Embedding a Full Linear Lambda Calculus in Haskell

Jeff Polakow

Abstract

We present an encoding of full linear lambda calculus in Haskell using higher order abstract syntax. By making use of promoted data kinds, multi-parameter type classes and functional dependencies, the encoding allows Haskell to do both linear type checking and linear type inference.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

Keywords linear lambda calculus, higher-order abstract syntax, domain specific language

1. Introduction

Higher order abstract syntax (HOAS) [17] is, in general, a technique for specifying a system which makes direct use of the abstraction mechanism of the specification langauge. The idea was explicitly stated and used in [7], although it's origins go back to [3]. For the purposes of this paper (and most programming language research), HOAS comes down to representing a binding structure for an object language, e.g. lambdas in a lambda calculus, with the binding structure of the meta-language, e.g. lambdas in the specification language. Here is a concrete example of representing a lambda calculus in Haskell:

```
data Exp a where
  Lam :: Exp a -> Exp b -> Exp (a -> b)
  App :: Exp (a -> b) -> Exp a -> Exp b
```

Note that this example makes use of a generalized algebraic datatype (GADT) [13] to describe a typed lambda calculus; we will only consider typed object languages in this paper.

Using HOAS is very convenient as there is no explicit substitution in the object language; i.e. in the above there is no explicit representation for variables which are implicitly represented by Haskell variables, nor any machinery for substituting an `Exp` for a variable. However, HOAS tends to only work when the system being described has the same substitution and typing behavior as the specification language; in the preceding example, Haskell's function variables behave identically to the those of the lambda calculus.

The purpose of this paper is to show that natural HOAS encodings are indeed possible, at least in Haskell, for languages with radically different type systems (though with similar substitution) by using tagless final encodings [2] along with some type-level machinery. The central idea is to explicitly represent (an abstraction of) the variable environment which can be analyzed during type checking/inference by the type class machinery.

In particular, this paper gives a HOAS encoding of a full linear lambda calculus (LLC) including additives and units [15, 21]. The encoding will allow the Haskell type checker to do both linear type checking and linear type inference. We believe that this general approach could be used to create HOAS encodings for a variety of "exotically" typed languages.

An interesting aspect of the technique for representing linear lambda caclulus is that it requires using multi parameter type class with functional dependencies [8]; it cannot be done with (closed) type families [4], at least not as currently implemented. Thus this paper presents another data point in the ongoing discussion of the design space for type class machinery, and perhaps an argument against the current design of closed type families.

2. Tagless Final Encodings

Rather than describing an object language as values of a given algebraic data type in the meta language, i.e. a `data` declaration as in the introduction, tagless final encodings use terms of the meta language to encode the object language.

```
class Exp repr where
  lam :: (repr a -> repr b) -> repr (a -> b)
  app :: repr (a -> b) -> repr a -> repr b
```

`repr` is an abstract type, representing an `Exp` term being constructed, which is decorated with the Haskell type of that `Exp` term.

The techniques employed in this paper to encode LLC should transfer smoothly from a tagless final approach to an initial algebraic approach, like the GADT based example in section 1. However, we prefer the tagless final approach as it does not require GADTs, and the type class machinery can be used to conveniently give multiple concrete instantiations of the abstract type.

It is worth noting that the types of the `Exp` combinators correspond to the typing rules for the simply typed lambda calculus (thinking of $\vdash$ as a function and Γ as implicit):

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\,hyp \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.e : A \to B}\,{\to_I}$$

$$\frac{\Gamma \vdash e_0 : A \to B \quad \Gamma \vdash e_1 : A}{\Gamma \vdash e_0\, e_1 : B}\,{\to_E}$$

where, since we are using HOAS, the treatment of hypotheses is handled implicitly for us by Haskell. We may think of constructing a HOAS encoding as an exercise in transcribing the typing rules of the object language into the types of the meta language. Since Haskell offers reasonably convenient machinery for type level computation, the door is open to creating HOAS tagless final encodings of languages significantly different from Haskell. In the remainder of this paper, we will show how to encode LLC using HOAS and the tagless final approach.

$$\frac{}{x:A \vdash x:A}lvar \qquad \frac{\Delta, x:A \vdash e:B}{\Delta \vdash \hat{\lambda}x.e:A \multimap B}\multimap_I$$

$$\frac{\Delta_0 \vdash e_0:A \multimap B \quad \Delta_1 \vdash e_1:A}{\Delta_0 \bowtie \Delta_1 \vdash e_0 \hat{} e_1:B}\multimap_E$$

$$\frac{}{\Delta \vdash ():\top}\top_I \qquad \frac{\Delta \vdash e_0:A \quad \Delta \vdash e_1:B}{\Delta \vdash (e_0,e_1):A \,\&\, B}\&_I$$

$$\frac{\Delta \vdash e:A \,\&\, B}{\Delta \vdash \mathtt{fst}\, e:A}\&_{E0} \qquad \frac{\Delta \vdash e:A \,\&\, B}{\Delta \vdash \mathtt{snd}\, e:B}\&_{E1}$$

Figure 1. Typing derivation rules for mini-LLC

3. Mini Linear Lambda Calculus

LLC can be thought of as the proof terms arising from a natural deduction style presentation of intutionistic linear logic (ILL) [1]. In order to simplify the exposition, we will restrict ourselves to the smallest fragment of LLC which captures the complexity of linear type checking (and inference). We refer to this subset of LLC as mini-LLC. Furthermore, we will gradually work up to this complete fragment, developing the Haskell encoding as we go. For reference, a presentation of full LLC is given in appendix A which closely follows the presentation in [15][1].

The types of mini-LLC are as follows:

A	$::=$	$A_0 \multimap A_1$	linear functions
	$\mid$	$A_0 \,\&\, A_1$	additive conjunction
	$\mid$	$\top$	additive unit

We will use capital letters, $A, B, \ldots$, to stand for types.

The terms of mini-LLC are as follows:

e	$::=$	x			variables
	$\mid$	$\hat{\lambda}x.e$	$\mid$	$e_0 \hat{} e_1$	linear functions
	$\mid$	(e_0, e_1)	$\mid$	$\mathtt{fst}\, e \mid \mathtt{snd}\, e$	additive conjunction
	$\mid$	$()$			additive unit

We use the following judgement for typing derivations

$$\Delta \vdash e:A$$

where Δ is the linear variable context.

A variable context is a list of typed variables:

$$\Delta ::= \cdot \mid \Delta, x:A$$

We will use Δ, Γ for contexts. We will sometimes overload , to mean list append as well as list cons (as above); i.e. $\Delta_1, x:A, \Delta_2$ denotes a list which contains x to the right of everything in Δ_1 and to the left of everything in Δ_2.

We define the following non-deterministic merge which we will use on linear contexts to allow exchange of linear hypotheses, i.e. the order of linear hypotheses doesn't matter:

$$\begin{aligned} \cdot \bowtie \cdot &= \cdot \\ \Delta, x:A \bowtie \Delta' &= (\Delta \bowtie \Delta'), x:A \\ \Delta \bowtie \Delta', x':A' &= (\Delta \bowtie \Delta'), x':A' \end{aligned}$$

In the following sections, we develop a HOAS encoding of mini-LLC in Haskell. We will start with the multiplicative fragment, i.e. $\multimap$, in section 4, and then include the additive fragment, i.e. $\&, \top$, in section 5. The complete typing derivations for mini-LLC are in figure 1.

[1] Online at http://www.cs.cmu.edu/~fp/courses/linear/lectures/lecture15.html

4. Multiplicatives

We begin with the implicational fragment, which just contains linear functions and application. The basic derivation rules follow.

$$\frac{}{x:A \vdash x:A}lvar$$

A variable can only be used if there are no other variables in scope; the lack of other variables prevents Weakening from holding, i.e. variables must be used at least once.

$$\frac{\Delta, x:A \vdash e:B}{\Delta \vdash \hat{\lambda}x.e:A \multimap B}\multimap_I \qquad \frac{\Delta_0 \vdash e_0:A \multimap B \quad \Delta_1 \vdash e_1:A}{\Delta_0 \bowtie \Delta_1 \vdash e_0 \hat{} e_1:B}\multimap_E$$

The context split, when reading from conclusion to premises, prevents Contraction from holding, i.e. variables must be used at most once, since no variable can be copied into both premises. Additionally, the context split is non-deterministic when reading from conclusion to premises[2], which is the natural reading when considering type checking, and inference, for a given term. Before attempting a Haskell encoding, we will need to remove this non-determinism from type checking.

4.1 IO system

To remove the non-determinism in the $\multimap_E$ rule, we will rely on a technique for linear logic proof search developed in the context of linear logic programming [6]; the presentation and proofs used throughout this section and section 5.1 are based on the development of an ordered linear logic programming language [19][3], a similar presentation directly on ILL can be found in [14][4].

The basic idea for removing the non-determinism is to lazily split the context by passing all available variables to the first premise and passing the remaining ones to the second premise. Thus we use the following judgement:

$$\Delta_I \backslash \Delta_O \vdash e:A$$

where Δ_I are the input variables and Δ_O are the output, i.e. unused, variables. In order to ensure that all variables are indeed consumed, we augment the linear context to be a list of variables and placeholders, $\square$:

$$\Delta ::= \cdot \mid \Delta, x:A \mid \Delta, \square$$

This presentation of linear contexts as lists with placeholders for consumed variables will be useful when we start writing our Haskell code in section 4.2.

The derivation rules for the IO system follow:

$$\frac{}{\Delta, x:A, \Delta' \backslash \Delta, \square, \Delta' \vdash x:A}lvar$$

Note that x is the only variable consumed (changed to a placeholder), and all other elements of the input context appear unchanged in the output context.

$$\frac{\Delta_I, x:A \backslash \Delta_O, \square \vdash e:B}{\Delta_I \backslash \Delta_O \vdash \hat{\lambda}x.e:A \multimap B}\multimap_I$$

The placeholder in the output context ensures that x really was consumed in the derivation.

$$\frac{\Delta_I \backslash \Delta \vdash e_0:A \multimap B \quad \Delta \backslash \Delta_O \vdash e_1:A}{\Delta_I \backslash \Delta_O \vdash e_0 \hat{} e_1:B}\multimap_E$$

There is no longer a non-deterministic context split in the $\multimap_E$ rule and the typing derivations as a whole are deterministic.

[2] The conclusion gives us no hint for splitting the context.

[3] ILL is a subset of the ordered system.

[4] Online at http://www.cs.cmu.edu/~fp/courses/linear/lectures/lecture16.html

Before showing an encoding of the preceding system in Haskell, we'd like to prove its correctness with respect to the previous system in section 3. To that end, we define a super-context relation to formalize the notion of superset on linear contexts:

$$\frac{}{\cdot \sqsupseteq \cdot} \qquad \frac{\Delta_I \sqsupseteq \Delta_O}{\Delta_I, x{:}A \sqsupseteq \Delta_O, \square}$$

$$\frac{\Delta_I \sqsupseteq \Delta_O}{\Delta_I, x{:}A \sqsupseteq \Delta_O, x{:}A} \qquad \frac{\Delta_I \sqsupseteq \Delta_O}{\Delta_I, \square \sqsupseteq \Delta_O, \square}$$

IO system derivation rules maintain the following property.

Lemma 1. *$\Delta_I \backslash \Delta_O \vdash e{:}A$ implies $\Delta_I \sqsupseteq \Delta_O$.*

Proof. By induction on given derivation. □

We also define context difference

$$\begin{array}{rcl} \cdot - \cdot & = & \cdot \\ \Delta_I, \square - \Delta_O, \square & = & \Delta_I - \Delta_O \\ \Delta_I, x{:}A - \Delta_O, x{:}A & = & \Delta_I - \Delta_O \\ \Delta_I, x{:}A - \Delta_O, \square & = & (\Delta_I - \Delta_O), x{:}A \end{array}$$

and show the following two useful properties of $-$ and $\bowtie$.

Lemma 2.

$$\begin{array}{c} \Delta_0 \sqsupseteq \Delta_1 \textit{ and } \Delta_1 \sqsupseteq \Delta_2 \textit{ implies} \\ \Delta_0 - \Delta_2 \;=\; (\Delta_0 - \Delta_1) \bowtie (\Delta_1 - \Delta_2). \end{array}$$

Proof. By induction on the structure of the given derivations. □

Lemma 3.

$$\begin{array}{c} \Delta_I - \Delta_O = \Delta_0 \bowtie \Delta_1 \textit{ implies} \\ \exists \Delta. \Delta_I - \Delta = \Delta_0 \textit{ and } \Delta - \Delta_O = \Delta_1 \end{array}$$

Proof. By induction on the length of Δ_I. □

We can now state and prove the correctness of the IO system in two parts as follows.

Theorem 1.

$$\Delta_I \backslash \Delta_O \vdash e{:}A \quad \textit{implies} \quad \Delta_I - \Delta_O \vdash e{:}A$$

Proof. By induction on the structure of the given derivation using lemmas 1 and 2. □

Theorem 2.

$$\Delta_I - \Delta_O \vdash e{:}A \quad \textit{implies} \quad \Delta_I \backslash \Delta_O \vdash e{:}A$$

Proof. By induction on the structure of the given derivation using lemma 3. □

4.2 Haskell Encoding of Multiplicatives

We would like to use HOAS to encode the language of section 4.1, but the behavior of Haskell variables clearly does not match that of linear variables. Although the typing rules do not match, the binding and substitution of linear variables do follow that of Haskell variables[5]; thus we can imagine a forgetful encoding of LLC into Haskell where the linearity is enforced statically when type checking but the underlying function is a regular Haskell function.

```
newtype a -<> b = Lolli {unLolli :: a -> b}
```

[5] We did not explicitly define substitution for LLC terms as it is identical to subsitution on regular lambda calculus terms.

Such an encoding would just require enough information for the type checker to decide whether a variable is being used linearly. We will accomplish this by decorating our representation type with an abstraction of the linear context.

We will make use of Haskell's `DataKinds` extension [22] to get a type level abstraction of the linear context

```
data Nat = Z | S Nat
data CtxElm = Box | Elm Nat
```

The information we need in our abstract linear context is just whether a variable has been used; we don't care about the type of the variable. Thus, we will tag each in-scope linear variable with a type level `Nat`, and store those tags in our abstract context. So our abstract representation type will look like

```
repr :: Nat -> [CtxElm] -> [CtxElm] -> * -> *
```

We intend `repr v i o a` to represent a LLC term of type `a`, or the derivation of a linear type `a`, with input (abstract) context `i` and output (abstract) context `o`; the `v` is a counter for generating a new tag.

Let us consider how to represent the $\multimap_I$ rule, or linear function. We'd like something of the form

```
llam :: (repr ? ? ? a ->
           repr (S v) (Elm v ': i) (Box ': o) b
        ) -> repr v i o (a -<> b)
```

so that we have linear functions represented by Haskell functions from `a` to `b`. How should the `?`s be filled in? The $lvar$ rule from section 4.1 should be our guide, since that is the derivation rule represented by the argument `repr` above. A direct transcription of the $lvar$ rule would be

```
type LVar repr v a =
  forall (v'::Nat) (i::[CtxElm]) (o::[CtxElm]) .
  Consume v i o => repr v' i o a
```

which states that any `i` and `o` for which the constraint `Consume v i o` holds can be used to form a derivation of the variable `v` of type `a`. `Consume` is a type level relation specifying that `v` occurs in `i` and is replaced by `Box` in `o`.

```
class Consume (v::Nat)
              (i::[CtxElm])
              (o::[CtxElm])
      | v i -> o
class Consume1 (b::Bool)
               (v::Nat)
               (x::Nat)
               (i::[CtxElm])
               (o::[CtxElm])
      | b v x i -> o

instance (Consume v i o)
      => Consume v (Box ': i) (Box ': o)
instance (EQ v x b, Consume1 b v x i o)
      => Consume v (Elm x ': i) o

instance Consume1 True v x i (Box ': i)
instance (Consume v i o)
      => Consume1 False v x i (Elm x ': o)

class EQ (x::k) (y::k) (b::Bool) | x y -> b
instance EQ x x True
instance (b ~ False) => EQ x y b
```

The `EQ` typeclass encodes equality with respect to the Haskell type checker's unification machinery subject to the constraint solver's

ability to make progress; i.e. if `EQ x y b` holds then `x` and `y` unify and `b` will be `'True`, or `b` will be `'False`. This form of type level equality which reflects unifiability is really necessary for our encoding to work since `Consume` needs to realize that `v` does not equal `S v`; note that `EQ` is a modern rendering of the `TypeEq` and `TypeCast` type classes in [11].

We can now finish up our encoding of linear functions:

```
class LLC
  (repr :: Nat -> [CtxElm] -> [CtxElm] -> * -> *)
 where
  llam :: (LVar repr v a ->
             repr (S v) (Elm v ': i) (Box ': o) b
           )
       -> repr v i o (a -<> b)

  (^) :: repr v i m (a -<> b)
      -> repr v m o a
      -> repr v i o b
```

Note that the type of `^` is a direct transcription of the $\multimap_E$ rule.

Now we can try typing some example LLC terms

```
*Main> :t llam $ \f -> llam $ \x -> f ^ x
llam $ \f -> llam $ \x -> f ^ x
  :: LLC repr => repr v i i ((a -<> b) -<> (a -<> b))
```

That is reassuring. However, when we try an ill-typed example:

```
*Main> :t llam $ \f -> llam $ \x -> f ^ x ^ x
llam $ \f -> llam $ \x -> f ^ x ^ x
  :: (Consume ('S v) i o, LLC repr) =>
     repr v i o ((a -<> (a -<> b)) -<> (a -<> b))
```

The `Consume` constraint can't be further analyzed because we haven't constrained the input and output linear contexts. We need a way to declare a closed linear term, or definition. We do not want to simply require that our definitions have empty input and output contexts since that would preclude using them in a context where there are linear variables in scope. Instead, we want to declare that definitions do not change their input linear context:

```
type Defn a = forall repr (v::Nat) (i::[CtxElm])
            . LLC repr => repr v i i a
defn :: Defn a -> Defn a
defn x = x
```

Similar considerations prevent us from using the concrete `Z` instead of `v` above.

Our first example still type checks

```
*Main> :t defn $ llam $ \f -> llam $ \x -> f ^ x
defn $ llam $ \f -> llam $ \x -> f ^ x
 :: LLC repr => repr v i i ((a -<> b) -<> (a -<> b))
```

Now when we try our ill-typed example as a `Defn` term

```
*Main> :t defn $ llam $ \f -> llam $ \x -> f ^ x ^ x

<interactive>:1:42:
    Could not deduce (Consume ('S v1) i1 i1) ...
```

we get the expected error[6]. The previous example shows that a linear variable cannot be used twice; we can also check that linear variables cannot be ignored:

```
*Main> :t defn $ llam $ \f -> llam $ \x -> f

<interactive>:1:34:
  Could not deduce
  (Consume1 'False v1 ('S v1) ('Elm v1 : i1)
    ('Box : 'Box : i1)
  )
```

One interesting aspect of our encoding is that it cannot be done with closed type families. The `Consume` type class is not equivalent to the closed type family

```
type family ConsumeF (v::Nat)
                     (i::[CtxElm])
         :: [CtxElm] where
  ConsumeF v (Elm v ': i) = Box ': i
  ConsumeF v (x ': i) = x ': ConsumeF v i
```

because the definition of apartness [4] does not allow `ConsumeF v (Elm (S v) ': i)` to rewrite[7]. Relying on universally quantified type variables in the `Defn` type is critical for the encoding to be compositional. In order to maintain a consistent style, we will continue to use type classes for the rest of the paper.

5. Additives

We now consider the additive types. We will focus on the two introduction rules, $\&_I$ and $\top_I$, as the other rules do not add any complexity to the typing derivations.

$$\frac{}{\Delta \vdash ():\top}\top_I \qquad \frac{\Delta \vdash e_0 : A \quad \Delta \vdash e_1 : B}{\Delta \vdash (e_0, e_1) : A \,\&\, B}\&_I$$

The direct translations of the above rules into the IO system of section 4.1 follow:

$$\frac{\Delta_I \sqsupseteq \Delta_O}{\Delta_I \backslash \Delta_O \vdash () : \top}\top_I$$

$$\frac{\Delta_I \backslash \Delta_O \vdash e_0 : A \quad \Delta_I \backslash \Delta_O \vdash e_1 : B}{\Delta_I \backslash \Delta_O \vdash (e_0, e_1) : A \,\&\, B}\&_I$$

For completeness, we state the IO versions of the two $\&_E$ rules:

$$\frac{\Delta_I \backslash \Delta_O \vdash e : A \,\&\, B}{\Delta_I \backslash \Delta_O \vdash \mathtt{fst}\, e : A}\&_{E0} \qquad \frac{\Delta_I \backslash \Delta_O \vdash e : A \,\&\, B}{\Delta_I \backslash \Delta_O \vdash \mathtt{snd}\, e : B}\&_{E1}$$

Note that theorems 1 and 2 still hold when the additive rules are included.

The IO $\top_I$ rule introduces non-determinism into the IO typing derivations since a term of type $\top$ can consume arbitrary variables. Following the development in [6], we remove this non-determinism by introducing a flag to keep track of whether the current linear context has passed through a $\top_I$ rule.

5.1 IO-$\top$ System

Our new judgements have the following form:

$$\Delta_I \backslash \Delta_O \vdash_v e : A$$

where v is a boolean, either t or f. The $\top$ flag, v, denotes whether there is a $\top$ somewhere in the derivation which could be used to consume leftover variables in the output.

The $\top_I$ rule is now deterministic

$$\frac{}{\Delta \backslash \Delta \vdash_t () : \top}\top_I$$

and effectively delays the choice of which variables the $\top$ consumes by setting the $\top$ flag to t. The $lvar$ rule then consumes its variable as before and sets the $\top$ flag to f:

$$\frac{}{\Delta, x : A, \Delta' \backslash \Delta, \Box, \Delta' \vdash_f x : A}\, lvar$$

[6] We have elided the location information in the error.

[7] See http://ghc.haskell.org/trac/ghc/ticket/9918 for an in depth discussion.

The $\multimap_I$ rule simply passes its $\top$ flag value on:

$$\frac{\Delta_I, x{:}A \backslash \Delta_O, \Box \vdash_v e{:}B}{\Delta_I \backslash \Delta_O \vdash_v \hat{\lambda} x.e{:}A \multimap B} \multimap_I$$

However, if the $\top$ flag is t then it is ok if the linear variable was not explicitly consumed; so we have another $\multimap_I$ rule:

$$\frac{\Delta_I, x{:}A \backslash \Delta_O, x{:}A \vdash_t e{:}B}{\Delta_I \backslash \Delta_O \vdash_t \hat{\lambda} x.e{:}A \multimap B} \multimap_{It}$$

Since a $\top$ in either premise of the $\multimap_E$ rule can consume a formula, the conclusion $\top$ flag is the disjunction of those in premises:

$$\frac{\Delta_I \backslash \Delta \vdash_{v_0} e_0{:}A \multimap B \quad \Delta \backslash \Delta_O \vdash_{v_1} e_1{:}A}{\Delta_I \backslash \Delta_O \vdash_{v_0 \vee v_1} e_0\, e_1{:}B} \multimap_E$$

The $\&_I$ is now complicated by the presence of $\top$ in either premise clouding which variables are actually in the output. Thus we cannot simply require that both premises have the same output context. Instead, we need to perform a kind of intersection on the two output contexts to get a conclusion output which accurately reflects which variables must have been consumed. We define linear context intersection in the presence of $\top$ as follows:

$$\begin{aligned}
\cdot \ {}_{v_0}\sqcap_{v_1}\ \cdot &= \cdot \\
\Delta_0, x{:}A \ {}_{v_0}\sqcap_{v_1}\ \Delta_1, x{:}A &= (\Delta_0 \ {}_{v_0}\sqcap_{v_1}\ \Delta_1), x{:}A \\
\Delta_0, \Box \ {}_{v_0}\sqcap_{v_1}\ \Delta_1, \Box &= (\Delta_0 \ {}_{v_0}\sqcap_{v_1}\ \Delta_1), \Box \\
\Delta_0, x{:}A \ {}_{t}\sqcap_{v_1}\ \Delta_1, \Box &= (\Delta_0 \ {}_{t}\sqcap_{v_1}\ \Delta_1), \Box \\
\Delta_0, \Box \ {}_{v_0}\sqcap_{t}\ \Delta_1, x{:}A &= (\Delta_0 \ {}_{v_0}\sqcap_{t}\ \Delta_1), \Box
\end{aligned}$$

We can then write the $\&_I$ rule as follows:

$$\frac{\Delta_I \backslash \Delta_0 \vdash_{v_0} e_0{:}A \quad \Delta_I \backslash \Delta_1 \vdash_{v_1} e_1{:}B}{\Delta_I \backslash (\Delta_0 \ {}_{v_0}\sqcap_{v_1}\ \Delta_1) \vdash_{v_0 \wedge v_1} (e_0, e_1){:}A \,\&\, B} \&_I$$

For completeness, we state the two $\&_E$ rules:

$$\frac{\Delta_I \backslash \Delta_O \vdash_v e{:}A \,\&\, B}{\Delta_I \backslash \Delta_O \vdash_v \mathtt{fst}\, e{:}A} \&_{E0} \qquad \frac{\Delta_I \backslash \Delta_O \vdash_v e{:}A \,\&\, B}{\Delta_I \backslash \Delta_O \vdash_v \mathtt{snd}\, e{:}B} \&_{E1}$$

We will now prove the correctness of the IO-$\top$ system with respect to the IO system. We start with a useful property of ${}_{v_0}\sqcap_{v_1}$

Lemma 4.

1. $\Delta_0 \ {}_{f}\sqcap_{v}\ \Delta_1 = \Delta_0$ *and* $\Delta_0 \ {}_{v}\sqcap_{f}\ \Delta_1 = \Delta_1$
2. $\Delta_0 \sqsupseteq (\Delta_0 \ {}_{v_0}\sqcap_{v_1}\ \Delta_1)$ *and* $\Delta_1 \sqsupseteq (\Delta_0 \ {}_{v_0}\sqcap_{v_1}\ \Delta_1)$

We can now state and prove the correctness of the IO-$\top$ system in two parts as follows.

Theorem 3.

$$\Delta_I \backslash \Delta_O \vdash_f e{:}A \quad \textit{implies} \quad \Delta_I \backslash \Delta_O \vdash e{:}A$$

$$\Delta_I \backslash \Delta_O \vdash_t e{:}A \ \textit{and}\ \Delta_I \sqsupseteq \Delta \sqsupseteq (\Delta_I - \Delta_O) \quad \textit{implies} \quad \Delta_I \backslash \Delta \vdash e{:}A$$

Proof. By induction on the structure of the given derivation using lemma 4. □

Theorem 4.

$$\Delta_I \backslash \Delta_O \vdash e{:}A \ \textit{and}\ \Delta \sqsupseteq \Delta_O \quad \textit{implies} \quad \Delta_I \backslash \Delta_O \vdash_f e{:}A \quad \textit{or} \quad \Delta_I \backslash \Delta \vdash_t e{:}A$$

Proof. By induction on the structure of the given derivation using lemma 4. □

We remark that an IO-$\top$ derivation with the $\top$ flag set to t can be interpreted as a derivation in an affine type theory, i.e. each hypothesis must be used *at most* once. Thus, we can trivially get an affine lambda calculus from the IO-$\top$ system by changing the $lvar$ rule to set the $\top$ flag to t :

$$\frac{}{\Delta, x{:}A, \Delta' \backslash \Delta, \Box, \Delta' \vdash_t x{:}A}$$

and leaving the other typing rules unchanged.

5.2 Haskell Encoding of Additives

The Haskell code for the multiplicatives in section 4.2 can be extended to encode the IO-$\top$ system by simply transcribing the extension from IO to IO-$\top$ judgments and rules. We start by encoding the supporting machinery we will need.

```
class Or (x::Bool) (y::Bool) (z::Bool) | x y -> z
instance Or True y True
instance Or False y y

class And (x::Bool) (y::Bool) (z::Bool) | x y -> z
instance And False y False
instance And True y y

class MrgL (h1::[CtxElm]) (tf1::Bool)
           (h2::[CtxElm]) (tf2::Bool)
           (h::[CtxElm])
  | h1 h2 -> h
instance MrgL '[] v1 '[] v2 '[]
instance (MrgL h1 v1 h2 v2 h) =>
   MrgL (x ': h1) v1 (x ': h2) v2 (x ': h)
instance (MrgL h1 True h2 v2 h) =>
   MrgL (Elm x ': h1) True (Box ': h2) v2 (Box ': h)
instance (MrgL h1 v1 h2 True h) =>
   MrgL (Box ': h1) v1 (Elm x ': h2) True (Box ': h)
```

Note that we have collapsed two cases of the ${}_{v_0}\sqcap_{v_1}$ definition into one `MrgL` instance.

In order to capture the two $\multimap_I$ rules, we will use the following relation

```
class VarOk (tf :: Bool) (v :: CtxElm)
instance VarOk True (Elm v)
instance VarOk True Box
instance VarOk False Box
```

which specifies the valid relations between the $\top$ flag and the newly introduced linear variable in the output context. `VarOk` will be a constraint on the `llam` method which lets us collapse the two $\multimap_I$ rules into one.

We can now write out the IO-$\top$ derivation rules in Haskell; we will start by creating two new types:

```
type a & b = (a, b)
type Top = ()
```

We use type synonyms as & and $\top$ really do correspond to Haskell pairs and unit. We next extend our `repr` type with a $\top$ flag

```
repr :: Nat -> Bool -> [CtxElm] -> [CtxElm] -> * -> *
```

We change the `LVar` definition to reflect that IO-$\top$ $lvar$ rule:

```
type LVar repr v a =
  forall (v'::Nat) (i::[CtxElm]) (o::[CtxElm]) .
  Consume v i o => repr v' False i o a
```

Now we add methods to `LLC` to represent the IO-$\top$ derivation rules:

```
class LLC
  (repr :: Nat -> Bool
```

```
       -> [CtxElm] -> [CtxElm] -> * -> *
  )
 where
  llam :: VarOk tf var
       => (LVar repr v a ->
             repr (S v) tf (Elm v ': i) (var ': o) b
          )
       -> repr v tf i o (a -<> b)

  (^) :: (Or tf0 tf1 tf)
      => repr v tf0 i m (a -<> b)
      -> repr v tf1 m o a
      -> repr v tf i o b

  top :: repr v True i i Top

  (&) :: ( MrgL h0 tf0 h1 tf1 o
         , And tf0 tf1 tf
         )
       => repr v tf0 i h0 a
       -> repr v tf1 i h1 b
       -> repr v tf i o (a & b)

  pi1 :: repr v tf i o (a & b)
      -> repr v tf i o a
  pi2 :: repr v tf i o (a & b)
      -> repr v tf i o b
```

Finally we modify `Defn` to apply to IO-$\top$ judgments:

```
type Defn tf a = forall repr (v::Nat) (i::[CtxElm])
               . LLC repr => repr v tf i i a
defn :: Defn tf a -> Defn tf a
defn x = x
```

Now we can try to type an additive LLC term:

```
*Main> :{
*Main| :t defn $
*Main|    llam $ \f -> llam $ \x -> llam $ \y ->
*Main|    (f ^ x ^ y) & (f ^ y ^ x)
*Main| :}

<interactive>:44:13:
    Could not deduce (MrgL i1 'False i1 'False i1)
```

It is not too hard to see from the error message[8] that the problem lies in the interaction of `Defn` and `MrgL`. We've put no constraints upon the linear context, `i`, which the `Defn` passes through; so of course the constraint solver does not know whether `MrgL i 'False i False i` is valid. The solution is simply to place the appropriate constraints in `Defn`. There are four combinations of $\top$ flag values possible, so we have the following new `Defn`:

```
type MrgLs i = ( MrgL i False i False i
               , MrgL i False i True i
               , MrgL i True i False i
               , MrgL i True i True i
               )

type Defn tf a =
    forall repr (v::Nat) (i::[CtxElm]) .
      (LLC repr, MrgLs i) => repr v tf i i a
defn :: Defn tf a -> Defn tf a
defn x = x
```

[8] Again we have left out the location information.

where we have used constraint kinds [22] to abstract out of `Defn` the four separate constraints. When we try our previous example again:

```
*Main> :{
*Main| :t defn $
*Main|    llam $ \f -> llam $ \x -> llam $ \y ->
*Main|    (f ^ x ^ y) & (f ^ y ^ x)
*Main| :}
defn $
   llam $ \f -> llam $ \x -> llam $ \y ->
   (f ^ x ^ y) & (f ^ y ^ x)
  :: ( MrgL i 'True i 'True i
     , MrgL i 'True i 'False i
     , MrgL i 'False i 'True i
     , MrgL i 'False i 'False i
     , LLC repr
     ) =>
     repr v 'False i i
     ((a -<> (a -<> b)) -<> (a -<> (a -<> (b & b))))
```

We get the expected type (although ghci inlines the `MrgLs` constraint definition). The following ill-typed example

```
*Main> :{
*Main| :t defn $
*Main|    llam $ \f -> llam $ \x -> llam $ \y ->
*Main|    (f ^ x ^ y) & (f ^ x)
*Main| :}

<interactive>:3:16:
  Could not deduce
    (MrgL ('Box : 'Box : 'Box : i1)
          'False
          ('Elm ('S ('S v1)) : 'Box : 'Box : i1)
          'False
          ('Box : 'Box : 'Box : i1))
```

behaves as expected[9]; so too does a well-typed example with $\top$:

```
*Main> :{
*Main| :t defn $
*Main|    llam $ \f -> llam $ \x -> llam $ \y ->
*Main|    (f ^ x ^ y) & (f ^ top)
*Main| :}
defn $
   llam $ \f -> llam $ \x -> llam $ \y ->
   (f ^ x ^ y) & (f ^ top)
  :: ( MrgL i 'True i 'True i
     , MrgL i 'True i 'False i
     , MrgL i 'False i 'True i
     , MrgL i 'False i 'False i
     , LLC repr
     ) =>
     repr v 'False i i
     ((Top -<> (a1 -<> a)) -<>
       (Top -<> (a1 -<> (a & (a1 -<> a)))))
```

6. Unrestricted Functions

Up to this point, we have shown how to encode the mini-LLC of section 3 using HOAS. In order to get an encoding of full LLC, described in appendix A, we just need to show how to include unrestricted functions. Allowing both linear and unrestricted functions introduces unrestricted variables which we will accomodate in a

[9] This is actually the last of three errors, the other two arise from this one and have to do with `VarOk` not having enough information since `MrgL` fails.

separate variable context; thus our typing judgments now have the form:

$$\Gamma\,;\,\Delta \vdash e : A$$

where is Γ is the unrestricted variable context.

The unrestricted function type adds three new typing derivation rules:

$$\frac{}{\Gamma_1, x:A, \Gamma_2\,;\,\cdot \vdash x:A}\,uvar \qquad \frac{\Gamma, x:A\,;\,\Delta \vdash e:B}{\Gamma\,;\,\Delta \vdash \lambda x.e : A \to B}\,{\to_I}$$

$$\frac{\Gamma\,;\,\Delta \vdash e_0 : A \to B \qquad \Gamma\,;\,\cdot \vdash e_1 : A}{\Gamma\,;\,\Delta \vdash e_0\, e_1 : B}\,{\to_E}$$

Note the linear context must be empty in the $uvar$ rule and, correspondingly, in the minor premise of the $\to_E$ rule.

Extending the IO and IO-$\top$ systems to accomodate unrestricted variables is straightforward. We simply add an unrestricted context to the typing judgments and translate the three $\to$ derivation rules accordingly; we shall only present the IO-$\top$ version as the IO version is quite similar. Here is the IO-$\top$ judgment for full LLC:

$$\Gamma\,;\,\Delta_I \backslash \Delta_O \vdash_v e : A$$

and here are the three $\to$ IO-$\top$ rules:

$$\frac{}{\Gamma_1, x:A, \Gamma_2\,;\,\Delta \backslash \Delta \vdash_f x : A}\,uvar$$

Note that $uvar$ consumes no linear variables and sets the $\top$ flag to f.

$$\frac{\Gamma, x:A\,;\,\Delta_I \backslash \Delta_O \vdash_v e : B}{\Gamma\,;\,\Delta_I \backslash \Delta_O \vdash_v \lambda x.e : A \to B}\,{\to_I}$$

$$\frac{\Gamma\,;\,\Delta_I \backslash \Delta_O \vdash_v e_0 : A \to B \quad \Gamma\,;\,\cdot \backslash \cdot \vdash_{v'} e_1 : A}{\Gamma\,;\,\Delta_I \backslash \Delta_O \vdash_v e_0\, e_1 : B}\,{\to_E}$$

The $\to_E$ enforces an empty linear context in the minor premise by passing in an empty context; it would also be correct to use Δ_O, or any context. This is in contrast to the $uvar$ rule which cannot require an empty context even though no linear formulas can be consumed. The reason for the difference is that the $\to_E$ rule can "decide" the input context of its second premise, while the $uvar$ rule cannot control what input context it receives. For reference, figure 2 contains the IO-$\top$ rules for mini-LLC plus unrestricted functions. Note that theorems 1, 2, 3, and 4 can all be trivially extended to apply to languages with unrestricted functions. Additionally, the complete IO-$\top$ system for full LLC is given in appendix B.

6.1 Haskell Encoding of Unrestricted Functions

Extending the Haskell code of section 5.2 to include unrestricted functions is surprisingly easy. Since Haskell variables behave the same as unrestricted variables, we may just transcribe the new derivation rules without adding any new machinery.

```
type UVar repr a =
  forall (v::Nat) (i::[CtxElm]) .
  repr v False i i a

class LLC
  (repr :: Nat -> Bool
        -> [CtxElm] -> [CtxElm] -> * -> *
  )
 where
  ulam :: (UVar repr a -> repr v tf i o b)
       -> repr v tf i o (a -> b)

  ($$) :: repr v tf i o (a -> b)
       -> repr v tf' '[] '[] a
       -> repr v tf i o b
```

$$\frac{}{\Gamma_1, x:A, \Gamma_2\,;\,\Delta \backslash \Delta \vdash_f x : A}\,uvar$$

$$\frac{\Gamma, x:A\,;\,\Delta_I \backslash \Delta_O \vdash_v e : B}{\Gamma\,;\,\Delta_I \backslash \Delta_O \vdash_v \lambda x.e : A \to B}\,{\to_I}$$

$$\frac{\Gamma\,;\,\Delta_I \backslash \Delta_O \vdash_v e_0 : A \to B \quad \Gamma\,;\,\cdot \backslash \cdot \vdash_{v'} e_1 : A}{\Gamma\,;\,\Delta_I \backslash \Delta_O \vdash_v e_0\, e_1 : B}\,{\to_E}$$

$$\frac{}{\Gamma; \Delta, x:A, \Delta' \backslash \Delta, \Box, \Delta' \vdash_f x : A}\,lvar$$

$$\frac{\Gamma; \Delta_I, x:A \backslash \Delta_O, \Box \vdash_v e : B}{\Gamma; \Delta_I \backslash \Delta_O \vdash_v \hat{\lambda} x.e : A \multimap B}\,{\multimap_I}$$

$$\frac{\Gamma; \Delta_I, x:A \backslash \Delta_O, x:A \vdash_t e : B}{\Gamma; \Delta_I \backslash \Delta_O \vdash_t \hat{\lambda} x.e : A \multimap B}\,{\multimap_{It}}$$

$$\frac{\Gamma; \Delta_I \backslash \Delta \vdash_{v_0} e_0 : A \multimap B \quad \Gamma; \Delta \backslash \Delta_O \vdash_{v_1} e_1 : A}{\Gamma; \Delta_I \backslash \Delta_O \vdash_{v_0 \vee v_1} e_0\, e_1 : B}\,{\multimap_E}$$

$$\frac{}{\Gamma; \Delta \backslash \Delta \vdash_t () : \top}\,{\top_I}$$

$$\frac{\Gamma; \Delta_I \backslash \Delta_0 \vdash_{v_0} e_0 : A \quad \Gamma; \Delta_I \backslash \Delta_1 \vdash_{v_1} e_1 : B}{\Gamma; \Delta_I \backslash (\Delta_0\ {}_{v_0}\!\sqcap_{v_1}\ \Delta_1) \vdash_{v_0 \wedge v_1} (e_0, e_1) : A \,\&\, B}\,{\&_I}$$

$$\frac{\Gamma; \Delta_I \backslash \Delta_O \vdash_v e : A \,\&\, B}{\Gamma; \Delta_I \backslash \Delta_O \vdash_v \mathtt{fst}\, e : A}\,{\&_{E0}} \qquad \frac{\Gamma; \Delta_I \backslash \Delta_O \vdash_v e : A \,\&\, B}{\Gamma; \Delta_I \backslash \Delta_O \vdash_v \mathtt{snd}\, e : B}\,{\&_{E1}}$$

Figure 2. IO-$\top$ derivation rules with unrestricted functions.

We elide the other methods of the `LLC` class which are unchanged from section 5.2.

We now show some terms. We start with an ill-typed purely linear term

```
*Main> :t defn $ llam $ \f -> llam $ \x -> f

<interactive>:1:34:
    Could not deduce (VarOk 'False ('Elm ('S v1)))
```

which can be made well-typed by changing the inner function to being unrestricted

```
*Main> :t defn $ llam $ \f -> ulam $ \x -> f
defn $ llam $ \f -> ulam $ \x -> f
  :: ( MrgL i 'True i 'True i
     , MrgL i 'True i 'False i
     , MrgL i 'False i 'True i
     , MrgL i 'False i 'False i
     , LLC repr
     ) =>
     repr v 'False i i (b -<> (a -> b))
```

Next we'll demonstrate that unrestricted application

```
*Main> :t defn $ llam $ \f -> llam $ \x -> f $$ x

<interactive>:1:34:
    Could not deduce (VarOk 'False ('Elm ('S v1)))
```

requires an unrestricted argument

```
*Main> :t defn $ llam $ \f -> ulam $ \x -> f $$ x
defn $ llam $ \f -> ulam $ \x -> f $$ x
  :: ( MrgL i 'True i 'True i
```

```
    , MrgL i 'True i 'False i
    , MrgL i 'False i 'True i
    , MrgL i 'False i 'False i
    , LLC repr
    ) =>
    repr v 'False i i ((a -> b) -<> (a -> b))
```

Appendix C contains a complete implementation of full LLC in Haskell.

7. A Concrete Instance

Up to this point, we have been working with pure syntax; i.e. we have been using methods of a type class with no actual instances. One of the nice features of tagless final encodings is their flexibility; they provide a convenient mechanism for isolating syntax which can be re-used under different concrete interpretations. A common interpretation of an object language syntax is its evaluation, see [2, 10] for many examples. HOAS often leads to almost trivial evaluation machinery since the object language tends to correspond closely with the meta language.

Even though Haskell and LLC significantly differ, the LLC encoding developed in this paper enjoys a trivial evaluation interpretation. This is due to the forgetful nature of our encoding as remarked in section 4.2; our linear functions are just regular haskell functions with some constraints on the argument. The following code implements an interpreter for our LLC type class.

```
newtype Ev (v::Nat)
           (tf::Bool)
           (i::[CtxElm])
           (o::[CtxElm])
           a
     = Ev {ev :: a}

instance LLC
    (Ev :: Nat -> Bool
        -> [CtxElm] -> [CtxElm] -> * -> *
    )
  where
    llam f = Ev $ Lolli $ \x -> ev (f (Ev x))
    f ^ x = Ev $ unLolli (ev f) (ev x)

    ulam f = Ev $ \x -> ev (f (Ev x))
    f $$ x = Ev $ ev f (ev x)

    top = Ev ()

    x & y = Ev $ (ev x, ev y)
    pi1 = Ev . fst . ev
    pi2 = Ev . snd . ev
```

We define the following top level evaluation function which checks that a term is closed by instantiating the various types which make up the linear constraint machinery.

```
eval :: Ev Z tf '[] '[] a -> a
eval = ev
```

The results of `eval` really are terms:

```
*Main> :{
*Main| putStrLn $
*Main|   (unLolli . eval $ llam (\x -> x)) "hello"
*Main| :}
hello
```

The `unLolli` coercion is necessary since we defined `-<>` as a `newtype` rather than a type synonym. If we made `-<>` a type synonym we would alleviate the need for coercions and we'd still have proper type inference; but Haskell would not be able to distinguish between `-<>` and `->` and would accept bad type ascriptions, i.e. `llam (\x -> x) :: Defn False (a -> a)` would be accepted.

8. Related work

Kiselyov presents a tagless final encoding of linear and unrestricted lambdas in [10][10]. However the encoding uses deBruijn indices which complicate the presentation by requiring two type classes to separate out derivation rules which require constraints on the output linear context, e.g. the $\multimap_I$ rule requires that the output context not have the newly introduced variable. The use of deBruijn indices additionally complicates the user experience.

The general idea of explicitly representing the context of in-scope variables to allow HOAS representations in Haskell of languages with "fancy" types has been used in [9] to encode a staged language with effects. The explicit contexts allow for type class machinery to statically ensure that various code generation techniques are only applicable in safe contexts, i.e. where the generated code will be well-typed.

9. Conclusions and Future Work

We have presented a HOAS encoding of a full LLC with multiplicatives, additives, and units. This encoding is fairly lightweight and allows Haskell to do both linear type checking and linear type inference. The encoding relies upon standard representation techniques from higher order logic programming and LF style logical frameworks [5, 18]. We think this general approach would work well for encoding other systems into Haskell such as an ordered LLC, lambda box [16], or even languages with session types [12, 20].

We also think there might be uses of this encoding as an embedded domain specific language. Since linear functions are directly represented by Haskell functions, it seems possible that this embedding could provide a reasonably lightweight mechanism to incorporate linear types into larger Haskell programs. We would like to explore techniques for turning this into an EDSL as well as potential uses of linear types in Haskell code.

We motivated, and proved correct, the translation of standard non-deterministic linear typing derivations into deterministic typing derivations. However, we did not try to prove the correctness of the Haskell encoding of the deterministic system. We would like to explore methods for formalizing the correctness of our encoding (and similar kinds of encodings) along the lines of the adequacy results advocated in [5].

Additionally, since our encoding cannot be done with closed type families, we have highlighted a concerete difference between type classes and type families. Hopefully we have shown something useful which is within the scope of type classes (with multiple parameters, functional dependencies, and overlapping instances) which is not possible with the current implementation of (closed) type families.

10. Acknowledgements

We'd like to thank the anonymous reviewers for helpful comments, and Oleg Kiselyov for helpful feedback and pointers to related work.

References

[1] Andrew Barber and Gordon Plotkin. *Dual intuitionistic linear logic*. University of Edinburgh, Department of Computer Science, Labora-

[10] Code available at http://okmij.org/ftp/tagless-final/course/LinearLC.hs

tory for Foundations of Computer Science, 1996.

[2] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(05):509–543, 2009.

[3] A. Church. A formulation of a simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[4] Richard A Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. Closed type families with overlapping equations. *ACM SIGPLAN Notices*, 49(1):671–683, 2014.

[5] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM (JACM)*, 40(1):143–184, 1993.

[6] Joshua S. Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design and Implementation*. PhD thesis, University of Pennsylvania, Department of Computer and Information Science, 1994.

[7] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica.*, 11(1):31–55, 1978.

[8] Mark P Jones. Type classes with functional dependencies. In *Programming Languages and Systems*, pages 230–244. Springer, 2000.

[9] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung chieh Shan. Combinators for impure yet hygienic code generation. In Wei-Ngan Chin and Jurriaan Hage, editors, *PEPM*, pages 3–14. ACM, 2014.

[10] Oleg Kiselyov. Typed tagless final interpreters. In *Generic and Indexed Programming*, pages 130–174. Springer, 2012.

[11] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM, 2004.

[12] Sam Lindley and J.Garrett Morris. A semantics for propositions as sessions. In Jan Vitek, editor, *Programming Languages and Systems*, volume 9032 of *Lecture Notes in Computer Science*, pages 560–584. Springer Berlin Heidelberg, 2015. .

[13] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for gadts. In *ACM SIGPLAN Notices*, volume 41, pages 50–61. ACM, 2006.

[14] Frank Pfenning. Linear functional programming. Lecture 16 from a course on Linear Logic., 2001.

[15] Frank Pfenning. Linear lambda calculus. Lecture 15 from a course on Linear Logic., 2001.

[16] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical structures in computer science*, 11(04): 511–540, 2001.

[17] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *ACM SIGPLAN Notices*, volume 23, pages 199–208. ACM, 1988.

[18] Frank Pfenning et al. Logic programming in the lf logical framework. *Logical frameworks*, pages 149–181, 1991.

[19] Jeff Polakow. *Ordered linear logic and applications*. PhD thesis, Carnegie Mellon University, 2001.

[20] Bernardo Toninho, Luis Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, pages 350–369, Berlin, Heidelberg, 2013. Springer-Verlag. .

[21] Philip Wadler. A taste of linear logic. In *Mathematical Foundations of Computer Science 1993*, pages 185–210. Springer, 1993.

[22] Brent A Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, pages 53–66. ACM, 2012.

A. Full Linear Lambda Calculus

Types

$$
\begin{array}{rcll}
A & ::= & A_0 \multimap A_1 & \text{linear functions} \\
& | & A_0 \to A_1 & \text{unrestricted functions} \\
& | & !A & \text{unrestricted modality} \\
& | & \top & \text{additive unit} \\
& | & A_0 \,\&\, A_1 & \text{additive conjunction} \\
& | & 1 & \text{multiplicative unit} \\
& | & A_0 \otimes A_1 & \text{multiplicative conjunction} \\
& | & 0 & \text{additive zero} \\
& | & A_0 \oplus A_1 & \text{additive disjunction}
\end{array}
$$

We will use capital letters, $A, B, \ldots$, to stand for types.

Terms

$$
\begin{array}{rcll}
e & ::= & x & \text{variables} \\
& | & \hat{\lambda} x.e \mid e_0 \hat{} e_1 & \text{linear functions} \\
& | & \lambda x.e \mid e_0\, e_1 & \text{unrestricted functions} \\
& | & !e \mid \texttt{let}\ !x = e_0\ \texttt{in}\ e_1 & \text{unrestricted modality} \\
& | & () & \text{additive unit} \\
& | & (e_0, e_1) \mid \texttt{fst}\, e \mid \texttt{snd}\, e & \text{additive conjunction} \\
& | & \langle\rangle \mid \texttt{let}\ \langle\rangle = e_0\ \texttt{in}\ e_1 & \text{multiplicative unit} \\
& | & \langle e_0, e_1 \rangle \mid & \text{multiplicative conjunction} \\
& & \texttt{let}\ x \otimes y = e_0\ \texttt{in}\ e_1 & \\
& | & \texttt{abort}^A\, e & \text{additive zero} \\
& | & \texttt{inl}^A e \mid \texttt{inr}^A e \mid & \text{additive disjunction} \\
& & \texttt{case}\ e_0\ \texttt{of inl}\ x \Rightarrow e_1 \mid \texttt{inr}\ x \Rightarrow e_2 &
\end{array}
$$

Variable Contexts

$$\Gamma ::= \cdot \mid \Gamma, x{:}A$$

We use Γ, Δ to stand for contexts. We will sometimes overload $,$ to mean list append as well as list cons (as above); i.e. $\Gamma_1, x{:}A, \Gamma_2$ denotes a list which contains x to the right of everything in Γ_1 and to the left of everything in Γ_2.

Non-deterministic Context Merge

$$
\begin{array}{rcl}
\cdot \bowtie \cdot & = & \cdot \\
\Delta, x{:}A \bowtie \Delta' & = & (\Delta \bowtie \Delta'), x{:}A \\
\Delta \bowtie \Delta', x'{:}A' & = & (\Delta \bowtie \Delta'), x'{:}A'
\end{array}
$$

Typing Judgement

$$\Gamma\,;\, \Delta \vdash e{:}A$$

Typing Derivations

$$\frac{}{\Gamma\,;\, x{:}A \vdash x{:}A}\, lvar \qquad \frac{\Gamma\,;\, \Delta, x{:}A \vdash e{:}B}{\Gamma\,;\, \Delta \vdash \hat{\lambda} x.e{:}A \multimap B} \multimap_I$$

$$\frac{\Gamma\,;\, \Delta_0 \vdash e_0{:}A \multimap B \quad \Gamma\,;\, \Delta_1 \vdash e_1{:}A}{\Gamma\,;\, \Delta_0 \bowtie \Delta_1 \vdash e_0 \hat{} e_1{:}B} \multimap_E$$

$$\frac{}{\Gamma_1, x{:}A, \Gamma_2\,;\, \cdot \vdash x{:}A}\, uvar \qquad \frac{\Gamma, x{:}A\,;\, \Delta \vdash e{:}B}{\Gamma\,;\, \Delta \vdash \lambda x.e{:}A \to B} \to_I$$

$$\frac{\Gamma\,;\, \Delta \vdash e_0{:}A \to B \quad \Gamma\,;\, \cdot \vdash e_1{:}A}{\Gamma\,;\, \Delta \vdash e_0\, e_1{:}B} \to_E$$

$$\frac{\Gamma\,;\, \cdot \vdash e{:}A}{\Gamma\,;\, \cdot \vdash !e{:}!A}\, !_I$$

$$\frac{\Gamma\,;\, \Delta_0 \vdash e_0{:}!A \qquad \Gamma, x{:}A\,;\, \Delta_1 \vdash e_1{:}B}{\Gamma\,;\, \Delta_0 \bowtie \Delta_1 \vdash \texttt{let}\ !x = e_0\ \texttt{in}\ e_1{:}B}\, !_E$$

$$\frac{}{\Gamma\,;\,\Delta \vdash\, ():\top}\top_I$$

$$\frac{\Gamma\,;\,\Delta \vdash\, e_0 : A \quad \Gamma\,;\,\Delta \vdash\, e_1 : B}{\Gamma\,;\,\Delta \vdash\, (e_0, e_1) : A \,\&\, B}\&_I$$

$$\frac{\Gamma\,;\,\Delta \vdash\, e : A \,\&\, B}{\Gamma\,;\,\Delta \vdash\, \mathtt{fst}\, e : A}\&_{E0} \qquad \frac{\Gamma\,;\,\Delta \vdash\, e : A \,\&\, B}{\Gamma\,;\,\Delta \vdash\, \mathtt{snd}\, e : B}\&_{E1}$$

$$\frac{}{\Gamma\,;\,\cdot \vdash\, \langle\rangle : 1}1_I \qquad \frac{\Gamma\,;\,\Delta_0 \vdash\, e_0 : 1 \quad \Gamma\,;\,\Delta_1 \vdash\, e_1 : C}{\Gamma\,;\,\Delta_0 \bowtie \Delta_1 \vdash\, \mathtt{let}\ \langle\rangle = e_0\ \mathtt{in}\ e_1 : C}1_E$$

$$\frac{\Gamma\,;\,\Delta_0 \vdash\, e_0 : A \quad \Gamma\,;\,\Delta_1 \vdash\, e_1 : B}{\Gamma\,;\,\Delta_0 \bowtie \Delta_1 \vdash\, \langle e_0, e_1\rangle : A \otimes B}\otimes_I$$

$$\frac{\Gamma\,;\,\Delta_0 \vdash\, e_0 : A \otimes B \quad \Gamma\,;\,\Delta_1, x:A, y:B \vdash\, e_1 : C}{\Gamma\,;\,\Delta_0 \bowtie \Delta_1 \vdash\, \mathtt{let}\ x \otimes y = e_0\ \mathtt{in}\ e_1 : C}\otimes_E$$

$$\frac{\Gamma\,;\,\Delta \vdash\, e : 0}{\Gamma\,;\,\Delta \bowtie \Delta' \vdash\, \mathtt{abort}^C\, e : C}0_E$$

$$\frac{\Gamma\,;\,\Delta \vdash\, e : A}{\Gamma\,;\,\Delta \vdash\, \mathtt{inl}^B e : A \oplus B}\oplus_{Il} \qquad \frac{\Gamma\,;\,\Delta \vdash\, e : B}{\Gamma\,;\,\Delta \vdash\, \mathtt{inr}^A e : A \oplus B}\oplus_{Ir}$$

$$\frac{\Gamma\,;\,\Delta_0 \vdash\, e_0 : A \oplus B \qquad \begin{array}{l}\Gamma\,;\,\Delta_1, x:A \vdash\, e_1 : C \\ \Gamma\,;\,\Delta_1, y:B \vdash\, e_2 : C\end{array}}{\Gamma\,;\,\Delta_0 \bowtie \Delta_1 \vdash\, \mathtt{case}\ e_0\ \mathtt{of}\ \mathtt{inl}\, x \Rightarrow e_1 \,|\, \mathtt{inr}\, x \Rightarrow e_2 : C}\oplus_E$$

B. IO-$\top$ Derivations for Full Linear Lambda Calculus

Variable Contexts

$$\Gamma ::= \cdot \mid \Gamma, x:A \mid \Gamma, \Box$$

We use Γ, Δ to stand for contexts. We will sometimes overload $,$ to mean list append as well as list cons (as above); i.e. $\Delta_1, x:A, \Delta_2$ denotes a list which contains x to the right of everything in Δ_1 and to the left of everything in Δ_2.

Typing Judgement

$$\Gamma\,;\,\Delta_I \backslash \Delta_O \vdash_v e : A$$

Typing Derivations

$$\frac{}{\Gamma; \Delta, x:A, \Delta' \backslash \Delta, \Box, \Delta' \vdash_f\, x : A}lvar$$

$$\frac{\Gamma; \Delta_I, x:A \backslash \Delta_O, X \vdash_v e : B}{\Gamma; \Delta_I \backslash \Delta_O \vdash_v \hat{\lambda} x.e : A \multimap B}\multimap_I$$

where $X \equiv \Box$ `or` $X \equiv x:A$
and $v \equiv f$ `implies` $X \equiv \Box$.

$$\frac{\Gamma; \Delta_I \backslash \Delta \vdash_{v_0} e_0 : A \multimap B \quad \Gamma; \Delta \backslash \Delta_O \vdash_{v_1} e_1 : A}{\Gamma; \Delta_I \backslash \Delta_O \vdash_{v_0 \vee v_1} e_0\, e_1 : B}\multimap_E$$

$$\frac{}{\Gamma_1, x:A, \Gamma_2\,;\,\Delta \backslash \Delta \vdash_f\, x : A}uvar$$

$$\frac{\Gamma, x:A\,;\,\Delta_I \backslash \Delta_O \vdash_v e : B}{\Gamma\,;\,\Delta_I \backslash \Delta_O \vdash_v \lambda x.e : A \to B}\to_I$$

$$\frac{\Gamma\,;\,\Delta_I \backslash \Delta_O \vdash_v e_0 : A \to B \quad \Gamma\,;\,\cdot \backslash \cdot \vdash_{v'} e_1 : A}{\Gamma\,;\,\Delta_I \backslash \Delta_O \vdash_v e_0\, e_1 : B}\to_E$$

$$\frac{\Gamma\,;\,\cdot \backslash \cdot \vdash_v e : A}{\Gamma\,;\,\Delta \backslash \Delta \vdash_f !e : !A}!_I$$

$$\frac{\Gamma\,;\,\Delta_I \backslash \Delta \vdash_{v_0} e_0 : !A \qquad \Gamma, x:A\,;\,\Delta \backslash \Delta_O \vdash_{v_1} e_1 : B}{\Gamma\,;\,\Delta_I \backslash \Delta_O \vdash_{v_0 \vee v_1} \mathtt{let}\ !x = e_0\ \mathtt{in}\ e_1 : B}!_E$$

$$\frac{}{\Gamma; \Delta \backslash \Delta \vdash_t\, () : \top}\top_I$$

$$\frac{\Gamma; \Delta_I \backslash \Delta_0 \vdash_{v_0} e_0 : A \quad \Gamma; \Delta_I \backslash \Delta_1 \vdash_{v_1} e_1 : B}{\Gamma; \Delta_I \backslash (\Delta_0\ {}_{v_0}\sqcap_{v_1}\ \Delta_1) \vdash_{v_0 \wedge v_1} (e_0, e_1) : A \,\&\, B}\&_I$$

$$\frac{\Gamma; \Delta_I \backslash \Delta_O \vdash_v e : A \,\&\, B}{\Gamma; \Delta_I \backslash \Delta_O \vdash_v \mathtt{fst}\, e : A}\&_{E0} \qquad \frac{\Gamma; \Delta_I \backslash \Delta_O \vdash_v e : A \,\&\, B}{\Gamma; \Delta_I \backslash \Delta_O \vdash_v \mathtt{snd}\, e : B}\&_{E1}$$

$$\frac{}{\Gamma\,;\,\Delta \backslash \Delta \vdash_f\, \langle\rangle : 1}1_I \qquad \frac{\Gamma\,;\,\Delta_I \backslash \Delta \vdash_{v_0} e_0 : 1 \quad \Gamma\,;\,\Delta \backslash \Delta_O \vdash_{v_1} e_1 : C}{\Gamma\,;\,\Delta_I \backslash \Delta_O \vdash_{v_0 \vee v_1} \mathtt{let}\ \langle\rangle = e_0\ \mathtt{in}\ e_1 : C}1_E$$

$$\frac{\Gamma\,;\,\Delta_I \backslash \Delta \vdash_{v_0} e_0 : A \quad \Gamma\,;\,\Delta \backslash \Delta_O \vdash_{v_1} e_1 : B}{\Gamma\,;\,\Delta_I \backslash \Delta_O \vdash_{v_0 \vee v_1} \langle e_0, e_1\rangle : A \otimes B}\otimes_I$$

$$\frac{\Gamma\,;\,\Delta_I \backslash \Delta \vdash_{v_0} e_0 : A \otimes B \quad \Gamma\,;\,\Delta, x:A, y:B \backslash \Delta_O, X, Y \vdash_{v_1} e_1 : C}{\Gamma\,;\,\Delta_I \backslash \Delta_O \vdash_{v_0 \vee v_1} \mathtt{let}\ x \otimes y = e_0\ \mathtt{in}\ e_1 : C}\otimes_E$$

where $X \equiv \Box$ `or` $X \equiv x:A$
and $Y \equiv \Box$ `or` $Y \equiv y:B$
and $v_1 \equiv f$ `implies` $(X \equiv \Box$ `and` $Y \equiv \Box)$.

$$\frac{\Gamma\,;\,\Delta_I \backslash \Delta_O \vdash_v e : 0}{\Gamma\,;\,\Delta_I \backslash \Delta_O \vdash_t \mathtt{abort}^C\, e : C}0_E$$

$$\frac{\Gamma\,;\,\Delta_I \backslash \Delta_O \vdash_v e : A}{\Gamma\,;\,\Delta_I \backslash \Delta_O \vdash_v \mathtt{inl}^B e : A \oplus B}\oplus_{Il} \qquad \frac{\Gamma\,;\,\Delta_I \backslash \Delta_O \vdash_v e : B}{\Gamma\,;\,\Delta_I \backslash \Delta_O \vdash_v \mathtt{inr}^A e : A \oplus B}\oplus_{Ir}$$

$$\frac{\Gamma\,;\,\Delta_I \backslash \Delta \vdash_{v_0} e_0 : A \oplus B \qquad \begin{array}{l}\Gamma\,;\,\Delta, x:A \backslash \Delta_1, X \vdash_{v_1} e_1 : C \\ \Gamma\,;\,\Delta, y:B \backslash \Delta_2, Y \vdash_{v_2} e_2 : C\end{array}}{\Gamma\,;\,\Delta_I \backslash (\Delta_1\ {}_{v_1}\sqcap_{v_2}\ \Delta_2) \vdash_v \mathtt{case}\ e_0\ \mathtt{of}\ \mathtt{inl}\, x \Rightarrow e_1 \,|\, \mathtt{inr}\, x \Rightarrow e_2 : C}\oplus_E$$

where $v \equiv v_0 \wedge (v_1 \vee v_2)$
and $X \equiv \Box$ `or` $X \equiv x:A$
and $Y \equiv \Box$ `or` $Y \equiv y:B$
and $v_1 \equiv f$ `implies` $X \equiv \Box$
and $v_2 \equiv f$ `implies` $Y \equiv \Box$.

C. Full Linear Lambda Calculus in Haskell

```
{-# LANGUAGE
  ConstraintKinds,
  DataKinds,
  FlexibleContexts,
  FlexibleInstances,
  FunctionalDependencies,
  KindSignatures,
  MultiParamTypeClasses,
  NoMonomorphismRestriction,
  OverlappingInstances,
  PolyKinds,
  RankNTypes,
  TypeFamilies,
  TypeOperators,
  UndecidableInstances
 #-}

import Prelude hiding( (^), (*), (+))

--
-- Linear types
```

```
--
newtype a -<> b = Lolli {unLolli :: a -> b}
newtype Bang a = Bang {unBang :: a}
type Top = ()
type a & b = (a, b)
data One = One
data a * b = Tensor a b
data a + b = Inl a | Inr b
data Zero

--
-- linear variable vid in Haskell context
--
type LVar repr (vid::Nat) a =
    forall (v::Nat)
           (i::[Maybe Nat])
           (o::[Maybe Nat])
    . Consume vid i o => repr v False i o a

--
-- unrestricted variable in Haskell context
--
type UVar repr a =
    forall (vid::Nat)
           (i::[Maybe Nat])
    . repr vid False i i a

--
-- The syntax of LLC.
--
class LLC (repr :: Nat
                -> Bool
                -> [Maybe Nat]
                -> [Maybe Nat]
                -> *
                -> *
               ) where
  llam
    :: (VarOk tf var)
    => (LVar repr vid a -> repr (S vid)
                                tf
                                (Just vid ': i)
                                (var ': o)
                                b
       )
    -> repr vid tf i o (a -<> b)
  (^)
    :: (Or tf1 tf2 tf)
    => repr vid tf1 i h (a -<> b)
    -> repr vid tf2 h o a
    -> repr vid tf i o b

  ulam
    :: (UVar repr a -> repr vid tf i o b)
    -> repr vid tf i o (a -> b)
  ($$)
    :: repr vid tf0 i o (a -> b)
    -> repr vid tf1 '[] '[] a
    -> repr vid tf0 i o b

  bang
    :: repr vid tf '[] '[] a
    -> repr vid False i i (Bang a)
  letBang
    :: (Or tf0 tf1 tf)
    => repr vid tf0 i h (Bang a)
    -> (UVar repr a -> repr vid tf1 h o b)
    -> repr vid tf i o b

  top
    :: repr vid True i i Top

  (&)
    :: ( MrgL h0 tf0 h1 tf1 o
       , And tf0 tf1 tf
       )
    => repr vid tf0 i h0 a
    -> repr vid tf1 i h1 b
    -> repr vid tf i o (a & b)
  pi1
    :: repr vid tf i o (a & b)
    -> repr vid tf i o a
  pi2
    :: repr vid tf i o (a & b)
    -> repr vid tf i o b

  one
    :: repr vid False i i One
  letOne
    :: (Or tf0 tf1 tf)
    => repr vid tf0 i h One
    -> repr vid tf1 h o a
    -> repr vid tf i o a

  (*)
    :: (Or tf0 tf1 tf)
    => repr vid tf0 i h a
    -> repr vid tf1 h o b
    -> repr vid tf i o (a * b)
  letStar
    :: ( VarOk tf1 var0
       , VarOk tf1 var1
       , Or tf0 tf1 tf
       )
    => repr vid tf0 i h (a * b)
    -> (LVar repr vid a
        -> LVar repr (S vid) b
        -> repr (S (S vid))
                tf1
                (Just vid ': Just (S vid) ': h)
                (var0 ': var1 ': o)
                c
       )
    -> repr vid tf i o c

  inl
    :: repr vid tf i o a
    -> repr vid tf i o (a + b)
  inr
    :: repr vid tf i o b
    -> repr vid tf i o (a + b)
  letPlus
    :: ( MrgL o1 tf1 o2 tf2 o
       , And tf1 tf2 tf3
       , Or tf0 tf3 tf
       , VarOk tf1 var1
       , VarOk tf2 var2
       )
    => repr vid tf0 i h (a + b)
    -> (LVar repr vid a -> repr (S vid)
```

```
                              tf1
                              (Just vid ': h)
                              (var1 ': o1)
                              c
      )
    -> (LVar repr vid b -> repr (S vid)
                              tf2
                              (Just vid ': h)
                              (var2 ': o2)
                              c
      )
    -> repr vid tf i o c

  abort
    :: repr vid tf i o Zero
    -> repr vid True i o a

--
-- A definition for a closed LLC term.
--
type MrgLs i = ( MrgL i False i False i
               , MrgL i False i True i
               , MrgL i True i False i
               , MrgL i True i True i
               )

type Defn tf a =
    forall repr i vid
    . (LLC repr, MrgLs i)
    => repr vid tf i i a
defn :: Defn tf a -> Defn tf a
defn x = x


{-------------------------------------------------

Type level machinery

-------------------------------------------------}

--
-- We will use type level Nats
--
data Nat = Z | S Nat

class Or (x::Bool) (y::Bool) (z::Bool) | x y -> z
instance Or True y True
instance Or False y y

class And (x::Bool) (y::Bool) (z::Bool) | x y -> z
instance And False y False
instance And True y y

--
-- Type level machinery for consuming a variable
-- in a list of variables.
--
class Consume (v::Nat)
              (i::[Maybe Nat])
              (o::[Maybe Nat])
      | v i -> o
class Consume1 (b::Bool)
               (v::Nat)
               (x::Nat)
               (i::[Maybe Nat])
                    (o::[Maybe Nat])
      | b v x i -> o

instance (Consume v i o)
      => Consume v (Nothing ': i) (Nothing ': o)
instance (EQ v x b, Consume1 b v x i o)
      => Consume v (Just x ': i) o

instance Consume1 True v x i (Nothing ': i)
instance (Consume v i o)
      => Consume1 False v x i (Just x ': o)

class EQ (x::k) (y::k) (b::Bool) | x y -> b
instance EQ x x True
instance (b ~ False) => EQ x y b


--
-- Type level machinery for merging outputs of
-- additive operations and getting right Top flag.
--
class MrgL (h1::[Maybe Nat])
           (tf1::Bool)
           (h2::[Maybe Nat])
           (tf2::Bool)
           (h::[Maybe Nat])
  | h1 h2 -> h
instance MrgL '[] v1 '[] v2 '[]
instance (MrgL h1 v1 h2 v2 h)
      => MrgL (x ': h1)
              v1
              (x ': h2)
              v2
              (x ': h)
instance (MrgL h1 True h2 v2 h)
      => MrgL (Just x ': h1)
              True
              (Nothing ': h2)
              v2
              (Nothing ': h)
instance (MrgL h1 v1 h2 True h)
      => MrgL (Nothing ': h1)
              v1
              (Just x ': h2)
              True
              (Nothing ': h)

--
-- Check, in -<> type rule, that Top flag
-- was set or hypothesis was consumed.
--
class VarOk (tf :: Bool) (v :: Maybe Nat)
instance VarOk True (Just v)
instance VarOk True Nothing
instance VarOk False Nothing
```