm Emp ]
```

To complete the semantics we pair the state for the thread pool and the channels, and lift the **fork** and communication operations into a monad M. We also nest an Error monad in this state monad, to account for an exception that indicates that the thread must be delayed, because there was no value present in the buffer to be received. We then implement the operation that dispatches these operations on a command:

```
handle : ∀ {Φ} → (c : Cmd Φ) → M (δ c) Φ
```

The top-level scheduling loop picks the first thread from the threadpool that is not done (using dequeue), tries to take a step, reschedules, and repeats. If popping a thread fails, then all threads have computed to a value, and the scheduler is done. If taking a step fails, then we recover (using orelse) by rescheduling it and continuing. The operation orelse is implemented for the state monad wrapped around the error monad, and resets the state for the recovery computation if the left-hand side computation fails:

```
run : ϵ[ M Emp ]
run = do
  thr? ← dequeue
  case thr? of λ where
    (inj₁ empty) → return empty
    (inj₂ thr)   → do
      empty ← (do
        thr′ ← step handle thr
        enqueue thr′) orelse (enqueue thr)
      run
```

Where the use of fuel to satisfy the termination checker (as well as the error handling for running out of fuel) has been ellided.

We have defined the typed representation for a concurrent runtime with asynchronous, buffered communication. Although the typing of the runtime for a session-typed language contains some subtleties, the staged interpretation of this language required no changes at the level of the logic. The logic and monadic abstractions that we defined scale to the functional session-typed language LTLC_{ses} and indeed yield interpreters whose clarity is not obscured by explicit proof terms.

6 Related Work

We discuss prior work on using dependent types to express linear and relevant scoping and typing of terms, and to prove

safety of session-typed languages. We also discuss the work on separation logic that was used to develop PRSAs.

Co-de-Bruijn Representation of Syntax The term co-de-Bruijn was coined by McBride [26], who exploits this representation in Agda to ensure that variable shifts in well-scoped, non-linear lambda terms do not require term traversals. He shows that hereditary substitutions on these terms become structurally recursive. Rather than working directly with objects in their relevant context, he works with objects wrapped in "thinnings" into a larger context, and develops the structure of these wrapped objects. He defines "relevant pairs", which are almost identical to our separating conjunctions for typing contexts, but permit overlap between the consumption of the left and right projections.

Abel and Kraus [2] describe an encoding of the binding in the simply-typed lambda calculus terms that is inspired by the typing of terms in a linearly-typed lambda calculus. They adapt it to non-linear lambda calculi and use it to avoid space leaks in interpreters that build function closures. This is achieved by separating environments along context separations in the interpreter, thus only capturing relevant values and avoiding leaks. This separation of the environment reappears in our linear interpreters in the generic frame operation of the Reader monad.

Neither Abel and Kraus [2], nor McBride [26] make the connection with separation logic, or make use of a magic wand-like connective, which was crucial in typing our functional abstractions.

Intrinsically-Typed Session-Type Semantics The most closely related work on semantics for session-typed languages is an intrinsically-typed small-step operational semantics in Agda by Thiemann [37]. The session-typed language that he uses is a superset of ours. It includes internal and external choice operators (with subtyping), a coinductive definition of session types, and unrestricted types. He gives a round-robin scheduler for threads, and implements synchronous communication. The semantics takes the form of an interruptible abstract machine [15], that operates by decomposing an expression with its value environment and evaluation context, into a command for the scheduler. In addition, he proves that his semantics implements the beta rule for unrestricted function application, and an eta rule for pairs. The complete development (minus import statements) comprises 2890 lines of Agda code. Of those lines, 1652 are used to define the semantics.

The cited work presents a mechanized semantics of a session-typed language in an executable, intrinsically typed style, and identifies the role of both static and dynamic separation in the type-safety proof. One of the key issues that loc. cit. identifies, which directly provoked present work, is the difficulty of managing the separation of the resources. Particularly, the pervasiveness of proof terms related to the separation of resources make many of the definitions "tedious exercise[s] in resource shuffling". This includes important definitions for the semantics, such as the function that decomposes expressions. Another example is the function that searches in the thread pool for the send command that corresponds to a read command.[10] This function comprises 50 lines of Agda and its type quantifies over 7 contexts, related by 3 separation witnesses. The interesting case of this function, which considers a send command, requires 11 lines of code, 10 of which are reorganizing the six separation terms that appear in the arguments of function.

We address the complexity and tedium of working with separated objects by composable abstractions. Each of the abstractions can be understood separately, making the complexity of the composite manageable. An important part of this is that we are able to give recognizable types to abstractions by working in an appropriate logic. Additionally, the abstractions help considerably to avoid duplication. For example, we are able to prove separation rotation lemmas for all PRSAs using just ⊎−assoc and ⊎−comm, whereas they are proven separately for both separation of typing contexts and session contexts in loc. cit. Another good example is his definition of session context separation, which has 6 constructors, baking in the separation on channels. Using our library, one can define it as the composition of two PRSAs, reusing our instance for lists. Besides the reuse one gets, this also simplifies the proofs.

Besides the proof terms, the styles of the semantics also differ significantly. Whereas a small-step semantics really shines in a relational setting, it is an indirect way to define an executable semantics. The computation steps of the language are hidden within the mechanics of the abstract machine—i.e., in decomposing expressions, and plugging values back into evaluation contexts. Some of the clarity is lost in these indirections that animate the small-step semantics. In our definitional interpreter, this tension between the small-step nature of concurrency and the functional style that we require, is resolved by using the free monad, representing continuations not as syntax, but as Agda functions. Additionally we avoid searching in evaluation contexts to find two threads that are both ready to communicate, by mediating between communicating threads using a buffer.

Monadic Intrinsically-Typed Interpreters Other prior work is an intrinsically-typed interpreter for Middleweight Java by Bach Poulsen et al. [6]. They show that monotone state can be encapsulated and programmed with using a strong monad in a category of monotone predicates. This directly inspired the use of monadic strength in this paper.

It is interesting to see that despite the fact monotonicity and separation are different concepts, both became easier to

[10] https://github.com/peterthiemann/definitional-session/blob/1e32d68f027cc3f1da2683a166da482be47fb1d2/src/Session.agda#L353

Table 1. Line counts of the Agda development

Module	LOC
Relation.Ternary.Separation.*	989
Typed.LTLC	50
Typed.LTLCRef	78
Sessions.*	367
*	1484

manipulate by using monadic strength. It would be interesting to investigate builtin support for inserting applications of strength in programs that use do-notation, not unlike how arrow-notation is desugared in Haskell [31].

Separation Algebras In the development of our abstractions and interpreters we have found a lot of inspiration in existing work on separation logic. We have already mentioned the work that contributed to the formulation of PRSAs in §4.1. For constructing the logic on top of PRSAs we were inspired by the construction of separation logic in Iris [21], which defines the type formers and connectives of separation logic in terms of *cameras*—i.e., the variant of separation algebras used in Iris. The Market PRSA is inspired by Iris's Auth camera [22], which serves a similar purpose: relating the provider of a resource to its clients. Elements of Auth A are essentially pairs $(x\ ,\ y)$ of A's. The left (*authoritative*) element can either be present or not, and cannot be separated. If it is present, then we must have the inclusion $x \uplus z \simeq y$, for some leftover resource z. Unfortunately, because of the inclusion evidence, the Auth construction does not transfer well to the proof-relevant setting. Moreover, to prove the laws of Auth A for an arbitrary PRSA A, one needs the higher structure of A—e.g., composing $\uplus-\mathsf{assoc}$ with $\uplus-\mathsf{unassoc}$ is the identity. The Market PRSA is a generalization of the model for counting permissions by Bornat et al. [7].

Linear Types and Separation Logic Whereas we used ideas from separation logic to write interpreters for linear languages that are intrinsically type safe, there has has been prior work on using separation logic to prove type safety in an extrinsic manner. The most notable development in this direction is RustBelt [20], where they used the technique of *logical relations* in Iris [25] to prove type safety and data race freedom of the Rust type system and some of its standard libraries. Contrary to our work, these developments on logical relations are based on an untyped operational semantics instead of a well-typed interpreter.

In another line of recent work, researchers developed a separation logic for proving functional correctness of message-passing programs called Actris [19]. While Actris is loosely based on session types, its goal (functional correctness) is different from our work.

7 Conclusions and Future Work

We presented the development of intrinsically-typed interpreters for $\mathrm{LTLC}_{\mathrm{ref}}$ and a session-typed language $\mathrm{LTLC}_{\mathrm{ses}}$. The Agda development consists of (1) the library for proof-relevant separation logic and the described functional abstractions, (2) an interpreter for LTLC and $\mathrm{LTLC}_{\mathrm{ref}}$, and (3) the syntax and semantics of $\mathrm{LTLC}_{\mathrm{ses}}$. The reader-, state-, and free-monad constructions are part of the library, whereas we count the communication operations as part of the $\mathrm{LTLC}_{\mathrm{ses}}$ semantics. The line counts in Table 1, which do not include module import statements, give an impression of the size of the library and the case studies.

The interpreters for $\mathrm{LTLC}_{\mathrm{ref}}$ (§4) and $\mathrm{LTLC}_{\mathrm{ses}}$ (§5) meet the requirements that we outlined in the introduction: the interpreters are executable and type-safe specifications, and the semantics is not obscured by explicit proof terms. The few proof terms that are still present are trivial to fulfill using the laws of PRSAs, and thus impact the clarity of the semantics less than the partiality present in untyped interpreters. Using monadic strength to enable programming with the external bind of these monads, and by using a free monad to implement concurrency and communication, we managed to preserve the familiar look of monadic interpreters.

Future Work In this work we focused on the *typing* and *usage* of a monadic interface for effects in the presence of separation. The *implementations* of the monadic operations were of lesser concern, because they are generic and reusable. These operations have to poke through the abstractions of its interface (i.e., the logic) and look at the separation witnesses. It would be interesting to further investigate if this can be improved. It seems that one could avoid some more manipulation of the separation witnesses if one adopts a completely point-free programming style. In current day dependently-typed languages this is not very appealing, because we cannot use the builtin support for defining functions using dependent pattern matching.

Because the remaining proofs of separation are mechanical, using only the axioms of the PRSAs, it seems likely that we can use some lightweight automation to fill them in, see e.g., [24, §9.6]. We would also like to investigate whether Agda's builtin rewriting [11] could be used to automatically eliminate separation witnesses containing an identity ϵ.

Acknowledgments

We thank the reviewers for their comments. We also thank Peter Thiemann for suggesting the topic and for discussing his previous work. This research was partially funded by the NWO VICI Language Designer's Workbench project (639.023.206), the NWO VENI Verified Programming Language Interaction project (016.Veni.192.259), and the NWO VENI Composable and Safe-by-Construction Programming Language Definitions project (VI.Veni.192.259).

References

[1] Andreas Abel and James Chapman. 2014. Normalization by evaluation in the delay monad: A case study for coinduction via copatterns and sized types. In *MSFP (EPTCS)*, Vol. 153. 51–67.

[2] Andreas Abel and Nicolai Kraus. 2011. A lambda term representation inspired by linear ordered logic. In *LFMTP (EPTCS)*, Vol. 71. 1–13.

[3] Guillaume Allais, James Chapman, Conor McBride, and James McKinna. 2017. Type-and-scope safe programs and their proofs. In *CPP*. 195–207.

[4] Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. 2015. Monads need not be endofunctors. *LMCS* 11, 1 (2015).

[5] Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. In *POPL*. 666–679.

[6] Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2018. Intrinsically-typed definitional interpreters for imperative languages. *PACMPL* 2, POPL (2018), 16:1–16:34.

[7] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. 2005. Permission accounting in separation logic. In *POPL*. 259–270.

[8] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. 2007. Local action and abstract separation logic. In *LICS*. 366–378.

[9] Venanzio Capretta. 2005. General recursion via coinductive types. *LMCS* 1, 2 (2005).

[10] Jesper Cockx. 2017. *Dependent pattern matching and proof-relevant unification.* Ph.D. Dissertation. Katholieke Universiteit Leuven.

[11] Jesper Cockx and Andreas Abel. 2016. Sprinkles of extensionality for your vanilla type theory. *Abstract of a talk at TYPES* (2016).

[12] Thierry Coquand. 1992. Pattern matching with dependent types. In *Informal proceedings of Logical Frameworks*, Vol. 92. 66–79.

[13] Nils Anders Danielsson. 2018. Up-to techniques using sized types. *PACMPL* 2, POPL (2018), 43:1–43:28.

[14] Robert Dockins, Aquinas Hobor, and Andrew W. Appel. 2009. A fresh look at separation algebras and share accounting. In *APLAS (LNCS)*, Vol. 5904. 161–177.

[15] Matthias Felleisen and Daniel P. Friedman. 1987. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts*. 193–222.

[16] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. 2019. Exceptional asynchronous session types: Session types without tiers. *PACMPL* 3, POPL (2019), 28:1–28:29.

[17] Simon J. Gay and Vasco T. Vasconcelos. 2010. Linear type theory for asynchronous session types. *JFP* 20, 1 (2010), 19–50.

[18] Peter Hancock and Anton Setzer. 2000. Interactive programs in dependent type theory. In *CSL (LNCS)*, Vol. 1862. 317–331.

[19] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2020. Actris: Session-type based reasoning in separation logic. *PACMPL* 4, POPL (2020), 6:1–6:30.

[20] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the foundations of the Rust programming language. *PACMPL* 2, POPL (2018), 66:1–66:34.

[21] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *JFP* 28 (2018), e20.

[22] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*. 637–650.

[23] Anders Kock. 1972. Strong functors and monoidal monads. *Archiv der Mathematik* 23, 1 (1972), 113–120.

[24] Robbert Krebbers. 2015. *The C standard formalized in Coq.* Ph.D. Dissertation. Radboud University Nijmegen.

[25] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *POPL*. 205–217.

[26] Conor McBride. 2018. Everybody's got to be somewhere. In *MSFP (EPTCS)*, Vol. 275. 53–69.

[27] Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55–92.

[28] Ulf Norell. 2009. Dependently typed programming in Agda. In *TLDI*. 1–2.

[29] Peter W. O'Hearn and David J. Pym. 1999. The logic of bunched implications. *Bulletin of Symbolic Logic* 5, 2 (1999), 215–244.

[30] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional big-step semantics. In *ESOP (LNCS)*, Vol. 9632. 589–615.

[31] Ross Paterson. 2001. A new notation for arrows. In *ICFP*. 229–240.

[32] Benjamin C. Pierce. 2002. *Types and programming languages.* MIT Press.

[33] John C. Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *LICS*. 55–74.

[34] Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. 2020. Agda development of "Intrinsically-Typed Definitional Interpreters for Linear, Session-typed Languages". https://github.com/metaborg/linear.agda

[35] Jeremy G. Siek. 2013. Type safety in three easy lemmas. http://siek.blogspot.co.uk/2013/05/type-safety-in-three-easy-lemmas.html.

[36] Wouter Swierstra and Tim Baanen. 2019. A predicate transformer semantics for effects (functional pearl). *PACMPL* 3, ICFP (2019), 103:1–103:26.

[37] Peter Thiemann. 2019. Intrinsically-typed mechanized semantics for session types. In *PPDP*. 19:1–19:15.

[38] Philip Wadler. 2014. Propositions as sessions. *JFP* 24, 2-3 (2014), 384–418.