EECS 762 - Project 3 Solution

Perry Alexander
The University of Kansas EECS Department
alex@ittc.ku.edu

February 25, 2004

1 Introduction

The objective of Project 3 is to write an interpreter for simply typed lambda calculus with subtyping and records ($\lambda_{<:}$) expressions as defined in *Types and Programming Languages* [1], Chapters 15 and 16. In addition, you were to include booleans and the `if` special form. The definition of the abstract syntax provides the following three forms for $\lambda_{<:}$ terms, values and types in:

$$\begin{array}{rcl} t & ::= & x \mid \lambda x:T.t \mid t\ t \mid \\ & & t.l \mid \{l_i = t_i^{i \in 1..n}\} \\ v & ::= & \lambda x:T.t \mid \texttt{true} \mid \texttt{false} \mid \{l_i = v_i^{i \in 1..n}\} \\ T & ::= & \texttt{Bool} \mid T \rightarrow T \mid \{l_i : T_i^{i \in 1..n}\} \mid Top \end{array}$$

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

$$\frac{t_1 \longrightarrow t_1^{'}}{t_1 t_2 \longrightarrow t_1^{'} t_2} \text{ E-APP1}$$

$$\frac{t_2 \longrightarrow t_2^{'}}{t_1 t_2 \longrightarrow t_1 t_2^{'}} \text{ E-APP2}$$

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

$$\frac{\texttt{if true then } t_2 \texttt{ else } t_3}{t_2} \text{ E-IFTRUE}$$

$$\frac{\texttt{if false then } t_2 \texttt{ else } t_3}{t_3} \text{ E-IFFALSE}$$

$$\frac{t \to t_1^{'}}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \to \texttt{if } t_1^{'} \texttt{ then } t_2 \texttt{ else } t_3} \text{ E-If}$$

$$\frac{}{\{l_i = v_i^{i \in 1..n}\}.l_j \longrightarrow v_j} \text{ E-ProjRcd}$$

$$\frac{t_1 \longrightarrow t_1^{'}}{t_1.l \longrightarrow t_1^{'}.l} \text{ E-Proj}$$

Finally, the definition provides the following typing rules that will define the type inference function:

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

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

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \; t_2 : T_{12}} \text{ T-App}$$

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

$$\frac{}{\texttt{true : Bool}}$$

$$\frac{}{\texttt{false : Bool}}$$

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

2 SubtypedLambdaMonad Module

module *SubtypedLambdaMonad* (*Term* (..))
    where

The *SubtypedLambdaMonad* module provides the basic definitions for manipulating simply typed lambda calculus expressions with subtyping and records added. As indicated by the module name, a monad is used to implement the evaluation and type checking processes. The data type representing terms and types is first defined, followed by the type checking function, subtyping function, and the evaluation function.

2.1 Data Types

An abstract type T, is defined to represent the two possible types in $\lambda_{<:}$. A constant represents the Boolean type while a pair of types represents the $\rightarrow$ type former application.

```
import Data.List
data Retval a =
    Value a | Error String
    deriving (Eq, Show)

instance Monad Retval where
    Error s >>= k = Error s
    Value a >>= k = k a
    return = Value
    fail = Error

data T =
    TyBool |
    TyTop |
    TyArr T T |
    TyTpl [T] |
    TyRec [(String, T)]
    deriving (Eq, Show)

dom :: T → Retval T
dom t =
    case t of
        TyArr d _ → return d
        TyBool → fail "Type Error - Cannot find domain of a boolean type"
        TyRec _ → fail "Type Error - Cannot find domain of a record type "
        TyTpl _ → fail "Type Error - Cannot find the domain of a tuple tye"
        TyTop → fail "Type Error - Cannot find the domain of top"

ran :: T → Retval T
ran t =
    case t of
        TyArr _ r → return r
        TyBool → fail "Type Error - Cannot find range of a boolean type"
        TyRec _ → fail "Type Error - Cannot find range of a record type"
        TyTpl _ → fail "Type Error - Cannot find the range of a tuple tye"
        TyTop → fail "Type Error - Cannot find range of top"

isArr :: T → Bool
isArr (TyArr _ _) = True
isArr _ = False

isRec :: T → Bool
isRec (TyRec _) = True
isRec _ = False

isTpl :: T → Bool
isTpl (TyTpl _) = True
```

$isTpl\ \_ = False$

A constructed type, *Term*, is defined to represent the abstract syntax for $\lambda_{<:}$ terms. Note that this definition is identical to that used in $\lambda_{<:}$ except *Lambda* includes a type annotation, the *If* construct is defined, and boolean costants have been added. The implementation of the *Term* type is a Maybe type that either represents a legal term or a form that cannot be evaluated.

```
data Term =
    TmTrue | TmFalse |
    If Term Term Term |
    Var Int |
    Lambda T Term |
    AppV Term Term |
    AppN Term Term |
    Rec [(String, Term)] |
    Tpl [Term] |
    ProjRcd Term String |
    ProjTpl Term Int
    deriving (Eq, Show)
```

The *value* function defines a predicate that specifies $\lambda_{<:}$ terms that are values. By definition, *TmTrue*, *TmFalse*, *Lambda* forms, *Rec* and *Tpl* are constructors for values.

```
value :: Term → Bool
value TmTrue = True
value TmFalse = True
value (Lambda _ _) = True
value (Rec _) = True
value (Tpl _) = True
value _ = False
```

The *TmTrue* and *TmFalse* values represent the Boolean values true and false respectively. The *If* constructor defines a classical if-expression. The *Var* construtor identifies the index for a variable and corresponds with the x form in the abstract syntax. The *Lambda* constructor defines an abstraction by specifying a term and corresponds with the $\lambda x : T.t$ form in the abstract syntax. The *AppV* cosntructor defines the application of one term to another using call-by-value semantics. Similarly, the *AppN* constructor defines the appliation of one term to another using call-by-name semantics. Both forms correspond with the $t\ t$ form in the abstract syntax.

The original λ language implemented untyped lambda calculus. Thus, the maintenance of context information was unnecessary. For $\lambda_{<:}$, the types of variables must be maintained as a part of context for type checking. The Γ type is used to store a list of variable bindings that will be used to maintain the types associated with variables in context.

```
type Γ = [T]
```

To mainipulate the context, the following functions are defined:

```
addBinding :: Γ → T → Γ
addBinding γ t = t : γ
```

```
getTypeFromContext :: Γ → Int → Retval T
getTypeFromContext c v = if ((length c) ⩾ v)
                             then return (c !! v)
                             else fail "Type Error - Variable out of scope"

findField :: Γ → String → T → Retval T
findField γ s1 (TyRec l) = findField2 γ s1 l

findField2 :: Γ → String → [(String, T)] → Retval T
findField2 _ _ [] = fail "Type Error - Field name not found"
findField2 γ s1 ((s2, t) : l) = if s1 ≡ s2
                                 then return t
                                 else findField2 γ s1 l

findProj :: Γ → Int → T → Retval T
findProj _ i (TyTpl tl) = if i ⩾ 0 ∧ i < (length tl)
                      then return (tl !! i)
                      else fail "Type Error - Tuple index out of range"
```

2.2 Type Derivation

Type derivation is achieved using the $typeof_{<:}$ function. Each case directly corresponds to one of the typing rules defined for $\lambda_{<:}$.

```
typeof<: :: Γ → Term → Retval T
typeof<: γ TmTrue = return TyBool
typeof<: γ TmFalse = return TyBool
typeof<: γ (Rec l) = do tl ← mapM (λ(_, x) → (typeof<: γ x)) l
                        ; return (TyRec (zip (map (λ(x, _) → x) l) tl))
typeof<: γ (Tpl l) = do tl ← mapM (λx → (typeof<: γ x)) l
                        ; return (TyTpl tl)
typeof<: γ (ProjRcd t s) =
    do rectype ← typeof<: γ t
       ; if (isRec rectype)
           then findField γ s rectype
           else fail "Type Error - Cannot project a non-record type."
typeof<: γ (ProjTpl t i) =
    do tpltype ← typeof<: γ t
       ; if (isTpl tpltype)
           then findProj γ i tpltype
           else fail "Type Error - Cannot project a non-tuple type."
typeof<: γ (If t1 t2 t3) =
    if (typeof<: γ t1) ≡ return TyBool
      then let tt1 = typeof<: γ t2; tt2 = typeof<: γ t3 in
               if tt1 ≡ tt2 then tt2
               else fail "Type Error - If branches of different types"
      else fail "Type Error - If conditional not boolean"
typeof<: γ (Var x) = (getTypeFromContext γ x)
typeof<: γ (AppV t1 t2) =
    do tt1 ← typeof<: γ t1
```

$$
\begin{array}{l}
\quad\quad ;\, tt_2 \leftarrow \mathit{typeof}_{<:}\ \gamma\ t_2 \\
\quad\quad ;\, dtt_1 \leftarrow \mathit{dom}\ tt_1 \\
\quad\quad ;\, \mathbf{if}\ \mathit{isArr}\ tt_1\ \mathbf{then} \\
\quad\quad\quad \mathbf{if}\ (\mathit{subtype}\ tt_2\ dtt_1)\ \mathbf{then}\ \mathit{return}\ tt_2 \\
\quad\quad\quad\quad \mathbf{else}\ \mathit{fail}\ \texttt{"Type Error - Argument type is not a subtype of domain"} \\
\quad\quad\quad \mathbf{else}\ \mathit{fail}\ \texttt{"Type Error - Term is not an abstraction"} \\
\mathit{typeof}_{<:}\ \gamma\ (\mathit{AppN}\ t_1\ t_2) = \\
\quad \mathbf{do}\ tt_1 \leftarrow \mathit{typeof}_{<:}\ \gamma\ t_1 \\
\quad\quad ;\, tt_2 \leftarrow \mathit{typeof}_{<:}\ \gamma\ t_2 \\
\quad\quad ;\, dtt_1 \leftarrow \mathit{dom}\ tt_1 \\
\quad\quad ;\, \mathbf{if}\ \mathit{isArr}\ tt_1\ \mathbf{then} \\
\quad\quad\quad \mathbf{if}\ (\mathit{subtype}\ tt_2\ dtt_1)\ \mathbf{then}\ \mathit{return}\ tt_2 \\
\quad\quad\quad\quad \mathbf{else}\ \mathit{fail}\ \texttt{"Type Error - Argument type is not a subtype of domain"} \\
\quad\quad\quad \mathbf{else}\ \mathit{fail}\ \texttt{"Type Error - Term is not an abstraction"} \\
\mathit{typeof}_{<:}\ \gamma\ (\mathit{Lambda}\ ty\ t) = \\
\quad \mathbf{do}\ tt \leftarrow \mathit{typeof}_{<:}\ (\mathit{addBinding}\ \gamma\ ty)\ t \\
\quad\quad\quad ;\, \mathit{return}\ (\mathit{TyArr}\ ty\ tt)
\end{array}
$$

The *subtype* function defines when one type is a subtype of another. *TyBool* is only a subtype of itself. A function type, *TyArr ty11 ty12* is a subtype of *TyArr ty21 ty22* if *subtype ty21 ty11* and *subtype ty12 ty22*. A record, *TyRec* is a subtype of another if it's list of label, type pairs intersects with the other's label, type pairs. *TyTop* is by definition a supertype of everything while all other subtyping attempts are illegal.

$$
\begin{array}{l}
\mathit{subtype} :: T \rightarrow T \rightarrow \mathit{Bool} \\
\mathit{subtype}\ \mathit{TyBool}\ \mathit{TyBool} = \mathit{True} \\
\mathit{subtype}\ (\mathit{TyArr}\ \mathit{ty11}\ \mathit{ty12})\ (\mathit{TyArr}\ \mathit{ty21}\ \mathit{ty22}) = \\
\quad \mathit{subtype}\ \mathit{ty21}\ \mathit{ty11} \wedge \mathit{subtype}\ \mathit{ty12}\ \mathit{ty22} \\
\mathit{subtype}\ (\mathit{TyRec}\ \mathit{l1})\ (\mathit{TyRec}\ \mathit{l2}) = (\mathit{intersect}\ \mathit{l1}\ \mathit{l2}) \equiv \mathit{l1} \\
\mathit{subtype}\ (\mathit{TyTpl}\ \mathit{l1})\ (\mathit{TyTpl}\ \mathit{l2}) = (\mathit{isPrefixOf}\ \mathit{l1}\ \mathit{l2}) \\
\mathit{subtype}\ \_\ \mathit{TyTop} = \mathit{True} \\
\mathit{subtype}\ \_\ \_ = \mathit{False}
\end{array}
$$

2.3 Shifting and Substituting

Our implementation uses de Brujin indices as a basis for representation and evaluation. Thus, definitions of *shift* and *subst* are required to implement elements of the evalation function. These functions are largely the same as those used in Project 1 except they must deal with type information.

The *shift* definition provides a case for shifting over each valid λ form as defined in the *Term* data type. This definition follows directly from the standard definition of shift from TPL Chapter 8:

$$
\begin{array}{l}
\mathit{shift} :: \mathit{Term} \rightarrow \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Term} \\
\mathit{shift}\ \mathit{TmTrue}\ c\ d = \mathit{TmTrue} \\
\mathit{shift}\ \mathit{TmFalse}\ c\ d = \mathit{TmFalse} \\
\mathit{shift}\ (\mathit{Rec}\ l)\ c\ d = \\
\quad (\mathit{Rec}\ (\mathit{map}\ (\lambda(\mathit{str}, t) \rightarrow (\mathit{str}, (\mathit{shift}\ t\ c\ d)))\ l)) \\
\mathit{shift}\ (\mathit{Tpl}\ l)\ c\ d = \\
\quad (\mathit{Tpl}\ (\mathit{map}\ (\lambda t \rightarrow (\mathit{shift}\ t\ c\ d))\ l)) \\
\mathit{shift}\ (\mathit{ProjRcd}\ r\ s)\ c\ d = (\mathit{ProjRcd}\ (\mathit{shift}\ r\ c\ d)\ s) \\
\mathit{shift}\ (\mathit{ProjTpl}\ t\ i)\ c\ d = (\mathit{ProjTpl}\ (\mathit{shift}\ t\ c\ d)\ i) \\
\mathit{shift}\ (\mathit{Var}\ x)\ c\ d = \mathbf{if}\ x < c\ \mathbf{then}\ (\mathit{Var}\ x)\ \mathbf{else}\ (\mathit{Var}\ (x + d))
\end{array}
$$

$$
\begin{aligned}
&\mathit{shift}\ (\mathit{Lambda}\ \mathit{ty}\ t)\ c\ d = (\mathit{Lambda}\ \mathit{ty}\ (\mathit{shift}\ t\ (c+1)\ d))\\
&\mathit{shift}\ (\mathit{AppV}\ t_1\ t_2)\ c\ d = (\mathit{AppV}\ (\mathit{shift}\ t_1\ c\ d)\ (\mathit{shift}\ t_2\ c\ d))\\
&\mathit{shift}\ (\mathit{AppN}\ t_1\ t_2)\ c\ d = (\mathit{AppN}\ (\mathit{shift}\ t_1\ c\ d)\ (\mathit{shift}\ t_2\ c\ d))
\end{aligned}
$$

Lambda, *AppV*, and *AppN* terms are shifted by shifting their identified terms. Shifting a *Var* term requires application of the definition from Chapter 8. Specifically, the index for a variable is shifted by d if its index is greater than c.

The *subst* definition again provides a case for substitution over each value λ as defined in the *Term* data type. This definition follows directly from teh standard definition of substitution from TPL Chapter 8:

$$
\begin{aligned}
&\mathit{subst} :: \mathit{Int} \to \mathit{Term} \to \mathit{Term} \to \mathit{Term}\\
&\mathit{subst}\ j\ s\ (\mathit{Var}\ x) = \textbf{if}\ x \equiv j\ \textbf{then}\ s\ \textbf{else}\ (\mathit{Var}\ x)\\
&\mathit{subst}\ j\ s\ (\mathit{Lambda}\ \mathit{ty}\ t) = (\mathit{Lambda}\ \mathit{ty}\ (\mathit{subst}\ (j+1)\ (\mathit{shift}\ s\ 0\ 1)\ t))\\
&\mathit{subst}\ j\ s\ (\mathit{AppV}\ t_1\ t_2) = (\mathit{AppV}\ (\mathit{subst}\ j\ s\ t_1)\ (\mathit{subst}\ j\ s\ t_2))\\
&\mathit{subst}\ j\ s\ (\mathit{AppN}\ t_1\ t_2) = (\mathit{AppN}\ (\mathit{subst}\ j\ s\ t_1)\ (\mathit{subst}\ j\ s\ t_2))\\
&\mathit{subst}\ \_\ \_\ \mathit{TmTrue} = \mathit{TmTrue}\\
&\mathit{subst}\ \_\ \_\ \mathit{TmFalse} = \mathit{TmFalse}\\
&\mathit{subst}\ j\ s\ (\mathit{Rec}\ l) =\\
&\qquad (\mathit{Rec}\ (\mathit{map}\ (\lambda(\mathit{str}, t) \to (\mathit{str}, (\mathit{subst}\ j\ s\ t)))\ l))\\
&\mathit{subst}\ j\ s\ (\mathit{Tpl}\ l) =\\
&\qquad (\mathit{Tpl}\ (\mathit{map}\ (\lambda t \to (\mathit{subst}\ j\ s\ t))\ l))\\
&\mathit{subst}\ j\ s\ (\mathit{ProjRcd}\ r\ \mathit{str}) =\\
&\qquad (\mathit{ProjRcd}\ (\mathit{subst}\ j\ s\ r)\ \mathit{str})\\
&\mathit{subst}\ j\ s\ (\mathit{ProjTpl}\ t\ i) =\\
&\qquad (\mathit{ProjTpl}\ (\mathit{subst}\ j\ s\ t)\ i)
\end{aligned}
$$

2.4 Call By Value Evaluation Function

The $\mathit{eval}_{<:}$ function provides a standard definition of call-by-value evaluation following from TPL Chapter 5. The function definition is split up into cases corresponding to the evaluation rules. Note that only *AppN* and *AppV* forms can be evaluated. *Lambda* forms are values and *Var* forms are not closed. Although passing a context is allowed in this function, in this project we are concerned only with close terms. Thus, the context can be safely ignored in all evaluation cases.

The form of the evaluation function is a mapping from Γ and *Term* to another *Term*:

$$
\begin{aligned}
&\mathit{expr} :: \mathit{Term} \to \mathit{Term}\\
&\mathit{expr}\ (\mathit{Lambda}\ \mathit{ty}\ t) = t
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{proj} :: [(\mathit{String}, \mathit{Term})] \to \mathit{String} \to \mathit{Retval}\ \mathit{Term}\\
&\mathit{proj}\ [\,]\ \_ = \mathit{fail}\ \texttt{"Evaluation Error - Field not defined"}\\
&\mathit{proj}\ ((s_1, v) : l)\ s_2 = \textbf{if}\ s_1 \equiv s_2\\
&\qquad\qquad\qquad\qquad\quad \textbf{then}\ \mathit{return}\ v\\
&\qquad\qquad\qquad\qquad\quad \textbf{else}\ (\mathit{proj}\ l\ s_2)
\end{aligned}
$$

$$\mathit{eval}_{<:} :: \mathit{Term} \to \mathit{Retval}\ \mathit{Term}$$

$$
\begin{aligned}
&\mathit{eval}_{<:}\ \mathit{TmTrue} = \mathit{return}\ \mathit{TmTrue}\\
&\mathit{eval}_{<:}\ \mathit{TmFalse} = \mathit{return}\ \mathit{TmFalse}\\
&\mathit{eval}_{<:}\ (\mathit{Rec}\ x) = \mathit{return}\ (\mathit{Rec}\ x)
\end{aligned}
$$

$eval_{<:}\ (Tpl\ x) = return\ (Tpl\ x)$

$eval_{<:}\ (AppV\ t_1\ t_2) =$
  do $nt1 \leftarrow eval_{<:}\ t_1$
  $;\ nt2 \leftarrow eval_{<:}\ t_2$
  $;\ eval_{<:}\ (shift\ (subst\ 0\ (shift\ nt2\ 0\ 1)\ (expr\ nt1))\ 0\ (-1))$

$eval_{<:}\ (AppN\ t_1\ t_2) =$
  do $nt1 \leftarrow eval_{<:}\ t_1$
  $;\ eval_{<:}\ (shift\ (subst\ 0\ (shift\ t_2\ 0\ 1)\ (expr\ nt1))\ 0\ (-1))$

$eval_{<:}\ (Lambda\ ty1\ t_1) = return\ (Lambda\ ty1\ t_1)$

$eval_{<:}\ (Var\ t_1) = fail$ `"Evaluation Error - Cannot evaluate free variable"`

$eval_{<:}\ (If\ TmTrue\ t_2\ t_3) =$
  do $tr \leftarrow eval_{<:}\ t_2$
  $;\ return\ tr$
$eval_{<:}\ (If\ TmFalse\ t_2\ t_3) =$
  do $fl \leftarrow eval_{<:}\ t_3$
  $;\ return\ fl$
$eval_{<:}\ (If\ t_1\ t_2\ t_3) =$
  do $cond \leftarrow (eval_{<:}\ t_1)$
  $;\ eval_{<:}\ (If\ cond\ t_2\ t_3)$
$eval_{<:}\ (ProjRcd\ (Rec\ r)\ l) = (proj\ r\ l)$
$eval_{<:}\ (ProjTpl\ (Tpl\ t)\ i) = return\ (t\ !!\ i)$

3 Type Checking and Evaluation

The objective of type checking is to statically predict the runtime behavior of a code element with respect to the kinds of values produced and expected. Specifically, does a program element always product the kind of value espected? The `typeof` function provides a capability for generating the type associated with a $\lambda_{<:}$ term. If such a type exists, then the term is well-typed and should be executed.

To combine type checking and evaluation is a simple matter of: (i) determining the type of a term; and (ii) executing the term if the type exists. This process can be specified using the following template:

$interpTemplate :: (Term \rightarrow Retval\ T) \rightarrow$
  $(Term \rightarrow Retval\ Term) \rightarrow$
  $Term \rightarrow$
  $Retval\ Term$
$interpTemplate\ typeof_{<:}\ eval_{<:}\ term =$
  do $termtype \leftarrow (typeof_{<:}\ term)$
  $;\ term' \leftarrow (eval_{<:}\ term)$
  $;\ return\ term'$

$evalTemplateStar :: (Term \rightarrow Retval\ Term) \rightarrow Term \rightarrow Retval\ Term$
$evalTemplateStar\ eval_{<:}\ term =$
  do $term' \leftarrow (eval_{<:}\ term)$
  $;$ **if** $(value\ term')$

```
        then return term'
        else (evalTemplateStar eval<:) term'

interpret :: Term → Retval Term
interpret = interpTemplate (typeof<: []) eval<:
```

4 Testing and Evaluation

To test the interpreter, some functions from the book are provided here. They include the identity combinator, Church Boolean funtions, some Church Numbers and the successor function defined for Church numbers.

Note that with the introduction of types, all parameters must have associated type annotations. Although significant static checking results, the flexibility of these functions is diminished. Specifically, for the `ident` combinator a new combinator must be written for each type. No polymorphism exists and the type system is strict, so there is no way to reuse the `ident` combinator definition. Such strictness is common in older programming languages, but new polymorphism implementations render the approach obscolete.

4.1 Identity Combinator

Redefine the *ident* combinator to operate over *Top* to take advantage of subtyping. Evaluating the (*App indent t*) combinator on term defined in $\lambda_{<:}$ should result in t regardless of its type:

```
ident :: Term
ident = (Lambda TyTop (Var 0))

testIdent :: [Retval Term]
testIdent =
    map interpret [ident,
                   (Rec [("1", TmTrue), ("2", TmFalse), ("3", ident)]),
                   (ProjRcd (Rec [("1", TmTrue), ("2", TmFalse), ("3", ident)]) "1")]

testRecord :: [Retval Term]
testRecord =
    let r = (Rec [("1", TmTrue), ("2", TmFalse), ("3", ident)]) in
      map interpret
        [(ProjRcd r "1"),
         (ProjRcd r "3"),
         (ProjRcd r "4"),
         (AppV (ProjRcd r "3") r)]

testRecSubtype :: [Retval Term]
testRecSubtype =
    let l = (Lambda (TyRec [("1", TyBool), ("2", TyBool), ("3", TyBool)])
            (ProjRcd (Var 0) "2")) in
      map interpret
           [(AppV l (Rec [("1", TmTrue), ("2", TmFalse), ("3", TmTrue)])),
            (AppV l (Rec [("1", TmTrue), ("2", TmFalse)])),
            (AppV l (Rec [("2", TmFalse), ("3", TmTrue)])),
```

$(AppV\ l\ (Rec\ [(\texttt{"1"}, TmTrue), (\texttt{"4"}, TmFalse), (\texttt{"3"}, TmTrue)]))]$

$testTuple :: [Retval\ Term]$
$testTuple =$
  $\textbf{let}\ r = (Tpl\ [TmTrue, TmFalse, ident])\ \textbf{in}$
    $map\ interpret$
      $[(ProjTpl\ r\ 0),$
      $(ProjTpl\ r\ 2),$
      $(ProjTpl\ r\ 3),$
      $(AppV\ (ProjTpl\ r\ 2)\ r)]$

$testTupleSubtype :: [Retval\ Term]$
$testTupleSubtype =$
  $\textbf{let}\ l = (Lambda\ (TyTpl\ [TyBool, TyBool, TyBool])$
      $(ProjTpl\ (Var\ 0)\ 1))\ \textbf{in}$
    $map\ interpret$
      $[(AppV\ l\ (Tpl\ [TmTrue, TmFalse, TmTrue])),$
      $(AppV\ l\ (Tpl\ [TmTrue, TmFalse])),$
      $(AppV\ l\ (Tpl\ [TmFalse, TmTrue])),$
      $(AppV\ l\ (Tpl\ [TmTrue, ident, TmTrue]))]$

4.2 Church Boolean Definitions

The boolean combinators must have typed parameters. Real booleans are chosen for simplicity. Church Booleans don't make a great deal of sense in a strongly typed language like this.

$tru :: Term$
$tru = (Lambda\ TyTop\ (Lambda\ TyTop\ (Var\ 1)))$
$fls :: Term$
$fls = (Lambda\ TyTop\ (Lambda\ TyTop\ (Var\ 0)))$

$testTru :: Retval\ Term$
$testTru = interpret\ (AppV\ (AppV\ tru\ ident)\ fls)$

$testFls :: Retval\ Term$
$testFls = interpret\ (AppV\ (AppV\ fls\ fls)\ ident)$

4.3 Church Number Definitions

The number combinators must have typed parameters. Real booleans are chosen for simplicity. In a word, these definitions are not appropriate anymore, but they are retained for testing.

$c_0 :: Term$
$c_0 = (Lambda\ TyBool\ (Lambda\ TyBool\ (Var\ 0)))$

$c_1 :: Term$
$c_1 = (Lambda\ TyBool\ (Lambda\ TyBool\ (AppV\ (Var\ 1)\ (Var\ 0))))$

$c_2 :: Term$

$c_2 = (Lambda\ TyBool\ (Lambda\ TyBool\ (AppV\ (Var\ 1)\ (AppV\ (Var\ 1)\ (Var\ 0)))))$

$scc :: Term$
$scc = (Lambda\ TyBool$
$\quad (Lambda\ TyBool$
$\quad\ (Lambda\ TyBool\ (AppV\ (Var\ 1)$
$\qquad\qquad (AppV\ (Var\ 2)$
$\qquad\qquad\ (AppV\ (Var\ 1)\ (Var\ 0)))))))$

4.4 Omega

With the introduction of types, the omega combinator must type its parameter. Boolean is selected here for the sake of simplicity, but other types could be used.

$\Omega :: Term$
$\Omega = (Lambda\ TyTop\ (AppV\ (Var\ 0)\ (Var\ 0)))$

5 Notes

The evaluation function is built to handle both call-by-value and call-by-name function application. This may end up causing problems if the arguments to record and tuple references are treated as types in the language rather than Haskell string and integer types.

Are record and tuple projections too aggressive for call-by-name application? They are not true functions in the langauge due to the types of their reference values.

References

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