Typability

let f = proc (x: int) { proc (y: int) { x+y }} in (f 2)

Typable. Take Γ to be {} and t to be int -> int.
{} |- let f = proc (x:int) { proc (y: int) { x+y }} in (f 2) : int -> int

```
------------------------ TVar ---------------------- TVar
{x:int, y:int} |- x: int      {x:int, y:int} |- y: int
---------------------------------------------------- TAdd
               {x:int, y:int} |- x+y: int
--------------------------------------------- TProc                 -------------------------------- TVar   ------------------- TInt
   {x:int} |- proc (y: int) { x+y } : int -> int                   {f:int->int->int} |- f:int->int->int     {f:int->int->int}|- 2: int
------------------------------------------------------------ TProc  ------------------------------------------------------------ TApp
{} |- proc (x:int) { proc (y: int) { x+y }} : int -> int -> int          {f: int->int->int} |- (f 2): int-> int
------------------------------------------------------------------------------------------------------------------------------ TLet
                          {} |- let f = proc (x:int) { proc (y: int) { x+y }} in (f 2) : int -> int
```

proc (x: int) { proc (y: bool) {if y then x else x-1} }

Typable. Set Γ to be {} and t to be int -> bool -> int.
{} |- proc (x: int) {proc (y: bool) {if y then x else x-1} }:t

```
                                                             {x: int, y: bool}(x) = int
                                                             ---------------------TVar   ----------------------TInt
{x: int, y: bool}(y) = bool    {x: int, y: bool}(x) = int    {x: int, y: bool }|- x: int    {x: int, y: bool} |- 1: int
---------------------------TVar  -----------------------TVar  -------------------------------------------------TSub
{x: int, y: bool} |- y: bool     {x: int, y: bool} |- x: int          {x: int, y: bool} |- x-1: int
-------------------------------------------------------------------------------------------------------------TITE
                                  {x: int, y: bool} |- if y then x else x-1: int
-------------------------------------------------------------------------------------------------------------TProc
                        {x: int} |- proc (y: bool) {if y then x else x-1}: bool -> int
-------------------------------------------------------------------------------------------------------------TProc
                {} |- proc (x: int) { proc (y: bool) {if y then x else x-1} }: int -> bool -> int
```

let f=proc(x: int) { proc (y: int) {x-y} } in (f (3-6))

Typable. Take Γ to be {} and f to be int->int->int
{} |- let f=proc(x: int) { proc(y: int) {x-y} } in (f (3-6)): int -> int

```
 {x: int, y: bool}(x) = int
  ----------------TVar  ------------TVar
 {x: int, y: int} |- x: i    {x: int, y: int} |- y: i
-----------------------------------------TSub                                       --------------TInt    -----------------TInt
          {x: int, y: int} |- x-y: i                                                {f: i->i->i} |- 3: i     {f: i->i->i} |- 6: i
-----------------------------------------TProc     -----------------TVar  ----------------------------------------TSub
      {x: int} |- proc(y: int) {x-y}: i->i         {f: i->i->i} |- f: i->i->i          {f: i->i->i} |- 3-6: i
------------------------------------------TProc    ---------------------------------------------------------------TApp
{} |- proc(x: int) { proc(y: int) {x-y} }: i->i->i                 {f: i->i->i} |- (f (3-6)): i->i
------------------------------------------------------------------------------------------------------------------TLet
                        {} |- let f=proc(x: int) { proc(y: int) {x-y} } in (f (3-6)): int -> int
```

Debug

Explicit-Refs

let a = 3
in let b = newref(if zero?(a) then 1 else 2)
in debug(a)
Env := [a => NumVal 3, b => RefVal 0]
Store := [0 => NumVal 2]

let a = newref(2+1)
in let b = newref(deref(a)+1)
in debug(a)
Env := [a => RefVal 0, b => RefVal 1]
Store:= [0 => NumVal, 1 => NumVal 4]

Implicit-Refs

let a = 2 in let b = 3
in begin
set a = b; debug(a)
end
Env := [a => RefVal 0, b => RefVal 1]
Store := [0 -> NumVal 3, 1 -> NumVal 3]

let x=2
in let y=set x=3
in let z=proc(y) {proc(w) {set x=y+w}}
in debug(3)
Env:= [x:=RefVal 0, y:=RefVal 1., z:=RefVal 2]
Store:= [0:=NumVal 3, 1:=UnitVal
2:=ProcVal("y", proc(w) {set x=y+w}, [x:=RefVal 0, y:=RefVal 1])]

Typability Notes

- if 3 then 88 else 69 //not typable: 3 is not a bool
- let x = 3 in (3 x) //not typable: 3 is not a function
- proc(x) { (x 3) } //typable: x has to be a func ocam

- Γ is the typing environment
  - Γ |- e:t -> look into Γ and retrieve the type of e
  - Γ, id:t1 |- e:t2 -> extend Γ with id of type t1 (extend_env id t1), then retrieve type of e
- {x: int} |- x+2: int -> valid
- {x: bool} |- x+2: int -> invalid

- functions are type (example) t1->t2->t3, where t3 is the return type and t1 is the first param, t2 is second. t2 only exists if it's a nested function (whose purpose is to enable a second parameter).

- typing expr, env, type

Pair Implementation

```
  Γ |- e1: t1   Γ |- e2: t2
---------------------------TPair
  Γ |- pair(e1, e2): t1*t2
```

```
| Pair(e1,e2) ->
  chk_expr e1 >>= fun t1 ->
  chk_expr e2 >>= fun t2 ->
  return @@ PairType(t1,t2)
```

```
  Γ |- e1: t1*t2  Γ, id1:t1, id:t2 |- e2:t
--------------------------------TUnpair
  Γ |- unpair(id1, id2) = e1 in e2: t
```

```
| Unpair(id1,id2,e1,e2) ->
  chk_expr e1 >>= fun t ->
  (match t with
   | PairType(t1,t2) ->
     extend_tenv id1 t1 >>+
     extend_tenv id2 t2 >>+
     chk_expr e2
   | _ -> error "unpair: expected a pair")
```

Debug Notes

- newref evaluates the expression, then allocates it to the Store
  - Store: 0 => ITE(IsZero?("a"), Int 1, Int 2) //**wrong**

Checked Implementation

```
(* EXPLICIT-REFS *)
| BeginEnd([]) ->
  return UnitType
| BeginEnd(es) ->
  let rec check = function
    | [] -> return UnitType
    | [h] -> chk_expr h
    | h::t -> chk_expr h >>= fun _ ->
           check t
  in check es
| NewRef(e) ->
  chk_expr e >>= fun t ->
  return @@ RefType(t)
| DeRef(e) ->
  chk_expr e >>= fun t ->
  (match t with
    | RefType(t) -> return t
    | _ -> error "deref: expected a reference type" )
| SetRef(e1,e2) ->
  chk_expr e1 >>= fun t1 ->
  chk_expr e2 >>= fun t2 ->
  (* check e1 is a ref, then check e2 = e1's type *)
  (match t1 with
    | RefType(t) -> if(t=t2)
                then return UnitType
                else error "setref: types of ref and expr don't match"
    | _ -> error "setref: expected a reference type" )
(* list *)
| EmptyList(None) ->
  return @@ ListType(UnitType)
| EmptyList(Some t) ->
  return @@ ListType(t)
| Cons(h, t) ->
  chk_expr h >>= fun t1 ->
  chk_expr t >>= fun t2 ->
  (match t2 with
    | ListType(t) -> if(t1=t)
                 then return @@ ListType(t)
                 else error "cons: type of head and tail don't match"
    | _ -> error "cons: expected a list type" )
| IsEmpty(e) ->
  chk_expr e >>= fun t ->
  (match t with
    | ListType(_) -> return BoolType
    | TreeType(_) -> return BoolType
    | _ -> error "isempty: expected a list type" )
| Hd(e) ->
  chk_expr e >>= fun te ->
  (match te with
    | ListType(t) -> return t
    | _ -> error "hd: expected a list type"
  )
| Tl(e) ->
  chk_expr e >>= fun te ->
  (match te with
    | ListType(t) -> return @@ ListType(t)
    | _ -> error "tl: expected a list type" )
```

Checked Implementation

```
 (* tree *)
 | EmptyTree(None) ->
   return @@ TreeType(UnitType)
 | EmptyTree(Some t) ->
   return @@ TreeType(t)
 | Node(de, le, re) ->
   chk_expr de >>= fun t1 ->
   chk_expr le >>= fun t2 ->
   chk_expr re >>= fun t3 ->
   (* t2 and t3 must be trees of the same type as t1. *)
   if t2=TreeType(t1) && t3=TreeType(t1)
   then return @@ TreeType(t1)
   else error "node: types of don't match or subtrees aren't trees"
 | CaseT(target,emptycase,id1,id2,id3,nodecase) ->
   chk_expr target >>= fun tr ->
   chk_expr emptycase >>= fun tec ->
     (match tr with
     | TreeType(t) ->
       extend_tenv id1 t >>+
       extend_tenv id2 (TreeType(t)) >>+
       extend_tenv id3 (TreeType(t)) >>+
       chk_expr nodecase >>= fun tnc ->
       (* e2 and e3 must be of same type s *)
       if(tec=tnc)
       then return tnc
       else error "caseT: types of emptycase and nodecase don't match"
   | _ -> error "caseT: expected a tree type"
   )
```

Record Example (Exam Implementation Guess)

```
 | Record(fs) ->
   sequence (List.map process_field fs) >>= fun evs ->
   return (RecordVal (addIds fs evs))
 | Proj(e,id) ->
   eval_expr e >>=
   fields_of_recordVal >>= fun f ->
   (match List.assoc_opt id f with
   | None -> error "Field does not exist"
   | Some (_,v) -> return v
     (*need (_,v) bc now we have a true/false field for mutable*)
     (*List.assoc_opt still works bc f is a key(string)-value pair. Just that the value is now
       a tuple rather than an exp_val*)
    )
 | SetField(e1,id,e2) ->
   eval_expr e1 >>=
   fields_of_recordVal >>= fun f ->
   eval_expr e2 >>= fun l ->
   (match List.assoc_opt id f with
   | None -> error "Field does not exist"
   | Some (b,v) ->
       match b with
       | false -> error "Error: Field not mutable"
       | true ->
         int_of_refVal v >>= fun t ->
         Store.set_ref g_store t l >>= fun _ -> return UnitVal
       (* mutable fields will be a RefVal with value of its value's address in Store. *)
       (* despite the pdf's example in 2.1, p.age will still return the RefVal value (the pointer vo
the actual NumVal. *)
       (* however, if you debug() after setting the ref, it'll show that the value has been corre
the Store. *) )
```

DO NOT PRINT THIS PAGE

Add:

- type derivations
- debug examples for explicit and implicit refs
- Copy+Paste checked hw + guess final content and maybe do it? -> record, probably!!
- figure out how to assign a bool to a var...
- pair unpair code