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:

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 Pfening 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

¹Here, → (pronounced "lollipop") denotes linear implication, ⊸ ("tensor") denotes the multiplicative linear product, and One denotes the multiplicative unit.
²Note 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.

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 → harray a → a → array a 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 Pfening 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 \( s \to s+s \), 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 \( s \to let! s' = s in !s'+!s' \). The type \( !\alpha \), pronounced "bang \( \alpha \)" , indicates that expressions of type \( \alpha \) 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 ! 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
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 injects 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 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 lambda calculus 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].

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. 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.

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 means that is the disjoint union of and ; it enforces the fact that variables cannot 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.

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 $\Gamma \vdash e : \sigma$, where the variables in $\Gamma$ are non-linear, the variables in $e$ are linear, and the result type $\sigma$ is also linear. The revised typing rules are given in Figure 3. The revised $\text{var}$ rule allows arbitrary non-linear variables, while the revised $\text{app}$ 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 $\Gamma, a : \alpha ; \vdash a : \alpha$ is not valid because $\alpha$ 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: $\sigma, \tau ::= \cdots \mid \text{Lower } \alpha$.

As shown in Figure 3, terms of type $\text{Lower } \alpha$ are constructed from arbitrary non-linear terms via an operation called $\text{suspend}$, so every linear expression of type $\text{Lower } \alpha$ morally holds a non-linear value. The elimination form, let $! a = e \text{ in } e'$, 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 $\text{Lift}$ connective, which embeds linear expressions in the non-linear world: $\alpha, \beta ::= \cdots \mid \text{Lift } \tau$. 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 $\text{Lift}$ type is introduced by $\text{suspend } e$, which internalizes a linearly-closed expression $e$ as a non-linear value.

The corresponding elimination form, $\text{force}$, moves such a value back into the linear world.

### 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.

#### 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 $\text{Lower}$ connective describes a simple way to use arbitrary host language terms, making the whole host language accessible from within the linear fragment. The $\text{Lift}$ 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.
consider the following sample derivation:

\[
\frac{\text{Just } (\sigma \rightarrow r) \vdash 0: \sigma \rightarrow r \quad \text{VAR}}{\text{Nothing, Just } \sigma \vdash 1: \sigma \quad \text{VAR}} \quad \text{APP} \quad \frac{\text{Nothing, Just } \sigma \vdash \lambda. 0.0: (\sigma \rightarrow r) \rightarrow r \quad \text{ABS}}{\text{Nothing, Just } \sigma \vdash \lambda. 0.0: (\sigma \rightarrow r) \rightarrow r \quad \text{ABS}}
\]

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

\[
\text{Just } (\sigma \rightarrow r) \not\subseteq \text{Nothing, Just } \sigma = \text{Just } \sigma, \text{ Just } (\sigma \rightarrow r)
\]

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:

\[
y1 \quad \uplus \quad [] = y1
\]
\[
[] \quad \uplus \quad y2 = y2
\]
\[
(\text{Just } : y1) \quad \uplus \quad (\text{Nothing } : y2) = \text{Just } : (y1 \uplus y2)
\]
\[
(\text{Nothing } : y1) \quad \uplus \quad (\text{Just } : y2) = \text{Just } : (y1 \uplus y2)
\]
\[
(\text{Nothing } : y1) \quad \uplus \quad (\text{Nothing } : y2) = \text{Nothing } : (y1 \uplus y2)
\]

This representation contains some redundancy: the lists [Just \(\sigma\)] and [Just \(\sigma\), 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 \(\sigma\).

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 \(y \vdash x: \sigma\) if \(y\) is the context containing only the single binding \(x: \sigma\). We formulate this relation in Haskell as a multi-parameter type class CSingleton \(x \times \sigma\), as shown in Figure 4. The class CSingletonN \(x \times \sigma\) records the same property, but for non-empty contexts—we use this helper type class to inductively build up the relation.

instance CSingletonN x \(\sigma\) \(\Rightarrow\) CSingleton x \(\sigma\) (NCtx y)
instance CSingletonN Z \(\sigma\) (End \(\sigma\))
instance CSingletonN x \(\sigma\) \(\Rightarrow\) CSingleton (\(S\ x\)) \(\sigma\) (Cons Nothing \(\gamma\))

The functional dependencies \(x \rightarrow y\) and \(y \rightarrow x\) 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 \(x \times \sigma\ y\ y'\) encodes the property that \(y' = y, x: \sigma\), where \(x\) does not already occur in \(y\). The class CMerge \(y1 y2 y\) says that \(y1 \uplus y2 = y\), or, in other words, that \(y\) is the disjoint union of \(y1\) and \(y2\). 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 \(\text{exp } \gamma \ y\ \tau\). The parameter \(\text{exp } ::\ Ctx \rightarrow \text{LType} \rightarrow \text{Type}\) 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
  \(\lambda\) :: (CSingleton x \(\sigma\) y', CAdd x \(\sigma\) y y', x ~ Fresh \(\gamma\))
  \(\Rightarrow\) (exp y' \(\sigma\) \(\rightarrow\) exp y' \(\tau\) \(\rightarrow\) exp y \(\sigma\) \(\rightarrow\) y)

The HasLolli type class asserts that the typing judgment exp contains abstraction (\(\lambda\)) 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 \(\lambda\) without the type class constraints:

\[(\text{exp } y' \(\rightarrow\) exp y' \(\tau\) \(\rightarrow\) exp y \(\sigma\) \(\rightarrow\) y)\]

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

\[(\text{CSingleton } x \sigma y', \text{CAdd } x \sigma y y', x \sim \text{Fresh } \gamma)\]

The last constraint says that \(x\) is a particular variable that is fresh in \(\gamma\): we define Fresh \(\gamma\) to be the smallest natural number that is undefined in \(\gamma\). The middle constraint says that the body of the function, of type \(\text{exp } y' \(\rightarrow\) \tau\), satisfies the relation \(y' = y, x: \sigma\). The first constraint says that the argument of the function, of type \(\text{exp } y' \(\rightarrow\) \sigma\), really is a variable, since \(y' = x: \sigma\). Put in a more functional notation, the type of \(\lambda\) could be described as follows:

\[(\text{exp } [x:\sigma] \sigma \rightarrow \text{exp } (y, x: \sigma) \rightarrow \text{exp } (\sigma \rightarrow \tau))\]

The HOAS encoding leads to very natural-looking code. The identity function is \(\lambda (x: \rightarrow \tau) \rightarrow \text{exp } (\sigma \rightarrow \tau)\) which composition is defined as:

\[\text{compose } :: \text{HasLolli exp}
  \Rightarrow \text{exp Empty ((r2 \rightarrow r3) \rightarrow (r1 \rightarrow r2)) \rightarrow (r1 \rightarrow r3))\]

\[\text{compose } \lambda g \rightarrow \lambda f \rightarrow \lambda x: g \wedge (f \wedge x)\]

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 @, and sums @. For the linear multiplicative unit, we have the following class:

class HasOne exp where
  unit :: exp Empty One
  letUnit :: CMerge y1 y2 y \Rightarrow exp y1 One \rightarrow exp y2 \(\tau\) \(\rightarrow\) exp y \(\tau\)

\(^4\)The linear abstraction function \(\lambda\) should not be confused with Haskell’s usual anonymous function abstraction, written \(\lambda a \rightarrow t\).
class CSingleton (x :: Nat) (σ :: LType) (γ :: Ctx)
class CAdd (x :: Nat) (σ :: LType) (y :: Ctx) (y' :: Ctx)
class CMerge (y1 :: Ctx) (y2 :: Ctx) (y :: Ctx)

letPair y, z → (y σ → γ, y → x σ) → y = [x:σ]

The LNL connective Lower x

The variables are represented in the higher-order abstract syntax by arguments exp y21 σ1 and exp y22 σ2 respectively, where γ21 = [x1:σ1] and γ22 = [x2:σ2]. The continuation of the letPair is in the context y'22 = y21, x1:σ1, x2:σ2. 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 ⊸ r → r ⊸ σ.

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.

It would certainly be more natural to write λ x \ (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 \Apair, and write our example as \Apair \ (y, z) → z ⊸ y. We discuss the issue of type checking and nested pattern matching more in Section 7.1.
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. 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 define a type of linear values 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 which must contain only a single variable, \( x \). We structure these three components as data types and family indexes by a signature \( \text{Family} \).

```haskell
data LExp (sig :: Type) (r :: LType) :: Type
data LVal (sig :: Type) (r :: LType) :: Type
type family Effect (sig :: Type) :: Type \to Type

An evaluation context is a finite map from variables, represented using singletons \cite{Eisenberg2014}, 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 \( \text{ECtx} \) maps variables \( x : \sigma \in \gamma \) to values of type \( \text{LVal} \) \( \text{sig} \).

```haskell
data ECtx sig y where
  ECtx :: (\gamma \times \sigma) \Rightarrow \text{ECtx} \gamma y \times \sigma \Rightarrow \sigma \Rightarrow \text{LVal} \text{sig} y \\
  \Rightarrow \text{ECtx} \text{sig} y \\

Evaluation is specified as a type class on signatures.

class Eval sig where
  eval :: Monad (Effect sig) \\
  \Rightarrow \text{ECtx} \text{sig} y \Rightarrow \text{LExp} \text{sig} y r \Rightarrow \text{Effect} \text{sig} (\text{LVal} \text{sig} r)
```

4.1 A deep embedding

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

```haskell
data Deep 
  where
    data instance LExp Deep y r where 
      Var :: CSingleton x r y \Rightarrow Sing \text{LExp} y r \\
      Abs :: CAdd x \sigma y y' \Rightarrow Sing \text{LExp} y' r \Rightarrow \text{LExp} y (\sigma \Rightarrow r) \\
      App :: Cmerge y! y2 y \Rightarrow \text{LExp} y1 (\sigma \Rightarrow r) \Rightarrow \text{LExp} y2 \sigma \Rightarrow \text{LExp} y r 

To instantiate the \( \text{HasLolli} \) type class, it is enough, therefore, to produce the singleton value \( x \) that corresponds to \( \text{Fresh} \ y \).

```haskell
instance HasLolli (LExp Deep) where
  \lambda :: \forall x \sigma y y'. (\text{Sing} x \sigma y y', \text{CAdd} x \sigma y y', x \Rightarrow \text{Fresh} y) \\
  \Rightarrow (\text{LExp} y' \sigma \Rightarrow \text{LExp} y (\sigma \Rightarrow r)) \Rightarrow \text{LExp} (\sigma \Rightarrow r) y \\

\lambda \sigma = \text{sing} (\text{sing} x) \\
(\sigma \Rightarrow r) = \text{App}

Values are defined by induction on \( \text{LType} \). A value of type \( \sigma \Rightarrow r \) is a closure containing an evaluation context paired with the body of the abstraction, while a value of type \( \text{Lower} \ a \) is the underlying Haskell value, and so on.

```haskell
data instance LVal Deep (Lower a) = VPut a \\
data instance LVal Deep One = VUnit \\
data instance LVal Deep (a \oplus r) = VPair (LVal Deep a) (LVal Deep r) \\

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

```haskell
instance Eval Deep where 
  eval y (Abs x e) = return 8 Vabs y x e 

If the expression is a variable, we know that the typing context \( x \) 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.

```haskell
  eval y (Var x) = return 8 lookup x y 

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 y x = Just \sigma, when \( \text{CSingleton} x \sigma y \). We discuss how to embed this constraint in the \( \text{CSingleton} \) type class in Appendix A.

To evaluate an application \( \text{App} e1 e2 \), 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 \( e2 \).

```haskell
  eval y (App (e1 :: LExp Deep y1 r1) (e2 :: LExp Deep y2 r2)) = do let (y1,y2) = split @y1 @y2 \\
                  Vabs y' x e'1 \Rightarrow eval y1 e1 \\
                  v2 \Leftarrow eval y2 e2 \\
                  eval (add x v2 y') e1' 
```

^In the development, the constraint \( \text{Sing} x \) is a superclass of \( \text{CSingleton} \).

^https://wiki.haskell.org/GHC/Type_families
This operation uses two additional helper functions to manipulate contexts in a way similar to Lookup. The function \texttt{add} takes a variable \(x\), an evaluation context for \(\gamma\), and a value of type \(\sigma\), and produces an evaluation context for \(\gamma, x : \sigma\). Similarly, \texttt{split} \(\theta y1 \theta y2\) takes an evaluation context for \(\gamma\) where \texttt{CMerge} \(\gamma1 \gamma2 y\), and outputs two evaluation contexts for \(\gamma1\) and \(\gamma2\) respectively; it uses visible type application [Eisenberg et al. 2016] (e.g., \(\theta y1\)) 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 \texttt{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 \(\exp y r\) is represented as a monadic function from evaluation contexts for \(\gamma\) to values of type \(\tau\). Evaluation in the shallow embedding is just unpacking this function.

```haskell
data Shallow = SExp { runSExp :: ECtx \(\gamma\) \(\rightarrow\) Effect Shallow (LVal r) }
instance Eval Shallow where eval \(\gamma\) \(f\) = runSExp \(f\) \(\gamma\)
```

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

```haskell
data instance LVal Shallow (\(\sigma \rightarrow \tau\)) =
  VAbs (LVal Shallow \(\sigma\) \rightarrow Effect Shallow (LVal Shallow \(\tau\)))
```

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

```haskell
instance Monad (Effect Shallow) \Rightarrow HasLower (LExp Shallow) where
  put a = SExp \(\empty\) \(\_\) \(\rightarrow\) return \(\$\) VPut a
  put ! f = SExp \(\_\) \(\rightarrow\) do let \((y1, y2) = \text{split} y\)
  VPut a \(\leftarrow\) runSExp e y1
  runSExp \((f\ a)\ y2\)
```

### 4.3 File Handles

Both embeddings can be given instances of the \texttt{HasFH} type class, where values of type \texttt{Handle} are built-in 10 file handles, and the effect is also 10. We sketch the shallow embedding here, and give the deep embedding in Appendix B.

```haskell
data instance LVal Shallow Handle = VHHandle 10.Handle
instance Effect Shallow = \(10\)
```

The file handle operations are easily given by their 10 counterparts; open and close are defined here, and read and write analogously.

```haskell
instance HasFH (LExp Shallow) where
  open s = SExp \(\empty\) \(\rightarrow\) do h \(\leftarrow\) 10.openFile s 10.ReadWriteMode
  close e = SExp \(\_\) \(\rightarrow\) do VHHandle h \(\leftarrow\) runSExp e \(\rho\)
  h.close 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 \texttt{Lower} \(\alpha\) are as follows:

```haskell
put a ! f = f a \[\beta\]
put ! f \(\_\) \(\rightarrow\) e ! \(\rightarrow\) put e \[\eta\]
\((e ! f) ! g\) = e ! \(\_\) \(\rightarrow\) f \(\_\) \(\rightarrow\) g \[\text{assoc}\]
```

We say that an instance for \texttt{Eval sig} satisfies the \texttt{Lower} laws if they are preserved by evaluation.

#### Proposition 4.1. The shallow embedding satisfies the \texttt{Lower} laws.

**Proof.** We start with the \(\beta\) rule. Unfolding definitions:

\[
\text{eval} \ (\text{put} a ! f) \ y = \text{do} \ (\text{let} \ ((y1, y2) = \text{split} y)) \ \\
\text{VPut} a \ (\leftarrow\) \text{runSExp} e y1 \ \\
\text{runSExp} \ ((f\ a)\ y2)
\]

Since \(y1\) is the empty context we know that \(y2 = y\). Since \texttt{HasLower} (\texttt{LExp Shallow}) assumes that \texttt{Effect Shallow} is a monad, the equation above is equal to \texttt{runSExp} \((f\ a)\ y\), as expected.

The proofs of the \(\eta\) and associativity laws are similarly obtained by unfolding definitions and applying the monad laws.

#### Proposition 4.2. The deep embedding satisfies the \texttt{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 \texttt{hask}, the idealized category of Haskell types and terms. The \texttt{LINEAR} category has objects that are elements of \texttt{LType}, and morphisms that are values of type \texttt{LExp sig Empty} (\(\sigma \rightarrow \tau\)).

The operators \texttt{Lift} and \texttt{Lower} are functors between these two categories. For any Haskell function \(\alpha \rightarrow \beta\) we have a linear morphism \texttt{Lower} \(\alpha \rightarrow \texttt{Lower} \beta\), and similarly for any linear morphism \(\sigma \rightarrow \tau\) we have a Haskell function \texttt{Lift} \(\sigma \rightarrow \texttt{Lift} \tau\).

```haskell
fmapLift :: (HasLolli (LExp sig), HasLower (LExp sig)) \Rightarrow \texttt{Lift} \sigma \rightarrow \texttt{Lift} \tau
fmapLift f = \lambda \_ \_ \text{do} \text{put} . f
fmapLift :: HasLolli (LExp sig) \Rightarrow \texttt{Lift} sig Empty \rightarrow \texttt{Lift} sig \leftarrow \texttt{Lift} sig
```

Note that we do not give an instance of the standard type class \texttt{Functor}, which only describes endofunctors on \texttt{hask}.

---

\(^3\)The astute reader will recognize a similarity to the monad laws, which we discuss in depth in Section 5.
Back in the linear/non-linear model, \textit{Lift} and \textit{Lower} form a (symmetric monoidal) adjunction \textit{Lower} \leadsto \textit{Lift}, which is what allows non-linear variables to occur in linear typing judgments. Mac Lane \cite{ML} 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 \( F \rightarrow G \) 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 \( \text{Bang} \ \sigma \, \tau = \text{Lower} \ (\text{Lift} \ \sigma \, \tau) \) (from Section 3.4) forms a comonad, and its dual \( \text{Lift} \ \sigma \) \( (\text{Lower} \ \alpha) \) forms a monad.

We write this linearity monad as \( \text{Lin} \ \sigma \). For convenience, the accessor functions \( \text{suspend} \) and \( \text{force} \) move directly between the monad and the linear category.

defnewtype \text{Lin} \ \sigma \ = \ \text{Lin} \ (\text{Lift} \ \sigma \ (\text{Lower} \ \alpha))
\[
\text{suspend} \cdot \text{Lin} \ \sigma \ (e) \ = \ \text{suspend L} \ \text{force} \ e
\]
The linearity monad does indeed have a monad instance.

\begin{theorem}
If a signature \( \sigma \) satisfies the lower laws, then the monad laws hold for \( \text{Lin} \ \sigma \): (1) \( \text{pure} \ a \ = \ \text{suspend} \ \text{force} \ (\text{pure} \ a) \), (2) \( \text{pure} \ e = e \), and (3) \( \text{force} \ (\text{pure} \ f) \ = \ \text{suspend} \ \text{force} \ e \).
\end{theorem}

Proof: For (1), expanding the definition for \( \text{Lin} \ \sigma \) we see that
\[
\text{pure} \ a \ = \ \text{suspend} \ \text{force} \ (\text{pure} \ a) \ \text{force} \ e \ = \ \text{suspend} \ \text{force} \ e
\]
By the \( \beta \) rule for \( \text{force} \), this is equal to \( \text{suspend} \ \text{force} \ e \), which is \( \eta \)-equivalent to \( f \) a itself.

The proofs of (2) and (3) are similarly by unfolding definitions and applying the lower laws.

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

\[
\text{run} \ e = \text{eval LEmpty} \ (\text{force} \ e) \ \text{force} \ e \ = \ \text{suspend EEmpty} \ \text{force} \ e
\]

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 \( \text{run} \) with \( \text{withFile} \) will produce an IO action that manipulates the file directly.

\[
\text{withFile :: HasFH (LExp sig) \Rightarrow String} \\
\rightarrow \text{Lift sig (Handle \→ Handle \⊙ Lower a) \→ Lin sig a} \\
\text{withFile s op =} \text{Suspend } \text{force op} \ `\text{open s} ` \text{letPair} ` \text{\(h, a\)}\rightarrow \text{\(close h ` \text{letUnit} ` a\)}
\]

Just like the state monad in Haskell, the type \( \sigma \ → \ \sigma \ ⊗ \tau \) forms a monad in the linear category. We can then define a type class of linear monads \( \text{LMonad m} \), where \( m \) has kind \( \text{LType} \ → \text{LType} \), with linear versions of \text{return} and \text{bind}.

To make an instance declaration for linear state, we first try to define a type synonym \( \text{LState} \ \sigma \) for \( \sigma \ → \ \sigma \ ⊗ \tau \). This approach fails for a rather silly reason: the monad \( \text{LState} \ \sigma \) 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 \text{Types}, not \text{LTypes}.

Our solution is to use a trick called defunctionalization \cite{ES}. The Singletons library \footnote{https://hackage.haskell.org/package/singletons} provides a type-level arrow \( \text{Lin} \ (\text{LinT} \ \sigma) \ (\text{LinT} \ \tau) \) that describes unsaturated type-level functions between kinds \( k_1 \) and \( k_2 \). To define a defunctionalized arrow, we first define an empty data type for the unsaturated version of \text{LState}, and then define a type instance for the (infix) type family \( (\text{Lin} \ \sigma) \ → \text{Lin} \ \tau \), which has kind \( (k_1 \ → k_2) \ → k_1 \ → k_2 \).

\[
data \text{LState'} (\sigma :: \text{LType}) :: \text{LType} → \text{LType}
data \text{instance LState'} \ σ \ τ = \ σ → \ τ \ τ
\]

When convenient, we use the notation \( \text{e as f} \) for \( \text{bind} ^ e \cap f \).

The laws for monads in \text{Linear} are the same as for those in \text{Haskell}: \( \text{Lin} \) \( \text{return} \ e = \ λ \ s → s \ ⊗ e \), \( \text{Lin} \) \( \text{bind} \ e \ (\text{Lin} \ f) = \ λ \ s → e \ (\text{Lin} \ f) \ (\text{Lin} \ s) \), and \( \text{Lin} \) \( \text{pure} \ (\text{Lin} \ e) = \ λ \ s → e \). We can now define our monad instance.

\[
\text{instance HasMSL (LExp sig) \Rightarrow LMonad (LinT' \ σ) \ where} \\
\text{LinT} \ e = \ \text{\(\lambda \ \text{st} \ → \text{lreturn} \ \text{st} \ \text{letPair} \ \text{(x, s) → f x} \ \text{^ x ^ s}\)}
\]

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

\[
\text{takeM :: HasFH (LExp sig) \Rightarrow Int → LExp sig Empty (LState Handle (Lower String))} \\
\text{takeM n | n ≤ 0 = \text{return} ""} \\
\text{otherwise = \text{readM} \ (\lambda \ \text{x} → \text{Sink} \ \text{x} \ \text{! c} → \text{takeM} \ (\text{n} - 1) \ (\lambda \ \text{y} → \text{Sink} \ \text{y} \ \text{! s} → \text{return} \ \text{put} \ (\text{c} \ : \ s))}
\]

The monadic \text{readM :: LState Handle (Lower Char)} is just \( \lambda \) read.

5.2 The monad transformer

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

\[
\text{newtype \text{LinT} sig m (m :: \text{LType} → \text{LType}) (α :: \text{Type}) =} \\
\text{LinT (Lin sig m (m ⊘ (Lower a)))}
\]

\[
\text{suspendT :: LExp sig Empty (m ⊘ (Lower a)) → LinT sig m α} \\
\text{forceT :: LinT sig m α → LExp sig Empty (m ⊘ (Lower a))}
\]

We can define the \text{Monad} instance just as we did for \text{Lin}:

\[
data \text{instance (LMonad m, HasLower (LExp sig)) \Rightarrow Monad (LinT sig m)} \ where} \\
\text{return = suspendT \ \text{pure} \ \text{put} \} \\
\text{x as f = \text{suspend \ (forceT \ x as λ \ y → y) \ (force \ . \ f)}}
We also define a monad transformer version of take.

Why, consider a non-linear program with purely functional arrays:

\[
\text{let arr1 = write } 0 \text{ arr "hello" in arr1[0]}
\]

We also define a monad transformer version of take.

\[
\begin{align*}
\text{read} &:: \text{HasFH (LExp sig) } \Rightarrow \text{LStateT sig Handle Char} \\
\text{write} &:: \text{HasFH (LExp sig) } \Rightarrow \text{Char } \rightarrow \text{LStateT sig Handle ()} \\
\text{withFile} &:: \text{HasFH (LExp sig) } \Rightarrow \text{String } \rightarrow \text{LStateT sig Handle a } \rightarrow \text{Lin sig a}
\end{align*}
\]

Putting these together we can actually evaluate our linear programs:

\[
\begin{align*}
\text{main} &\equiv \text{run } s \Rightarrow \text{do withFileT } "foo" \Rightarrow \text{mapM_ writeT } "Hello world" \\
&\Rightarrow \text{withFileT } "foo" \Rightarrow \text{takeT } (7-1) \\
&\Rightarrow \text{return } s \Rightarrow \text{c}::s
\end{align*}
\]

Putting these together we can actually evaluate our linear programs:

\[
\begin{align*}
\text{main} &\equiv \text{run } s \Rightarrow \text{do withFileT } "foo" \Rightarrow \text{mapM_ writeT } "Hello world" \\
&\Rightarrow \text{withFileT } "foo" \Rightarrow \text{takeT } (7-1) \\
&\Rightarrow \text{return } s \Rightarrow \text{c}::s
\end{align*}
\]

6.1 Arrays

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

6.1.1 Implementation

We can implement the HasMELL signature in the shallow embedding. A value of type Array k a 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.

\[
\text{data instance LVal Shallow (Array k a) } = \text{VarArray } [\text{Int}] (\text{IOArray } \text{Int})
\]

\[
\text{type instance Effect Shallow } = \text{IO}
\]

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.

\[
\begin{align*}
\text{slice i e1 e2 } &\equiv \text{SExp } \gamma \Rightarrow \text{do VarArray } \text{bond arr } \Rightarrow \text{runSExp } e y \\
&\Rightarrow \text{let arr1 = filter } (< i) \text{ bond arr} \\
&\Rightarrow \text{let arr2 = filter } (\geq i) \text{ bond arr} \\
&\Rightarrow \text{return } VPair arr1 arr2
\end{align*}
\]

\[
\begin{align*}
\text{join e1 e2 } &\equiv \text{SExp } \gamma \Rightarrow \text{do let } (y_1,y_2) = \text{split } y \\
&\Rightarrow \text{let VarArray } \text{bdn1 arr } \Rightarrow \text{runSExp } e y_1 \\
&\Rightarrow \text{let VarArray } \text{bdn2 arr } \Rightarrow \text{runSExp } e y_2 \\
&\Rightarrow \text{return } \text{VarArray } (\text{bdn1+bdn2}) \text{ arr}
\end{align*}
\]

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 a is Lin sig (LState' a).

\[
\begin{align*}
\text{domT } &:: \text{HasArray (LExp sig) } \Rightarrow \text{LStateT sig (Array k a)} \\
\text{readT } &:: \text{HasArray (LExp sig) } \Rightarrow \text{Int } \rightarrow \text{LStateT sig (Array k a)} \\
\text{writeT } &:: \text{HasArray (LExp sig) } \Rightarrow \text{Int } \rightarrow \text{LStateT sig (Array k a)}
\end{align*}
\]

We can also derive a lift 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 being sliced, and returns a new array with the same valid indices.

\[
\text{sliceT } i e1 e2 = \text{SExp } \gamma \Rightarrow \text{do VarArray } \text{bond arr } \Rightarrow \text{runSExp } e y \\
&\Rightarrow \text{let arr1 = filter } (< i) \text{ bond arr} \\
&\Rightarrow \text{let arr2 = filter } (\geq i) \text{ bond arr} \\
&\Rightarrow \text{return } VPair arr1 arr2
\]

\[
\begin{align*}
\text{join e1 e2 } &\equiv \text{SExp } \gamma \Rightarrow \text{do let } (y_1,y_2) = \text{split } y \\
&\Rightarrow \text{let VarArray } \text{bdn1 arr } \Rightarrow \text{runSExp } e y_1 \\
&\Rightarrow \text{let VarArray } \text{bdn2 arr } \Rightarrow \text{runSExp } e y_2 \\
&\Rightarrow \text{return } \text{VarArray } (\text{bdn1+bdn2}) \text{ arr}
\end{align*}
\]

12As 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 a) ()
    -> LStateT sig (Array k a) ()
```

```
sliceT i st1 st2 = Suspend . \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 or equal to 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 a) ()
```

```
swap i j = do a <- readT i
    b <- readT j
    writeT i b >>= writeT j a
```

```
partition :: (HasArray (LExp sig), Ord a) => LStateT sig (Array k a) (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 a) => LStateT sig (Array k a) ()
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 -> \tau` receives a channel of type `\sigma`, then continues with the protocol `\tau`. A channel with protocol `\sigma \times \tau` sends a channel of type `\sigma` and then continues as `\tau`. 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 `\Rightarrow`. Consider the following implementation of `Market`:

```
marketplace :: HasMELL exp => Lift exp Market
```

```
marketplace = Suspend . \arr . \_ \_ \_ \_ \_
    \x > \_ \item y \_ \_ \_
        (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 . \lambda \c -> \c \_ \_ \_ \_ \_
    c \_ \_ \_ \_ \_
        (put $ "Tea" \_ \_ \_ \_ \_
            \c \_ \_ \_ \_ \_
                (put 1234 \_ \_ \_ \_ \_
                    \c \_ \_ \_ \_ \_
                        "letPair" \_ \_ \_ \_ \_
                        \_ \_ \_ \_ \_
                    \_ \_ \_ \_ \_
                        "letUnit" \_ \_ \_ \_ \_
                    \_ \_ \_ \_ \_
                \_ \_ \_ \_ \_
            \_ \_ \_ \_ \_
        \_ \_ \_ \_ \_
    \_ \_ \_ \_ \_
```

```
transaction :: HasMELL exp => Lin exp String
transaction = suspendL $ 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 `\sigma` is sent on the channel, the recipient will coerce it back to that same type `\sigma`. 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 parameterized 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:

\[
\text{class HasMEll \ exp \Rightarrow HasQuantum \ exp \ where} \\
\text{new :: Bool \Rightarrow Empty \ Qubit} \\
\text{unitary :: Unitary \ \sigma \rightarrow \exp \ \gamma \ \sigma} \\
\text{meas :: exp \ Qubit \rightarrow \exp \ (Lower \ Bool)}
\]

The new operation creates a qubit in a so-called “classical” state, corresponding to either \(0\) (\(\text{False}\)) or \(1\) (\(\text{True}\)). 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 \(\text{Unitary} \ \sigma\), each of which corresponds to a linear transformation \(\sigma \Rightarrow \sigma\). 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 \(\text{yin}/\text{yout} \Rightarrow e : r\) that threads an input context into every judgment. Eisenberg et al. use the standard typing judgment \(y \Rightarrow e : r\) 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 \(\lambda \ y \ x \rightarrow (\lambda \ y \rightarrow y) \ x\) * type checks in Haskell, but not \((\lambda \ y \ x \rightarrow x) \ x\). Type checks in Haskell, but not \((\lambda \ y \ x \rightarrow x) \ x\) * type checks in Haskell, but not \((\lambda \ y \ x \rightarrow x) \ x\). Haskell cannot infer that both sides of the application are typed in the empty context; knowing \(y1 \cup y2 = \text{Empty}\) is not enough to infer that \(y1 = y2 = \text{Empty}\). 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 \(\text{yin}/\text{yout} \Rightarrow e : r\) 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 plug-in that uses an external solver to search for the intermediate typing contexts.

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 \(\sigma\) 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 \(\lambda \ (\forall \ \gamma \rightarrow x \ \otimes \ x)\), 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:

\[
\text{Couldn’t match type ‘Empty’ with ‘\text{\textbf{\texttt{\textbackslash nEmpty}}} (\text{End} \ \sigma)’} \\
\text{arising from a use of ‘\texttt{\textbackslash o}\’ in the expression: x \ \otimes \ x}
\]

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 \(\text{LExp Shallow} \ y \ r\) that do not correspond to anything in the linear lambda calculus.\(^\text{14}\) 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 \(\text{SExp}\) 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 \(\text{LExp Deep} \ y \ r\) 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 \(\otimes\) or the adjoint decomposition considered here. Their proposal would allow the implementation of efficient garbage collection and

\(^\text{13}\)https://ghc.haskell.org/trac/ghc/wiki/Plugins/TypeChecker

\(^\text{14}\)For example, \(\text{SExp} (\langle y \rightarrow \text{VPut} ()\rangle)\) has type \(\text{LExp Shallow} \ y \ (\text{Lower} ()\rangle)\) for any context \(y\).
explicit memory management, and could conceivably be adapted to a wide variety of different domains using foreign function interface calls. 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: \( (\alpha \to \bot) \to \bot \).

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 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 \( \lambda f \) has type \( \exp y \ (\sigma \to \tau) \), it suffices to show that \( f : \exp y \gamma \sigma \to \exp y \gamma \tau \) where

1. CSingleton \( x \sigma y \gamma \gamma \);
2. CAdd \( x \sigma y \gamma \gamma \); and
3. \( x \sim \) Fresh \( y \).

In many cases, we know the value of \( y \)—in practice the top-level of an expression will often be the empty context—and we can proceed in the following way. From \( y \) we can compute \( x \), as Fresh \( y \) is a type family that produces the smallest natural number \( x \) that does not occur in \( y \). The functional dependencies of CSingleton state that because we know \( x \) and \( \sigma \), we can compute \( y' \); it is the context with Just \( \sigma \) at index \( 1 \) and Nothing everywhere else. Furthermore, knowing the values of \( x \), \( \sigma \) and \( y \) we can deduce \( y' \) based on the functional dependencies of CAdd.

Unfortunately, when we do not know the value of \( y \), we cannot in general deduce the types of the other variables. This situation arises whenever a merge occurs. For example, even if we know \( y \) when type-checking \( e \leftarrow e' \ colon \ exp y \tau \), we do not a priori know the contexts \( y_1 \) and \( y_2 \) such that \( e : \exp y_1 (\sigma \to \tau) \), and \( e' : \exp y_2 \sigma \), such that \( y_1 \cup y_2 = y \). Knowing \( y \) is not enough to compute \( y_1 \) and \( y_2 \), 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 \( y_1 \) or \( y_2 \) respectively, which allows us to compute the other.

This explains why \( \lambda (x \to (\lambda (y \to y'\times x)) \times x) \) does type check in our system, but \( \lambda (\langle x \to x \rangle \times \lambda (y \to y) \) 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 \( x \sigma y y' \), we need to enforce the functional dependency that \( y' \times \sigma \gamma \gamma \). Naively one might expect the following instances of this class:

\[
\begin{align*}
\text{instance CSingletonCtx x \sigma y y'} & \Rightarrow \text{CAdd x \sigma y y'} \\
\text{instance CAddN x \sigma y y'} & \Rightarrow \text{CAdd x y (N y) (N y')} \\
\end{align*}
\]

where \( \text{CAddN} \) 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 \( y' \), when \( y' \) is non-empty. The decision of which case to apply depends on the size of the output context \( y' \); when the size of \( y' \) 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 CountMinus1 \( y' \) to be one less than the number of the elements in \( y' \).

\[
\begin{align*}
\text{type family CountMinus1 (y :: NCtx) :: Nat where} \\
\text{CountMinus1 (End _)} & = 2 \\
\text{CountMinus1 (Cons (Just _)} & = \text{CountMinus1 y} \\
\text{CountMinus1 (Cons Nothing _)} & = \text{CountMinus1 y} \\
\text{type family Count (y :: Ctx) :: Nat where} \\
\text{Count Empty} & = 2 \\
\text{Count (N y)} & = \text{S (CountMinus1 y)} \\
\end{align*}
\]

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.

\[
\begin{align*}
\text{instance CAdd' x \sigma y y'} (\text{CountMinus1 y'}) & \Rightarrow \text{CAdd} x \sigma y (N y') \\
\text{class len Cnt y} & \Rightarrow \text{CAdd'} x \sigma y (y :: Cnt x) (y' :: NCtx y) \\
& | x \sigma y \to \text{len y'} , x \sigma y' \text{len} \to \sigma y \\
\text{class len Cnt y} & \Rightarrow \text{CAddN'} x \sigma (y :: NCtx) (y' :: NCtx) \\
& | x \sigma y \to \text{len y'} , x \sigma y' \text{len} \to \sigma y \\
\end{align*}
\]

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:

\[
\begin{align*}
\text{class Merge y1 y2 y1 y2 y1 y2 y1 y2 y1) & \Rightarrow \text{Empty (N y) (N y) (N y} \\
\end{align*}
\]

In particular, knowing any two of \( y_1, y_2 \) and \( y \) 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 \( y_1 y \to y_2 \) because of the overlap between the instance MergeForward (N y) Empty (N y) and MergeForward (N y1) (N y2) (N y) (and similarly for \( y_2 y \to y_1 \) ). Since the merge relation is in fact functional in this direction, we can define a type family that computes that function. The type

\[\text{https://wiki.haskell.org/Template_Haskell}\]
The Linearity Monad

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

instance CAdd' x σ y y' (CountNMinus1 y') ⇒ CAdd x σ (N y')
instance CSingletonNCtx x σ y ⇒ CAdd' x σ Empty y' Z
instance CAddNN' x σ y y' n ⇒ CAdd' x σ (N y') (S n)
instance Count y ~ n ⇒ CAddN' Z σ (Cons Nothing y) (Cons (Just σ) y) n
instance CSingletonNCtx x σ y ⇒ CAddN' (S x) σ (End τ) (Cons (Just τ) y') (S Z)
instance CAddNN' x σ y y' n ⇒ CAddN' (S x) σ (Cons Nothing y) (Cons Nothing y') n
instance CAddN' x σ y y' (S n) ⇒ CAddN' (S x) σ (Cons (Just τ) y) (Cons (Just τ) y') (S (S n))

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

class CMergeForwardN (γ) instance CMergeForwardN (Cons Nothing σ γ)
instance CMergeNForward (Cons Nothing γ σ)
instance CMergeNForward (Cons (Just τ) γ σ)
instance CMergeForwardN (Cons (Just τ) γ σ)

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

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

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 y.

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

instance (CMergeForwardN y1 y2 y, CMergeForwardN y2 y1 y, Div y1 ~ y2, Div y2 ~ y1) ⇒ CMerge x y1 y2 y

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 y where
  ECtx :: (γ x σ. Lookup x x ~ Just σ ⇒ Sing x → LVal sig σ)
  lookup :: CSingleton x x σ γ ⇒ Sing x → ECtx sig y → LVal sig σ
  add :: CAdd x σ y y' γ
       ⇒ Sing x → LVal sig σ → ECtx sig y → ECtx sig y'
  split :: CMergeN y1 y2 y
        ⇒ ECtx sig y → (ECtx sig y1, ECtx sig y2)

  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 x σ γ, then Lookup x 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 x y ~ Just σ ⇒ CSingleton x x σ γ
class LookupN x y ~ Just σ ⇒ CSingletonN x 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 x σ γ, then:
     a. y is a well-formed context;
     b. Lookup y x ~ Just σ;
     c. y ~ Singleton x σ; and
     d. Remove x γ ~ Empty.

  2. If CAdd x σ y y' γ then:
     a. y and y' are well-formed contexts;
     b. Lookup y' x ~ Just σ;
c. Remove x y’ ∼ y; and
  d. Addf x σ y ∼ y’.

3. If 

   Cmerge y1 y2 y then:
   a. Div y y2 ∼ y1;
   b. Div y y1 ∼ y2; and
   c. each of y1, y2, and y are well-formed.

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

1. Div y Empty ∼ y;
2. Div y y ∼ Empty;
3. CmergeForward Empty y y; and
4. CmergeForward y Empty y.

Using this, we can define lookup:

lookup :: CSingleton x σ y ⇒ Sing x → Ectx sig y → LVal sig σ
lookup x (Ectx f) = f x

For add, we will need to construct an evaluation context for y’, so add x v (Ectx f) will map y to f y when y ≠ 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 ⊕add x σ y y’ then Lookup y’ x ∼ Just σ (which follows from the theorems stated above); and
2. If ⊕add x σ y y’ and y ≠ x then Lookup y’ y ∼ Lookup y x.

The second result is proved by induction over the structure of the ⊕add relation, but because it refers to a type variable y that is not mentioned in the ⊕add 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 ⊕add class is as follows:

class (y’ ∼ ⊕add x σ y, y’ ∼ ⊕rem x y’, Lookup y’ x ∼ Just σ) ⇒ ⊕add x σ y y’ (σ :: LType) (y’ :: Ctx) (y :: Ctx) x σ (σ :: LType)
where
  ⊕lookupEq :: (x ≈ y) ⇒ Sing x → Sing y → Dict (Lookup y’ y ∼ Lookup y x)

The add helper function calls out to this proof in the case that y ≠ x.

add :: ∀ x σ y y’. ⊕add x σ y y’ ⇒ Sing x → LVal sig σ → Ectx sig y → Ectx sig y’
add x v (Ectx f) = Ectx $ \y → case eqSNat x y of
  Left Dict → v
  Right Dict → case ⊕lookupEq x σ v y’ x of
  Dict ⊕f f y

The split operation runs into a similar problem. When ⊕merge y1 y2 y, an evaluation context ρ for y is also an evaluation context for both y1 and y2, since Lookup y1 x ∼ Just σ implies Lookup y x ∼ Just σ, and similarly for y2. We add the following proofs to the ⊕mergeForward type class:

class ⊕mergeForward y1 y2 y | y1 y2 y where
  ⊕lookupMerge1 :: Lookup y1 x ∼ Just σ
  ⇒ Sing x → Dict (Lookup y x ∼ Just σ)

The definition of split follows:

split :: ∀ y1 y2 y. ⊕merge y1 y2 y ⇒ Ectx y → Ectx y1, Ectx y2
split (Ectx f) = (Ectx x \case ⊕lookupMerge1 @y1 @y2 @y x of
  Dict → f x
  ,Ectx x \case ⊕lookupMerge2 @y1 @y2 @y x of
  Dict → f x)

B Modularity 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 y r where
  Var :: CSingleton x σ y ⇒ LExp Deep y σ
  Dom :: Domain Deep dom ⇒ dom (LExp Deep y r)
  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-parametrized data structure dom. For example, file handles use the following domain, which closely resembles the HasFh type class.

data FDom (exp :: Ctx → LType → Type) :: Ctx → LType → Type where
  Open :: String → FDom exp Empty Handle
  Read :: exp y Handle → FDom exp y (Handle @ Lower Char)
  Write :: exp y Handle → Char → FDom exp y Handle
  Close :: exp y Handle → FDom exp y 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 y → dom (LExp sig y r) → Effect sig (LVal sig r)
instance Eval Deep where
  eval y Var = return $ lookup y
  eval y (Dom e) = evalDomain y 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

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

instance Domain Deep FDom where
  evalDomain _ (Open s) =
    VHandle <$> IO.openFile s IO.ReadWriteMode
evalDomain y (Read e) =
  VHandle h ← eval y e
c c ← IO.hGetChar h
  return $ VHandle h `VPair` VPut c
We use a variant of the shallow embedding to encode expressions, and we call runSExp when we construct a UChan. To evaluate an expression, we first construct a new channel with newU, which outputs the two endpoints of the new channel. Then 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.

\[
e >! f = SExp \rho c \rightarrow do \text{let } (p1,p2) = \text{split } \rho (x,x') \leftarrow \text{newU forkIO } $ runSExp e p1 x a \leftarrow \text{recvU } x' \text{ runSExp } (f a) \rho2 c
\]

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.

\[
\text{type UChan } = (\text{Chan } Any, \text{ Chan Any})
\]

\[
\text{newU } :: \text{ IO (UChan, UChan)}
\]

\[
\text{newU } = \text{ do } (c1, c2) \leftarrow 10. \text{newChan}
\]

\[
\text{c2 } \leftarrow \text{10.newChan}
\]

\[
\text{return } ((c1,c2),(c2,c1))
\]

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

\[
\text{sendU } :: \text{ UChan } \rightarrow a \rightarrow \text{ IO ()}
\]

\[
\text{sendU } (\text{cin,cout}) a \rightarrow \text{writeChan cout } a \leftarrow \text{unsafeCoerce } a
\]

\[
\text{recvU } :: \text{ UChan } \rightarrow \text{ IO a}
\]

\[
\text{recvU } (\text{cin,cout}) = \text{unsafeCoerce } <\Rightarrow \text{ readChan cin}
\]

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 10 channels under the hood, the effect of the signature is 10. All values with this signature, no matter the type, are UChans.

\[
\text{data Sessions}
\]

\[
\text{data instance LVal Sessions } \tau = \text{ Chan UChan}
\]

\[
\text{type instance Effect Sessions } = \text{ IO ()}
\]

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 \(\sigma\) on its output channel, for example.

\[
\text{data instance LExp Sessions } y \tau =
\]

\[
\text{SExp } \langle \text{runSExp } :: \text{ SCtx Sessions } y \rightarrow \text{ UChan } \rightarrow \text{ IO ()} \rangle
\]

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.

\[
\text{instance Eval Sessions where}
\]

\[
\text{eval } e y = \text{ do } (c,c') \leftarrow \text{newU forkiO } $ runSExp e y c \text{ return } $ \text{Chan c'}
\]

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 \(\tau\) via put \(a\), we simply send the Haskell value \(a\) over the output channel.

\[
\text{put } a = \text{ SExp } \langle \_ c \rightarrow \text{sendU } c a \rangle
\]

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.

\[
\text{type family } (\sigma :: \text{ LType}) \prod (n :: \text{ Nat}) :: \text{ LType where}
\]

\[
\sigma \prod Z = \text{ One}
\]

\[
\sigma \prod (S (S n)) = \sigma \otimes (\sigma \prod S n)
\]

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.

\[
\text{fourier } :: \text{ HasQuantum exp } \Rightarrow \text{ Sing } n \rightarrow \text{ LStateT (Qubit } \prod n) \text{ ()}
\]

\[
\text{fourier } (\text{SS } S Z) = \text{ return } ()
\]

\[
\text{fourier } (\text{SS } m) = \text{ suspendT } \lambda q \text{ unitary Hadamard } \otimes \text{ put } t
\]

\[
\begin{align*}
\text{fourier } & (\text{SS } (S m)) = \text{ suspendT } \lambda \text{ pair } q (q,qs) \\
& \lambda \text{ forceT (fourier } m ^ q \text{ \ ‘letin’ \ qs } \\
& \lambda \text{ forceT (rotations (SS } m) ^ q \text{ \ @ qs})}
\end{align*}
\]

\[
\text{where rotations } :: \text{ Sing } m \rightarrow \text{ Sing } n
\]

\[
\rightarrow \text{ Lift exp (Qubit } \prod S n \rightarrow \text{ Qubit } \prod S n)
\]

The Sing \(n\) data family is a runtime representation of the natural number \(n\), from the singletons library, with constructors SS::Sing 2 and SS::Sing \(n \rightarrow \text{ Sing } (S n)\). The operation \(\lambda\)pair combines abstraction and letPair to match against the input to the \(\lambda\).

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:

\[
\text{data QuantumExp exp } :: \text{ Ctx } \rightarrow \text{ LType } \rightarrow \text{ Type where}
\]

\[
\text{New } :: \text{ Bool } \rightarrow \text{ QuantumExp exp Empty Qubit}
\]

\[
\text{Meas } :: \text{ exp } \text{ Qubit } \rightarrow \text{ QuantumExp exp } \text{ exp } \text{ (Lower Bool)}
\]

\[
\text{Unitary } :: \text{ Unitary } \sigma \rightarrow \text{ exp } \gamma \sigma \rightarrow \text{ QuantumExp exp } \gamma \sigma
\]

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.

\[
\text{newM } :: \text{ Bool } \rightarrow \text{ DensityMonad Int}
\]

\[
\text{applyUnitaryM } :: \text{ Mat } (2^m) (2^n) \rightarrow \text{ [Int] } \rightarrow \text{ DensityMonad } ()
\]

\[
\text{measM } :: \text{ Int } \rightarrow \text{ DensityMonad Bool}
\]
Values of type Qubit are integer qubit identifiers, and DenseTyMonad is the effect.

data instance LVal Deep Qubit = QId Int
type instance Effect Deep = DenseTyMonad

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

D.5 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 calculus have been proposed, using which 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-Zabuzsky, Kenneth Foner, Richard Eisenberg, and Stephanie Weirich for many helpful discussions about this work.

References


