The Linearity Monad

JENNIFER PAYKIN, University of Pennsylvania
STEVE ZDANCEWIC, University of Pennsylvania

We introduce a technique for programming with domain-specific linear languages using a 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 our 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. 2016. The Linearity Monad. 1, 1, Article 1 (January 2016), 26 pages.
DOI:

1 INTRODUCTION

For years, linear types have been used for effectful domain-specific languages with great success. For the domains of memory management (Amani et al. 2016; Fluet et al. 2006; Pottier and Protzenko 2013) and 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 typing disciplines.

Consider the following interface for linear file handles. Here, ⊸ (pronounced "lollipop") denotes linear implication, ⊗ ("tensor") denotes the multiplicative linear product, and `One` denotes the multiplicative unit.

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

In this setting linearity rules out two specific kinds of errors.[1] 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.

Linear types are useful in this domain because they statically enforce properties that are inexpressible using conventional "unrestricted" types. This principle extends to other domains as well. For mutable state, linear types enforce a single-threadedness property that allows functional operations such as `writeArray :: Array` α ⊸ α ⊸ `Array` α to be implemented as mutable updates (Wadler 1990). For concurrent session types, linearity statically enforces the fact that every channel has exactly two endpoints that obey complementary communication protocols (Caires

[1]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.

and Pfenning 2010). For quantum computing, linear types enforce the "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 typing disciplines are often domain-specific, meaning that new applications of linear types must be added by the language designer, not the user. Second, linear type systems are often unwieldy, with linear typing information bleeding into programs that are entirely non-linear. Language designers must take great care to ensure that the costs of a linear system aren't paid for programs that work entirely with unrestricted data.[2]

To see the problem of "linearity creep", 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 have to be written something like `\s → let! s' = s in !s' ++ !s'`, of type `!String ⊸ !String`. The type $!\alpha$, pronounced "bang α", denotes 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 a `!`, as in `!e`, which guarantees 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 code bases. The problem is that $!\alpha$ presents the wrong defaults, presenting linearity as the default and non-linearity as the exception; programmers expect the opposite.

Over the years, various linear type systems have been introduced to mitigate the problems 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). To date, these proposals have seen little adoption, at least in part because they are not compatible with existing languages. In addition, in each of these cases the application domain is fixed by the language designers and cannot be easily extended.

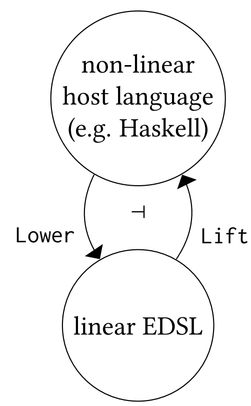

Fig. 1. The linear/non-linear programming model.

We propose a different approach, inspired by Benton's linear/non-linear (LNL) logic (1994). The linear/non-linear 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 an embedding of a simple linear lambda calculus inside an existing non-linear programming language. The embedded language approach is easily extensible to 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.

When the host language supports monadic programming, as Haskell does, the LNL interface reveals a connection with monads. It is already 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 (Benton and Wadler 1996).

This structure, which we call the *linearity monad*, is the main focus of this work.

[2]A notable exception is the use of ownership types for safe memory management in languages like Rust (Matsakis and Klock 2014) and Clean (Smetsers et al. 1994). Ownership types integrate well with nonlinear data, but they are weaker than full linear types.

1.1 Contributions

In this paper we show how to realize the LNL structure as the embedding of 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 aim for a design that allows various application domains to be expressed modularly in the system.

In exploring this design space, we make 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 employs Haskell's type class mechanism to automatically discharge linearity constraints, which requires careful structuring of the proof search space. We can instantiate the framework with either a shallow embeddings of judgments as Haskell functions, or a deep embedding using generalized algebraic data types (GADTs). Throughout, the framework uses the dependently-typed features of the Glasgow Haskell Compiler[3] (GHC) to enforce the linear use of typing judgments.

(3) Finally, we demonstrate the effectiveness of our framework by implementing numerous examples of domain-specific linear languages in our framework (Section 6), including:
  - Safe file handles in the style of Mazurak et al. (2010);
  - Mutable arrays in the style of Wadler's "Linear types can change the world!" (1990);
  - Session types in the style of Caires and Pfenning (2010); and
  - Quantum computing in the style of Selinger and Valiron (2009).

The implementation and all of the examples described in this paper are included in the supplementary materials.

1.2 The File Handle Example

To see how the file handle example plays out when implemented in our framework, we first express the desired interface via the following type class:[4]

```
class HasFH (exp :: Ctx → LType → Type) 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 parameter `exp` represents the typing judgment for a linear language: a Haskell value `e` of type `exp` γ σ satisfies $\gamma \vdash e : \sigma$, where σ is a linear type and γ is a linear typing context. The inhabitants of the class represent inference rules: for any string s we have $\cdot \vdash$ `open` s :: `Handle`, and for any $\gamma \vdash e$ :: `Handle` we have $\gamma \vdash$ `close` e :: `One`.

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

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

[3] https://www.haskell.org/ghc/

[4] Here, `Type` is the kind of Haskell types, `LType` is the kind of linear types, and `Handle`, `One`, and `Lower Char` all have kind `LType`.

To write more substantial programs we use syntax for manipulating pairs $\sigma \otimes \tau$, units `One`, functions $\sigma \multimap \tau$, and lowered Haskell types `Lower` α. Each of these has an associated type class—`HasTensor`, `HasOne`, `HasLolli`, and `HasLower`—that classifies the introduction and elimination forms for that type. Consider the following function, which reads a character from a file and writes that same character back to the file twice:

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

The λ constructx a linear function using higher-order abstract syntax, *i.e.,* the identity function is λ `(\x → x)`. The operation `letPair` decomposes the result of the read ($\gamma \vdash$ `read h : Handle` $\otimes$ `Lower Char`) into variables `h : Handle` and `x : Lower Char`. The infix bind operation >! (pronounced "let bang") turns a linear expression of type `Lower Char` into an ordinary character c :: `Char`, which can be duplicated like any other Haskell value.

1.2.1 The monad. In the `readWriteTwice` example we saw the linear `Lower` type that holds arbitrary Haskell values. The `Lift` operator is the opposite: an ordinary Haskell data type containing linear expressions. The only exception is that `Lift` only holds *closed* linear expressions, which we can think of as reproducible procedures that produce linear results. In other words, a value of type `Lift exp` σ is a *suspended* linear computation. The composition of `Lift` and `Lower` makes up the linearity monad, which we write `Lin`.

```
data Lift exp σ = Suspend { force :: exp Empty σ }
data Lin exp α = Lin (Lift exp (Lower α))
```

The linearity monad is often the outward interface to a linear program. Consider the following function that takes in a file name, opens a file handle with that name, performs some operation, and closes the handle again.

```
withFile :: HasFH exp ⇒ String → Lift exp (Handle ⊸ Handle ⊗ Lower α) → Lin exp α
withFile name f = Lin . Suspend $ force f ^ open name `letPair` \(h,a) →
                                    close h `letUnit` a
```

Note, the infix operator `(^)` encodes linear application, and `letUnit` eliminates terms of type `One`. In the remainder of this section we assume that `HasFH exp` also includes the constraints `HasLolli exp`, `HasTensor exp`, *etc.*

1.2.2 The linear monad type class. We can go a step further by recognizing that the intermediate operation in `withFile`, *i.e.,* the function of type `Lift exp (Handle ⊸ Handle ⊗ Lower a)`, is a linear version of a state monad. We can define a type class for such `LMonads`, which have the usual monad operations, but operate on linear data:

```
class LMonad exp (m :: LType → LType) where
   lreturn :: Lift exp (τ ⊸ m τ)
   lbind   :: Lift exp (m σ ⊸ (σ ⊸ m τ) ⊸ m τ)
```

We can reformulate the `read` and `write` operations so that they highlight the linear state monad, with `readM :: LState Handle (Lower Char)` and `writeM :: Char → LState Handle One`. We then present the `readWriteTwice` example using the `LMonad` operators, where ⋙= is an infix version of `lbind`:

```
readWriteTwiceM :: HasFH exp ⇒ Lift exp (LState Handle One)
readWriteTwiceM = Suspend $ readM   ⋙= \x → x >! \c →
                            writeM c ⋙= \y → y `letUnit` writeM c
```

1.2.3 The monad transformer. Since `readM` always returns a lowered non-linear `Char`, and `writeM` always returns a unit type, we rather inconveniently need to eliminate the results of a bind using >! and `letUnit`. We can avoid this extra step by combining the linearity monad and the `LMonad` type class to form a *monad transformer* when the result of the `LMonad` is a lowered type.

	linear	**non-linear**
types	$\sigma, \tau ::= \sigma \multimap \tau \mid \sigma \otimes \tau \mid \cdots$	$\alpha, \beta ::= \alpha \to \beta \mid (\alpha, \beta) \mid \cdots$
variables	x, y	a, b
typing contexts	$\gamma ::= \cdot \mid \gamma, x : \sigma$	$\Gamma ::= \cdot \mid \Gamma, a : \alpha$
expressions	e	t
typing judgment	$\gamma \vdash e : \sigma$	$\Gamma \vdash t : \alpha$

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

```
data LinT exp (m :: LType → LType) α = LinT (Lift exp (m (Lower α)))
```

For any `m` with an `LMonad` instance, we can give a `Monad` instance to `LinT exp m`.

Again, we can construct versions of `read` and `write` in this monad: `readT :: LinT (LState Handle) Char` and `writeT :: Char → LinT (LState Handle) ()`. Now the `readWriteTwice` function can be defined entirely in the non-linear world, since the linearity monad hides all of the linear "plumbing."

```
readWriteTwiceT :: HasFH exp ⇒ LinT exp (LState Handle) ()
readWriteTwiceT = do c ← readT
                     writeT c
                     writeT c
```

2 LINEAR/NON-LINEAR TYPES

Linear/non-linear (LNL) logic, introduced by Benton (1994), is a model of linear logic obtained by combining two very simple type systems. One system is a entirely linear language, meaning that all variables are linear and there is no unrestricted modality $!\sigma$. The other system 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.* We assume only that the linear type system follows the usual constraints of linear logic. As a starting point, consider the standard presentation of a linear lambda calculus with application and abstraction, where we write $\gamma \vdash e : \tau$ for its typing judgment.

$$\frac{\gamma = x : \tau}{\gamma \vdash x : \tau}\ \textsc{var} \qquad \frac{\gamma' = \gamma, x : \sigma \qquad \gamma' \vdash e : \tau}{\gamma \vdash \lambda x.e : \sigma \multimap \tau}\ \textsc{abs} \qquad \frac{\gamma = \gamma_1 \uplus \gamma_2 \qquad \gamma_1 \vdash e_1 : \sigma \multimap \tau \qquad \gamma_2 \vdash e_2 : \sigma}{\gamma \vdash e_1 e_2 : \tau}\ \textsc{app}$$

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 as usual. In APP, the relation $\gamma = \gamma_1 \uplus \gamma_2$ means that γ is the disjoint union of γ_1 and γ_2; it enforces the fact that variables cannot be duplicated so as to occur on both sides of an application.

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.

The linear/non-linear type system modifies these two languages so that they interact in a predictable way. This modification happens in three steps.

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.

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

$$\frac{\gamma = x : \tau}{\Gamma; \gamma \vdash x : \tau}\ \textsc{var} \qquad \frac{\gamma' = \gamma, x : \sigma \qquad \Gamma; \gamma' \vdash e : \tau}{\Gamma; \gamma \vdash \lambda x.e : \sigma \multimap \tau}\ \textsc{abs} \qquad \frac{\gamma = \gamma_1 \uplus \gamma_2 \qquad \Gamma; \gamma_1 \vdash e_1 : \sigma \multimap \tau \qquad \Gamma; \gamma_2 \vdash e_2 : \sigma}{\Gamma; \gamma \vdash e_1 e_2 : \tau}\ \textsc{app}$$

$$\frac{\Gamma \vdash t : \alpha}{\Gamma; \cdot \vdash \mathsf{put}\ t : \texttt{Lower}\ \alpha}\ \textsc{put} \qquad \frac{\gamma = \gamma 1 \uplus \gamma 2 \qquad \Gamma; \gamma 1 \vdash e : \texttt{Lower}\ \alpha \qquad \Gamma, a : \alpha; \gamma 2 \vdash e' : \tau}{\Gamma; \gamma \vdash \texttt{let!}\ a = e\ \texttt{in}\ e' : \tau}\ \textsc{let!}$$

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

Fig. 3. Typing rules for the combined linear/non-linear type system

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 we call `put`. Thus, every linear expression of type `Lower` α morally holds a non-linear value. The elimination form, `let!` $a = e$ `in` e', allows us to 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, in fact, safe to duplicate it. 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.[5]

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.

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

[5]The `suspend` and `force` notation is inspired by Call-By-Push-Value (Levy 2003), 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.

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.

3 EMBEDDING A LINEAR TYPE SYSTEM IN HASKELL

To embed a linear language in Haskell, there are a number of design decisions to make. How will we encode variables and typing contexts? How are linear expressions and their typing judgments represented? How can we infer the typing contexts and ensure that variables are used linearly? This section answers these questions by building up successively more expressive linear languages. The general strategy we use is to build Haskell data structures for linear types and contexts, and then to impose constraints on those contexts using type classes. As we will see, our design permits an extremely flexible representation of linear terms.

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.

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 `LType`s. 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, consider the following sample derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{[\texttt{Nothing}, \texttt{Nothing}, \texttt{Just}\ (\sigma \multimap \tau)] \vdash 2 : \sigma \multimap \tau}\ \textsc{var} \quad \dfrac{}{[\texttt{Just}\ \sigma \vdash 0 : \sigma]}\ \textsc{var}}{[\texttt{Just}\ \sigma, \texttt{Nothing}, \texttt{Just}\ (\sigma \multimap \tau)] \vdash 2\ 0 : \tau}\ \textsc{app}}{[\texttt{Just}\ \sigma] \vdash \lambda\, 2.\ 2\ 0 : (\sigma \multimap \tau) \multimap \tau}\ \textsc{abs}}{[] \vdash \lambda\, 0.\ \lambda\, 2.\ 2\ 0 : \sigma \multimap (\sigma \multimap \tau) \multimap \tau}\ \textsc{abs}$$

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

$$\texttt{[Nothing,Nothing,Just}(\sigma \multimap \tau)\texttt{]} \uplus \texttt{[Just}\ \sigma\texttt{]} = \texttt{[Just}\ \sigma\texttt{,Nothing,Just}\ (\sigma \multimap \tau)\texttt{]}$$

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, the merge relation is defined as:

```
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
```

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. 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.

```
class CSingleton  (x :: Nat) (σ :: LType) (γ :: Ctx)  | x σ → γ, γ → x σ
instance CSingletonN x σ γ ⇒ CSingleton x σ (NCtx γ)
class CSingletonN (x :: Nat) (σ :: LType) (γ :: NCtx) | x σ → γ, γ → x σ
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. 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 $\gamma 2$.

```
class CAdd (x :: Nat) (σ :: LType) (γ :: Ctx) (γ' :: Ctx) | x σ γ → γ', x γ' → σ γ, γ γ' → x σ
class CMerge (γ1 :: Ctx) (γ2 :: Ctx) (γ :: Ctx) | γ1 γ2 → γ, γ1 γ → γ2, γ2 γ → γ1
```

Deriving these functional dependencies is not straightforward, and in the implementation we use a number of helper classes to convince GHC that they hold. 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 term $\gamma \vdash e : \tau$ in the linear lambda calculus is represented as a value `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.[6] The application operator corresponds closely to the APP inference rule given in Figure 3, where `CMerge` encodes the disjoint union of contexts. The abstraction operation, which we write λ, 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 $\gamma'' = x : \sigma$.

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

The HOAS encoding leads to very natural-looking code. The identity function is just `λ (\x → x)`, while composition is defined as follows:[7]

```
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 Units, pairs, and sums

It easy to extend the language for other operators of linear logic, such as units, pairs ⊗, and sums ⊕. 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
```

For the operators ⊗ and ⊕, 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`:

```
data LType = One | Lolli LType LType | MkLType (ext LType)
```

Extensions, denoted with the meta-variable `ext`, are paramaterized by a type. The multiplicative product ⊗ 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.

$$\frac{\gamma 1 \vdash e1 : \tau 1 \qquad \gamma 2 \vdash e2 : \tau 2 \qquad \gamma = \gamma 1 \uplus \gamma 2}{\gamma \vdash e1 \otimes e2 : \tau 1 \otimes \tau 2} \qquad \frac{\gamma 1 \vdash e : \sigma 1 \otimes \sigma 2 \qquad \gamma 2, x1 : \sigma 1, x2 : \sigma 2 \vdash e' : \tau \qquad \gamma = \gamma 1 \uplus \gamma 2}{\gamma \vdash \text{let } (x1, x2) = e \text{ in } e' : \tau}$$

We overload the constructor (⊗) 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 :: ( CMerge γ1 γ2 γ, CAdd x1 σ1 γ2 γ2', CAdd x2 σ2 γ2' γ2''
               , CSingleton x1 σ1 γ21, CSingleton x2 σ2 γ22, x1 ~ Fresh γ2, x2 ~ Fresh γ2')
             ⇒ 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 $\gamma 2'' = \gamma 2, x1 : \sigma 1, x2 : \sigma 2$. 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 `σ ⊗ τ ⊸ τ ⊗ σ`.[8]

In the implementation we also provide 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.

---

[7]The ($) operator is Haskell notation for ordinary function application.

[8]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.

```
curry :: HasMILL exp ⇒ Lift exp ((σ1 ⊗ σ2 ⊸ τ) ⊸ σ1 ⊸ σ2 ⊸ τ)
curry = Suspend . λ $ \f → λ $ \x1 → λ $ \x2 → f ^ (x1 ⊗ x2)
uncurry :: HasMILL exp ⇒ Lift exp ((σ1 ⊸ σ2 ⊸ τ) ⊸ σ1 ⊗ σ2 ⊸ τ)
uncurry = Suspend . λ $ \f → λ $ \x → x `letPair` \(x1,x2) → f ^ x1 ^ x2

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
```

Fig. 4. Examples of linear code. `HasMILL exp` is a synonym for `(HasLolli exp, HasTensor exp, HasOne exp)`, and `HasMELL exp` is a synonym for `(HasMILL exp, HasLower exp)`.

```
    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 type:

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

3.5 Examples

Figure 4 presents some simple examples of linear programs. First we define operations to curry and uncurry linear functions. For convenience, we define synonyms for classes of constraints, such as `HasMILL` for multiplicative intuitionistic linear logic connectives (⊸, ⊗, and `One`) and `HasMELL` for multiplicative exponential linear logic connectives (the MILL connectives plus `Lower`). Figure 4 also encodes the $!\tau$ operator from linear logic as the composition of `Lower` and `Lift` and shows that terms of type `Bang τ` can in fact be duplicated and discarded.

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. Consequently, we define a separate type of value for every linear type using data families[9]. 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 Haskell's primitive libraries, which

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

means in this domain case 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 will thus have three connected 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, which can be thought of as a store, is a finite map from variables to values. It is indexed by a signature indicating which values will be carried, 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`. The structure of an evaluation context mirrors the structure of the typing context it is indexed by.

```
data ECtx (sig :: Type) (γ :: Ctx) where
    ENothing :: ECtx sig Empty
    ENEmpty  :: ENCtx sig γ → ECtx sig (NEmpty γ)
data ENCtx (sig :: Type) (γ :: NCtx) where ⋯
```

Now evaluation is given 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)
```

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 ⇒ LExp Deep γ τ
    Abs :: CAdd x σ γ γ'     ⇒ 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
```

It is therefore quite easy to instantiate the `HasLolli` type class, although we must use explicit type application[10] (*e.g.*, `@x`) to specify which variable should be used in the `Var` and `Abs` constructors.

```
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)
    (^)  = 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 One      = VUnit
data instance LVal Deep (σ ⊗ τ) = VPair (LVal Deep σ) (LVal Deep τ)
data instance LVal Deep (σ ⊸ τ) where
    VAbs :: CAddCtx x σ γ γ' ⇒ ECtx Deep γ → LExp Deep γ' τ → LVal Deep (σ ⊸ τ)
```

[10] https://ghc.haskell.org/trac/ghc/wiki/ExplicitTypeApplication

```
data instance LVal Deep (Lower α) = VPut α
```

Next we define evaluation as an interpreter. When the expression is an abstraction we return the closure.

```
instance Eval Deep where
    eval :: Monad (Effect Deep) ⇒ ECtx Deep γ → LExp Deep γ τ → Effect Deep (LVal Deep τ)
    eval γ (Abs e) = return $ VAbs γ 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 a `lookup` operation.

```
    eval γ Var = return $ lookup γ
```

In the application case, first we evaluate `e1` to obtain a closure, then evaluate `e2`. We evaluate the body of the closure under its evaluation context extended with the value of `e2`.

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

This operation uses two additional helper functions to manipulate contexts in a way similar to `lookup`. The function `add` takes an evaluation context for γ and a value of type σ, and produces an evaluation context for $\gamma, x : \sigma$. The use site must specify x, which in this case is `Fresh γ2`. 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.

These helper functions are readily defined by induction over the structure of the relations `CSingleton`, `CMerge`, and `CAdd`, so we amend the declarations of these type classes to include these functions. They are instantiated when the instances for each class are given.

```
class CSingleton  (x :: Nat) (σ :: LType) (γ :: Ctx)  | x σ → γ, γ → x σ where
    lookup :: ECtx γ → LVal σ
class CAdd (x :: Nat) (σ :: LType) (γ :: Ctx) (γ' :: Ctx) | x σ γ → γ', x γ' → σ γ where
    add :: LVal σ → ECtx γ → ECtx γ'
class CMerge (γ1 :: Ctx) (γ2 :: Ctx) (γ :: Ctx) | γ1 γ2 → γ, γ1 γ → γ2, γ2 γ → γ1 where
    split :: ECtx γ → (ECtx γ1, ECtx γ2)
```

4.2 Modularly extending the deep embedding

To naively extend the syntax of the deep embedding, 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 redefine the data type and the entire evaluation function. Instead, we want a solution that lets modularly extend the `LExp Deep` data type. We do this using 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
```

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 → exp γ Handle
    Close :: exp γ Handle → 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
```

4.3 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 all 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
```

4.4 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:

$$\texttt{put a >! f} =_\beta \texttt{f a} \qquad \texttt{e} :: \texttt{exp}\ \gamma\ (\texttt{Lower}\ \alpha) =_\eta \texttt{e >! put}$$

(The astute reader may recognize a similarity to the monad laws, which we will discuss in depth in Section 5.)

Proposition 4.1. *For the shallow embedding, the β and η equalities for* `Lower` α *hold.*

Proof Sketch. We start with the β rule. Unfolding definitions, we see that:

```
put a >! f = (SExp $ \_ → return $ VPut a) >! f
           = SExp $ \γ → let (γ1,γ2) = split γ
                         in (\_ → return $ VPut a) γ1 ≫= \(VPut a) → runSExp (f a) γ2
```

Since `γ1 :: SCtx Empty` we know that `γ1=SEmpty` and `γ2=γ`. Recall that the instance of `HasLower (LExp Shallow)` assumes that `Effect Shallow` is a monad. By the monad laws, this is therefore equal to the desired result:

```
(return (VPut a) ≫= \(VPut a) → f a γ)  = runSExp (f a) γ
```

The proof of the η rule is similarly obtained by unfolding defitions and applying the monad laws. □

For the deep embedding, the situation is less straightforward. Clearly `LetBang (Put a) f` is not syntactically equal to `f a`, but they are semantically equal under evaluation.

Definition 4.2. An interpretation instance for `Eval sig` is correct for `HasLower` if, for all evaluation contexts γ:

- for all `a` :: α and `f` :: $\alpha \to$ `LExp sig` τ, we have `eval` γ `(put a >! f) = eval` γ `(f a)`; and
- for all `e` :: `LExp sig (Lower` α`)`, we have `eval` γ `e = eval` γ `(e >! put)`.

It is the immediate consequence of Proposition 4.1 that the `Eval` instance for `Lower` α is correct. We can prove a similar result for the deep embedding, although we omit the proof.

Proposition 4.3. *The interpretation instance for* `Eval Deep` *is correct for* `HasLower`.

5 THE MONAD

Benton (1994) originally proposed linear/non-linear logic as a proof theory for understanding linear logic. Through the Curry-Howard correspondence we have interpreted it as a type system, but we can also draw on the categorical interpretation also explored by Benton. 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, which we call LINEAR, has objects that are elements of `LType`. Morphisms in LINEAR between σ and τ consist of values of type `LExp sig Empty` $(\sigma \multimap \tau)$.

The operators `Lift` and `Lower` are functors between these two categories. For any Haskell function $\alpha \to \beta$ we have a linear morphism `Lower` $\alpha \multimap$ `Lower` β, and similarly for any linear morphism $\sigma \multimap \tau$ we have a Haskell function `Lift` $\sigma \to$ `Lift` τ.[11]

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

```
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
```

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 3.5) forms a comonad, and its dual `Lift sig (Lower α)` forms a monad—the linearity monad discussed in Section 1.

We write the linearity monad as `Lin sig α`. For convenience, we define accessor functions `suspendL` and `forceL` to 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
```

We can define a monad instance for `Lin sig` using only the `Lift` and `Lower` connectives.[12]

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

Theorem 5.1. *If the β and η laws for the* `HasLower` *type class hold for the signature* `sig`*, then the monad laws hold for* `Lin sig`*: (1)* `pure a ≫= f = f a`*; and (2)* `e ≫= pure = e`.

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

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

The β equality rule for >! states that `put a >! g = g a`, and so the code above is equal to `suspendL (forceL $ f a)`, which is η-equivalent to `f a` itself.

Similarly for (2), buy unfolding definitions we see that `e ≫= pure` is equal to `suspendL (forceL e >! put)`. By η-equality for `Lower` this is equal to `suspendL (forceL e)`, the η-expanded form of `e`. □

When we evaluate the body of an expression in `Lin sig α`, the result is always an effectful lowered Haskell value `LVal sig (Lower α)`. It is thus possible to 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
```

Terms in the linearity have no linear input and no linear output, so they consist of operations that look non-linear to the end user, but are implemented in a linear domain. For example, let us revisit our file handle example from the introduction. Recall the function `withFile`, which opens a file, performs some transformations, and closes the file again.

```
withFile :: HasFH (LExp sig) ⇒ String → Lift sig (Handle ⊸ Handle ⊗ Lower a) → Lin sig a
withFile s f = suspendL $ force f ^ (Open s) `letPair` \(h,a) → Close h `letUnit` a
```

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

The run operation seamlessly connects the embedded linear language with its effectful implementation in Haskell. When applied to the result of withFile, for example, we obtain a program in IO that manipulates primitive IO file handles.

5.1 Monads in the linear category

The withFile operation exposes a common pattern: it takes in a linear morphism of type $\sigma \multimap \sigma \otimes \tau$. Just like the state monad in Haskell, this type forms a monad in the linear category. To make this observation formal, we first define a type class of linear monads.

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

When convenient, we may use the notation `e >>= f` for `lbind ^ e ^ f`. The laws for monads in LINEAR are essentially the same as for those in HASK: `lreturn e >>= f = f ^ e` and `e >>= lreturn = e`.

To make an instance declaration for linear state, we first attempt to define a type synonym `LState σ τ` for $\sigma \multimap \sigma \otimes \tau$ and give it an LMonad instance. This approach fails for a rather silly reason: `LState σ` is a partially defined type synonym, which are not allowed in Haskell. The ordinary solution would be to define a newtype, but newtypes (and regular algebraic data types) produce Types, not LTypes.

Our solution is to use a trick called defunctionalization (Eisenberg and Stolarek 2014). The Singletons library[13] 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)
```

With this boilerplate out of the way, we can 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
```

Figure 5 illustrates how we can write transformations over file handles in a monadic way.

5.2 The monad transformer

We saw in Section 1.2 that when an LMonad returns a lowered Haskell type, as is the case of readM and takeM above, we can push the monadic programming style a step further: the adjunction Lower ⊣ Lift also makes an LMonad transformer. In particular, 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.

```
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))
```

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

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

Fig. 5. Examples of functions on file handles using the linear state monad. `takeM` reads the first *n* characters from a handle.

```
takeT  :: HasFH (LExp sig) ⇒ Int → LinT (LState' Handle) String
takeT n | n ≤ 0     = return ""
        | otherwise = do c ← readT
                         s ← take (n-1)
                         return $ c:s
writeString :: HasFH (LExp sig) ⇒ String → LinT sig (LState' Handle) ()
writeString s = mapM_ writeT s
withFile :: HasFH (LExp sig) ⇒ String → LinT sig (LState' Handle) a → Lin sig a
withFile s f = suspendL $ forceT f ^ (open s) `letPair` \(h,a) → close h `letUnit` a
```

Fig. 6. Examples of functions on file handles using the linear state monad lifted through the monad transformer.

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 a
    x >>= f = suspend $ forceT x =>>= λ $ \y → y >! (force . f)
```

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

The proof is straightforward by unfolding definitions and applying the `LMonad` laws.

Now our interface to `read` and `write` can be specified relative to the monad transformer.

```
readT  :: HasFH (LExp sig) ⇒ LinT sig (LState' Handle) Char
writeT :: HasFH (LExp sig) ⇒ Char → LinT sig (LState' Handle) ()
```

In Section 5.2 we define a series of operations in the lifted linear state monad. First we have the monad transformer version of `take`, as well as a version of the function `writeString` from Section 1.2, which writes an entire string to a file. In addition, we can restructure the operation `withFile` to use the monad transformer as its intermediate state. Putting all of these parts together we can actually evaluate our linear code:

```
main = run $ do withFile "foo" $ writeLine "Hello world"
                withFile "foo" $ takeT 7
> "Hello w"
```

6 EXAMPLES

In this section we present three additional application domains in the linear/nonlinear framework: 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 given a pure functional interface if they are only accessed linearly. To understand why, consider a non-linear program with functional arrays:

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

If `write` updates the array in place, the program returns `"hello"` instead of `"world"`, as we would expect. 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 just 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 = ArraySig Type Type            type Array token α = MkLType ('ArraySig token α)
```

The token argument to an 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 to arrays can allocate new arrays and drop the pointer to existing arrays; once all slices of an array have been dropped, garbage collection can deallocate the array. Each array is associated with a set of valid indices, which can be obtained via the operation `domain`. 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 usual `read` and `write` operations will fail at runtime if their arguments are not in the domain of their slice.

```
class (HasLolli exp, HasTensor exp, HasOne exp, HasLower exp) ⇒ HasArray exp where
    alloc  :: Int → α → SomeArray exp α
    drop   :: exp γ (Array tok α) → exp γ One
    domain :: exp γ (Array tok α) → exp γ (Array tok α ⊗ Lower [Int])
    slice  :: Int → exp γ (Array tok α) → exp γ (Array tok α ⊗ Array tok α)
    join   :: CMerge γ1 γ2 γ ⇒ exp γ1 (Array tok α) → exp γ2 (Array tok α) → exp γ (Array tok α)
    read   :: Int → exp γ (Array tok α) → exp γ (Array tok α ⊗ Lower α)
    write  :: Int → exp γ (Array tok α) → α → exp γ (Array tok α)
```

6.1.1 Implementation. We can implement the `HasArray` signature in the shallow embedding. A value of type `Array tok 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`. Since we use `IOArrays`, the effect of this language will be in `IO`.

```
data instance LVal Shallow (Array tok α) = VArray [Int] (IO.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.

```
slice i e1 e = SExp $ \γ → do
    VArray bounds arr ← runSExp e γ
    return $ VPair (VArray (filter (< i) bounds) arr) (VArray (filter (≥ i) bounds arr))
```

The `join` operation evaluates its arguments and combines the two resulting bounds.[14]

```
join e1 e2 = SExp $ \γ → do let (γ1,γ2) = split γ
                             VArray bounds1 arr ← runSExp e1 γ1
                             VArray bounds2 _   ← runSExp e2 γ2
                             return $ VArray (bounds1 ++ bounds2) arr
```

6.1.2 Arrays in the lifted state monad. We can lift `domain`, `read`, and `write` into the linear state monad transformer with the following signatures, where we write `LStateT σ α` for `LinT (LState' σ) α`.

```
domainT :: HasArray (LExp sig) ⇒ LStateT (Array tok α) Int
readT   :: HasArray (LExp sig) ⇒ Int → LStateT (Array tok α) α
writeT  ::  HasArray (LExp sig) ⇒ Int → α → LStateT (Array tok 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 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 tok α) () → LStateT sig (Array tok α) () → LStateT sig (Array tok 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 of values than the pivot value 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 tok 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 tok 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 tok α) ()
quicksort = partition ≫= \case Nothing    → return ()
                               Just pivot → sliceT pivot quicksort quicksort
```

6.1.4 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

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

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. In this case the session protocol for the marketplace would be:

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

In Caires and Pfenning's formulation, a channel with session protocol $\sigma \multimap \tau$ will receives a channel of type σ, then continues with the protocol τ. A channel with protocol $\sigma \otimes \tau$ will send 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 `MarketProtocol`

```
marketplace :: HasMELL exp ⇒ Lift exp MarketplaceProtocol
marketplace = Suspend . λ $ \x → x >! \item →
                        λ $ \y → 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 (MarketplaceProtocol ⊸ 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
```

Using some simple aliases like `send` for `(⊗)` and `recv f` for `λ (>! f)`, the `marketplace` implementation starts to seem much more like a process than a λ term, but these details are superficial.

```
marketplace = Suspend . recv $ \item →
                        recv $ \cc →
                        send (put $ "Processed order for " ++ item) done
```

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 with a pair 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 `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.

```
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 `UChan`s.

```
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
```

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).

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 a quantum value 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 $\sigma \multimap \sigma$. For example:

```
data Unitary σ where
    Hadamard :: Unitary Qubit
    CNOT     :: Unitary (Qubit ⊗ Qubit)
    ...
```

Finally, `meas` performs quantum measurement, which probabilistically outputs a boolean value.

6.3.1 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:

```
fourier :: HasQuantum exp ⇒ Sing n → LStateT (Qubit ∏ n) ()
fourier SZ            = return ()
fourier (SS SZ)       = suspendT . λ $ unitary Hadamard ⊗ put ()
fourier (SS m@(SS _)) = 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 `λpair` combines abstraction and `letPair` to match against the input to the λ.

6.3.2 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. Next 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 the monad describes transformations between 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.

6.3.3 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).

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 more awkward.

In this paper we combine the two representations to get a HOAS encoding of the direct-style typing judgment. Doing so has some drawbacks, however; as expressions become more complex, the type class mechanism starts to show its weaknesses. 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)`. The problem is that Haskell cannot infer that both sides of the application are typed in the empty context; knowing $\gamma 1 \uplus \gamma 2$ = `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 at 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.[15]

Eisenberg et al. and Polakow use $!\alpha$ as an embedded connective, which, compared to the linear/non-linear decomposition of ! that we present in this paper, is less connected to regular Haskell programming and requires significantly more maintenance in the linear system. With regards to applications, Eisenberg et al. are focused on the domain of quantum computing, while Polakow focuses on a pure linear lambda calculus. Our paper shows that the same principles can be applied to a variety of domains and a variety of implementation techniques

7.2 Deep versus shallow embeddings

The prior work by Eisenberg et al. and Polakow describe only shallow embeddings, which should be more efficient than deep embeddings (although we have not performed a thorough performance analysis). 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. For example, `SExp (`$\gamma \rightarrow$ `VPut ())` has type `LExp Shallow` γ `(Lower ())` for any context γ. However, there are two different consumers of our framework: DSL *implementers* and *users* of the DSL. 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. This may be beneficial from a soundness perspective, although of course the language implementer could make an error in defining the evaluation function. The deep embedding also makes it possible to express program transformations and optimizations in that language.

7.3 Further integration with Haskell

A recent proposal suggests how to integrate linear types directly into GHC based on a model of linear logic that uses weighted type annotations instead of $!\alpha$ or the adjoint decomposition considered here, which would allow the implementation of efficient garbage collection and explicit memory management.[16] Compared to our approach, the proposal requires significant changes to GHC and is for a fixed domain, whereas our approach works out of the box and is extensible to numerous domains.

The 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[17] to define data structures and functions with implementations in both the linear and non-linear world. Likewise, support for other features such as nested pattern matching could make our framework more accessible.

7.4 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 ideas presented here are not specific to Haskell, but could be applicable in even richer languages like Coq or Agda.

[15]https://ghc.haskell.org/trac/ghc/wiki/Plugins/TypeChecker
[16]https://ghc.haskell.org/trac/ghc/wiki/LinearTypes
[17]https://wiki.haskell.org/Template_Haskell

REFERENCES

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

Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. 2016. Cogent: Verifying High-Assurance File System Implementations. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16).* ACM, New York, NY, USA, 175–188. DOI:http://dx.doi.org/10.1145/2872362.2872404

Erik Barendsen and Sjaak Smetsers. 1993. Conventional and uniqueness typing in graph rewrite systems. In *Foundations of Software Technology and Theoretical Computer Science: 13th Conference Bombay, India, December 15–17, 1993 Proceedings*, Rudrapatna K. Shyamasundar (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 41–51. DOI:http://dx.doi.org/10.1007/3-540-57529-4_42

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. DOI:http://dx.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. DOI:http://dx.doi.org/10.1109/LICS.1996.561458

P. N. Benton. 1994. *A mixed linear and non-linear logic: proofs, terms and models (preliminary report).* Technical Report 352. Computer Laboratory, University of Cambridge.

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. DOI:http://dx.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). DOI:http://dx.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. DOI:http://dx.doi.org/10.1145/2633357.2633361

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. DOI:http://dx.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. DOI:http://dx.doi.org/10.1145/2491956.2462177

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. DOI:http://dx.doi.org/10.1007/3-540-46425-5_15

Paul Blain Levy. 2003. Call-By-Push-Value: A Subsuming Paradigm. In *Call-By-Push-Value.* Springer Science Business Media, 27–47. DOI:http://dx.doi.org/10.1007/978-94-007-0954-6_2

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 (Lecture Notes in Computer Science)*, Jan Vitek (Ed.), Vol. 9032. Springer Berlin Heidelberg, London, UK, 560–584. DOI:http://dx.doi.org/10.1007/978-3-662-46669-8_23

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. DOI:http://dx.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. DOI:http://dx.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). DOI:http://dx.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. DOI:http://dx.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. DOI:http://dx.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. DOI: http://dx.doi.org/10.1007/BFb0054091

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. DOI: http://dx.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. DOI: http://dx.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. DOI: http://dx.doi.org/10.1145/2544174.2500598

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. DOI: http://dx.doi.org/10.1017/CBO9781139193313.005

Sjaak Smetsers, Erik Barendsen, Marko van Eekelen, and Rinus Plasmeijer. 1994. *Guaranteeing safe destructive updates through a type system with uniqueness information for graphs*. Springer Berlin Heidelberg, Berlin, Heidelberg, 358–379. DOI: http://dx.doi.org/10.1007/3-540-57787-4_23

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, 347–359.

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

Received April 2017