# Monadic Typed Lambda Calculus Interpreter

Perry Alexander
ITTC - The University of Kansas
2335 Irving Hill Rd
Lawrence, KS 66045
alex@ittc.ku.edu

August 12, 2004

## 1 Introduction

The objective of this project is to write an interpreter for an extended simply typed lambda calculus ($\lambda_{\rightarrow}$) based on definitions from *Types and Programming Languages* [1], Chapter 8, Figure 8-1 and Chapter 9, Figure 9-1. We will enhance the basic language to include integers and integer sum and difference in addition to the basic operations. The definition of the abstract syntax provides the following forms for $\lambda_{\rightarrow}$ terms, values and types in:

$$\begin{array}{rcl} t & ::= & x \mid v \mid \lambda x:T.t \mid t\ t \mid \texttt{plus}\ t\ t \mid \texttt{sub}\ t\ t \\ v & ::= & \lambda x:T.t \mid \mathcal{I} \mid \texttt{true} \mid \texttt{false} \\ T & ::= & \texttt{Bool} \mid \texttt{Int} \mid T \rightarrow T \end{array}$$

The definition for call-by-value evaluation provides the following evaluation rules that will define the evaluation function:

$$\frac{t_1 \longrightarrow t_1^{'}}{t_1 t_2 \longrightarrow t_1^{'} t_2}\ \textsc{E-App1}$$

$$\frac{t_2 \longrightarrow t_2^{'}}{t_1 t_2 \longrightarrow t_1 t_2^{'}}\ \textsc{E-App2}$$

$$\frac{}{(\lambda x:T.t_{12})v_2 \longrightarrow [x \rightarrow v_2]t_{12}}\ \textsc{E-AppAbs}$$

$$\frac{\texttt{if true then}\ t_2\ \texttt{else}\ t_3}{t_2}\ \textsc{E-IfTrue}$$

$$\frac{\texttt{if false then}\ t_2\ \texttt{else}\ t_3}{t_3}\ \textsc{E-IfFalse}$$

$$\frac{t_1 \rightarrow t_1^{'}}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \rightarrow \texttt{if } t_1^{'} \texttt{ then } t_2 \texttt{ else } t_3} \text{ E-IF}$$

$$\frac{t_1 \rightarrow t_1^{'} \quad t_2 \rightarrow t_2^{'}}{\texttt{plus } t_1^{'} \ t_2^{'}} \text{ E-PLUS1}$$

$$\frac{\texttt{plus } \mathcal{I}_1 \ \mathcal{I}_2}{\mathcal{I}_1 + \mathcal{I}_2} \text{ E-PLUS2}$$

$$\frac{t_1 \rightarrow t_1^{'} \quad t_2 \rightarrow t_2^{'}}{\texttt{sub } t_1^{'} \ t_2^{'}} \text{ E-MINUS1}$$

$$\frac{\texttt{sub } \mathcal{I}_1 \ \mathcal{I}_2}{\mathcal{I}_1 - \mathcal{I}_2} \text{ E-MINUS2}$$

where $\mathcal{I}$ is any constant integer value.

The following typing rules that define the type inference function:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ T-VAR}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \rightarrow T_2} \text{ T-ABS}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \ t_2 : T_{12}} \text{ T-APP}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T} \text{ T-IF}$$

$$\frac{}{\texttt{true : Bool}} \text{ T-TRUE}$$

$$\frac{}{\texttt{false : Bool}} \text{ T-FALSE}$$

$$\frac{t_1 : \texttt{Int} \quad t_2 : \texttt{Int}}{\texttt{plus } t_1 \ t_2 : \texttt{Int}} \text{ T-PLUS}$$

$$\frac{t_1 : \texttt{Int} \quad t_2 : \texttt{Int}}{\texttt{sub } t_1 \ t_2 : \texttt{Int}} \text{ T-MINUS}$$

Our objective is to: (i) define a data structure for representing $\lambda_{\rightarrow}$ terms embodying the abstract syntax; (ii) a type derivation function for $\lambda_{\rightarrow}$ terms embodying the type rules; and (iii) an evaluation function for $\lambda_{\rightarrow}$ terms embodying the evaluation rules.

## 2 Abstract Syntax

**module** *TypedLambdaAST*
  **where**

**import** *LangUtils*

### 2.1 Type Language

**data** $TyBase\ ty = TyBool \mid TyInt$ **deriving** $(Eq, Show)$

**data** $TyAbs\ ty = ty :->: ty$ **deriving** $(Eq, Show)$

**type** $TyLangSum = (Sum\ TyBase\ TyAbs)$

**type** $TyLang = Rec\ TyLangSum$

**instance** $Eq\ TyLang$ **where**
  $x \equiv y = (unS\ (out\ x)) \equiv (unS\ (out\ y))$

### 2.2 Term Language

The term language include Boolean values and integer values, addition and subtraction operators, if-then-else expressions, and lambda expressions and lambda application.

**data** $TmBool\ te = TmTrue \mid TmFalse$ **deriving** $(Eq, Show)$

**instance** $Functor\ TmBool$ **where**
  $map_f\ f\ TmTrue = TmTrue$
  $map_f\ f\ TmFalse = TmFalse$

**data** $TmInt\ te = TmConstInt\ Int$ **deriving** $(Eq, Show)$

**instance** $Functor\ TmInt$ **where**
  $map_f\ f\ (TmConstInt\ x) = (TmConstInt\ x)$

**data** $TmOp\ te = TmAdd\ te\ te \mid TmSub\ te\ te$ **deriving** $(Eq, Show)$

**instance** $Functor\ TmOp$ **where**
  $map_f\ f\ (TmAdd\ x\ y) = (TmAdd\ (f\ x)\ (f\ y))$
  $map_f\ f\ (TmSub\ x\ y) = (TmSub\ (f\ x)\ (f\ y))$

**data** $TmIf\ te = If\ te\ te\ te$ **deriving** $(Eq, Show)$

**instance** $Functor\ TmIf$ **where**
  $map_f\ f\ (If\ c\ t\ e) = (If\ (f\ c)\ (f\ t)\ (f\ e))$

**data** $TmVar\ t = TmVar\ String$ **deriving** $(Show, Eq)$

```
instance Functor TmVar where
    map_f f (TmVar x) = (TmVar x)

data TmFn t = TmLambda String TyLang t
            | TmApp t t
            deriving (Eq)

instance Functor TmFn where
    map_f f (TmLambda s ty te) = (TmLambda s ty (f te))
    map_f f (TmApp te1 te2) = (TmApp (f te1) (f te2))

type TmLangSum = (Sum TmBool
                  (Sum TmInt
                   (Sum TmOp
                    (Sum TmIf
                     (Sum TmVar TmFn)))))

type TmLang = Rec TmLangSum


toTmLang :: (Subsum f TmLangSum) => f TmLang -> TmLang
toTmLang = toSum
```

# 3 Environment

This very simple module defines a standard environment parameterized over a stored type. It is used to define both $\Gamma$ for the type checking routine and the environment for the evaluation routine.

```
module TypedLambdaEnv where

type Environment a = [(String, a)]

lookupEnv :: (Eq a) => String -> (Environment a) -> (Maybe a)
lookupEnv s e = lookup s e
```

# 4 Type Checking

## 4.1 Type Values

These are the type values available in our language. For the type language, this will serve as the carrier set or value space for both the type langauge and the term language under type checking. $\phi$ for the type language is defined over $Ty_{\mathcal{D}}\ a$ while $\phi$ for the term language type checker is defined over $T_{n-1}\ a$. In effect, $\phi$ evaluates the term language to a type value rather than a term value.

```
module TypedLambdaTypesT where
  import LangUtils
  import TypedLambdaAST
  import TypedLambdaEnv
```

```
import Monad
import Control.Monad.Error
import Control.Monad.Reader
```

Note that values are not interpreted, so no *Algebra* is needed. Technically, we could make $\phi = id$ for values, but it's not necessary to think about this right now.

#### 4.1.1 Boolean and Integer Type Value

```
data TyBaseVal ty = TyBoolVal | TyIntVal deriving (Eq, Show)

instance Functor TyBaseVal where
    fmap f TyBoolVal = TyBoolVal
    fmap f TyIntVal = TyIntVal
```

#### 4.1.2 Abstraction Type Value

```
data TyAbsVal ty = TyAbsVal ty ty deriving (Eq, Show)

instance Functor TyAbsVal where
    fmap f (TyAbsVal x y) = TyAbsVal (f x) (f y)
```

#### 4.1.3 Type Value

The value space sum for types is the sum of the base values (integer and boolean) and the abstraction value and is called *TyValSum*. The set of type values is the fixed point, *TyVal*. *TyVal* is an instance of *Show* and *Eq* to allow printing and comparing values. *toTyVal* injects elements from *TyVal* components into the value space.

```
type TyValSum = (Sum TyBaseVal TyAbsVal)

instance (Show (f a), Show (g a)) => Show (Sum f g a) where
    show (S (Prelude.Left x)) = ("(Left " ++ (show x) ++ ")")
    show (S (Prelude.Right x)) = ("(Right " ++ (show x) ++ ")")

type TyVal = Rec TyValSum

instance Show TyVal where
    show x = show (out x)

instance Eq TyVal where
    x == y = (unS (out x)) == (unS (out y))

toTyVal :: (Subsum f TyValSum) => f TyVal -> TyVal
toTyVal = toSum
```

## 4.2 The Reader Error Monad

The monad used for handling the environment and error messages will be formed by composing a *Reader* with and *ErrorMonad*. First we define the error handling aspects, then embed the *ErrorMonad* in a *Reader* using *ReaderT*.

The *Either* type constructor is already an instance of the *MonadError* class. Thus, it is not necessary to define *throwError* and *catchError* explicitly for the type. The definitions are included here for documentation, but are not loaded.

**instance** *MonadError* (*Either e*) **where**
  *throwError* = *Left*
  *catchError* (*Left e*) *handler* = *handler e*
  *catchError a* _ = *a*

*TyError* is a simple data type for storing errors. We could simply store the error string rather than create a type. However, *TyError* serves as a placeholder if we want to do fancier things later. *TyError* is also an instance of the standard *Error*.

**data** *TyError* = *Err String* **deriving** (*Show*, *Eq*)

**instance** *Error TyError* **where**
  *noMsg* = *Err* `"Type Error"`
  *strMsg s* = *Err s*

$\Gamma$ defines the data structure used for a binding list. It is simply a list of (*String*, *TyVal*) pairs. Adding a binding appends it to the front of a binding list and looking up a binding is handled in the canonical fashion.

**type** $\Gamma$ = *Environment TyVal*

$addBinding :: \Gamma \to (String, TyVal) \to \Gamma$
$addBinding\ g\ t = (t : g)$

$lookupGamma :: String \to \Gamma \to Maybe\ TyVal$
$lookupGamma = lookup$

*TyMonad* defines the actual monad used by the type checker. The signature of *TyMonad* is a bit odd. It must be a type constructor and thus must have one argument. *ReaderT* is applied to a $\Gamma$ and (*Either TyError*) leaving the last argument to *TyError* as an argument to *TyMonad*.

**type** *TyMonad* = *ReaderT* $\Gamma$ (*Either TyError*) *TyVal*

**instance** *Subtype TyError* (*Either TyError TyVal*) **where**
  $\uparrow x = (Left\ x)$
  $\downarrow (Left\ x) = Just\ x$
  $\downarrow (Right\ x) = Nothing$

**instance** *Subtype TyVal* (*Either TyError TyVal*) **where**
  $\uparrow x = (Right\ x)$
  $\downarrow (Right\ x) = Just\ x$
  $\downarrow (Left\ x) = Nothing$

## 4.3 Type Language

The type language defines the language for types over the type values. The type language will be $f$ and defined over the type value space serving as $a$ in an algebra definition.

### 4.3.1 Base Types

The Base Types represent integer and boolean atomic types.

$$\begin{array}{l}
\textbf{instance}\ \mathit{Functor}\ \mathit{TyBase}\ \textbf{where} \\
\quad \mathit{map}_f\ f\ \mathit{TyBool} = \mathit{TyBool} \\
\quad \mathit{map}_f\ f\ \mathit{TyInt} = \mathit{TyInt}
\end{array}$$

$$\begin{array}{l}
\textbf{instance}\ \mathit{Algebra}\ \mathit{TyBase}\ \mathit{TyMonad}\ \textbf{where} \\
\quad \phi\ \mathit{TyBool} = \mathit{return}\ \$ \uparrow \$\ \mathit{toTyVal}\ \mathit{TyBoolVal} \\
\quad \phi\ \mathit{TyInt} = \mathit{return}\ \$ \uparrow \$\ \mathit{toTyVal}\ \mathit{TyIntVal}
\end{array}$$

### 4.3.2 Abstraction Type

Typically thought of as a function type, the abstraction type represents a mapping from a range type to a domain type.

$$\begin{array}{l}
\textbf{instance}\ \mathit{Functor}\ \mathit{TyAbs}\ \textbf{where} \\
\quad \mathit{map}_f\ f\ (x : - >: y) = (f\ x) : - >: (f\ y)
\end{array}$$

$$\begin{array}{l}
\textbf{instance}\ \mathit{Algebra}\ \mathit{TyAbs}\ \mathit{TyMonad}\ \textbf{where} \\
\quad \phi\ (x : - >: y) = \textbf{do}\ \{ x' \leftarrow x \\
\qquad\qquad\qquad\qquad\quad ; y' \leftarrow y \\
\qquad\qquad\qquad\qquad\quad ; \mathit{return}\ \$ \uparrow \$\ \mathit{toTyVal}\ (\mathit{TyAbsVal}\ x'\ y') \\
\qquad\qquad\qquad\qquad\quad \}
\end{array}$$

Define a utility function for converting a type term into the type language. The *evalTy* function is a separate function for evaluating elements of the type language.

$$\begin{array}{l}
\mathit{toTyLang} :: (\mathit{Subsum}\ f\ \mathit{TyLangSum}) \Rightarrow f\ \mathit{TyLang} \rightarrow \mathit{TyLang} \\
\mathit{toTyLang} = \mathit{toSum}
\end{array}$$

$$\begin{array}{l}
\mathit{evalTy} :: \mathit{TyLang} \rightarrow \mathit{TyMonad} \\
\mathit{evalTy} = \mathit{cata}
\end{array}$$

## 4.4 Type Checking Functions

The type checking functions are defined by defining an algebra from *TmLang* to *TyMonad*. Thus, *TyMonad* is the carrier set for the *TmLang* algebra and $\phi$ defines the evaluation function.

$$\begin{array}{l}
\textbf{instance}\ \mathit{Algebra}\ \mathit{TmBool}\ \mathit{TyMonad}\ \textbf{where} \\
\quad \phi\ \mathit{TmTrue} = \mathit{return}\ \$ \uparrow \$\ \mathit{toTyVal}\ \mathit{TyBoolVal} \\
\quad \phi\ \mathit{TmFalse} = \mathit{return}\ \$ \uparrow \$\ \mathit{toTyVal}\ \mathit{TyBoolVal}
\end{array}$$

```
instance Algebra TmInt TyMonad where
    φ (TmConstInt x) = return $ ↑ $ toTyVal TyIntVal

instance Algebra TmOp TyMonad where
    φ (TmAdd x y) = do { x′ ← x
                       ; y′ ← y
                       ; if (x′ ≡ (toTyVal TyIntVal) ∧
                             y′ ≡ (toTyVal TyIntVal))
                          then return $ ↑ $ toTyVal TyIntVal
                          else throwError $ Err "Argument to Add not Integer"
                       }
    φ (TmSub x y) = do { x′ ← x
                       ; y′ ← y
                       ; if (x′ ≡ (toTyVal TyIntVal) ∧
                             y′ ≡ (toTyVal TyIntVal))
                          then return $ ↑ $ toTyVal TyIntVal
                          else throwError $ Err "Argument to Sub not Integer"
                       }

instance Algebra TmIf TyMonad where
    φ (If c t e) = do { c′ ← c
                      ; t′ ← t
                      ; e′ ← e
                      ; if (c′ ≡ (toTyVal TyBoolVal) ∧
                            t′ ≡ e′)
                         then return $ ↑ t′
                         else throwError $ Err "Either condition is not boolean or then and else are not of
                      }

instance Algebra TmVar TyMonad where
    φ (TmVar s) = do { val ← asks (lookupGamma s)
                     ; case val of
                        Just x → return x
                        Nothing → throwError $ Err ("Variable " ++ (s ++ " not found"))
                     }

instance Algebra TmFn TyMonad where
    φ (TmLambda s ty te) = do { γ ← ask
                              ; ty′ ← evalTy ty
                              ; te′ ← local (const (addBinding γ (s, ty′))) te
                              ; return $ ↑ $ toTyVal (TyAbsVal ty′ te′)
                              }

    φ (TmApp te1 te2) = do { te1′ ← te1
                           ; te2′ ← te2
                           ; checkLambda (out te1′) te2′
                           }

checkLambda l te2 = case (↓_S l) of
                      (Just (TyAbsVal tty tte)) → if tty ≡ te2
```

```
                                then return $ ↑ tte
                                else throwError $ Err "Actual parameter type does not match fo
                   _ → throwError $ Err "First argument to application must be a Lambda"
```

The basic $typeof_{\mathcal{D}}$ function is a catamorphism over the *TmLang TyMonad*. The signature is specified to explicitly identify types. The *runTypeof* function is a utilty function that evaluates the *Reader* monad. The initial environment is empty because there are no predefined symbols in our language. *runTypeof* should be used to integrate the type checker with other language elements.

```
typeof_D :: TmLang → TyMonad
typeof_D = cata

runTypeof t = (runReaderT (typeof_D t) [])
```

# 5 Evaluation

```
module TypedLambdaEval where

import LangUtils
import TypedLambdaEnv
import TypedLambdaAST
import Control.Monad.Reader
import Control.Monad.Error
```

## 5.1 Value Representation

There are three values associated with the Lambda language that all interpretable functions must converge to - booleans, integers, and lambda values. Together, these are specified in the *TmVal* constructed type. Note that this type is recursive, unlike the term language and type language specifications. The `Haskell` types used to represent primitive values are defined to be subtypes of the aggregate `TmVal` type. Thus, ↓ and ↑ are define between types.

```
data TmVal
    = TmBoolVal Bool
    | TmIntVal Int
    | LambdaVal (TmValEnv → TmValEnv)

instance Show TmVal where
    show (TmBoolVal x) = show x
    show (TmIntVal x) = show x
    show (LambdaVal x) = "<Lambda Value>"

instance Subtype Bool TmVal where
    ↑ x = (TmBoolVal x)
    ↓ (TmBoolVal x) = Just x
    ↓ (TmIntVal _) = Nothing
    ↓ (LambdaVal _) = Nothing
```

```
instance Subtype Int TmVal where
    ↑ x = (TmIntVal x)
    ↓ (TmBoolVal _) = Nothing
    ↓ (TmIntVal x) = Just x
    ↓ (LambdaVal _) = Nothing

instance Subtype (TmValEnv → TmValEnv) TmVal where
    ↑ x = (LambdaVal x)
    ↓ (TmBoolVal _) = Nothing
    ↓ (TmIntVal _) = Nothing
    ↓ (LambdaVal x) = Just x

type Env = Environment TmVal
```

## 5.2 The Evaluator Monad

The monad used to support evaluation is a composition of the *ErrorMonad* and the *Reader* monad with the *ErrorMondad* encapsulated by the *Reader*.

```
data TmError = Err String deriving (Show, Eq)

instance Error TmError where
    noMsg = Err "Type Error"
    strMsg s = Err s

type TmValEnv = ReaderT Env (Either TmError) TmVal
```

## 5.3 Expressions as Algebras

```
instance Algebra TmBool TmValEnv where
    φ TmTrue = return $ ↑ True
    φ TmFalse = return $ ↑ False

instance Algebra TmInt TmValEnv where
    φ (TmConstInt x) = return $ ↑ x

instance Algebra TmOp TmValEnv where
    φ (TmAdd x y) =
      do { x′ ← x
         ; y′ ← y
         ; case (↓ x′) of
           Just x″ → case (↓ y′) of
                     Just y″ → return $ ↑ ((x″ :: Int) + (y″ :: Int))
                     Nothing → error ((show y′) ++ " not an integer")
           Nothing → error ((show x′) ++ " not an integer")
         }

    φ (TmSub x y) =
      do { x′ ← x
         ; y′ ← y
```

```
        ; case (↓ x′) of
          Just x″ → case (↓ y′) of
                      Just y″ → return $ ↑ ((x″ :: Int) − (y″ :: Int))
                      Nothing → error ((show y′) ++ " not an integer")
          Nothing → error ((show x′) ++ " not an integer")
        }

instance Algebra TmIf TmValEnv where
  φ (If b t e) =
    do { b′ ← b
        ; case (↓ b′) of
          Just b″ → if b″ then t else e
          Nothing → error ((show b′) ++ " is not boolean")
        }

instance Algebra TmVar TmValEnv where
  φ (TmVar v) = do { val ← asks (lookup v)
                   ; case val of
                     Just x → return x
                     Nothing → error ("Variable " ++ (v ++ " not found"))
                   }

instance Algebra TmFn TmValEnv where
  φ (TmLambda s ty te) =
    do { env ← ask
        ; return $ ↑ $ (λv → do { v′ ← v
                                ; local (const ((s, v′) : env)) te
                                })
        }
  φ (TmApp te1 te2) =
    do { te1′ ← te1
        ; case (↓ te1′) of
          (Just (LambdaVal f)) → (f te2)
          a → error ((show a) ++ " is not a lambda value")
        }
```

The $eval_{\mathcal{D}}$ function generates a monad from a term language element. The monad is an *ErrorMonad* composed with a *Reader* monad, thus the result of applying *runReader* is either a value or an error message. *runEval* applies *runReaderT* to the *Reader* monad resulting from $eval_{\mathcal{D}}$ on an environment parameter. *execute* applies *runEval* with an empty environment.

```
eval_D :: TmLang → TmValEnv
eval_D = cata

runEval t e = (runReaderT (eval_D t) e)

execute t = runEval t []
```

## 6 Interpretation

Here the type checker and the evaluator are put together to form an interpreter.

```
module TypedLambdaInterpreter where

import LangUtils
import TypedLambdaEnv
import TypedLambdaAST
import TypedLambdaEval
import TypedLambdaTypesT
```

The *interpret* function is primarily a command line, testing function. It accepts a term and generates an *IO* monad representing either the error message or value generated by the evaluator. Most of the work here is simply getting the output in a reasonably well formatted form.

```
interpret :: TmLang → IO ()
interpret t = case (runTypeof t) of
                (Left (TypedLambdaTypesT.Err y)) →
                    do { putStr "Type Error: "
                       ; putStr y; putStr "\n"
                       }
                (Right y) → case (runEval t [ ]) of
                    (Left (TypedLambdaEval.Err z)) →
                        do { putStr "Runtime Error: "
                           ; putStr (show z)
                           ; putStr "\n"
                           }
                    (Right z) →
                        do { putStr "Value: "
                           ; putStr (show z)
                           ; putStr ":: "
                           ; putStr (show y)
                           ; putStr "\n"
                           }
```

# References


[1] B. Pierce. *Types and Programming Languages.* MIT Press, Cambridge, MA, 2002.