The Linearity Monad

Jennifer Paykin
University of Pennsylvania
jpaykin@cis.upenn.edu

Steve Zdancewic
University of Pennsylvania
stevez@cis.upenn.edu

Abstract

We introduce a technique for programming with domain-specific linear languages using the monad that arises from the theory of linear/non-linear logic. In this work we interpret the linear/non-linear model as a simple, effectful linear language embedded inside an existing non-linear host language. We implement a modular framework for defining these linear EDSLs in Haskell, allowing both shallow and deep embeddings. To demonstrate the effectiveness of the framework and the linearity monad, we implement languages for file handles, mutable arrays, session types, and quantum computing.

ACM Reference format:
Jennifer Paykin and Steve Zdancewic. 2017. The Linearity Monad. In *Proceedings of 10th ACM SIGPLAN International Haskell Symposium, Oxford, UK, September 7-8, 2017 (Haskell'17),* 16 pages.
https://doi.org/10.1145/3122955.3122965

1 Introduction

Linear types have been used successfully for a variety of effectful domain-specific programming languages. For the domains of memory management [Fluet et al. 2006; Pottier and Protzenko 2013], mutable state [Chen and Hudak 1997; Wadler 1990], concurrency [Caires and Pfenning 2010; Mazurak and Zdancewic 2010], and quantum computing [Selinger and Valiron 2009], linearity statically enforces properties, specific to each domain, that are inexpressible in non-linear settings.

Consider the following interface for linear file handles.[1]

```
open  :: String ⊸ Handle      read  :: Handle ⊸ Handle ⊗ Char
close :: Handle ⊸ One         write :: Handle ⊸ Char ⊸ Handle
```

In this example, linearity rules out two specific kinds of errors. First, it ensures that file handles cannot be used more than once in a term, which means that once a handle has been closed, it cannot be read from or written to again. Second, linearity ensures that all open handles are eventually closed (at least for terminating computations) since variables of type `Handle` cannot be dropped. Linearity allows us to think of a file handle as a consumable resource that gets used up when it is closed.[2]

Linearity is useful here because it statically enforces properties that are inexpressible using conventional "non-linear" types. For mutable state, linear types enforce a single-threadedness property that allows a functional operation such as `writeArray` of type `Int ⊸ Array α ⊸ α ⊸ Array α` to be implemented as a mutable update [Wadler 1990]. For concurrent session types, linearity statically enforces the fact that every channel has exactly two endpoints that obey complementary communication protocols [Caires and Pfenning 2010]. For quantum computing, linear types enforce the so-called "no-cloning" theorem by restricting function spaces to linear transformations [Selinger and Valiron 2009].

Unfortunately, few mainstream programming languages offer support for linear types, for two reasons. First, linear type systems are often unwieldy, with linear typing information bleeding into programs that are entirely non-linear. For example, consider an ordinary, unrestricted function that concatenates a string to itself, given by `\s → s++s`, of type `String → String`. In traditional presentations of linear types [Benton et al. 1993] this function would instead be given type `!String ⊸ !String` and would be written `\s → let! s' = s in !s'++!s'`. The type $!\alpha$, pronounced "bang α", indicates that expressions of type α can be duplicated, but the programmer must make such uses explicit by means of the binding `let!`. Conversely, to create a value of type `!String`, the programmer must explicitly mark an expression with `!`, as in `!e`, which promises that `e` contains no free linear variables. For simple examples like this one, the explicit management of linearity isn't too bothersome, but it quickly becomes painful for larger pieces of code. Put another way, the traditional presentation of linearity using the $!\alpha$ type presents linearity as the default and non-linearity as the exception, while programmers expect the opposite.

Over the years, various linear type systems have been introduced to mitigate the problem of mixing linear and non-linear programming, using techniques based on subtyping [Selinger and Valiron 2009], constraint solving [Morris 2016], weights [McBride 2016], and kind polymorphism [Mazurak et al. 2010]. However, these techniques introduce complicated typing rules, can be difficult to use, and require significant modifications to existing non-linear language design.

The second problem with integrating substructural type systems with mainstream languages is that linear typing disciplines are almost always domain-specific, meaning that new applications of linear types must be added by the language designer, not the user. In the past few years, a growing number of general purpose languages have begun integrating features from substructural logics into their type systems, in order to express some of these domain-specific features. Ownership types in Rust [Matsakis and Klock 2014] and uniqueness types in Clean [Nöcker et al. 1991] and Idris [Idris Community 2017] are limited to a specific domain—shared memory management—and are weaker than full linear types. Recently, Bernardy et al. [2017] proposed a plan to add full linear types to Haskell, which could in the future be integrated with new

[1] Here, ⊸ (pronounced "lollipop") denotes linear implication, ⊗ ("tensor") denotes the multiplicative linear product, and `One` denotes the multiplicative unit.

[2] Note that linearity does not prevent all runtime errors: `open` could fail if there is a problem with the file name, or `read` could fail with an end-of-file error, etc. These later errors depend on the state of the system external to the program, while the errors avoided by linear types depend only on the program itself.

Haskell'17, September 7-8, 2017, Oxford, UK

domains. Their approach, discussed further in Section 7, is intriguing but requires thinking about linearity in a new way, as a property of arrows rather than as a property of data.

We propose a different approach, inspired by Benton's linear/non-linear (LNL) presentation of linear logic [1995]. The LNL model, illustrated in Figure 1, describes a categorical adjunction between two separate type systems, one linear and the other non-linear. In this paper we interpret the LNL model as the embedding of a simple linear lambda calculus inside an existing non-linear programming language. The embedded language approach easily extends to a variety of different application domains, and the adjoint functors `Lift` and `Lower` form a straightforward interface between the embedded and host languages: `Lower` inserts host language terms into the embedded language, and `Lift` injects closed linear terms into the host language as suspended computations.

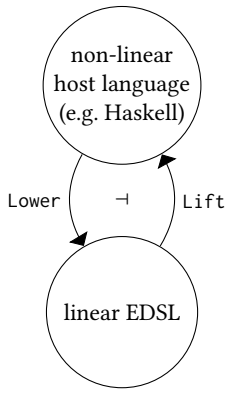

Figure 1. The linear/non-linear programming model.

When the host language supports monadic programming, as Haskell does, the LNL interface reveals a connection with monads. It is well-known that the ! modality from linear logic forms a comonad on the linear category. In Figure 1, the ! modality corresponds to the composition `Lower ∘ Lift`; we can think of it as the perspective of looking "up" at the non-linear category from the linear one. In this work, we propose to also look "down" at the linear category from the unrestricted world. The adjoint structure of the LNL model ensures that the result, the composition `Lift ∘ Lower`, forms a monad. This structure, the linearity monad, is the main focus of this work.

1.1 Contributions

In this paper we show how to realize linear/non-linear type theory by embedding a linear language inside of an unrestricted language, using `Lower` and `Lift` to move between the two fragments (Section 2). For concreteness we choose Haskell as the host language, since it already has good support for monadic programming; we expect our techniques could be readily adapted to other host languages as well. Importantly, we target a design that allows various application domains to be expressed modularly in the system.

To that aim, the paper makes the following contributions:

1. We show how our realization of the LNL model as an embedded language gives rise to a linearity monad (Section 5). The relationship between linear types and monads is well-known from a categorical perspective, but the consequences for programming have not been widely explored [Benton and Wadler 1996; Chen and Hudak 1997]. We justify the monad laws and describe how the monad extends to a monad transformer.
2. We develop a framework for implementing linear EDSLs using higher-order abstract syntax in Haskell (Sections 3 and 4). The framework draws on prior embeddings of linear types in Haskell [Eisenberg et al. 2012; Polakow 2015] by employing Haskell's type class mechanism to automatically discharge linearity constraints. We can instantiate the framework with both shallow embeddings of judgments as Haskell functions, or with deep embeddings using generalized algebraic data types (GADTs). Throughout, the framework uses the dependently-typed features of the Glasgow Haskell Compiler (GHC) to enforce the linear use of typing judgments.
3. Finally, we demonstrate the effectiveness of linear monadic programming by implementing examples of domain-specific linear languages in our framework, including our running example of safe file handles in the style of Mazurak et al. [2010], as well as, in Section 6: (a) Mutable arrays in the style of Wadler's "Linear types can change the world!" [1990]; (b) Session types in the style of Caires and Pfenning [2010]; and (c) Quantum computing in the style of Selinger and Valiron [2009].

	linear	**non-linear**
types	$\sigma, \tau ::= \sigma \multimap \tau \mid \cdots$	$\alpha, \beta ::= \alpha \to \beta \mid \cdots$
variables	x, y	a, b
typing contexts	$\gamma ::= \cdot \mid \gamma, x : \sigma$	$\Gamma ::= \cdot \mid \Gamma, a : \alpha$
expressions	$e ::= \lambda x.e \mid ee' \mid \cdots$	$t ::= \lambda x.t \mid tt' \mid \cdots$
typing judgment	$\gamma \vdash e : \sigma$	$\Gamma \vdash t : \alpha$

Figure 2. Meta-variables for the purely linear and purely non-linear language fragments

The implementation and all of the examples described in this paper are available at the following URL:

https://github.com/jpaykin/LNLHaskell/tree/Haskell2017

2 Linear/Non-Linear types

Linear/non-linear (LNL) logic, introduced by Benton [1995], is a model of linear logic obtained by combining two very simple type systems. The first is an entirely linear lambda calculus, meaning that all variables are linear and there is no unrestricted modality $!\sigma$. The other is an entirely non-linear lambda calculus, in which resources are not tracked. We can think of these two systems independently, each containing their own syntax of types, variables, typing contexts, and typing judgments, as shown in Figure 2.

These fragments may contain arbitrary extra features, such as operations for manipulating file handles in the linear language, as in the example from the introduction. Alternatively, the non-linear type system may have algebraic data types, dependent types, *etc.* As a starting point, consider the standard presentation of a linear lambda calculus with application and abstraction.

$$\frac{}{\gamma = x : \tau \vdash x : \tau}\ \textsc{var} \qquad \frac{\gamma' = \gamma, x : \sigma \qquad \gamma' \vdash e : \tau}{\gamma \vdash \lambda x.e : \sigma \multimap \tau}\ \textsc{abs}$$

$$\frac{\gamma = \gamma_1 \Cup \gamma_2 \qquad \gamma_1 \vdash e_1 : \sigma \multimap \tau \qquad \gamma_2 \vdash e_2 : \sigma}{\gamma \vdash e_1 e_2 : \tau}\ \textsc{app}$$

In the VAR rule, no other variables occur in the context besides the one being declared, meaning that linear variables cannot be discarded (weakened) from a context. The ABS rule introduces a fresh linear variable into the context. In APP, the relation $\gamma = \gamma_1 \Cup \gamma_2$ means that γ is the disjoint union of γ_1 and γ_2; it enforces the fact that variables cannot occur on both sides of an application.

	linear	**non-linear**
types	$\sigma, \tau ::= \cdots \mid \texttt{Lower}\ \alpha$	$\alpha, \beta ::= \cdots \mid \texttt{Lift}\ \tau$
typing judgment	$\Gamma; \gamma \vdash e : \sigma$	$\Gamma \vdash t : \alpha$

$$\frac{}{\Gamma; \gamma \vdash x : \tau}\ \textsc{var} \qquad \frac{\gamma' = \gamma, x : \sigma \qquad \Gamma; \gamma' \vdash e : \tau}{\Gamma; \gamma \vdash \lambda x.e : \sigma \multimap \tau}\ \textsc{abs}$$

$$\frac{\gamma = \gamma_1 \uplus \gamma_2 \qquad \Gamma; \gamma_1 \vdash e_1 : \sigma \multimap \tau \qquad \Gamma; \gamma_2 \vdash e_2 : \sigma}{\Gamma; \gamma \vdash e_1 e_2 : \tau}\ \textsc{app}$$

$$\frac{\Gamma \vdash t : \alpha}{\Gamma; \cdot \vdash \texttt{put}\ t : \texttt{Lower}\ \alpha}\ \textsc{put}$$

$$\frac{\gamma = \gamma_1 \uplus \gamma_2 \qquad \Gamma; \gamma_1 \vdash e : \texttt{Lower}\ \alpha \qquad \Gamma, a : \alpha; \gamma_2 \vdash e' : \tau}{\Gamma; \gamma \vdash \texttt{let!}\ a = e\ \texttt{in}\ e' : \tau}\ \textsc{let!}$$

$$\frac{\Gamma; \cdot \vdash e : \tau}{\Gamma \vdash \texttt{suspend}\ e : \texttt{Lift}\ \tau}\ \textsc{suspend} \qquad \frac{\Gamma \vdash t : \texttt{Lift}\ \tau}{\Gamma; \cdot \vdash \texttt{force}\ t : \tau}\ \textsc{force}$$

Figure 3. Typing rules for the combined linear/non-linear language

For the non-linear language, we start with unrestricted typing rules as in the simply-typed lambda calculus, and write $\Gamma \vdash t : \alpha$ to denote its typing judgments.

A linear/non-linear type system modifies these two languages so that they interact in a predictable way.

First, we extend the linear typing judgment so it can refer to non-linear variables. The resulting judgment has the form $\Gamma; \gamma \vdash e : \sigma$, where the variables in Γ are non-linear, the variables in γ are linear, and the result type σ is also linear. The revised typing rules are given in Figure 3. The revised VAR rule allows arbitrary non-linear variables, while the revised APP rule allows non-linear variables to be used on both sides of the application.

Note that a non-linear variable is not a linear expression itself; the inference rule $\Gamma, a : \alpha; \cdot \vdash a : \alpha$ is not valid because α is not a linear type. In order to use non-linear data in the linear world, the second step in creating the linear/non-linear model is to extend the linear language with a new type: $\sigma, \tau ::= \cdots \mid \texttt{Lower}\ \alpha$.

As shown in Figure 3, terms of type `Lower` α are constructed from arbitrary non-linear terms via an operation called `put`, so every linear expression of type `Lower` α morally holds a non-linear value. The elimination form, `let!` $a = e$ `in` e', lets us use that value non-linearly as long as we use it to construct another linear expression; otherwise, the linear variables used to construct e would be lost.

The third step in creating a linear/non-linear system is to introduce the `Lift` connective, which embeds linear expressions in the non-linear world: $\alpha, \beta ::= \cdots \mid \texttt{Lift}\ \tau$. Of course, it is not always safe to treat linear expressions non-linearly—that is the entire point of linear logic! However, when a linear expression doesn't use any linear variables, it is safe to duplicate. Consider the term `open "filename"` from the file handle example; multiple invocations will create different handles to the file. Such an expression can be thought of as an effectful "suspended" computation that can be "forced" as many times as necessary, since running that computation doesn't consume any linear resources.

Also in Figure 3, the `Lift` type is introduced by `suspend e`, which internalizes a linearly-closed expression `e` as a non-linear value. The corresponding elimination form, `force`, moves such a value back into the linear world.[3]

2.1 LNL as an Embedded Language

One contribution of this paper is the recognition that the LNL model lends itself well to describing a linear language embedded in a non-linear one. The embedded structure means that host language's non-linear variables are, by default, accessible to the linear sub-language. As a result, the linear embedding only needs to keep track of the linear variables, since the non-linear variables are automatically handled by the host language. This vastly simplifies the representation of the embedded language. The `Lower` connective describes a simple way to use arbitrary host language terms, making the whole host language accessible from within the linear fragment. The `Lift` connective exposes linear expressions to the rest of the host language without exposing linear variables directly.

In the rest of this paper, we use Haskell as the host language, exploiting the dependently-typed features of GHC 8 to enforce linearity in the embedding. Haskell has been used as a host language for linear types before [Eisenberg et al. 2012; Polakow 2015], and we draw on ideas from these previous embeddings (deferring a more technical comparison to Section 7). The next section describes these implementation details and how we accomodate domain-specific linear types like file handles in our linear/non-linear interpretation. Ours isn't the only possible implementation—other design decisions will present different tradeoffs—but the implementation helps illustrate the main focus of this paper, which is the programming model that arises from linear/non-linear logic.

3 Embedding a linear type system in Haskell

To embed a linear language in Haskell we build data structures for linear types and contexts, and enforce linearity constraints on those contexts using type classes. The choice of how to encode variables, contexts, and typing judgments was made to maximize the type class mechanism's ability to automatically discharge these constraints during type checking, while also keeping the types and terms of our EDSL legible.

For the first iteration of our linear language, we will restrict linear types to the unit type and linear implication.

```
data LType = One | Lolli LType LType
```

We use the infix notation $\sigma \multimap \tau$ as a synonym for `Lolli` σ τ.

Following Eisenberg et al. [2012] we represent variables in our embedding as unary natural numbers (`data Nat = Z | S Nat`) and typing contexts as finite maps from natural numbers to `LTypes`. The operations we define later rely heavily on the inductive structure of both variables and contexts. The finite map is represented as a list `[Maybe LType]`, where the variable `i` maps to the type stored in the list at index `i`. The `Maybe` type marks the presence (`Just` σ) or absence (`Nothing`) of the variable in the context. As an example,

[3] The `suspend` and `force` notation is inspired by Call-By-Push-Value [Levy 2012], which separates pure and effectful computations into two parts, much in the same way linear and non-linear type systems are separated in LNL. Indeed, the effectful linear/non-linear type system presented in this paper can be thought of as the combination of CBPV and linear logic.

consider the following sample derivation:

$$\dfrac{\dfrac{\dfrac{}{[\texttt{Just}\ (\sigma \multimap \tau)] \vdash 0 : \sigma \multimap \tau}\ \textsc{var} \quad \dfrac{}{[\texttt{Nothing}, \texttt{Just}\ \sigma \vdash 1 : \sigma]}\ \textsc{var}}{[\texttt{Just}\ (\sigma \multimap \tau), \texttt{Just}\ \sigma] \vdash 0\ 1 : \tau}\ \textsc{app}}{[\texttt{Nothing}, \texttt{Just}\ \sigma] \vdash \lambda\ 0.\ 0\ 1 : (\sigma \multimap \tau) \multimap \tau}\ \textsc{abs}}{[] \vdash \lambda\ 1.\ \lambda\ 0.\ 0\ 1 : \sigma \multimap (\sigma \multimap \tau) \multimap \tau}\ \textsc{abs}$$

To enforce the desired linearity constraints, the application rule in this derivation satisfies the side condition that

```
[Just(σ ⊸ τ)] ⩔ [Nothing, Just σ] = [Just σ, Just (σ ⊸ τ)]
```

The merge relation is not defined when two contexts hold the same variable, or, equivalently, when `Just` appears at the same index in both contexts. Mathematically, merge is defined as follows:

```
γ1              ⩔ []              = γ1
[]              ⩔ γ2              = γ2
(Just σ  : γ1) ⩔ (Nothing : γ2) = Just σ  : (γ1 ⩔ γ2)
(Nothing : γ1) ⩔ (Just σ  : γ2) = Just σ  : (γ1 ⩔ γ2)
(Nothing : γ1) ⩔ (Nothing : γ2) = Nothing : (γ1 ⩔ γ2)
```

This representation contains some redundancy: the lists `[Just σ]` and `[Just σ, Nothing]` both correspond to the same context, $0 : \sigma$. So instead of using the built-in list type `[Maybe LType]`, we say that a context `Ctx` is either empty, or is a non-empty context `NCtx`, which ends in a `Just σ`.

```
data Ctx  = Empty | NEmpty NCtx
data NCtx = End LType | Cons (Maybe LType) NCtx
```

Note that this is *not* a De Bruijn representation of variables; it is a nominal representation where the map from names to types is defined by indexing into the array.

3.1 Relations on typing contexts

The type system in Figure 3 uses three relations on contexts to enforce linearity. The VAR rule says that $\gamma \vdash x : \sigma$ if γ is the context containing only the single binding $x : \sigma$. We formulate this relation in Haskell as a multi-parameter type class `CSingleton x σ γ`, as shown in Figure 4. The class `CSingletonN x σ γ` records the same property, but for non-empty contexts—we use this helper type class to inductively build up the relation.

```
instance CSingletonN x σ γ ⇒ CSingleton x σ (NCtx γ)
instance CSingletonN Z σ (End σ)
instance CSingletonN x σ γ
      ⇒ CSingletonN (S x) σ (Cons Nothing γ)
```

The functional dependencies `x σ → γ` and `γ → x σ` tell GHC that the `CSingleton` relations are functional and injective [Jones 2000]. They are vital to linear type checking as they guide unification: for any concrete context, Haskell will automatically search for the proof that it forms a singleton context, and for any concrete variable and type, Haskell will automatically infer the singleton context containing that variable.

To handle the side conditions on the abstraction and application rules, we introduce two additional type classes, also shown in Figure 4. The class `CAdd x σ γ γ'` encodes the property that $\gamma' = \gamma, x : \sigma$, where x does not already occur in γ. The class `CMerge γ1 γ2 γ` says that $\gamma 1 \uplus \gamma 2 = \gamma$, or, in other words, that γ is the disjoint union of $\gamma 1$ and $\gamma 2$. Proving the functional dependencies for these classes is not straightforward, and in the implementation we use a number of helper classes to convince GHC that they hold, which we describe in Appendix A. The functional dependencies, which permit the typechecker to do some amount of inversion, are the main reason we use type classes (which encode relations), rather than type families (which encode functions) to describe the `CSingleton`, `CAdd`, and `CMerge` operations.

3.2 Typing judgments

A well-typed expression $\gamma \vdash e : \tau$ in the linear lambda calculus is represented as a Haskell term `e :: exp γ τ`. The parameter `exp :: Ctx → LType → Type` is a *typing judgment* characterized via a type class interface, the members of which correspond to the typing rules of the linear lambda calculus. For example:

```
class HasLolli (exp :: Ctx → LType → Type) where
  λ   :: (CSingleton x σ γ'', CAdd x σ γ γ', x ~ Fresh γ)
      ⇒ (exp γ'' σ → exp γ' τ) → exp γ (σ ⊸ τ)
  (^) :: CMerge γ1 γ2 γ ⇒ exp γ1 (σ ⊸ τ) → exp γ2 σ → exp γ τ
```

The `HasLolli` type class asserts that the typing judgment `exp` contains abstraction (λ) and application (`^`) operations.[4] The application operator corresponds closely to the APP inference rule given in Figure 3, where `CMerge` encodes the disjoint union of contexts. Abstraction uses higher-order abstract syntax, which means that it covers both the variable and abstraction rules at once. Let's take a look at the type of λ without the type class constraints:

```
(exp γ'' σ → exp γ' τ) → exp γ (σ ⊸ τ)
```

This type says that, in order to construct a linear function $\sigma \multimap \tau$, it suffices to provide an ordinary Haskell function from expressions of type σ to expressions of type τ. In order to ensure that this function uses its argument exactly once, we have the following constraints, where `~` is equality on types:

```
(CSingleton x σ γ'', CAdd x σ γ γ', x ~ Fresh γ)
```

The last constraint says that `x` is a particular variable that is fresh in γ: we define `Fresh γ` to be the smallest natural number that is undefined in γ. The middle constraint says that the body of the function, of type `exp γ' τ`, satisfies the relation $\gamma' = \gamma, x : \sigma$. The first constraint says that the argument of the function, of type `exp γ'' σ`, really is a variable, since $\gamma'' = x : \sigma$. Put in a more functional notation, the type of λ could be described as follows:

```
(exp [x:σ] σ → exp (γ,x:σ) τ) → exp γ (σ ⊸ τ)
```

The HOAS encoding leads to very natural-looking code. The identity function is `λ (\x → x)`, while composition is defined as:

```
compose :: HasLolli exp
        ⇒ exp Empty ((τ2 ⊸ τ3) ⊸ (τ1 ⊸ τ2) ⊸ (τ1 ⊸ τ3))
compose = λ $ \g → λ $ \f → λ $ \x → g ^ (f ^ x)
```

We do not have to add any special infrastructure to handle polymorphism; Haskell takes care of it for us.

3.3 Multiplicative unit and pairs

It is easy to extend the language to other operators of linear logic, such as units, pairs $\otimes$, and sums $\oplus$. For the linear multiplicative unit, we have the following class:

```
class HasOne exp where
  unit :: exp Empty One
  letUnit :: CMerge γ1 γ2 γ ⇒ exp γ1 One → exp γ2 τ → exp γ τ
```

[4] The linear abstraction function λ should not be confused with Haskell's usual anonymous function abstraction, written `\a → t`.

```
class CSingleton (x :: Nat) (σ :: LType) (γ :: Ctx)           | x σ → γ, γ → x σ                    -- γ = [x:σ]
class CAdd       (x :: Nat) (σ :: LType) (γ :: Ctx) (γ' :: Ctx) | x σ γ → γ', x γ' → σ γ, γ γ' → x σ -- γ' = γ,x:σ
class CMerge     (γ1 :: Ctx) (γ2 :: Ctx) (γ :: Ctx)           | γ1 γ2 → γ, γ1 γ → γ2, γ2 γ → γ1     -- γ1 ⩔ γ2 = γ
```

Figure 4. Type classes encoding relations on typing contexts

For the operators $\otimes$ and $\oplus$, we need to first extend the syntax of linear types. We could add constructors for tensor products, *etc.*, directly to the `LType` definition, but doing so would commit to a particular choice of linear connectives. Instead, we build in a way to extend linear types by introducing `MkLType`, which existentially introduces a new linear type:

```
data LType where MkLType :: ext LType → LType
```

Extensions, denoted with the meta-variable `ext`, are paramaterized by a type. For example, the multiplicative product $\otimes$ can be encoded as an extension `TensorExt` using GHC data type promotion [Eisenberg and Stolarek 2014], as follows:

```
data TensorExt ty = MkTensor ty ty
type σ ⊗ τ = MkLType (MkTensor σ τ)
```

Multiplicative products are pairs whose components come from disjoint typing contexts.

$$\frac{\gamma 1 \vdash e1 : \tau 1 \qquad \gamma 2 \vdash e2 : \tau 2 \qquad \gamma = \gamma 1 \Cup \gamma 2}{\gamma \vdash e1 \otimes e2 : \tau 1 \otimes \tau 2}$$

$$\frac{\gamma 1 \vdash e : \sigma 1 \otimes \sigma 2 \qquad \gamma 2, x1 : \sigma 1, x2 : \sigma 2 \vdash e' : \tau \qquad \gamma = \gamma 1 \Cup \gamma 2}{\gamma \vdash \text{let } (x1, x2) = e \text{ in } e' : \tau}$$

We overload the constructor $(\otimes)$ to construct multiplicative pairs. The HOAS version of the elimination form, which we write `letPair`, has a structure that mirrors the type of λ.

```
class HasTensor exp where
  (⊗) ::CMerge γ1 γ2 γ⇒exp γ1 τ1→exp γ2 τ2→exp γ (τ1 ⊗ τ2)
  letPair :: ( CAdd x1 σ1 γ2 γ2', CAdd x2 σ2 γ2' γ2''
             , CSingleton x1 σ1 γ21, CSingleton x2 σ2 γ22,
             , x1 ~ Fresh γ2, x2 ~ Fresh γ2', CMerge γ1 γ2 γ)
           ⇒ exp γ1 (σ1 ⊗ σ2)
           → ((exp γ21 σ1, exp γ22 σ2) → exp γ2'' τ)
           → exp γ τ
```

The variables `x1` and `x2` are represented in the higher-order abstract syntax by arguments `exp γ21 σ1` and `exp γ22 σ2` respectively, where $\gamma 21 = [x1 : \sigma 1]$ and $\gamma 22 = [x2 : \sigma 2]$. The continuation of the `letPair` is in the context $\gamma 22 = \gamma 2, x1 : \sigma 1, x2 : \sigma 2$. The result is that we are able to bind pairs in a natural way, as in `λ $ \x → x ‘letPair‘ \(y,z) → z ⊗ y`, of type $\sigma \otimes \tau \multimap \tau \otimes \sigma$.[5]

In the implementation we provide similar interfaces for additive sums, products, and units.

3.4 The `Lift` and `Lower` types

The LNL connective `Lower` can be added to the linear language just like any other linear connective. The only difference is that `Lower` takes an argument of kind `Type`—the kind of Haskell types.

[5]It would certainly be more natural to write `λ $ \(y,z) → z ⊗ y` directly, but type checking for nested pattern matching is a difficult problem we leave for future work. We can however define a top-level pattern match `λpair`, and write our example as `λpair $ \ (y,z) → z ⊗ y`. We discuss the issue of type checking and nested pattern matching more in Section 7.1.

```
data LowerExp ty = MkLower Type
type Lower α = MkLType (MkLower α)
```

Figure 3 introduces the syntax `put` to introduce terms of type `Lower α` and `let! a = e in e'` to eliminate them. In Haskell we write the `let!` operator in higher-order abstract syntax as `(>!)`.

```
class HasLower exp where
    put  :: α → exp Empty (Lower α)
    (>!) :: CMerge γ1 γ2 γ
         ⇒ exp γ1 (Lower α) → (α → exp γ2 τ) → exp γ τ
```

Figure 3 also introduces syntax for the `Lift` type, a non-linear type carrying linear expressions with no free linear variables. We define `Lift` in Haskell as an ordinary record.

```
data Lift exp τ = Suspend { force :: exp Empty τ }
```

Figure 5 shows how we can embed the usual $!\sigma$ operation from linear logic using `Lift` and `Lower`. For convenience, we define the synonym `HasMELL` for the class of constraints corresponding to multiplicative exponential linear logic ($\multimap$, $\otimes$, `One`, and `Lower`).

3.5 File Handles

We now return to our running example from the introduction of safe file handles.

```
class HasMELL exp ⇒ HasFH exp where
    open  :: String → exp Empty Handle
    read  :: exp γ Handle → exp γ (Handle ⊗ Lower Char)
    write :: exp γ Handle → Char → exp γ Handle
    close :: exp γ Handle → exp γ One
```

The `open` and `write` operations, which take ordinary Haskell data as input, demonstrate how linear operations can take advantage of existing Haskell infrastructure. For example, the function that writes an entire string to a file (rather than just a single character) can be implemented as an ordinary fold over the string.

```
writeString :: HasFH exp ⇒ String→exp γ Handle→exp γ Handle
writeString s h = foldl write h s
```

File handles interact nicely with the other linear connectives. The following function reads a character from a file and writes that same character back to the file twice:

```
readWriteTwice :: HasFH exp ⇒ exp Empty (Handle ⊸ Handle)
readWriteTwice = λ $ \h → read h `letPair` \(h,x) →
                          x >! \c →
                          writeString [c,c] h
```

The linear type system does enforce the fact that the same file handle cannot have more than one alias to it, which prevents a handle from being read from after it is closed. So, the following does not type check:

```
readAfterClose :: exp Empty (Handle ⊸ Handle ⊗ Lower Char)
readAfterClose = λ $ \h → close h `letUnit` read h
```

```
type Bang τ = Lower (Lift τ)
dup  :: HasMELL exp ⇒ Lift exp (Bang τ ⊸ Bang τ ⊗ Bang τ)
dup  = Suspend . λ $ \x → x >! \a → put a ⊗ put a
drop :: HasMELL exp ⇒ Lift exp (Bang τ ⊸ One)
drop = Suspend . λ $ \x → x >! \_ → unit
```

Figure 5. Encoding the exponential from linear logic. `HasMELL exp` is a synonym for `(HasLolli exp, HasTensor exp, HasOne exp, HasLower exp)`.

4 Evaluation and Implementation

Our goal in embedding a linear language in Haskell is not just to represent programs in those languages, but to actually run those programs. In this section we define both deep and shallow embeddings that implement the `HasLolli` and `HasFH` type classes of the previous sections. In both cases, a correct implementation is expected to validate a number of coherence laws (akin to the monad laws) that we explain below.

We focus on large-step semantics rather than a small-step semantics, which would be both less efficient and, in the case of a shallow embedding, less appropriate. For each linear type we define a type of *linear values* using data families[6]. We also adopt environment semantics, evaluating open linear terms within an accompanying evaluation context. As a consequence we do not have to define an explicit substitution function, which is slow and type-theoretically challenging as it requires extensive manipulation of typing contexts. Evaluation is effectful—for example, file handles will be implemented using the Haskell primitive, which in this case means that evaluation will take place in the `IO` monad. Different domains (see Section 6) have different effects, so we need to ensure that the effect is a parameter of the framework.

Every implementation thus has three components: a typing judgment `exp :: Ctx → LType → Type`; a value judgment `val :: LType → Type`, and a (monadic) effect `m :: Type → Type`. We structure these three components as data and type families indexed by a signature `sig :: Type`.

```
data family LExp   (sig :: Type) (γ :: Ctx) (τ :: LType) :: Type
data family LVal   (sig :: Type) (τ :: LType) :: Type
type family Effect (sig :: Type) :: Type → Type
```

An evaluation context is a finite map from variables, represented using singletons [Eisenberg and Stolarek 2014], to values. It is indexed by a signature corresponding to the signature of values, as well as a typing context specifying the domain. That is, an evaluation context of type `ECtx sig γ` maps variables $x : \sigma \in \gamma$ to values of type `LVal sig σ`.

```
data ECtx sig γ where
  ECtx :: (∀ x σ. Lookup γ x ~ Just σ ⇒ Sing x → LVal sig σ)
       ⇒ ECtx sig γ
```

Evaluation is specified as a type class on signatures.

```
class Eval sig where
  eval :: Monad (Effect sig)
       ⇒ ECtx sig γ → LExp sig γ τ → Effect sig (LVal sig τ)
```

[6] https://wiki.haskell.org/GHC/Type_families

4.1 A deep embedding

First we consider a deep embedding, where linear lambda terms are defined as a GADT in Haskell. The `LExp` data type bears a strong resemblance to the `HasLolli` type class, although without higher-order abstract syntax.

```
data Deep
data instance LExp Deep γ τ where
  Var :: CSingleton x τ γ ⇒ Sing x → LExp Deep γ τ
  Abs :: CAdd x σ γ γ'
      ⇒ Sing x → LExp Deep γ' τ → LExp Deep γ (σ ⊸ τ)
  App :: CMerge γ1 γ2 γ
      ⇒ LExp Deep γ1 (σ ⊸ τ) → LExp Deep γ2 σ → LExp Deep γ τ
```

To instantiate the `HasLolli` type class, it is enough, therefore, to produce the singleton value x that corresponds to `Fresh γ`.[7]

```
instance HasLolli (LExp Deep) where
  λ :: ∀ x σ γ γ' γ''.
       (CSingleton x σ γ'', CAdd x σ γ γ', x ~ Fresh γ)
    ⇒ (LExp Deep γ'' σ→LExp Deep γ' τ)→LExp Deep γ (σ ⊸ τ)
  λ f = Abs x (f $ Var x) where x = (sing :: Sing x)
  (^) = App
```

Values are defined by induction on `LType`. A value of type $\sigma \multimap \tau$ is a closure containing an evaluation context paired with the body of the abstraction, while a value of type `Lower α` is the underlying Haskell value, and so on.

```
data instance LVal Deep (Lower α) = VPut α
data instance LVal Deep One       = VUnit
data instance LVal Deep (σ ⊗ τ)   =
  VPair (LVal Deep σ) (LVal Deep τ)
data instance LVal Deep (σ ⊸ τ) where
  VAbs :: CAdd x σ γ γ' ⇒ ECtx Deep γ
       → Sing x → LExp Deep γ' τ → LVal Deep (σ ⊸ τ)
```

Next we instantiate `Eval Deep` by defining the evaluation function. When the expression is an abstraction we return the closure.

```
instance Eval Deep where
  eval γ (Abs x e) = return $ VAbs γ x e
```

If the expression is a variable, we know that the typing context γ must contain only a single variable, $x :: \sigma$. In that case we want to return the value stored in the evaluation context, which we access via an operation we call `lookup`.

```
  eval γ (Var x) = return $ lookup x γ
```

The `lookup` operation is simply the result of looking up a variable in the evaluation context, so `lookup x (ECtx f) = f x`. However, this application is only valid if the constraint `Lookup γ x ~ Just σ`, when `CSingleton x σ γ`. We discuss how to embed this constraint in the `CSingleton` type class in Appendix A.

To evaluate an application `App e1 e2`, we first evaluate `e1` to obtain a closure, then evaluate `e2`. Then we evaluate the body of the closure, extending its evaluation context with the value of `e2`.

```
  eval γ (App (e1::LExp Deep γ1 τ1) (e2::LExp Deep γ2 τ2)) =
    do let (γ1,γ2) = split @γ1 @γ2 γ
       VAbs γ' x e1' ← eval γ1 e1
       v2            ← eval γ2 e2
       eval (add x v2 γ') e1'
```

[7] In the development, the constraint `SingI x` is a superclass to `CSingleton`.

This operation uses two additional helper functions to manipulate contexts in a way similar to `lookup`. The function `add` takes a variable x, an evaluation context for γ, and a value of type σ, and produces an evaluation context for $\gamma, x : \sigma$. Similarly, `split @γ1 @γ2` takes an evaluation context for γ where `CMerge γ1 γ2 γ`, and outputs two evaluation contexts for $\gamma 1$ and $\gamma 2$ respectively; it uses visible type application [Eisenberg et al. 2016] (*e.g.,* `@γ1`) to specify the appropriate contexts.

These helper functions are described thoroughly in Appendix A.

To extend the syntax of the deep embedding to additional domains such as file handles, we would need to modify the `LExp Deep` data type with each new constructor. However, this is not modular; every time a programmer wanted to use the embedding in a different domain, she would have to define or modify the data type and the entire evaluation function. In Appendix B we describe a design that allows the deep embedding to be modularly extended to arbitrary application domains.

4.2 A shallow embedding

Next we consider a shallow embedding, where an expression `exp γ τ` is represented as a monadic function from evaluation contexts for γ to values of type τ. Evaluation in the shallow embedding is just unpacking this function.

```
data Shallow
data instance LExp Shallow γ τ =
  SExp { runSExp :: ECtx γ → Effect Shallow (LVal τ) }
instance Eval Shallow where eval γ f = runSExp f γ
```

Values in the shallow embedding are almost the same as those in the deep embedding, except that a value of type $\sigma \multimap \tau$ in the shallow embedding is represented as a function from values of type σ to values of type τ, instead of as an explicit closure.

```
data instance LVal Shallow (σ ⊸ τ) =
  VAbs (LVal Shallow σ → Effect Shallow (LVal Shallow τ))
```

We can show that the shallow embedding simulates all the features of our linear language by instantiating the type classes for `HasLolli`, `HasLower`, `HasFH`, *etc.* Unsurprisingly, all of these constructions mirror the evaluation functions from the deep embedding. For example, here we give the instantiation of `HasLower`:

```
instance Monad (Effect Shallow) ⇒ HasLower (LExp Shallow)
  where put a  = SExp $ \_ → return $ VPut a
        e >! f = SExp $ \γ → do let (γ1,γ2) = split γ
                                VPut a ← runSExp e γ1
                                runSExp (f a) γ2
```

4.3 File Handles

Both embeddings can be given instances of the `HasFH` type class, where values of type `Handle` are built-in `IO` file handles, and the effect is also `IO`. We sketch the shallow embedding here, and give the deep embedding in Appendix B.

```
data instance LVal Shallow Handle = VHandle IO.Handle
type instance Effect Shallow = IO
```

The file handle operations are easily given by their `IO` counterparts; `open` and `close` are defined here, and `read` and `write` analogously.

```
instance HasFH (LExp Shallow) where
  open s  = SExp $ \ρ → do h←IO.openFile s IO.ReadWriteMode
                           return (VHandle h)
  close e = SExp $ \ρ → do VHandle h ← runSExp e ρ
                           IO.hClose h
                           return VUnit
```

4.4 Laws and correctness

In Haskell we often associate type classes with mathematical laws that characterize the properties of correct instances of those classes. In this setting, such laws describe an equational theory on the embedded language. For example, the laws for the type `Lower α` are as follows:[8]

```
put a >! f = f a  [β]              e >! put = e  [η]
   (e >! f) >! g = e >! \x → f x >! g   [assoc]
```

We say that an instance for `Eval sig` satisfies the `Lower` laws if they are preserved by evaluation.

Proposition 4.1. *The shallow embedding satisfies the* `Lower` *laws.*

Proof. We start with the β rule. Unfolding definitions:

```
eval (put a >! f) γ = do let (γ1,γ2) = split γ
                         VPut a ← return $ VPut a
                         runSExp (f a) γ2
```

Since γ_1 is the empty context we know that $\gamma_2 = \gamma$. Since `HasLower (LExp Shallow)` assumes that `Effect Shallow` is a monad, the equation above is equal to `runSExp (f a) γ`, as expected.

The proofs of the η and associativity laws are similarly obtained by unfolding definitions and applying the monad laws. □

Proposition 4.2. *The deep embedding satisfies the* `Lower` *laws.*

Proof. By unfolding definitions and applying monad laws. □

5 The monad

Benton [1995] originally proposed linear/non-linear logic as a proof theory, and through the Curry-Howard correspondence we have interpreted it as a type system; we can also draw on its categorical interpretation. Illustrated back in Figure 1, the LNL categorical model consists of two categories, one corresponding to the linear language, and the other corresponding to the non-linear language.

In our implementation, the non-linear category is HASK, the idealized category of Haskell types and terms. The LINEAR category has objects that are elements of `LType`, and morphisms that are values of type `LExp sig Empty (σ ⊸ τ)`.

The operators `Lift` and `Lower` are functors between these two categories. For any Haskell function $\alpha \to \beta$ we have a linear morphism `Lower α ⊸ Lower β`, and similarly for any linear morphism $\sigma \multimap \tau$ we have a Haskell function `Lift σ → Lift τ`.[9]

```
fmapLower :: (HasLolli (LExp sig), HasLower (LExp sig))
          ⇒ (α → β) → LExp sig Empty (Lower α ⊸ Lower β)
fmapLower f = λ $ \x → x >! put . f
fmapLift :: HasLolli (LExp sig)
         ⇒ LExp sig Empty (σ ⊸ τ) → Lift sig σ → Lift sig τ
fmapLift f s = Suspend $ f ^ force s
```

[8] The astute reader will recognize a similarity to the monad laws, which we discuss in depth in Section 5.

[9] Note that we do not give an instance of the standard type class `Functor`, which only describes endofunctors on HASK.

Back in the linear/non-linear model, `Lift` and `Lower` form a (symmetric monoidal) adjunction `Lower ⊣ Lift`, which is what allows non-linear variables to occur in linear typing judgments. Mac Lane [1978] famously says that "adjoint functors arise everywhere", but they seem to have found less ground in Haskell than their close cousin, the monad. Every adjunction $F \dashv G$ gives rise to a monad, $G \circ F$, as well as a comonad, $F \circ G$. As is usual in linear logic, the type operator `Bang sig τ = Lower (Lift sig τ)` (from Section 3.4) forms a comonad, and its dual `Lift sig (Lower α)` forms a monad.

We write this linearity monad as `Lin sig α`. For convenience, the accessor functions `suspendL` and `forceL` move directly between the monad and the linear category.

```
newtype Lin sig α = Lin (Lift sig (Lower α))
suspendL = Lin . Suspend
forceL (Lin e) = force e
```

The linearity monad does indeed have a monad instance.[10]

```
instance HasLower (LExp sig) ⇒ Monad (Lin sig) where
  return a = suspendL $ put a
  e >>= f  = suspendL $ forceL e >! forceL . f
```

Theorem 5.1. *If a siganture* `sig` *satisfies the* `Lower` *laws, then the monad laws hold for* `Lin sig`*: (1)* `pure a >>= f = f a`*; (2)* `e >>= pure = e`*; and (3)* `(e >>= f) >>= g = e >>= (\x → f x >>= g)`

Proof. For (1), expanding the definition for `Lin sig` we see that

```
pure a >>= f = suspendL $ forceL (pure a) >! forceL . f
             = suspendL $ put a >! forceL . f
```

By the β rule for `>!`, this is equal to `suspendL (forceL $ f a)`, which is η-equivalent to `f a` itself.

The proofs of (2) and (3) are similarly by unfolding definitons and applying the `Lower` laws. □

When we evaluate the body of an expression in `Lin sig α`, the result is an effectful lowered Haskell value `LVal sig (Lower α)`. We can always extract the underlying value of type α, meaning that we get a result in `Effect sig α`. We call this operation `run`.

```
run :: Eval sig ⇒ Lin sig a → Effect sig a
run e = eval EEmpty (forceL e) >>= \(VPut a) → return a
```

5.1 Monads in the linear category

Consider the following function, which opens a file, performs some transformations, and closes the file again. Note that composing `run` with `withFile` will produce an `IO` action that manipulates the file directly.

```
withFile :: HasFH (LExp sig) ⇒ String
         → Lift sig (Handle ⊸ Handle ⊗ Lower a) → Lin sig a
withFile s op = Suspend $ force op ^ open s `letPair` \(h,a)→
                          close h `letUnit` a
```

Just like the state monad in Haskell, the type $\sigma \multimap \sigma \otimes \tau$ forms a monad in the linear category. We can then define a type class of linear monads `LMonad m`, where `m` has kind `LType → LType`, with linear versions of `return` and `bind`.

To make an instance declaration for linear state, we first try to define a type synonym `LState σ τ` for $\sigma \multimap \sigma \otimes \tau$. This approach fails for a rather silly reason: the monad `LState σ` is a partially defined type synonym, which is not allowed in GHC. The ordinary solution would be to define a newtype, but these (and regular algebraic data types) produce `Type`s, not `LType`s.

Our solution is to use a trick called defunctionalization [Eisenberg and Stolarek 2014]. The Singletons library[11] provides a type-level arrow `k1 ⇝ k2` that describes unsaturated type-level functions between kinds `k1` and `k2`. To define a defunctionalized arrow, we first define an empty data type for the unsaturated version of `LState`, and then define a type instance for the (infix) type family `(@@)`, which has kind `(k1 ⇝ k2) → k1 → k2`.

```
data LState' (σ :: LType) :: LType ⇝ LType
type instance LState' σ @@ τ = σ ⊸ σ ⊗ τ
```

We can then define `LState σ τ = LState' σ @@ τ`. Instead of defining the `LMonad` type class for `m :: LType → LType`, we instead define it for defunctionalized arrows `m :: LType ⇝ LType`.

```
class LMonad sig (m :: LType ⇝ LType) where
  lreturn :: LExp sig γ τ → LExp γ (m @@ τ)
  lbind   :: LExp sig 'Empty (m @@ σ ⊸ (σ ⊸ m @@ τ) ⊸ m @@ τ)
```

When convenient, we use the notation `e >>= f` for `lbind ^ e ^ f`.

The laws for monads in LINEAR are the same as for those in HASK: (1) `lreturn e >>= f = f ^ e`; (2) `e >>= lreturn = e`; and (3) `(e >>= f) >>= g = e >>= (\x → f x >>= g)`

We can now define our monad instance.

```
instance HasMILL (LExp sig) ⇒ LMonad sig (LState' σ) where
  lreturn e = λ $ \s → s ⊗ e
  lbind     = λ $ \st → λ $ \f → λ $ \s →
                st ^ s `letPair` \(s,x) → f ^ x ^ s
```

To illustrate monadic linear programming, consider the following operation that reads the first n characters from a file handle:

```
takeM :: HasFH (LExp sig)
      ⇒ Int → LExp sig Empty (LState Handle (Lower String))
takeM n | n ≤ 0    = lreturn $ put ""
        | otherwise = readM >>= λ $ \x → x >! \c →
                      takeM (n-1) >>= λ $ \y → y >! \s →
                      lreturn $ put (c : s)
```

The monadic `readM :: LState Handle (Lower Char)` is just `λ read`.

5.2 The monad transformer

When an `LMonad` returns a lowered Haskell type, such as in `readM` and `takeM` above, we can push the monadic programming style a step further: the adjunction `Lower ⊣ Lift` also induces an `LMonad` *transformer*. Given an `LMonad` of type `LType ⇝ LType`, we can define a Haskell monad `LinT m`. As we did for `Lin`, it is convenient to have versions of `suspend` and `force`; we omit their definitions.

```
newtype LinT sig (m :: LType ⇝ LType) (α :: Type) =
        LinT (Lift sig (m @@ (Lower α)))
suspendT :: LExp sig Empty (m @@ (Lower α)) → LinT sig m α
forceT   :: LinT sig m α → LExp sig Empty (m @@ (Lower α))
```

We can define the `Monad` instance just as we did for `Lin`:

```
instance (LMonad m, HasLower (LExp sig)) ⇒ Monad (LinT sig m)
  where
    return = suspendT . lpure . put
    x >>= f = suspend $ forceT x >>= λ $ \y → y >! (force . f)
```

[10] The appropriate `Functor` and `Applicative` instances can be found in the implementation.

[11] https://hackage.haskell.org/package/singletons

Proposition 5.2. *If* `m` *satisfies the* `LMonad` *laws, then* `LinT sig m` *satisfies the* `Monad` *laws.*

Proof. By unfolding definitions and applying the `LMonad` laws. □

The `read`, `write`, and `withFile` operations have natural presentations in terms of `LinT`, where `LStateT sig σ α` is a synonym for `LinT sig (LState' σ) α`.

```
readT    :: HasFH (LExp sig) ⇒ LStateT sig Handle Char
writeT   :: HasFH (LExp sig) ⇒ Char → LStateT sig Handle ()
withFile :: HasFH (LExp sig)
         ⇒ String → LStateT sig Handle a → Lin sig a
```

We also define a monad transformer version of `take`.

```
takeT :: HasFH (LExp sig) ⇒ Int → LStateT sig Handle String
takeT n | n ≤ 0    = return ""
        | otherwise = do c ← readT
                         s ← takeT (n-1)
                         return $ c:s
```

Putting these together we can actually evaluate our linear programs:

```
main = run $ do withFileT "foo" $ mapM_ writeT "Hello world"
                withFileT "foo" $ takeT 7
> "Hello w"
```

6 Examples

In this section we present three additional application domains in the linear/non-linear framework: mutable arrays, session types, and quantum computing.

6.1 Arrays

In his paper "Linear types can change the world!", Wadler [1990] argues that mutable data structures like arrays can be given a pure functional interface if they are only accessed linearly. To understand why, consider a non-linear program with purely functional arrays:

```
let arr1 = write 0 arr "hello" in
let arr2 = write 0 arr "world" in arr1[0]
```

If `write` were to update the array in place, the program would return ``world'' instead of ``hello''. Linear types force us to serialize the operations on arrays so that reasonable equational laws still hold, even when performing destructive updates.

Here we expand Wadler's example to describe *slices* of an array. Consider an operation `slice i`, which splits an array into two disjoint sub-arrays determined by the index `i`. As long as the operations on each slice are restricted to their domains, the implementation of `slice` can alias the same array. Furthermore, as long as we keep track of when two slices alias the same array, we can merge slices back together with zero cost.

To implement linear arrays in the LNL framework, we first add a new type for arrays of non-linear values.

```
data ArraySig ty = MkArray Type Type
type Array token α = MkLType (MkArray token α)
```

The token argument to the array keeps track of the array being aliased. Constructing a new array will result in an array with an existentially quantified token, as required by the following type:

```
data SomeArray exp α where
  SomeArray :: exp Empty (Array token α) → SomeArray exp α
```

The linear interface can allocate new arrays and drop the pointers to existing ones. Each array is associated with a domain of valid indices, which can be obtained via the operation `dom`. The operation `slice` takes an index and an array, and outputs two aliases to that same array with domains partitioned around the index. Dually, `join` takes two aliases to the same array and combines their bounds. The `read` and `write` operations will fail at runtime if their arguments are not in the domain of their slice.

```
class HasMELL exp ⇒ HasArray exp where
  alloc :: Int → α → SomeArray exp α
  drop  :: exp γ (Array k α) → exp γ One
  dom   :: exp γ (Array k α)→exp γ (Array k α⊗Lower [Int])
  read  :: Int→exp γ (Array k α)→exp γ (Array k α⊗Lower α)
  write :: Int → exp γ (Array k α) → α → exp γ (Array k α)
  slice :: Int→exp γ (Array k α)→exp γ (Array k α⊗Array k α)
  join  :: CMerge γ1 γ2 γ ⇒ exp γ1 (Array k α)
        → exp γ2 (Array k α) → exp γ (Array k α)
```

6.1.1 Implementation

We can implement the `HasArray` signature in the shallow embedding. A value of type `Array k α` will be a pair of a domain of valid indices (of type `[Int]`) as well as a primitive Haskell array; in this case, an `IOArray`; the effect of this language will be `IO`.

```
data instance LVal Shallow (Array k α) =
    VArray [Int] (IOArray Int α)
type instance Effect Shallow = IO
```

The implementation of `alloc`, `read`, and `write` call to the primitive operations on `IOArrays`. The implementation of `drop` simply returns a unit value—it does not explicitly deallocate the array, which would be inappropriate when dropping partial slices. The `slice` operation partitions the bounds of its input array according to its index, while `join` evaluates its arguments and combines the resulting bounds.[12]

```
slice i e1 e = SExp $ \γ → do VArray bnd arr←runSExp e γ
                              let arr1 = filter (< i) bnd arr
                              let arr2 = filter (≥ i) bnd arr
                              return $ VPair arr1 arr2
join e1 e2 = SExp $ \γ→do let (γ1,γ2) = split γ
                          VArray bnd1 arr ← runSExp e1 γ1
                          VArray bnd2 _   ← runSExp e2 γ2
                          return $ VArray (bnd1⧺bnd2) arr
```

6.1.2 Arrays in the lifted state monad

We can lift `dom`, `read`, and `write` into the linear state monad transformer with the following signatures, where `LStateT sig σ α` is `LinT sig (LState' σ) α`.

```
domT :: HasArray (LExp sig) ⇒ LStateT sig (Array k α) Int
readT::HasArray (LExp sig) ⇒ Int → LStateT sig (Array k α) α
writeT :: HasArray (LExp sig)
       ⇒ Int → α → LStateT sig (Array k α) ()
```

We can also derive a lifted operation that combines slicing and joining. The function `sliceT` takes an index and two state transformations on arrays. The resulting state transformation takes in an

[12] As an aside, the structure of sliced arrays lends itself naturally to concurrency in the style of separation logic, and in the git repository we implement `join` so that it evaluates its two sub-arrays concurrently.

array, slices it around the input index, and applies the two state transformations to the two sub-arrays.

```
sliceT::HasArray (LExp sig)⇒Int→LStateT sig (Array k α) ()
    →LStateT sig (Array k α) ()→LStateT sig (Array k α) ()
sliceT i st1 st2 = Suspend . λ $ \arr →
    slice i arr `letPair` \(arr1,arr2) →
    forceT st1 ^ arr1 `letPair` \(arr1,res) → res >! \_ →
    forceT st2 ^ arr2 `letPair` \(arr2,res) → res >! \_ →
    join arr1 arr2 ⊗ put ()
```

6.1.3 Quicksort

We will use the `LStateT` interface to implement an in-place quicksort. Quicksort relies on a helper function `partition` that chooses a pivot value and swaps elements of the array until all values less than the pivot occur to the left of the pivot in the array, and all values greater than or equal to the pivot occur to the right. The `partition` function returns to us the index of the pivot after all the swapping occurs; if the list is too short to successfully partition, it returns `Nothing`. We omit the definition here but it uses the simple operation `swap`, which swaps two indices in the array.

```
swap :: HasArray (LExp sig)
     ⇒ Int → Int → LStateT sig (Array k α) ()
swap i j = do a ← readT i
              b ← readT j
              writeT i b ≫ writeT j a
partition :: (HasArray (LExp sig), Ord α)
          ⇒ LStateT sig (Array k α) (Maybe Int)
```

The quicksort algorithm slices its input according to the partition and recurses. The base case occurs when `partition` returns `Nothing`.

```
quicksort :: (HasArray (LExp sig), Ord α)
          ⇒ LStateT sig (Array k α) ()
quicksort = partition ≫= \case
    Nothing    → return ()
    Just pivot → sliceT pivot quicksort quicksort
```

6.1.4 Performance

Preliminary tests indicate that the linear typing framework for arrays reduces performance by a significant constant factor, although we have not performed a thorough analysis to confirm and quantify these results. Because we are implementing an embedded language, this is not entirely unexpected, but we expect a number of design choices could be tweaked to increase performance. In the shallow embedding of arrays, the runtime artifacts introduced by the linear framework are variables and evaluation contexts; the constraint-based type checking is only relevant at compile time.

6.1.5 Related work

Mutable state and memory management is one of the most common applications of linear type systems in the literature. Wadler [1990] formalizes the connection between mutable arrays and linear logic, and Chen and Hudak [1997] expand on this connection to show that when mutable abstract data types treat their data linearly in a precise way, they can be automatically transformed into monadic operations. Their monad corresponds more closely to Haskell's `IO` monad than the linearity monad described in this paper; it formally justifies Haskell's treatment of mutable update. Going beyond arrays, linear types have informed the use of regions [Fluet et al. 2006], uniqueness types [Barendsen and Smetsers 1993] and borrowing [Noble et al. 1998], all of which seek to safely manage memory usage in an unobtrusive way.

6.2 Session types

Session types are a language mechanism for describing communication protocols between two actors. A *session* is a channel with exactly two endpoints. Caires and Pfenning [2010] draw a Curry-Howard connection between session types and intuitionistic linear types, which we implement in this section.

Consider a protocol for an online marketplace: the marketplace will receive a request for an item in the form of a string, followed by a credit card number. After processing the order, the marketplace will send back a receipt in the form of a string. The session protocol for the marketplace is described by the following `LType`:

```
type Market = Lower String ⊸ Lower Int ⊸ Lower String ⊗ One
```

In Caires and Pfenning's formulation, a channel with session protocol $\sigma \multimap \tau$ receives a channel of type σ, then continues with the protocol τ. A channel with protocol $\sigma \otimes \tau$ sends a channel of type σ and then continues as τ. The Curry-Howard formulation means that we do not have to define a new syntax for session-typed programming, since we can just reuse the syntax we already have for $\otimes$ and $\multimap$. Consider the following implementation of `Market`:

```
marketplace :: HasMELL exp ⇒ Lift exp Market
marketplace = Suspend . λ $ \x → λ $ \y →
              x >! \item → y >! \cc →
              (put $ "Processed order for " ++ item) ⊗ unit
```

A consumer interacts with the opposite end of the protocol, and then the two actors can be plugged together to form a complete transaction.

```
buyer :: HasMELL exp ⇒ Lift exp (Market ⊸ Lower String)
buyer = Suspend . λ $ \c → c ^ put "Tea" `letin` \c →
                           c ^ put 1234  `letin` \c →
                           c `letPair` \(receipt,c) →
                           c `letUnit` receipt
transaction :: HasMELL exp ⇒ Lin exp String
transaction = supendL $ marketplace ^ buyer
```

6.2.1 Implementation

Although we use the same syntax as the pure linear lambda calculus, we really want an implementation that communicates data over channels. Since session-typed channels change their protocol over time, we implement them via untyped channels, and we use `unsafeCoerce`. This is appropriate (and safe!) because the session protocols—enforced by the linear types— ensure that each time a value of type α is sent on the channel, the recipient will coerce it back to that same type α. Details of the implementation can be found in Appendix C.

6.2.2 Related work

Session types have gained popularity in recent years as a model of concurrency. The connection to intuitionistic linear logic was first highlighted by Caires and Pfenning [2010], though connections have also been drawn with classical linear logic, which highlights the duality between sending and receiving on a channel [Lindley and Morris 2015; Wadler 2014]. Lindley and Morris [2016] provide

an embedding of their functional classical session types language GV in Haskell based on Polakow's linear embedding. Other implementations of session types in Haskell wrap enforce linearity dynamically by means of paramaterized monads [Orchard and Yoshida 2016; Pucella and Tov 2008], which we expect corresponds closely to the linearity monad.

6.3 Quantum computing

Quantum computing is the study of computing with qubits, entanglement, and other quantum-mechanical forces that are not expressible on classical (*e.g.,* non-quantum) machines. Mathematically, quantum computations are expressed as linear transformations (specifically unitary transformations) and as a result, non-linear computations such as copying quantum values are prohibited. Selinger and Valiron [2009] introduce a linear lambda calculus for describing quantum computations that they call the *quantum lambda calculus*. The details of quantum computation are beyond the scope of this paper; see Selinger and Valiron's presentation for a gentler introduction.

The quantum lambda calculus consists of a linear lambda calculus extended with a type for qubits (the quantum equivalent of a bit) and three additional operations:

```
class HasMELL exp ⇒ HasQuantum exp where
    new     :: Bool → exp Empty Qubit
    unitary :: Unitary σ → exp γ σ → exp γ σ
    meas    :: exp γ Qubit → exp γ (Lower Bool)
```

The `new` operation creates a qubit in a so-called "classical" state, corresponding to either 0 (`False`) or 1 (`True`). These qubits can be put into probabilistic states by applying unitary transformations, which correspond to the class of valid quantum computations. We assume there exists some universal set of unitary transformations `Unitary` σ, each of which corresponds to a linear transformation $\sigma \multimap \sigma$. Finally, `meas` performs quantum measurement, which probabilistically outputs a boolean value.

In Appendix D we show how to define a dependently-typed quantum Fourier transform using type families, drawing on Paykin et al. [2017], and give some implementation details.

7 Discussion and Related Work

7.1 Design of the embedded language

The embedding described in this paper is very similar to the work of Eisenberg et al. [2012] and Polakow [2015], who also describe embeddings of linear lambda calculi in Haskell using dependently-typed features of GHC to enforce linearity. We adapt features from both embeddings: Polakow introduces higher-order abstract syntax (HOAS) for linear types, but to achieve this he uses a non-standard typing judgment $\gamma in / \gamma out \vdash e : \tau$ that threads an input context into every judgment. Eisenberg et al. use the standard typing judgment $\gamma \vdash e : \tau$ but without HOAS, which makes linear programming awkward.

In this paper we combine the two representations to get a HOAS encoding of the direct-style typing judgment. Doing so has some limitations, however. For example, lambda abstractions can be used in either the left-hand side or the right-hand side of an application, but not both: the expression `λ $ \x → (λ $ \y → y) ^ x` type checks in Haskell, but not `(λ $ \x → x) ^ (λ $ \y → y)`. Haskell cannot infer that both sides of the application are typed in the empty context; knowing $\gamma 1 \Cup \gamma 2 =$ `Empty` is not enough to infer that $\gamma 1 = \gamma 2 =$ `Empty`. Although inconvenient, we find that this problem can often be circumvented by writing helper functions, *e.g.,* `id ^ id`.

Although we did not find this property prohibitively restrictive while writing our examples, it does represent a tradeoff in the design space. For example, one challenge we have not yet been able to overcome is type checking nested linear pattern matches. Polakow [2015]'s representation of typing judgments as a threaded relation $\gamma in / \gamma out \vdash e : \tau$ may be better suited for automatic type checking, but we find it less natural than the direct style. In future work many possibilities exist to enhance type checking for the direct style, including more robust type classes or a type checker plugin that uses an external solver to search for the intermediate typing contexts.[13]

Crucially, the contribution of this work in contrast to that of Eisenberg et al. and Polakow is not so much the design of the embedding in Haskell, but rather the use of the linear/non-linear model that gives rise to the linearity monad. Eisenberg et al. and Polakow introduce $!\alpha$ as an embedded connective, which, compared to the LNL decomposition of !, requires significantly more maintenance in the linear system, and introduces a divide between linear and regular Haskell programming.

7.2 Error messages

As in any type-heavy language embedded in Haskell, the error messages are not ideal. For example, the type checker will fail on `λ (\x → x ⊗ x)`, but instead of reporting that the program has attempted to duplicate a linear argument, the error message simply states that it expects an empty context where a non-empty context has been provided:

```
Couldn't match type `Empty' with `NEmpty (End σ)'
arising from a use of `⊗' in the expression: x ⊗ x
```

7.3 Deep versus shallow embeddings

The prior implementations by Eisenberg et al. [2012] and Polakow [2015] include only shallow embeddings, which should be more efficient than deep embeddings. However, the shallow embedding is not "adequate," because it is possible to write down terms of type `LExp Shallow` γ τ that do not correspond to anything in the linear lambda calculus.[14] This may be acceptable in some cases, as there are two different consumers of our framework: DSL *implementers* and DSL *users*. Implementers have access to unsafe features of the embedding, and so they must be careful to only expose an abstract linear interface (*e.g.,* one not containing the `SExp` constructor) to the *clients* of the language to enforce the linearity invariants.

In the deep embedding, linear expressions are entirely syntax so by definition all terms of type `LExp Deep` γ τ correspond to real linear expressions. The deep embedding also makes it possible to express program transformations and optimizations.

7.4 Further integration with Haskell

A recent proposal by Bernardy et al. [2017] suggests how to integrate linear types directly into GHC as Hask-LL, based on a model of linear logic that uses weighted annotations on arrows instead of $!\alpha$ or the adjoint decomposition considered here. Their proposal would allow the implementation of efficient garbage collection and

[13] https://ghc.haskell.org/trac/ghc/wiki/Plugins/TypeChecker

[14] For example, `SExp (\γ → VPut ())` has type `LExp Shallow` γ `(Lower ())` for any context γ.

explicit memory management, and could conceivably be adapted to a wide variety of different domains using foreign function interface calls.[15] Compared to our approach, the proposal requires significant changes to GHC; our framework works out-of-the box. We hypothesize that the linearity monad arises in their work as the (linear) CPS monad: $(\alpha \multimap \bot) \multimap \bot$.

Bernardy et al.'s proposal is also adamant about eliminating code duplication, meaning that data structures and operations on data structures should be parametric over linear versus non-linear data. It is certainly a drawback of our work that the user may have to duplicate Haskell code in the linear fragment, as we saw when defining the linear versions of the monad type classes in Section 5. Future work might address this by using Template Haskell[16] to define data structures and functions with implementations in both the linear and non-linear worlds.

7.5 Conclusion and future work

In this paper we present a new perspective on linear/non-linear logic as a programming model for embedded languages that integrates well with monadic programming. We develop a framework in Haskell to demonstrate our design, and implement a number of domain-specific languages. We expect the techniques presented in this paper to extend to many areas not covered here, such as affine and other substructural type systems, as well as bounded linear logic. In addition, the idea of an LNL model as an embedded language is not specific to Haskell, but could be applicable in a wide range of languages [Rand et al. 2017].

Appendices

A Type checking and type class resolution

The type classes `CAdd` and `CMerge` in Figure 4 are used to type-check linear expressions in Haskell, and they depend critically on functional dependencies to perform type checking. For example, consider type checking for a lambda abstraction. To show that λ `f` has type `exp` γ `(`$\sigma \multimap \tau$`)`, it suffices to show that `f :: exp` γ`''` $\sigma \to$ `exp` γ`'` τ where

1. `CSingleton x` σ γ`''`;
2. `CAdd x` σ γ γ`'`; and
3. `x ~ Fresh` γ.

In many cases, we know the value of γ—in practice the top-level of an expression will often be the empty context—and we can proceed in the following way. From γ we can compute x, as `Fresh` γ is a type family that produces the smallest natural number x that does not occur in γ. The functional dependencies of `CSingleton` state that because we know x and σ, we can compute γ''; it is the context with `Just` σ at index `i` and `Nothing` everywhere else. Furthermore, knowing the values of x, σ and γ we can deduce γ' based on the functional dependencies of `CAdd`.

Unfortunately, when we do not know the value of γ, we cannot in general deduce the types of the other variables. This situation arises whenever a merge occurs. For example, even if we know γ when type-checking `e ^ e'` `::exp` γ τ, we do not a priori know the contexts γ_1 and γ_2 such that `e ::exp` γ`1 (`$\sigma \multimap \tau$`)`, and `e' ::exp` γ`2` σ, such that $\gamma_1 \Cup \gamma_2 = \gamma$. Knowing γ is not enough to compute γ_1 and γ_2, although knowing any two of these three contexts is enough to compute the third, thanks to the functional dependencies of `CMerge`. When one of `e` or `e'` is, for example, a variable, we then know the value of γ_1 or γ_2 respectively, which allows us to compute the other. This explains why λ`(\x →` `(`λ `(\y →` `y) ^ x))` does type check in our system, but λ `(\x →` `x) ^` λ `(\y →` `y)` does not.

In order to enforce the functional dependencies required by this technique, it is necessary to design the type classes with some subtlety. For example, in the class `CAdd x` σ γ γ`'`, we need to enforce the functional dependency that γ`' x` $\to$ σ γ. Naively one might expect the following instances of this class:

```
instance CSingletonCtx x σ γ' ⇒ CAdd x σ Empty γ'
instance CAddN x σ γ γ' ⇒ CAdd x σ (N γ) (N γ')
```

where `CAddN` is the same relation, but on non-empty contexts. Unfortunately, these instances overlap, since there is not a unique instance that applies from just knowing x and γ', when γ' is non-empty. The decision of which case to apply depends on the *size* of the output context γ': when the size of γ' is one, the first rule applies, and when the size is greater than one, the second rule applies.

We can define a type class that counts the size of a non-empty context; for technical reasons (so we start counting at zero), we define `CountNMinus1` γ`'` to be one less than the number of elements in γ'.

```
type family CountNMinus1 (γ :: NCtx) :: Nat where
  CountNMinus1 (End _)           = Z
  CountNMinus1 (Cons (Just _) γ) = S (CountNMinus1 γ)
  CountNMinus1 (Cons Nothing γ)  = CountNMinus1 γ
type family Count (γ :: Ctx) :: Nat where
  Count Empty = Z
  Count (N γ) = S (CountNMinus1 γ)
```

Now the type class `CAdd` depends on a helper class, `CAdd'`, that itself depends on an additional argument corresponding to the size of the input context. The class `CAddN'` similarly applies when both the input and output contexts are non-empty, and the length argument to `CAddN'` corresponds with the length of the input context.

```
instance CAdd' x σ γ γ' (CountNMinus1 γ')⇒CAdd x σ γ (N γ')
class len ~ Count γ ⇒ CAdd'  x σ (γ :: Ctx)  (γ' :: NCtx) len
    | x σ γ → len γ', x γ' len → σ γ
class len ~ Count γ ⇒ CAddN' x σ (γ :: NCtx) (γ' :: NCtx) len
    | x σ γ → len γ', x γ' len → σ γ
```

The instances of `CAdd'` and `CAddN'` are then guided by this extra parameter, as shown in Figure 6.

We run into a similar problem for the `Merge` relation, which has the following functional dependencies:

```
class CMerge γ1 γ2 γ | γ1 γ2 → γ, γ1 γ → γ2, γ2 γ → γ1
```

In particular, knowing any two of γ`1`, γ`2` and γ determines the third. The naive instance declarations satisfy only the first functional dependency, however, as shown in Figure 7.

This class does not satisfy the dependency γ`1` γ $\to$ γ`2` because of the overlap between the instance `CMergeForward (N` γ`) Empty (N` γ`)` and `CMergeForward (N` γ`1) (N` γ`2) (N` γ`)` (and similarly for γ`2` γ $\to$ γ`1`). Since the merge relation is in fact functional in this direction, we can define a type family that computes that function. The type

[15] https://wiki.haskell.org/Foreign_Function_Interface
[16] https://wiki.haskell.org/Template_Haskell

```
instance CAdd' x σ γ γ' (CountNMinus1 γ') ⇒ CAdd x σ γ (N γ')
instance CSingletonCtx x σ γ'  ⇒ CAdd' x σ Empty γ' Z
instance CAddN' x σ γ γ' n     ⇒ CAdd' x σ γ (N γ') (S n)
instance Count γ ~ n           ⇒ CAddN' Z σ (Cons Nothing γ) (Cons (Just σ) γ) n
instance CSingletonCtx x σ γ'  ⇒ CAddN' (S x) σ (End τ) (Cons (Just τ) γ') (S Z)
instance CAddN' x σ γ γ' n     ⇒ CAddN' (S x) σ (Cons Nothing γ) (Cons Nothing γ') n
instance CAddN' x σ γ γ' (S n) ⇒ CAddN' (S x) σ (Cons (Just τ) γ) (Cons (Just τ) γ') (S (S n))
```

Figure 6. Instances of the `CAdd`, `CAdd'`, and `CAddN'` type classes.

```
class CMergeForward  (γ1 :: Ctx)  (γ2 :: Ctx)  (γ :: Ctx)  | γ1 γ2 → γ
class CMergeForwardN (γ1 :: NCtx) (γ2 :: NCtx) (γ :: NCtx) | γ1 γ2 → γ
instance CMergeForward Empty Empty Empty
instance CMergeForward Empty (N γ) (N γ)
instance CMergeForward (N γ) Empty (N γ)
instance CMergeNForward γ1 γ2 γ ⇒ CMergeForward (N γ1) (N γ2) (N γ)
instance CMergeNForward (End σ) (Cons Nothing γ2) (Cons (Just σ) γ2)
instance CMergeNForward (Cons Nothing γ1) (End σ) (Cons (Just σ) γ1)
instance CMergeNForward γ1 γ2 γ ⇒ CMergeNForward (Cons Nothing γ1) (Cons Nothing γ2) (Cons Nothing γ)
instance CMergeNForward γ1 γ2 γ ⇒ CMergeNForward (Cons (Just σ) γ1) (Cons Nothing γ2) (Cons (Just σ) γ)
instance CMergeNForward γ1 γ2 γ ⇒ CMergeNForward (Cons Nothing γ1) (Cons (Just σ) γ2) (Cons (Just σ) γ)
```

Figure 7. Type classes and instances for the merge relation.

```
type family Div (γ :: Ctx) (γ' :: Ctx) :: Ctx where
  Div γ     Empty  = γ
  Div (N γ) (N γ') = DivN γ γ'
type family DivN (γ :: NCtx) (γ' :: NCtx) :: NCtx where
  DivN (End _) (End _) = Empty
  DivN (Cons (Just _) γ) (End _) = N (Cons Nothing γ)
  DivN (Cons (Just _) γ) (Cons (Just _) γ') =
    ConsN Nothing (DivN γ γ')
  DivN (Cons (Just σ) γ) (Cons Nothing γ')  =
    ConsN (Just σ) (DivN γ γ')
  DivN (Cons Nothing  γ) (Cons Nothing  γ') =
    ConsN Nothing  (DivN γ γ')
type family ConsN (m :: Maybe LType) (γ :: Ctx) :: Ctx where
  ConsN Nothing  Empty = Empty
  ConsN (Just σ) Empty = N (End σ)
  ConsN m        (N γ) = N (Cons m γ)
```

Figure 8. The `Div` and `ConsN` type families on contexts

family `ConsN`, shown in Figure 8, adds an entry to the head of a possibly-empty context γ.

With the `Div` type family, also shown in Figure 8, we can satisfy the functional dependencies in `CMerge` as follows:

```
instance ( CMergeForward γ1 γ2 γ, CMergeForward γ2 γ1 γ
         , Div γ γ1 ~ γ2, Div γ γ2 ~ γ1)
      ⇒ CMerge γ1 γ2 γ
```

A.1 Helper functions: `lookup`, `add`, and `split`

Recall from Section 4 the definition of evaluation contexts and the signatures of the helper functions `lookup`, `add`, and `split`, which are used to instantiate embeddings of the linear languages.

```
data ECtx sig γ where
  ECtx :: (∀ x σ. Lookup γ x ~ Just σ ⇒ Sing x → LVal sig σ)
       ⇒ ECtx sig γ
lookup :: CSingleton x σ γ ⇒ Sing x → ECtx sig γ → LVal sig σ
add    :: CAdd x σ γ γ'
       ⇒ Sing x → LVal sig σ → ECtx sig γ → ECtx sig γ'
split  :: CMerge γ1 γ2 γ
       ⇒ ECtx sig γ → (ECtx sig γ1, ECtx sig γ2)
```

The runtime representation of evaluation contexts is just a function, and the implementations of `lookup` and `split` should not really modify this function; they should be no-ops at runtime. For `add`, the function should simply be updated to reflect the new binding. In order to convince the type system that this is valid, we must prove that the `Lookup` type family behaves appropriately with respect to the relations `CSingleton`, `CAdd`, and `CMerge`.

For the `CSingleton` type class, this amounts to showing that if `CSingleton x σ γ`, then `Lookup γ x ~ Just σ`. A common technique for doing this is adding this constraint to the `CSingleton` and `CSingletonN` type classes directly; the proof is build up by induction with each instance declaration, and it leaves no trace at runtime.

```
class Lookup  γ x ~ Just σ ⇒ CSingleton  x σ γ
class LookupN γ x ~ Just σ ⇒ CSingletonN x σ γ
```

In the implementation, we actually take this opportunity to prove other theorems in the same way, which aids in type checking:

1. If `CSingleton x σ γ`, then:
   a. γ is a well-formed context;
   b. `Lookup γ x ~ Just σ`;
   c. `γ ~ SingletonF x σ`; and
   d. `Remove x γ ~ Empty`.
2. If `CAdd x σ γ γ'` then:
   a. γ and γ' are well-formed contexts;
   b. `Lookup γ' x ~ Just σ`;

   c. `Remove x γ' ~ γ`; and
   d. `AddF x σ γ ~ γ'`.
3. If `CMerge γ1 γ2 γ` then:
   a. `Div γ γ2 ~ γ1`;
   b. `Div γ γ1 ~ γ2`; and
   c. each of $\gamma 1$, $\gamma 2$, and γ are well-formed.

Here, `SingletonF` is the type family that computes the singleton typing context, and similarly for `AddF`. The property that γ *is a well-formed context* further states that

1. `Div γ Empty ~ γ`;
2. `Div γ γ ~ Empty`;
3. `CMergeForward Empty γ γ`; and
4. `CMergeForward γ Empty γ`.

Using this, we can define `lookup`:

```
lookup :: CSingleton x σ γ ⇒ Sing x → ECtx sig γ → LVal sig σ
lookup x (ECtx f) = f x
```

For `add`, we will need to construct an evaluation context for γ', so `add x v (ECtx f)` will map `y` to `f y` when $y \neq x$, and to `v` when $y = x$. We can compare two singleton natural numbers using the `eqSNat` function, which produces either a proof that two nats are equal, or a proof that they are not equal. The `Dict` type provides type-level representation of constraints.

```
eqSNat :: ∀ (m :: Nat) (n :: Nat). Sing m → Sing n
       → Either (Dict (m ~ n)) (Dict ((m==n)~False))
```

Now we need to prove the following two properties:

1. If `CAdd x σ γ γ'` then `Lookup γ' x ~ Just σ` (which follows from the theorems stated above); and
2. If `CAdd x σ γ γ'` and $y \neq x$ then `Lookup γ' y ~ Lookup γ x`.

The second result is proved by induction over the structure of the `CAdd` relation, but because it refers to a type variable y that is not mentioned in the `CAdd` type class, we cannot prove it in the same way. Instead, we will need to add a "proof" of this theorem as a run-time method to the class, and build up the proof manually. The final type class declaration for the `CAdd` class is as follows:

```
class (γ'~AddF x σ γ, γ~Remove x γ', Lookup γ' x~Just σ)
  ⇒ CAdd (x :: Nat) (σ :: LType) (γ :: Ctx) (γ' :: Ctx)
  | x σ γ → γ', x γ' → σ γ
  where
    addLookupNEq :: (x == y) ~ False ⇒ Sing x → Sing y
                 → Dict (Lookup γ' y ~ Lookup γ y)
```

The `add` helper function calls out to this proof in the case that $y \neq x$.

```
add :: ∀ x σ γ γ'. CAdd x σ γ γ'
    ⇒ Sing x → LVal sig σ → ECtx sig γ → ECtx sig γ'
add x v (ECtx f) = ECtx $ \y → case eqSNat x y of
    Left  Dict → v
    Right Dict → case addLookupNEq @x @σ @γ @γ' x y of
                   Dict → f y
```

The `split` operation runs into a similar problem, and uses a similar technique to solve it. When `CMerge γ1 γ2 γ`, an evaluation context ρ for γ is also an evaluation context for both γ_1 and γ_2, since `Lookup γ1 x ~ Just σ` implies `Lookup γ x ~ Just σ`, and similarly for γ_2. We add the following proofs to the `CMergeForward` type class:

```
class CMergeForward γ1 γ2 γ | γ1 γ2 → γ where
  lookupMerge1 :: Lookup γ1 x ~ Just σ
               ⇒ Sing x → Dict (Lookup γ x ~ Just σ)
  lookupMerge2 :: Lookup γ2 x ~ Just σ
               ⇒ Sing x → Dict (Lookup γ x ~ Just σ)
```

The definition of `split` follows:

```
split :: ∀ γ1 γ2 γ. CMerge γ1 γ2 γ⇒ECtx γ→(ECtx γ1,ECtx γ2)
split (ECtx f) = (ECtx $ \x→case lookupMerge1 @γ1 @γ2 @γ x of
                   Dict → f x
                 ,ECtx $ \x→case lookupMerge2 @γ1 @γ2 @γ x of
                   Dict → f x)
```

B Modularly extending the deep embedding

In this section we describe an approach that lets us modularly extend the `LExp Deep` data type of the deep embedding from Section 4.1. Our approach uses the same trick of open recursion that we used for extending linear types.

```
data instance LExp Deep γ τ where
  Var :: CSingleton x σ γ ⇒ LExp Deep γ σ
  Dom :: Domain Deep dom ⇒ dom (LExp Deep) γ τ→LExp Deep γ τ
```

Notice that we elide `Abs` and `App` from our definition now; they can be defined independently as domains.

The `Dom` constructor takes an expression from a recursively-paramaterized data structure `dom`. For example, file handles use the following domain, which closely resembles the `HasFH` type class.

```
data FHDom (exp :: Ctx → LType → Type) :: Ctx → LType → Type
  where
    Open  :: String → FHDom exp Empty Handle
    Read  :: exp γ Handle → FHDom exp γ (Handle ⊗ Lower Char)
    Write :: exp γ Handle → Char → FHDom exp γ Handle
    Close :: exp γ Handle → FHDom exp γ One
```

When used by the `Dom` constructor, the parameter `exp` is replaced by `LExp Deep`, tying the knot. It is trivial to define the `HasFH` operators by wrapping their constructors with `Dom`, *e.g.*, `open = Dom . Open`.

The type class `Domain Deep dom` defines evaluation particularly for that domain, from which we can give a complete instance of `Eval` for the deep embedding.

```
class Domain sig dom where
  evalDomain :: Monad (Effect sig) ⇒ ECtx sig γ
             → dom (LExp sig) γ τ → Effect sig (LVal sig τ)
instance Eval Deep where
  eval γ Var     = return $ lookup γ
  eval γ (Dom e) = evalDomain γ e
```

All that remains now is to define an instance of `Domain` for file handles. First we define values of type `Handle` to be Haskell's time of built-in `IO` file handles, and we define the effect of the embedding to be `IO`.

```
data instance LVal Deep Handle = VHandle IO.Handle
type instance Effect Deep = IO
```

We implement evaluation using `IO` primitives to open and read from files (and similarly for `Write` and `Close`).

```
instance Domain Deep FHDom where
  evalDomain _ (Open s)    =
    VHandle <$> IO.openFile s IO.ReadWriteMode
  evalDomain γ (Read e)    = do
    VHandle h ← eval γ e
    c         ← IO.hGetChar h
    return $ VHandle h `VPair` VPut c
```

```
evalDomain γ (Write e c) = do VHandle h ← eval γ e
                              IO.hPutChar h c
                              return $ VHandle h
evalDomain γ (Close e)   = do VHandle h ← eval γ e
                              IO.hClose h
                              return VUnit
```

C Implementation of Session Types

We implement sessions as a pair `UChan` of untyped channels. We use a pair so that an actor will never send data and then receive that same data the next time they receive from the channel. Every time we construct a `UChan`, we also construct its swap, which corresponds to the other end of the channel.

```
type UChan = (Chan Any, Chan Any)
newU :: IO (UChan,UChan)
newU = do c1 ← IO.newChan
          c2 ← IO.newChan
          return ((c1,c2),(c2,c1))
```

These channels are untyped, but we will send and receive data of arbitrary types along them using

```
sendU :: UChan → a → IO ()
sendU (cin,cout) a = writeChan cout $ unsafeCoerce a
recvU :: UChan → IO a
recvU (cin,cout) = unsafeCoerce <$> readChan cin
```

The final operation on untyped channels is `linkU`, which takes as input two channels, and forwards all communication between them in both directions.

We define a new signature for sessions. Since we are using `IO` channels under the hood, the effect of the signature is `IO`. All values with this signature, no matter the type, are `UChans`.

```
data Sessions
data instance LVal Sessions τ = Chan UChan
type instance Effect Sessions = IO
```

We use a variant of the shallow embedding to encode expressions, which we represent as a function from evaluation contexts and an extra `UChan` to `IO ()`. The extra `UChan` is the output channel of the expressions; an expression of type $\sigma \otimes \tau$ will send a value σ on its output channel, for example.

```
data instance LExp Sessions γ τ =
    SExp {runSExp :: SCtx Sessions γ → UChan → IO ()}
```

To evaluate an expression, we first construct a new channel with `newU`, which outputs the two endpoints of the new channel. Then we call `runSExp` on the expression with one of the endpoints, and return the other endpoint.

```
instance Eval Sessions where
    eval e γ = do (c,c') ← newU
                  forkIO $ runSExp e γ c
                  return $ Chan c'
```

In the implementation we provide instances for `HasLolli`, `HasTensor`, `HasOne`, and `HasLower`, the last of which we illustrate here. To construct an expression of type `Lower` τ via `put a`, we simply send the Haskell value `a` over the output channel.

```
    put a = SExp $ \_ c → sendU c a
```

To implement `e >! f`, we spawn a new channel and pass one end to `e`. Then we wait for a value from the other end, to which we apply `f`.

```
    e >! f = SExp $ \ρ c → do let (ρ1,ρ2) = split ρ
                              (x,x') ← newU
                              forkIO $ runSExp e ρ1 x
                              a ← recvU x'
                              runSExp (f a) ρ2 c
```

D Quantum computing

D.1 A dependently typed Quantum Fourier Transform

We can take advantage of GHC's dependent types to describe a dependent quantum Fourier transform (QFT) [Paykin et al. 2017]. First, we define a `Nat`-indexed type family describing the `n`-ary tensor of a linear type.

```
type family (σ :: LType) ∏ (n :: Nat) :: LType where
    σ ∏ Z = One
    σ ∏ (S (S n)) = σ ⊗ (σ ∏ S n)
```

The quantum fourier transform depends on an operation `rotations`, which we omit here. The quantum fourier transform is defined recursively as follows, where `Hadamard::Unitary Qubit`.

```
fourier :: HasQuantum exp ⇒ Sing n → LStateT (Qubit ∏ n) ()
fourier SZ      = return ()
fourier (SS SZ) = suspendT . λ $ unitary Hadamard ⊗ put ()
fourier (SS m)  = suspendT . λpair $ \(q,qs) →
                  forceT (fourier m) ^ qs `letin` \qs →
                  forceT (rotations (SS m) m) ^ (q ⊗ qs)
  where rotations :: Sing m → Sing n
                  → Lift exp (Qubit ∏ S n ⊸ Qubit ∏ S n)
```

The `Sing n` data family is a runtime representation of the natural number `n`, from the singletons library, with constructors `SZ :: Sing Z` and `SS :: Sing n → Sing (S n)`. The operation λ`pair` combines abstraction and `letPair` to match against the input to the λ.

D.2 Implementation

We implement the quantum signature using the deep embedding rather than the shallow, as in the future we are interested in compiling and optimizing quantum computations. Thus we define a domain to plug into the deep embedding:

```
data QuantumExp exp :: Ctx → LType → Type where
  New     :: Bool → QuantumExp exp Empty Qubit
  Meas    :: exp γ Qubit → QuantumExp exp γ (Lower Bool)
  Unitary :: Unitary σ → exp γ σ → QuantumExp exp γ σ
```

As is usual with the deep embedding, it is easy to show that it satisfies the `HasQuantum` class.

There are many computational models available for simulating quantum computations, and our implementation chooses one based on density matrices [Nielsen and Chuang 2010]. We will not go into the details of this simulation here, but the outward-facing interface has three (monadic) operations, where `DensityMonad` is a probabilistic state monad on density matrices. Qubits are identified with integers that index into the matrix.

```
  newM          :: Bool → DensityMonad Int
  applyUnitaryM :: Mat (2^m) (2^m) → [Int] → DensityMonad ()
  measM         :: Int → DensityMonad Bool
```

Values of type `Qubit` are integer qubit identifiers, and `DensityMonad` is the effect.

```
data instance LVal Deep Qubit = QId Int
type instance Effect Deep = DensityMonad
```

The implementation is completed with a `Domain` instance, which we omit here.

D.3 Related work

Other approaches to higher-order quantum computing in Haskell have been proposed. The Quantum IO monad [Altenkirch and Green 2009] features a monadic approach to quantum computing that separates reversible (*e.g.,* unitary) computations from those containing measurement. Unlike the quantum lambda calculus, the Quantum IO monad is not type safe and may fail at runtime. Quipper [Green et al. 2013] is a scalable quantum circuit language embedded in Haskell and has a similar problem, although two closely related core calculi have been proposed that use linear types for safe quantum circuits [Paykin et al. 2017; Ross 2015].

Acknowledgments

This work is supported by the NSF under Grant No. CCF-1421193. Thanks to the anonymous reviewers, as well as Antal Spector-Zabusky, Kenneth Foner, Richard Eisenberg, and Stephanie Weirich for many helpful discussions about this work.

References

Thorsten Altenkirch and Alexander S. Green. 2009. *The Quantum IO Monad.* Cambridge University Press, 173–205. https://doi.org/10.1017/CBO9781139193313.006

Erik Barendsen and Sjaak Smetsers. 1993. Conventional and uniqueness typing in graph rewrite systems. In *Proceedings of the 13th Conference of Foundations of Software Technology and Theoretical Computer Science*, Rudrapatna K. Shyamasundar (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 41–51. https://doi.org/10.1007/3-540-57529-4_42

Nick Benton. 1995. A mixed linear and non-linear logic: Proofs, terms and models. In *Computer Science Logic*, Leszek Pacholski and Jerzy Tiuryn (Eds.). Lecture Notes in Computer Science, Vol. 933. Springer Berlin Heidelberg, 121–135. https://doi.org/10.1007/BFb0022251

Nick Benton, Gavin Bierman, Valeria de Paiva, and Martin Hyland. 1993. A term calculus for Intuitionistic Linear Logic. In *Typed Lambda Calculi and Applications*, Marc Bezem and JanFriso Groote (Eds.). Lecture Notes in Computer Science, Vol. 664. Springer Berlin Heidelberg, 75–90. https://doi.org/10.1007/BFb0037099

Nick Benton and Philip Wadler. 1996. Linear logic, monads and the lambda calculus. In *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science, 1996. LICS '96.* 420–431. https://doi.org/10.1109/LICS.1996.561458

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. Retrofitting Linear Types. (2017). https://www.microsoft.com/en-us/research/wp-content/uploads/2017/03/haskell-linear-submitted.pdf

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR 2010 - Concurrency Theory*, Paul Gastin and François Laroussinie (Eds.). Lecture Notes in Computer Science, Vol. 6269. Springer Berlin Heidelberg, 222–236. https://doi.org/10.1007/978-3-642-15375-4_16

Chih-Ping Chen and Paul Hudak. 1997. Rolling your own mutable ADT—a connection between linear types and monads. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '97.* Association for Computing Machinery (ACM). https://doi.org/10.1145/263699.263708

Richard Eisenberg, Benoıt Valiron, and Steve Zdancewic. 2012. Typechecking Linear Data: Quantum Computation in Haskell. (2012).

Richard A. Eisenberg and Jan Stolarek. 2014. Promoting Functions to Type Families in Haskell. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell (Haskell '14).* ACM, New York, NY, USA, 95–106. https://doi.org/10.1145/2633357.2633361

Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. 2016. Visible Type Application. In *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, April 2–8, 2016*, Peter Thiemann (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 229–254. https://doi.org/10.1007/978-3-662-49498-1_10

Matthew Fluet, Greg Morrisett, and Amal Ahmed. 2006. Linear Regions Are All You Need. In *Programming Languages and Systems*. Springer Science + Business Media, 7–21. https://doi.org/10.1007/11693024_2

Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: A Scalable Quantum Programming Language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13).* ACM, New York, NY, USA, 333–342. https://doi.org/10.1145/2491956.2462177

The Idris Community. 2017. Uniqueness Types. (2017). http://docs.idris-lang.org/en/latest/reference/uniqueness-types.html

Mark P. Jones. 2000. Type Classes with Functional Dependencies. In *Programming Languages and Systems: 9th European Symposium on Programming, ESOP 2000*, Gert Smolka (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 230–244. https://doi.org/10.1007/3-540-46425-5_15

P.B. Levy. 2012. *Call-By-Push-Value: A Functional/Imperative Synthesis.* Springer Netherlands.

Sam Lindley and J. Garrett Morris. 2015. A Semantics for Propositions as Sessions. In *Proceedings of Programming Languages and Systems, 24th European Symposium on Programming, ESOP 2015*, Jan Vitek (Ed.), Vol. 9032. Springer Berlin Heidelberg, London, UK, 560–584. https://doi.org/10.1007/978-3-662-46669-8_23

Sam Lindley and J. Garrett Morris. 2016. Embedding Session Types in Haskell. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016).* ACM, New York, NY, USA, 133–145. https://doi.org/10.1145/2976002.2976018

Saunders Mac Lane. 1978. *Categories for the working mathematician.* Vol. 5. Springer Science & Business Media.

Nicholas D. Matsakis and Felix S. Klock, II. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology (HILT '14).* ACM, New York, NY, USA, 103–104. https://doi.org/10.1145/2663171.2663188

Karl Mazurak and Steve Zdancewic. 2010. Lolliproc: To Concurrency from Classical Linear Logic via Curry-Howard and Control. *SIGPLAN Not.* 45, 9 (Sept 2010), 39–50. https://doi.org/10.1145/1932681.1863551

Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. 2010. Lightweight linear types in system F°. In *Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation - TLDI '10.* Association for Computing Machinery (ACM). https://doi.org/10.1145/1708016.1708027

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

J. Garrett Morris. 2016. The Best of Both Worlds: Linear Functional Programming Without Compromise. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016).* New York, NY, USA, 448–461. https://doi.org/10.1145/2951913.2951925

M.A. Nielsen and I.L. Chuang. 2010. *Quantum Computation and Quantum Information: 10th Anniversary Edition.* Cambridge University Press.

James Noble, Jan Vitek, and John Potter. 1998. Flexible alias protection. In *ECOOP'98 — Object-Oriented Programming: 12th European Conference Brussels, Belgium, July 20–24, 1998 Proceedings*, Eric Jul (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 158–185. https://doi.org/10.1007/BFb0054091

E. G. J. M. H. Nöcker, J. E. W. Smetsers, M. C. J. D. van Eekelen, and M. J. Plasmeijer. 1991. *Concurrent clean.* Springer Berlin Heidelberg, Berlin, Heidelberg, 202–219. https://doi.org/10.1007/3-540-54152-7_66

Dominic Orchard and Nobuko Yoshida. 2016. Effects As Sessions, Sessions As Effects. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16).* ACM, New York, NY, USA, 568–581. https://doi.org/10.1145/2837614.2837634

Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: A Core Language for Quantum Circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017).* ACM, New York, NY, USA, 846–858. https://doi.org/10.1145/3009837.3009894

Jeff Polakow. 2015. Embedding a full linear Lambda calculus in Haskell. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015.* 177–188. https://doi.org/10.1145/2804302.2804309

François Pottier and Jonathan Protzenko. 2013. Programming with permissions in Mezzo. *ACM SIGPLAN Notices* 48, 9 (nov 2013), 173–184. https://doi.org/10.1145/2544174.2500598

Riccardo Pucella and Jesse A. Tov. 2008. Haskell Session Types with (Almost) No Class. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell (Haskell '08).* ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/1411286.1411290

Robert Rand, Jennifer Paykin, and Steve Zdancewic. 2017. QWIRE Practice: Formal Verification of Quantum Verification in Coq. (2017). Quantum Physics and Logic (QPL), July 3–7, 2017, Nijmegen, Amsterdam.

Neil J. Ross. 2015. *Algebraic and Logical Methods in Quantum Computation.* Ph.D. Dissertation. Dalhousie University.

Peter Selinger and Benoît Valiron. 2009. *Quantum Lambda Calculus.* Cambridge University Press, 135–172. https://doi.org/10.1017/CBO9781139193313.005

Philip Wadler. 1990. Linear types can change the world!. In *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel.* North Holland.

Philip Wadler. 2014. Propositions as sessions. *Journal of Functional Programming* 24 (2014), 384–418. Issue Special Issue 2-3. https://doi.org/10.1017/S095679681400001X