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 language. The idea was explicitly stated and used in [7], although its 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 explicit representation for variables which are implicitly represented in the object language; i.e. in the above there is no explicit substitution in the object language.

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 calculus 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 ⊢ as a function and Γ as implicit):

\[
\begin{align*}
  \text{x:A} & \in \Gamma \quad h \triangleright x \colon A \\
  \Gamma \vdash x : A & \\
  \Gamma, x:A & \vdash e : B \\
  \Gamma & \vdash \text{Ax.e} : A \rightarrow B \rightarrow t \\
  \Gamma & \vdash e_0 : A \rightarrow B \\
  \Gamma & \vdash e_1 : A \\
  \Gamma & \vdash e_0 e_1 : B \rightarrow E
\end{align*}
\]

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.
3. Mini Linear Lambda Calculus

LLC can be thought of as the proof terms arising from a natural deduction style presentation of intuitionistic 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].

The types of mini-LLC are as follows:

\[
A ::= A_0 \rightarrow A_1 \quad \text{linear functions} \\
| A_0 \& A_1 \quad \text{additive conjunction} \\
| T \quad \text{additive unit}
\]

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

The terms of mini-LLC are as follows:

\[
e ::= x \quad \text{variables} \\
| \lambda x.e \quad \text{linear functions} \\
| e_0, e_1, \ldots \quad \text{linear functions} \\
| \text{fst } e \quad \text{additive conjunction} \\
| \text{snd } e \quad \text{additive unit}
\]

We use the following judgement for typing derivations

\[
\Delta \vdash e : A
\]

where \(\Delta\) is the linear variable context.

A variable context is a list of typed variables:

\[
\Delta ::= \cdot | \Delta, x : A
\]

We will use \(\Delta, \Gamma\) 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 \(\Delta_1\) and to the left of everything in \(\Delta_2\).

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:

\[
\cdot \otimes \cdot = \cdot \\
\Delta, x : A \otimes \Delta' = (\Delta \otimes \Delta'), x : A \\
\Delta \otimes \Delta', x' : A' = (\Delta \otimes \Delta'), x' : A'
\]

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

4. Multiplicatives

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

\[
x : A \vdash x : A
\]

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.

\[
\Delta, x : A \vdash e : B \quad \Delta \vdash e_0 : A \rightarrow B \quad \Delta_1 \vdash e_1 : A \quad \Gamma \vdash e : A \\
\Delta \vdash \lambda x.e : A \rightarrow B
\]

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, 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 \(-\otimes\) 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], a similar presentation directly on ILL can be found in [14].

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:

\[
\Delta \setminus \Delta_0 \vdash e : A
\]

where \(\Delta_0, \Delta_I\) are the input variables and \(\Delta_0\) 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, \(\square\):

\[
\Delta ::= \cdot | \Delta, x : A | \Delta, \square
\]

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:

\[
\Delta, x : A, \Delta \setminus \Delta, \square, \Delta' \vdash x : A
\]

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.

\[
\Delta, x : A, \Delta \setminus \Delta, \square, \Delta' \vdash x : A
\]

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

\[
\Delta, \Delta \setminus e_0 : A \rightarrow B, \Delta \setminus e_1 : A \\
\Delta \vdash e_0, e_1 : B
\]

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

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

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

3 ILL is a subset of the ordered system.

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:

\[
\frac{\Delta_I \supsetneq \Delta_O}{\Delta_I \supsetneq \Delta_O, \square}
\]

\[
\frac{\Delta_I \supsetneq \Delta_O}{\Delta_I, x:A \supsetneq \Delta_O, x:A}
\]

\[
\frac{\Delta_I \supsetneq \Delta_O}{\Delta_I, \square \supsetneq \Delta_O, \square}
\]

IO system derivation rules maintain the following property.

**Lemma 1.** \(\Delta_I \backslash \Delta_O \vdash e:A \text{ implies } \Delta_I \supsetneq \Delta_O\).

**Proof.** By induction on given derivation.

We also define context difference

\[
\frac{}{\Delta_I, \square \supsetneq \Delta_O, \square}
\]

\[
\frac{\Delta_I, x:A \supsetneq \Delta_O, x:A}{\Delta_I, \square \supsetneq \Delta_O, \square}
\]

\[
\frac{\Delta_I, x:A \supsetneq \Delta_O, \square}{(\Delta_I, \square, \square \supsetneq \Delta_O, \square)}
\]

and show the following two useful properties of \(\supsetneq\) and \(\sqsubseteq\).

**Lemma 2.**

\[
\Delta_0 \sqsubseteq \Delta_1 \text{ and } \Delta_1 \supsetneq \Delta_2 \text{ implies } \Delta_0 \sqsubseteq \Delta_2 = (\Delta_0 \sqsubseteq \Delta_1) \sqsubseteq (\Delta_1 \sqsubseteq \Delta_2).
\]

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

**Lemma 3.**

\[
\Delta_I - \Delta_O \supsetneq \Delta_I \supsetneq \Delta_1 \text{ implies } \exists \Delta_I, \Delta = \Delta_1 \text{ and } \Delta - \Delta_O = \Delta_1.
\]

**Proof.** By induction on the length of \(\Delta_I\).

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

**Theorem 1.**

\[
\Delta_I \backslash \Delta_O \vdash e:A \text{ implies } \Delta_I - \Delta_O \vdash e:A.
\]

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

**Theorem 2.**

\[
\Delta_I - \Delta_O \vdash e:A \text{ implies } \Delta_I \backslash \Delta_O \vdash e:A.
\]

**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; 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.

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

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

```haskell
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

```haskell
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 \(-o\) rule, or linear function. We’d like something of the form

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

so that we have linear functions represented by Haskell functions from `a` to `b`. How should the `?`s be filled in? The `Ivar` 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 `Ivar` rule would be

```haskell
type IVar 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`.

```haskell
class Consume (v::Nat)
  (i::[CtxElm])
  (o::[CtxElm])
  | v i -> o
```

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

```haskell
instance Consume True v x i (Box 'i)
instance Consume v i o
  => Consume False v x i (Elm x 'o)
```

```haskell
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 \( '\text{True} \), or \( b \) will be \( '\text{False} \). This form of type level equality which reflects unifiability is really necessary for our encoding to work since \( \text{Consume} \) needs to realize that \( v \) does not equal \( S \ v \); note that \( EQ \) is a modern rendering of the \( \text{TypeEq} \) and \( \text{TypeCast} \) type classes in \[11\].

We can now finish up our encoding of linear functions:

\[
\text{class LLC} \quad \text{(repl :: Nat -> [CtxEml] -> [CtxEml] -> * -> *) where}
\]
\[
\llam :: (LVar repl v a -> repl (S v) (Elm v ': i) (Box ': o) b )
\]
\[
\rightarrow repl v i o (a -> b)
\]
\[
(\ast) :: repl v i m (a -> b)
\]
\[
\rightarrow repl v m o a
\]
\[
\rightarrow repl v i o b
\]

Note that the type of \( \ast \) is a direct transcription of the \( \rightarrow_E \) rule.

Now we can try typing some example LLC terms

\[
*Main> :t llam $ \text{\{defn} $ llam $ \lambda f \rightarrow llam $ \lambda x \rightarrow f^x
\]
\[
\text{\{defn} $ llam $ \lambda f \rightarrow llam $ \lambda x \rightarrow f^x
\]

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

\[
*Main> :t llam $ \text{\{defn} $ llam $ \lambda f \rightarrow llam $ \lambda x \rightarrow f^x
\]
\[
\text{\{defn} $ llam $ \lambda f \rightarrow llam $ \lambda x \rightarrow f^x
\]

The \( \text{Consume} \) 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:

\[
\text{type Defn a} = \forall \text{repl} \ (v::\text{Nat}) \ (i::[\text{CtxEml}]) \ . \ \text{LLC repl} \Rightarrow \text{repl v i i a}
\]
\[
\text{defn} :: \text{Defn a} \rightarrow \text{Defn a}
\]
\[
\text{defn x} = x
\]

Similar considerations prevent us from using the concrete \( Z \) instead of \( v \) above.

Our first example still type checks

\[
*Main> :t \text{\{defn} $ llam $ \lambda f \rightarrow llam $ \lambda x \rightarrow f^x
\]
\[
\text{\{defn} $ llam $ \lambda f \rightarrow llam $ \lambda x \rightarrow f^x
\]

Now when we try our ill-typed example as a \( \text{Defn} \) term

\[
*Main> :t \text{\{defn} $ llam $ \lambda f \rightarrow llam $ \lambda x \rightarrow f^x
\]
\[
\text{\{defn} $ llam $ \lambda f \rightarrow llam $ \lambda x \rightarrow f^x
\]

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:

\[
*Main> :t \text{\{defn} $ llam $ \lambda f \rightarrow llam $ \lambda x \rightarrow f
\]
\[
\text{\{defn} $ llam $ \lambda f \rightarrow llam $ \lambda x \rightarrow f
\]

\(^6\)See http://ghc.haskell.org/trac/ghc/ticket/9918 for an in depth discussion.
Lemma 4. For completeness, we state the two context intersection in the presence of two output contexts to get a conclusion output which accurately reflects which variables have must been consumed. We define linear context intersection in the presence of \( T \) as follows:

\[
\Delta_0, x: A \rightarrow \bigoplus_{v_0, v_1} \Delta_1, x: A = (\Delta_0 \rightarrow \bigoplus_{v_0, v_1} \Delta_1), x: A
\]

Since a \( T \) in either premise of the \(-\bigoplus\) rule can consume a formula, the conclusion \( T \) is the disjunction of those in premises:

\[
\Delta_0 \rightarrow \bigoplus_{v_0, v_1} e_0: A \rightarrow B, \quad \Delta_1 \rightarrow \bigoplus_{v_0, v_1} e_1: A \rightarrow B
\]

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

\[
\Delta_0 \rightarrow \bigoplus_{v_0, v_1} e_0: A \quad \Delta_1 \rightarrow \bigoplus_{v_0, v_1} e_1: B \\
\Delta_0 \& \Delta_1 \rightarrow \bigoplus_{v_0, v_1} e_0 \& e_1: A \& B
\]

We will now prove the correctness of the IO-\( T \) system with respect to the IO system. We start with a useful property of \( \bigoplus_{v_0, v_1} \):

**Lemma 4.**

1. \( \Delta_0 \not\rightarrow \bigoplus_{v_0, v_1} \Delta_1 = \Delta_0 \) and \( \Delta_0 \not\rightarrow \bigoplus_{v_0, v_1} \Delta_1 = \Delta_1 \)
2. \( \Delta_0 \supseteq (\Delta_0 \rightarrow \bigoplus_{v_0, v_1} \Delta_1) \) and \( \Delta_1 \supseteq (\Delta_0 \rightarrow \bigoplus_{v_0, v_1} \Delta_1) \)

We can now state and prove the correctness of the IO-\( T \) system in two parts as follows.

**Theorem 3.**

\( \Delta_0 \rightarrow \bigoplus_{v_0, v_1} e: A \rightarrow B \) implies \( \Delta_0 \rightarrow \bigoplus_{v_0, v_1} e: A \)

\( \Delta_0 \rightarrow \bigoplus_{v_0, v_1} e: A \) and \( \Delta_1 \supseteq (\Delta_0 \rightarrow \bigoplus_{v_0, v_1} \Delta_1) \)

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

**Theorem 4.**

\( \Delta_0 \rightarrow \bigoplus_{v_0, v_1} e: A \rightarrow B \) implies \( \Delta_0 \rightarrow \bigoplus_{v_0, v_1} e: A \)

We remark that an IO-\( T \) derivation with the \( T \) 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-\( T \) system by changing the \textit{ivar} rule to set the \( T \) flag to \( t \):

\[
\Delta, x: A, \Delta \rightarrow \bigoplus_{v_0, v_1} \Delta_0, \Delta_1 \rightarrow x: A
\]

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-\( T \) system by simply transcribing the extension from IO to IO-\( T \) judgments and rules. We start by encoding the supporting machinery we will need.

```haskell
class VarOk (tf :: CtxtElm) (v :: CtxtElm) | x y -> z
instance Or True y True
instance Or False y False
instance And (x :: Bool) (y :: 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 h2 v1 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 \( v \rightarrow \bigoplus_{v_0, v_1} \) definition into one MrgL instance.

In order to capture the two \(-\bigoplus\) rules, we will use the following relation

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

which specifies the valid relations between the \( T \) flag and the newly introduced linear variable in the output context. VarOk will be a constraint on the \textit{lam} method which lets us collapse the two \(-\bigoplus\) rules into one.

We can now write out the IO-\( T \) 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 \( T \) really do correspond to Haskell pairs and unit. We next extend our \textit{repr} type with a \( T \) flag

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

We change the \textit{LVar} definition to reflect that IO-\( T \) \textit{ivar} rule:

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

Now we add methods to \textit{LLC} to represent the IO-\( T \) derivation rules:

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

```
lvar : repr v a => 
```
where

\[ \text{llam :: VarOk tf var} \]
\[ \Rightarrow (\text{LVar repr v a}) \]
\[ \Rightarrow (\text{repr (S v) tf (Elm v 'i) (var 'o) b}) \]
\[ \Rightarrow \text{repr v tf i o (a -<> b}) \]

\[ (\cdot) :: (\text{Or tf0 tf1 tf}) \]
\[ \Rightarrow \text{repr v tf0 i m (a -<> b}) \]
\[ \Rightarrow \text{repr v tf1 m o a} \]
\[ \Rightarrow \text{repr v tf i o b} \]

\[ \text{top :: repr v True i i Top} \]

\[ (\&) :: (\text{MrgL h0 tf0 h1 tf1 o}) \]
\[ \Rightarrow \text{repr v tf0 i h0 a} \]
\[ \Rightarrow \text{repr v tf1 i h1 b} \]
\[ \Rightarrow \text{repr v tf i o (a & b}) \]

Finally we modify Defn to apply to IO-⊤ judgments:

\[ \text{type Defn tf a = forall repr (v::Nat) (i::[CtxElm]) . LLC repr} \Rightarrow \text{repr v tf i i a} \]
\[ \text{defn :: Defn tf a \rightarrow Defn tf a} \]
\[ \text{defn x = x} \]

Now we can try to type an additive LLC term:

\[ {\text{Main}} :{\text{defn \& \& (f ^ x ^ y) \& (f ^ y ^ x)}} \]
\[ {\text{Main}} :{\text{defn \& \& (f ^ x ^ y) \& (f ^ y ^ x)}} \]
\[ {\text{Main}} :{\text{top \& \& (f ^ x ^ y) \& (f ^ y ^ x)}} \]

\[ \text{type MrgLn i = ( MrgL i False i False i}) \]
\[ \Rightarrow \text{repr v \& \& (a & b}) \]
\[ \Rightarrow \text{repr v \& \& (a & b}) \]
\[ \Rightarrow \text{repr v \& \& (a & b}) \]

\[ \text{Finally we modify Defn to apply to IO-⊤ judgments:} \]
\[ \text{type Defn tf a = forall repr (v::Nat) (i::[CtxElm]) . LLC repr} \Rightarrow \text{repr v \& \& (a & b}) \]
\[ \Rightarrow \text{repr v \& \& (a & b}) \]
\[ \Rightarrow \text{repr v \& \& (a & b}) \]

\[ \text{defn :: Defn tf a \rightarrow Defn tf a} \]
\[ \text{defn x = x} \]

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

\[ \text{type MrgLs i = ( MrgL i False i False i}) \]
\[ \Rightarrow \text{repr v \& \& (a & b}) \]
\[ \Rightarrow \text{repr v \& \& (a & b}) \]
\[ \Rightarrow \text{repr v \& \& (a & b}) \]

\[ \text{Finally we modify Defn to apply to IO-⊤ judgments:} \]
\[ \text{defn :: Defn tf a \rightarrow Defn tf a} \]
\[ \text{defn x = x} \]

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

\[ {\text{Main}} :{\text{defn \& \& (f ^ x ^ y) \& (f ^ y ^ x)}} \]
\[ {\text{Main}} :{\text{defn \& \& (f ^ x ^ y) \& (f ^ y ^ x)}} \]
\[ {\text{Main}} :{\text{defn \& \& (f ^ x ^ y) \& (f ^ y ^ x)}} \]

\[ \text{We get the expected type (although ghci inlines the MrgLs constraint definition). The following ill-typed example} \]
\[ {\text{Main}} :{\text{defn \& \& (f ^ x ^ y) \& (f ^ y ^ x)}} \]
\[ {\text{Main}} :{\text{defn \& \& (f ^ x ^ y) \& (f ^ y ^ x)}} \]

\[ \text{behaves as expected}^{9}; \text{ so too does a well-typed example with ⊤:} \]

\[ {\text{Main}} :{\text{defn \& \& (f ^ x ^ y) \& (f ^ y ^ x)}} \]
\[ {\text{Main}} :{\text{defn \& \& (f ^ x ^ y) \& (f ^ y ^ x)}} \]

\[ \text{6. Unrestricted Functions} \]

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

\[ ^{8} \text{Again we have left out the location information.} \]

\[ ^{9} \text{This is actually the last of three errors, the other two arise from this one and have to do with VarOk not having enough information since MrgL fails.} \]
separate variable context; thus our typing judgments now have the form:

\[ \Gamma; \Delta \vdash e : A \]

where is \( \Gamma \) is the unrestricted variable context.

The unrestricted function type adds three new typing derivation rules:

\[
\begin{align*}
\Gamma_1, x : A; \Gamma_2; \vdash x : A & \quad \text{unvar} \\
\Gamma_1; \Delta \vdash e_0 : A \rightarrow B & \vdash \lambda x. e : A \rightarrow B \rightarrow I \\
\Gamma_1; \Delta \vdash e_0 : A \rightarrow B & \vdash e_1 : A \rightarrow E
\end{align*}
\]

Note the linear context must be empty in the unvar rule and, correspondingly, in the minor premise of the \( \rightarrow E \) 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 \( \rightarrow \) 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:

\[
\Gamma; \Delta; \Delta_0 \vdash_v e : A
\]

and here are the three \( \rightarrow \) IO-\( \top \) rules:

\[
\begin{align*}
\Gamma_1, x : A; \Gamma_2; \Delta; \Delta_0 & \vdash x : A \quad \text{unvar} \\
\Gamma_1; \Delta; \Delta_0 \vdash_v e_0 : A \rightarrow B & \vdash \lambda x. e : A \rightarrow B \rightarrow I \\
\Gamma_1; \Delta; \Delta_0 \vdash_v e_0 : A \rightarrow B & \vdash e_1 : A \rightarrow E
\end{align*}
\]

The \( \rightarrow E \) enforces an empty linear context in the minor premise by passing in an empty context; it would also be correct to use the linear context must be empty in the unvar rule and, correspondingly, in the minor premise of the \( \rightarrow E \) 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 \( \rightarrow \) 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:

\[
\Gamma; \Delta; \Delta_0 \vdash_v e : A
\]

and here are the three \( \rightarrow \) IO-\( \top \) rules:

\[
\begin{align*}
\Gamma_1, x : A; \Gamma_2; \Delta; \Delta_0 & \vdash x : A \quad \text{unvar} \\
\Gamma_1; \Delta; \Delta_0 \vdash_v e_0 : A \rightarrow B & \vdash \lambda x. e : A \rightarrow B \rightarrow I \\
\Gamma_1; \Delta; \Delta_0 \vdash_v e_0 : A \rightarrow B & \vdash e_1 : A \rightarrow E
\end{align*}
\]

The unrestricted function type adds three new typing derivation

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 :: [CtzElm]) .
  repr v False i i a

class LLC
  (repr :: Nat -> Bool
    -> [CtzElm] -> [CtzElm] -> * -> *)
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

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 -> ulam $ \ x -> f $$ x

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

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

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

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

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

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

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

Figure 2. IO-\( \top \) derivation rules with unrestricted functions.
linear lambda calculus

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.

def newtype Ev (v::Nat)
  tff:Bool
  [CtxElm]
  [CtxElm]
  
  a
  = Ev {ev :: a}

instance LLC
  (Ev :: Nat -> Bool
   -> [CtxElm] -> [CtxElm] -> * -> *)

  where
  lla x = Ev $ Lolli $ \x -> ev (f (Ev x))
  f x = Ev $ unLolli (ev f) (ev x)

  ula x = 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.

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

The results of eval really are terms:

*Main> :
*Main| putStrLn (\x -> x) "hello"
*Main> :

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. lla (\x -> x) :: Defn False (a -> a) would be accepted.

Related work

Kiselyov presents a tagless final encoding of linear and unrestricted lambdas in [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 --<> 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.

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 concrete 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.

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
A. Full Linear Lambda Calculus

Types

\[ A ::= A \rightarrow A \quad \text{linear functions} \]
\[ A \rightarrow A \quad \text{unrestricted functions} \]
\[ !A \quad \text{unrestricted modality} \]
\[ T \quad \text{additive unit} \]
\[ A \& A \quad \text{additive conjunction} \]
\[ 1 \quad \text{multiplicative unit} \]
\[ A \otimes A \quad \text{multiplicative conjunction} \]
\[ 0 \quad \text{additive zero} \]
\[ A \oplus A \quad \text{additive disjunction} \]

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

Terms

\[ e ::= x \quad \text{variables} \]
\[ \lambda x. e \mid e_0 \cdot e_1 \quad \text{linear functions} \]
\[ !e \mid \text{let}\; !x = e_0 \; \text{in} \; e_1 \quad \text{unrestricted modality} \]
\[ () \quad \text{additive unit} \]
\[ (e_0, e_1) \mid \text{fst}\; e \mid \text{snd}\; e \quad \text{additive conjunction} \]
\[ () \mid \text{let}\; () = e_0 \; \text{in} \; e_1 \quad \text{multiplicative unit} \]
\[ (e_0, e_1) \mid \text{fst}\; e \quad \text{multiplicative conjunction} \]
\[ \text{let}\; x \otimes y = e_0 \; \text{in} \; e_1 \quad \text{additive zero} \]
\[ \text{add}^A e \quad \text{additive zero} \]
\[ \text{in}^A e \mid \text{in}^A e \quad \text{additive disjunction} \]
\[ \text{case}\; e_0\; \text{of}\; \text{in}\; x \Rightarrow e_1 \; \text{in}\; x \Rightarrow e_2 \quad \text{additive disjunction} \]

Variable Contexts

\[ \Gamma ::= \cdot \mid \Gamma, x : A \]

We use \( \Gamma, \Delta \) to stand for contexts. We will sometimes overload \( \cdot \), to mean list append as well as list cons (as above); i.e. \( \Gamma, x : A, \Gamma_2 \) denotes a list which contains \( x \) to the right of everything in \( \Gamma \) and to the left of everything in \( \Gamma_2 \).

Non-deterministic Context Merge

\[ \Delta, x : A \bowtie \Delta' = (\Delta \bowtie \Delta'), x : A \]
\[ \Delta \bowtie \Delta', x' : A' = (\Delta \bowtie \Delta'), x' : A' \]

Typing Judgement

\[ \Gamma ; \Delta \vdash e : A \]

Typing Derivations

\[ \Gamma ; x : A \vdash x : A_{\text{var}} \]
\[ \Gamma ; \Delta, x : A \vdash e : B \rightarrow_I \]
\[ \Gamma ; \Delta \vdash !x, e : A \rightarrow B \rightarrow_I \]
\[ \Gamma ; \Delta_0 \bowtie \Delta_1 \vdash e_0 \cdot e_1 : B \rightarrow_E \]
\[ \Gamma_1, x : A, \Gamma_2 ; \vdash x : A_{\text{var}} \]
\[ \Gamma ; \Delta \vdash e : B \rightarrow_I \]
\[ \Gamma ; \Delta_0 \vdash e_0 : A \rightarrow B \rightarrow_E \]
\[ \Gamma ; \Delta_1 \vdash e_1 : A \rightarrow E \]
\[ \Gamma ; \Delta_0 \bowtie \Delta_1 \vdash \text{let}\; !x = e_0 \; \text{in} \; e_1 : B \rightarrow_E \]
### B. IO-⊥ Derivations for Full Linear Lambda Calculus

#### Variable Contexts

\[ Γ ::= · | Γ, x : A | Γ, □ \]

We use \( Γ, Δ \) to stand for contexts. We will overload \( \rightarrow \), to mean list append as well as list cons (as above); i.e. \( Δ, x : A, Δ_{2} \) 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

\[ Γ; Δ ⊢ e : A \]

#### Typing Derivations

\[ \Gamma; \Delta \vdash e : A; \Gamma ; Δ \vdash \lambda x . e : A \rightarrow B \]

\[ \Gamma; Δ \vdash e : A; \Gamma ; Δ \vdash \text{let } x = e_0 \text{ in } e_1 : B \]

---

**C. Full Linear Lambda Calculus in Haskell**

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

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

--
-- Linear types
```

---

**linear lambda calculus**

10

2015/7/21
newtype a -<> b = Lolli (unLolli :: a -> b)
newtype Bang a = Bang (unBang :: a)
type Top = ()
type a & b = (a, b)
data One = One
ndata a * b = Tensor a b
data a + b = Inl a | Inr b

t-- linear variable vid in Haskell context

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

t-- unrestricted variable in Haskell context

type UVar repr a = 
  forall (vid::Nat)
  (i::[Maybe Nat])
  . repr vid False i i a

t-- 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)

  ($$) :: repr vid tf0 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)

  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 tf 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

(*): repr vid 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)

linear lambda calculus
\[
\begin{align*}
t_{f1} & \quad (\text{Just } \text{vid } ': \ h) \\
& \quad (\text{var1 } ': \ o1) \\
& \quad c \\
\end{align*}
\]

\[
\begin{align*}
t_{f2} & \quad (\text{Just } \text{vid } ': \ h) \\
& \quad (\text{var2 } ': \ o2) \\
& \quad c \\
\end{align*}
\]

\[
\begin{align*}
\text{-> (LVar repr vid b -> repr (S vid))} \\
\text{ -> repr vid tf i o c} \\
\text{abort } :: \text{ repr vid tf i o Zero} \\
\text{ -> repr vid True i o a} \\
\end{align*}
\]

---

-- 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])
  | b v i -> o

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

instance Consumer True v x i (Nothing ': v)
instance (Consume v i o)
  => Consumer 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 | v2 | h1]
instance (MrgL h1 v1 h2 v2 h)
  => MrgL (x ': h1)
instance (x ': h2)
instance (x ': h)

instance (MrgL h1 True h2 v2 h)
  => MrgL (Just x ': h1)
instance True
instance (Nothing ': h2)
instance v2
instance (Nothing ': h)

instance (MrgL h1 v1 h2 True h)
  => MrgL (Nothing ': h1)
instance v1
instance (Just x ': h2)
instance True
instance (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

linear lambda calculus