Monadic Subtyped Lambda Calculus Interpreter

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

August 17, 2004

1 Introduction

The objective of this project is to write an interpreter for System F ($\rightarrow \forall$) based on definitions from *Types and Programming Languages* [1], Chapter 23, Figure 23-1. We will enhance the basic language to include integers and integer sum and difference in addition to the basic operations.

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

2 Abstract Syntax

```
module SystemFAST
    where

import LangUtils
```

2.1 Type Language

```
data TyBase ty = TyBool | TyInt deriving (Eq, Show)

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

data TyTuple ty = TyTuple [ty] deriving (Eq, Show)

data TyVar ty = TyVar String deriving (Eq, Show)

    -- data TyAll ty = TyAll String ty

  type TyLangSum = (Sum TyBase (Sum TyAbs (Sum TyTuple TyVar)))

  type TyLang = Rec TyLangSum
```

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.

$$\textbf{data}\ \mathit{TmBool}\ \mathit{te} = \mathit{TmTrue} \mid \mathit{TmFalse}\ \textbf{deriving}\ (\mathit{Eq}, \mathit{Show})$$

$$\begin{array}{l} \textbf{instance}\ \mathit{Functor}\ \mathit{TmBool}\ \textbf{where} \\ \quad \mathit{map}_f\ f\ \mathit{TmTrue} = \mathit{TmTrue} \\ \quad \mathit{map}_f\ f\ \mathit{TmFalse} = \mathit{TmFalse} \end{array}$$

$$\textbf{data}\ \mathit{TmInt}\ \mathit{te} = \mathit{TmConstInt}\ \mathit{Int}\ \textbf{deriving}\ (\mathit{Eq}, \mathit{Show})$$

$$\begin{array}{l} \textbf{instance}\ \mathit{Functor}\ \mathit{TmInt}\ \textbf{where} \\ \quad \mathit{map}_f\ f\ (\mathit{TmConstInt}\ x) = (\mathit{TmConstInt}\ x) \end{array}$$

$$\begin{array}{l} \textbf{data}\ \mathit{TmTuple}\ \mathit{te} \\ \quad = \mathit{TmTuple}\ [\mathit{te}] \\ \quad \mid \mathit{TmPrj}\ \mathit{Int}\ \mathit{te} \\ \quad\ \ \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}) \end{array}$$

$$\begin{array}{l} \textbf{instance}\ \mathit{Functor}\ \mathit{TmTuple}\ \textbf{where} \\ \quad \mathit{map}_f\ f\ (\mathit{TmTuple}\ x) = \mathit{TmTuple}\ (\mathit{map}\ f\ x) \\ \quad \mathit{map}_f\ f\ (\mathit{TmPrj}\ x\ y) = \mathit{TmPrj}\ x\ (f\ y) \end{array}$$

$$\begin{array}{l} \textbf{data}\ \mathit{TmOp}\ \mathit{te} \\ \quad = \mathit{TmAdd}\ \mathit{te}\ \mathit{te} \\ \quad \mid \mathit{TmSub}\ \mathit{te}\ \mathit{te} \\ \quad \mid \mathit{TmMul}\ \mathit{te}\ \mathit{te} \\ \quad \mid \mathit{TmDiv}\ \mathit{te}\ \mathit{te} \\ \quad\ \ \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}) \end{array}$$

$$\begin{array}{l} \textbf{instance}\ \mathit{Functor}\ \mathit{TmOp}\ \textbf{where} \\ \quad \mathit{map}_f\ f\ (\mathit{TmAdd}\ x\ y) = (\mathit{TmAdd}\ (f\ x)\ (f\ y)) \\ \quad \mathit{map}_f\ f\ (\mathit{TmSub}\ x\ y) = (\mathit{TmSub}\ (f\ x)\ (f\ y)) \\ \quad \mathit{map}_f\ f\ (\mathit{TmMul}\ x\ y) = (\mathit{TmMul}\ (f\ x)\ (f\ y)) \\ \quad \mathit{map}_f\ f\ (\mathit{TmDiv}\ x\ y) = (\mathit{TmDiv}\ (f\ x)\ (f\ y)) \end{array}$$

$$\textbf{data}\ \mathit{TmIf}\ \mathit{te} = \mathit{If}\ \mathit{te}\ \mathit{te}\ \mathit{te}\ \textbf{deriving}\ (\mathit{Eq}, \mathit{Show})$$

$$\begin{array}{l} \textbf{instance}\ \mathit{Functor}\ \mathit{TmIf}\ \textbf{where} \\ \quad \mathit{map}_f\ f\ (\mathit{If}\ c\ t\ e) = (\mathit{If}\ (f\ c)\ (f\ t)\ (f\ e)) \end{array}$$

$$\begin{array}{l} \textbf{data}\ \mathit{TmVar}\ t = \mathit{TmVar}\ \mathit{String} \\ \qquad\qquad\quad \mid \mathit{TmTVar}\ \mathit{String} \\ \qquad\qquad\quad\ \ \textbf{deriving}\ (\mathit{Show}, \mathit{Eq}) \end{array}$$

$$\begin{array}{l} \textbf{instance}\ \mathit{Functor}\ \mathit{TmVar}\ \textbf{where} \\ \quad \mathit{map}_f\ f\ (\mathit{TmVar}\ x) = (\mathit{TmVar}\ x) \\ \quad \mathit{map}_f\ f\ (\mathit{TmTVar}\ x) = (\mathit{TmTVar}\ x) \end{array}$$

```
data TmFn t = TmLambda String TyLang t
            | TmApp t t
            | TmTLambda String t
            | TmTApp t TyLang

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))
    map_f f (TmTLambda s te) = (TmTLambda s (f te))
    map_f f (TmTApp te1 ty1) = (TmTApp (f te1) ty1)

type TmLangSum = (Sum TmBool
                   (Sum TmInt
                    (Sum TmOp
                     (Sum TmTuple
                      (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 Γ for the type checking routine and the environment for the evaluation routine.

```
module SystemFEnv 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. ϕ for the type language is defined over $Ty_{\mathcal{D}}\ a$ while ϕ for the term language type checker is defined over $T_{n-1}\ a$. In effect, ϕ evaluates the term language to a type value rather than a term value.

```
module SystemFTypesT where
  import LangUtils
  import SystemFAST
```

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

type TyMonFn = TyMonad → TyMonad

instance Show TyMonFn where
    show t = "<Monad Function>"

instance Eq TyMonFn where
    x ≡ y = False

data TyVal
    = TyAbsVal TyVal TyVal
    | TyBoolVal
    | TyIntVal
    | TyTupleVal [TyVal]
    | TyAllVal (TyMonad → TyMonad)
    | TyVarVal String
      deriving (Show, Eq)
```

This quite possibly dangerous, but we only use the $\leqslant$ operator during subtyping. It is not generally true that there is a order over type values, so the *Ord* class will never be completely satisfied by *TyVal*.

```
instance Ord TyVal where
    (TyTupleVal vs1) ⩽ (TyTupleVal vs2) = (vs2 ⩽ vs1)
    (TyAbsVal d1 r1) ⩽ (TyAbsVal d2 r2) = ((d2 ⩽ d1) ∧ (r1 ⩽ r2))
    TyBoolVal ⩽ TyBoolVal = True
    TyIntVal ⩽ TyIntVal = True
    _ ⩽ _ = False
```

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
```

Γ 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 Γ = Environment TyVal

addBinding :: Γ → (String, TyVal) → Γ
addBinding g t = (t : g)

lookupGamma :: String → Γ → 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 Γ and (*Either TyError*) leaving the last argument to *TyError* as an argument to *TyMonad*.

```
type TyMonad = ReaderT Γ (Either TyError) TyVal

instance Subtype TyError (Either TyError TyVal) where
    ↑ x = (Left x)
    ↓ (Left x) = Just x
    ↓ (Right x) = Nothing

instance Subtype TyVal (Either TyError TyVal) where
    ↑ x = (Right x)
    ↓ (Right x) = Just x
    ↓ (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.

```
instance Functor TyBase where
    map_f f TyBool = TyBool
    map_f f TyInt = TyInt

instance Algebra TyBase TyMonad where
    φ TyBool = return $ ↑ TyBoolVal
    φ TyInt = return $ ↑ TyIntVal

mkBool = toTyLang TyBool
mkInt = toTyLang TyInt
```

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.

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

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

$$\mathit{mkTyAbs}\ x\ y = \mathit{toTyLang}\ (x :->: y)$$

4.3.3 Tuple Type

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Functor}\ \mathit{TyTuple}\ \textbf{where} \\
\quad \mathit{map}_f\ f\ (\mathit{TyTuple}\ \mathit{ts}) = \mathit{TyTuple}\ (\mathit{map}\ f\ \mathit{ts})
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Algebra}\ \mathit{TyTuple}\ \mathit{TyMonad}\ \textbf{where} \\
\quad \phi\ (\mathit{TyTuple}\ s) = \textbf{do}\ \{ s' \leftarrow \mathit{sequence}\ s \\
\qquad\qquad\qquad\qquad ; \mathit{return}\ \$ \uparrow \$\ (\mathit{TyTupleVal}\ s') \\
\qquad\qquad\qquad\qquad \}
\end{array}
$$

$$\mathit{mkTupleTy}\ \mathit{ts} = \mathit{toTyLang}\ \$\ \mathit{TyTuple}\ \mathit{ts}$$

4.3.4 Type Variables

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Functor}\ \mathit{TyVar}\ \textbf{where} \\
\quad \mathit{map}_f\ f\ (\mathit{TyVar}\ x) = (\mathit{TyVar}\ x)
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Algebra}\ \mathit{TyVar}\ \mathit{TyMonad}\ \textbf{where} \\
\quad \phi\ (\mathit{TyVar}\ x) = \textbf{do}\ \{ \mathit{val} \leftarrow \mathit{asks}\ (\mathit{lookup}\ x) \\
\qquad\qquad\qquad ; \textbf{case}\ \mathit{val}\ \textbf{of} \\
\qquad\qquad\qquad\quad \mathit{Just}\ x \rightarrow \mathit{return}\ \$ \uparrow x \\
\qquad\qquad\qquad\quad \mathit{Nothing} \rightarrow \mathit{throwError}\ \$\ \mathit{Err}\ \texttt{"Type Variable not found"} \\
\qquad\qquad\qquad \}
\end{array}
$$

$$\mathit{mkTyVar}\ s = \mathit{toTyLang}\ \$\ \mathit{TyVar}\ s$$

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 ϕ defines the evaluation function.

```
instance Algebra TmBool TyMonad where
    φ TmTrue = return $ ↑ TyBoolVal
    φ TmFalse = return $ ↑ $ TyBoolVal

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

instance Algebra TmTuple TyMonad where
    φ (TmTuple xs) = do {xs′ ← sequence xs
                        ; return $ ↑ (TyTupleVal xs′)
                        }

    φ (TmPrj i t) = do {t′ ← t
                      ; case t′ of
                        (TyTupleVal tys) →
                          if ((i ⩾ 0) ∧ (i < (length tys)))
                          then return (↑ (tys !! i))
                          else throwError $ Err "Tuple index out of range"
                        _ → throwError $ Err "Project argument not a tuple type"
                      }

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

    φ (TmSub x y) = do {x′ ← x
                       ; y′ ← y
                       ; if (x′ ⩽ TyIntVal ∧
                              y′ ⩽ TyIntVal)
                         then return $ ↑ TyIntVal
                         else throwError $ Err "Argument to Sub not Integer"
                       }

    φ (TmMul x y) = do {x′ ← x
                       ; y′ ← y
                       ; if (x′ ⩽ TyIntVal ∧
                              y′ ⩽ TyIntVal)
                         then return $ ↑ TyIntVal
                         else throwError $ Err "Argument to Mul not Integer"
                       }
    φ (TmDiv x y) = do {x′ ← x
                       ; y′ ← y
```

```
                          ; if (x' <= TyIntVal &&
                                y' <= TyIntVal)
                            then return $ ↑ TyIntVal
                            else throwError $ Err "Argument to Div not Integer"
                          }


instance Algebra TmIf TyMonad where
    φ (If c t e) = do { c' <- c
                      ; t' <- t
                      ; e' <- e
                      ; if (c' == 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 $ ↑ (TyAbsVal ty' te')
                              }

    φ (TmApp te1 te2) = do { te1' <- te1
                           ; te2' <- te2
                           ; checkLambda te1' te2'
                           }
        where checkLambda l te2 =
                case l of
                    (TyAbsVal tty tte) -> if tty <= te2
                                          then return $ ↑ tte
                                          else throwError $ Err "Actual parameter type is not a subt
                    _ -> throwError $ Err "First argument to application must be a Lambda"

    φ (TmTLambda s te) = do { γ <- ask
                            ; return $ ↑ $ TyAllVal
                              (λtv -> do { tv' <- tv
                                         ; local (const (addBinding γ (s, tv'))) te
                                         })
                            }

    φ (TmTApp te ty) = do { te' <- te
                          ; case te' of
```

```
        (TyAllVal f) → (f (evalTy ty))
        _ → throwError $ Err "Not a universal"
       }
```

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 SystemFEval where

import LangUtils
import SystemFEnv
import SystemFAST
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)
    | TmTupleVal [TmVal]

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

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

$\downarrow (\mathit{LambdaVal}\ \_) = \mathit{Nothing}$
$\downarrow (\mathit{TmTupleVal}\ \_) = \mathit{Nothing}$

instance $\mathit{Subtype}\ \mathit{Int}\ \mathit{TmVal}$ **where**
$\uparrow x = (\mathit{TmIntVal}\ x)$
$\downarrow (\mathit{TmBoolVal}\ \_) = \mathit{Nothing}$
$\downarrow (\mathit{TmIntVal}\ x) = \mathit{Just}\ x$
$\downarrow (\mathit{LambdaVal}\ \_) = \mathit{Nothing}$
$\downarrow (\mathit{TmTupleVal}\ \_) = \mathit{Nothing}$

instance $\mathit{Subtype}\ (\mathit{TmValEnv} \to \mathit{TmValEnv})\ \mathit{TmVal}$ **where**
$\uparrow x = (\mathit{LambdaVal}\ x)$
$\downarrow (\mathit{TmBoolVal}\ \_) = \mathit{Nothing}$
$\downarrow (\mathit{TmIntVal}\ \_) = \mathit{Nothing}$
$\downarrow (\mathit{LambdaVal}\ x) = \mathit{Just}\ x$
$\downarrow (\mathit{TmTupleVal}\ \_) = \mathit{Nothing}$

instance $\mathit{Subtype}\ [\mathit{TmVal}]\ \mathit{TmVal}$ **where**
$\uparrow x = (\mathit{TmTupleVal}\ x)$
$\downarrow (\mathit{TmBoolVal}\ \_) = \mathit{Nothing}$
$\downarrow (\mathit{TmIntVal}\ \_) = \mathit{Nothing}$
$\downarrow (\mathit{LambdaVal}\ \_) = \mathit{Nothing}$
$\downarrow (\mathit{TmTupleVal}\ x) = \mathit{Just}\ x$

type $\mathit{Env} = \mathit{Environment}\ \mathit{TmVal}$

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 $\mathit{TmError} = \mathit{Err}\ \mathit{String}$ **deriving** $(\mathit{Show}, \mathit{Eq})$

instance $\mathit{Error}\ \mathit{TmError}$ **where**
$\mathit{noMsg} = \mathit{Err}$ `"Type Error"`
$\mathit{strMsg}\ s = \mathit{Err}\ s$

type $\mathit{TmValEnv} = \mathit{ReaderT}\ \mathit{Env}\ (\mathit{Either}\ \mathit{TmError})\ \mathit{TmVal}$

5.3 Expressions as Algebras

instance $\mathit{Algebra}\ \mathit{TmBool}\ \mathit{TmValEnv}$ **where**
$\phi\ \mathit{TmTrue} = \mathit{return}\ \$ \uparrow \mathit{True}$
$\phi\ \mathit{TmFalse} = \mathit{return}\ \$ \uparrow \mathit{False}$

$\mathit{mkTrue} = \mathit{toTmLang}\ \mathit{TmTrue}$
$\mathit{mkFalse} = \mathit{toTmLang}\ \mathit{TmFalse}$

instance $\mathit{Algebra}\ \mathit{TmInt}\ \mathit{TmValEnv}$ **where**
$\phi\ (\mathit{TmConstInt}\ x) = \mathit{return}\ \$ \uparrow x$

```
mkInt x = toTmLang $ TmConstInt 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")
       }

  φ (TmMul 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 (TmDiv x y) =
    do { x′ ← x
       ; y′ ← y
       ; case (↓ x′) of
         Just x″ → case (↓ y′) of
                     Just y″ → if (y″ :: Int) ≡ 0
                                 then throwError $ Err "Division by zero"
                                 else return $ ↑ ((x″ :: Int) ‘div‘ (y″ :: Int))
                     Nothing → error ((show y′) ++ " not an integer")
         Nothing → error ((show x′) ++ " not an integer")
       }

mkAdd x y = toTmLang $ TmAdd x y
mkSub x y = toTmLang $ TmSub x y
mkMul x y = toTmLang $ TmMul x y
mkDiv x y = toTmLang $ TmDiv x y
```

```
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")
         }

mkIf a b c = toTmLang $ If a b c

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"))
                     }

    φ (TmTVar v) = do { val ← asks (lookup v)
                      ; case val of
                         Just x → return x
                         Nothing → error ("Type variable " ++ (v ++ " not found"))
                      }

mkVar s = toTmLang $ TmVar s
mkTVar s = toTmLang $ TmTVar s

instance Algebra TmTuple TmValEnv where
    φ (TmTuple vs) = do { vs' ← sequence vs
                        ; return $ ↑ vs'
                        }

    φ (TmPrj i t) = do { t' ← t
                       ; case (↓ t') of
                          (Just (TmTupleVal vs)) → return $ ↑ (vs !! i)
                          Nothing → error "Bad tuple value"
                       }

mkTuple vs = toTmLang $ TmTuple vs
mkTmPrj i t = toTmLang $ TmPrj i t

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")
       }

  φ (TmTLambda s te) =
    do { te' ← te
       ; return $ ↑ te'
       }

  φ (TmTApp te1 te2) =
    do { te1' ← te1
       ; return $ ↑ $ te1'
       }

mkLambda s ty te = toTmLang $ TmLambda s ty te
mkApp t1 t2 = toTmLang $ TmApp t1 t2
mkTLambda s te = toTmLang $ TmTLambda s te
mkTApp t1 t2 = toTmLang $ TmTApp t1 t2
```

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 SystemFInterpreter where

import LangUtils
import SystemFEnv
import SystemFAST
import SystemFEval
import SystemFTypesT
```

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 (SystemFTypesT.Err y)) →
                do { putStr "Type Error: "
                   ; putStr y; putStr "\n"
                   }
            (Right y) → case (runEval t [ ]) of
                          (Left (SystemFEval.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"
                                 }

t0 = mkTuple [mkTrue, mkFalse]
t1 = mkTLambda "X"
     (mkLambda "x" (mkTyVar "X")
      (mkVar "x"))
t2 = mkTLambda "X"
     (mkLambda "x" (mkTyVar "X")
      (mkVar "x"))
t3 = mkApp
     (mkTApp
      (mkTLambda "X" (mkLambda "x" (mkTyVar "X") (mkVar "x")))
      SystemFTypesT.mkInt)
     (SystemFEval.mkInt 1)
```

References

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