Table of Contents

Introduction

Hello Professor, I mainly wrote this to ensure that my logic was correct for each part of the project. I Chose to leave this in to explain my logic and help you grade

Evaluation

Basic Definitions

```
#[derive(Debug)]
```

```
#[derive(Clone)]
#[derive(PartialEq)]
pub struct Slot {
    pub value: Pvalue,
    pub lifetime: Lifetime,

}

#[derive(Debug)]
#[derive(PartialEq)]
#[derive(Clone)]
pub struct Store(pub HashMap<Location, Slot>);
```

Implementing The Store

`locs_by_lifetime`

```
pub fn locs_by_lifetime(& self, l:Lifetime) -> Vec<Pvalue>{
    let mut result = Vec::new();
    for locs in self.0.keys(){
    let Some(s) = self.0.get(locs) else {panic!("Impossible")};
    if s.lifetime == l{
                result.push(Some(Value::Ref(locs.to_string(),Owned::Yes)));
                }}
        return result;}
```

Pretty Simple Function, It just iterates through the keys of the Hashmap and finds all the values with the lifetime `l`

`insert`

```
pub fn insert(&mut self, loc:&str, val:Pvalue, l:Lifetime){


    let slot = Slot{value:val, lifetime:l};
    self.0.insert(loc.to_string(),slot);}
```

Again Pretty Self Explanatory, We just insert the location and whatever slot it points to!

`locate`

```
 pub fn locate<'a>(&'a self, w: &'a Lval) -> &'a Location {
    let mut current_def = &w.ident;
    for i in 0..w.derefs{

        let Some(slot) = self.0.get(current_def) else {panic!("Impossible")};
        match &slot.value{
        Some(Value::Unit) => {panic!("Impossible");;},
        Some(Value::Int(x)) => {panic!("Impossible");;},
        Some(Value::Ref(loc,owned)) =>{current_def = loc;},
        None =>{panic!("Impossible");}
        }

        }
    return current_def;

 }
```

In the paper the Locate Function is defined as the following

$$loc(S, w) =_w$$

Because there is very much an abstraction in the paper, I took a lot more liberty and simply followed the path of deference to get to a location

`read`

```
pub fn read(&self, x: &Lval) -> &Slot {
    let loc = self.locate(x);
    return self.0.get(loc).expect("Impossible");}
```

$$\text{read}(S, w) = S(\ell_w) \quad \text{where loc}(S, w) = \ell_w$$

I followed this rule pretty literally and just used the locate function to fetch the location from our store

`write`

```
pub fn write(&mut self, x: &Lval, v: Pvalue) -> Pvalue {
    //First we Must Create the Life time.
    //We can set the Location as the Variable itself

    let loc = self.locate(x).clone();
    let Some(old_slot)= self.0.remove(&loc) else {panic!("Impossible");};
    let s=  Slot{value: v.clone(), lifetime:old_slot.lifetime.clone()};
    self.0.insert(loc.to_string(),s);


    return old_slot.value.clone();

    }
```

$$\mathrm{write}(S, w, v^{\perp}) = S[\ell_w \mapsto \langle v^{\perp}\rangle^m] \quad \text{where } \mathrm{loc}(S,w) = \ell_w \text{ and } S(\ell_w) = \langle\cdot\rangle^m$$

Once again pretty literal, I fetched the location and remapped it. Exactly as the function is defined

drop

```
    pub fn drop(&mut self, values: Vec<Pvalue>) {
        let mut result = Vec::new();
        for value in values{
            match value {

            Some(Value::Ref(loc,Owned::Yes)) =>{
            let Some(next) = self.0.remove(&loc) else {panic!("Impossible")};;
```

```
                result.push(next.value);}

            Some(Value::Ref(loc,Owned::No)) =>{},


            Some(literal) =>{},
            None =>{}}}
        if result.len() !=0{
        self.drop(result);

            }

    }
```

$$
\begin{aligned}
\text{drop}(S, \overrightarrow{\ }) &= S \\
\text{drop}(S, \psi \cup \{v^{\perp}\}) &= \text{drop}(S, \psi) \quad \text{if } v^{\perp} \neq \ell^{\bullet} \\
\text{drop}(S, \psi \cup \{\ell^{\bullet}\}) &= \text{drop}(S - \{\ell \mapsto \langle v^{\perp} \rangle^{*}\},\ \psi \cup \{v^{\perp}\}) \quad \text{where } S(\ell) = \langle v^{\perp} \rangle^{*}
\end{aligned}
$$

Here I recursively deallocated the location with in our store as described in the definition

`eval_expr`

Base Cases

```
    Expr::Unit =>{return Value::Unit},
    Expr::Int(x)=>{return Value::Int(x.clone())}
```

`Expr::Lv(x, Copyable::Yes)=>`

```
Expr::Lv(x, Copyable::Yes)=>{

    let loc = self.store.locate(x);
    let Some( x_slot) = self.store.0.get(loc) else{panic!("Impossible");};
    let Some(ref x_val) = x_slot.value else {panic!("Impossible");};
    return x_val.clone(); }
```

Because the Function is Copyable We can simply return its cloned value !

`Expr::Lv(x, Copyable::No)=>`

```
Expr::Lv(x, Copyable::No)=>{

    let old_val = self.store.write(x, None)
     .expect("move out of a non-existent slot");

    return old_val; }
```

Because this Lval is not Copyable we must perform a move:

$$\frac{read(S1, w) = <v>^m \quad S_2 = write(S_1, w, \perp)}{S_1 B\ w \to S_2 B v}$$

Hence, we simply perform a *destructive read* and return the value that was in our previous Store

`Expr:: OBox(e1)`

```
Expr:: OBox(e1)=>{
    let val_e1 = self.eval_expr(e1,l.clone());
    let s = Slot{value:Some(val_e1.clone()), lifetime:Lifetime::global()};
    let loc = self.fresh();
    self.store.0.insert(loc.clone(), s);
```

$$\frac{\langle S_1 \triangleright e \Downarrow S_2 \triangleright v\rangle^l \quad \ell_n \notin \mathrm{dom}(S_2) \quad S_3 = S_2[\ell_n \mapsto \langle v\rangle^*]}{\langle S_1 \triangleright \texttt{box}\ e \Downarrow S_3 \triangleright \ell_n\rangle^l} \quad \text{(R-Box-Big)}$$

Essentially, according to the rule, we evaluate the expression within our store, choose a fresh storage location and map our value to said location

`Expr::Borrow(x, mutability)`

```
Expr::Borrow(x, mutability)=>{
 let place = self.store.locate(x);
 return Value::Ref(place.clone(), Owned::No);}
```

\$\$ $\mathrm{loc}(\mathrm{S}^{\mathbb{I}}, w) = \ell_w \frac{}{\langle S \triangleright \&[\mathrm{mut}]\, w \longrightarrow S \triangleright \ell_w^{\circ}\rangle^l \,\text{(R-Borrow)}}$

\$\$

Simply put according to the rule all we have to do is return a reference to the location in our store!

`Expr::Block(statements, e1, l1)`

```
Expr::Block(statements, e1, l1)=>{
 for expr_st in statements{
  self.eval_stmt(expr_st, l1.clone());}

 let value = self.eval_expr(e1,l1.clone());
 self.store.drop(self.store.locs_by_lifetime(l1.clone()));
 return value;

  }}}
```

$$\frac{\langle S_1 \triangleright \overrightarrow{t_1} \longrightarrow S_2 \triangleright \overrightarrow{t_2}\rangle^m}{\langle S_1 \triangleright \{\overrightarrow{t_1}\}^m \longrightarrow S_2 \triangleright \{\overrightarrow{t_2}\}^m\rangle^{l1}}(\text{R-BlockA}), \qquad \frac{S_2 = \text{drop}(S_1, m)}{\langle S_1 \triangleright \{v\}^m \longrightarrow S_2 \triangleright v\rangle^1}(\text{R-BlockB}).$$

Essentially what this rule is saying is that we need to recursively evaluate each expression in our Store until we get to the last value in which case we drop the current lifetime and return the last value

`eval_stmt`

`Stmt::Assign`

```
 Stmt::Assign(lv, expr) =>{
 let val_expr = self.eval_expr(expr,l);

 let old_val= self.store.read(lv);

 self.store.drop([old_val.value.clone()].to_vec());
 self.store.write(lv, Some(val_expr));
```

$$\frac{\text{read}(S_1, w) = \langle v_1^{\perp} \rangle^m \quad S_2 = \text{drop}(S_1, \{v_1^{\perp}\}) \quad S_3 = \text{write}(S_2, w, v_2)}{\langle S_1 \triangleright w = v_2 \longrightarrow S_3 \triangleright \epsilon \rangle^l}$$

For an assignment, we read the old value stored at w then we drop it and write $v2$ to w

`Stmt::LetMut`

```
Stmt::LetMut(x, expr) =>{

let val_expr = self.eval_expr(expr,l.clone());
let s = Slot{value:Some(val_expr.clone()), lifetime:l.clone()};
  self.store.0.insert(x.to_string(),s);


 },
```

$$\frac{S_2 = S_1[\ell_x \mapsto \langle v \rangle^l]}{\langle S_1 \triangleright \texttt{let mut } x = v \longrightarrow S_2 \triangleright \epsilon \rangle^l}$$

This one is thankfully simple, we map the location of x to its value within our store. Note I chose to assert $x = l_x$ for simplicity and because it doesn't change our logic!

Type Checking Part 1

Basic Definitions

```
#[derive(Debug)]
#[derive(Clone)]
```

```
#[derive(PartialEq)]
pub struct Slot {
  pub tipe: Type,
  pub lifetime: Lifetime,
}

#[derive(Clone)]
#[derive(PartialEq)]
#[derive(Debug)]
pub struct Env(pub HashMap<Ident, Slot>);
```

Implementing The Environment

`insert`

```
    pub fn insert(&mut self, var: &str, tipe: Type, lifetime: Lifetime) {
        let s = Slot{tipe:tipe, lifetime:lifetime};
        self.0.insert(var.to_string(), s);
    }
```

Simply Put, we Map the Slot into our context

`type_lval`

```
    pub fn type_lval(&self, lval: &Lval) -> TypeResult<Slot> {
            let  Some(slot) = self.0.get(&lval.ident) else{return Err(Error::UnknownVar(lval.ident.clone()))};
```

```
        let mut current_type = &slot.tipe;
        let mut current_lifetime = &slot.lifetime;
        for i in 0..lval.derefs{
            if let Type::Undefined(_) = current_type {
                return Err(Error::MovedOut(lval.clone()));
            }
            match current_type{
            Type::TBox(x)=> {current_type=&*x;}
            Type::Ref(x,y)=>{

                let  Some(slot) = self.0.get(&x.ident) else{return Err(Error::UnknownVar(x.ident.clone()))};
                let Some(tipe) = self.contained(&x.ident)  else{return Err(Error::UnknownVar(x.ident.clone()))};
                 current_type =tipe ;
                 current_lifetime= &slot.lifetime;


            }
            x=>{return Err(Error::CannotDeref(x.clone()));}
            }
    }


        let slot = Slot{tipe:current_type.clone(), lifetime:current_lifetime.clone()};
        //self.0.insert(lval.ident.clone(), slot);
        return Ok(slot);

}
```

I assumed that this function takes in an Lval, and returns its type. Essentially we walk down the path of deferences, if we find an `Undefined` then that means that the Lval has already been moved out and we throw an error. Otherwise we return the fully deferenced type within a slot

`contained`

```
pub fn contained(&self, var: &Ident) -> Option<&Type> {

 let Some(slot) = self.0.get(var) else {panic!("Impossible")};
 let mut current_type = &slot.tipe;
 loop{

  match current_type{
  Type::TBox(x)=> {current_type=&*x;}
  Type::Undefined(x) =>{return None;}
  x=>{return Some(&x);},


  }

 }

 return None;

}
```

I used the following definition as inspiration :

$$\text{contains}(\Gamma, \widetilde{T}, T_y) = \begin{cases} \text{contains}(\Gamma, \widetilde{T}', T_y) & \text{if } \widetilde{T} = \Box\widetilde{T}' \\ \text{true} & \text{if } \widetilde{T} = T_y \\ \text{false} & \text{otherwise} \end{cases}$$

Essentially we recursively go inside each Box until we either arrive at a type or an undefined. If we arrive at an undefined we return `None` otherwise we return the type

`read_prohibited`

```
pub fn read_prohibited(&self, lval: &Lval) -> bool {

 for vars in self.0.keys(){
 let value = self.contained(vars);
 if let Some(Type::Ref(x, Mutable::Yes)) = value{
   if x.ident==lval.ident{

    return true;
    }
   }}



return false;
}
```

I came up with the following Definition from the Paper:

$$ReadProhibited = \{x|\Gamma \vdash x \leadsto \&mut\ \overline{u}\ \land \exists(PathConflict(x)\}$$

Where a Path conflict is defined as :

`$$PathConflict = \{ w| \exists x: x\leadsto{y} \land w\leadsto{y}   \}$$`

Essientiallly if there is a series of deferences between two Lvals that Point to the same variable we have a Path Conflict.

Thus if there is a Path Conflict AND the lval is a mutable reference, then the Lval is read prohibited

`write_prohibited`

```
    pub fn write_prohibited(&self, lval: &Lval) -> bool {
        if self.read_prohibited(lval){
            return true;
        }


        for vars in self.0.keys(){
            let value = self.contained(vars);
            if let Some(Type::Ref(x, y)) = value{
                if *x.ident==lval.ident{

                    return true;
                    }
                }}
        return false;

    }
```

I came up with the following Definition from the Paper:

$$WriteProhibitied\{x|ReadProhibited(x) \vee (x \rightsquigarrow \&\overline{y} \wedge Pathconflict(x))\}$$

Thus if the Lval is read prohibited OR Path Conflict AND the lval is a reference, then the Lval is Write Prohibited

`moove`

```
pub fn moove(&mut self, lval: &Lval) -> TypeResult<()> {
 if self.write_prohibited(lval){
 return Err(Error::MutBorrowAfterBorrow(lval.clone()));}

 let  Some(current) = self.0.get(&lval.ident) else {return Err(Error::UnknownVar(lval.ident.clone()))};
 let result = self.moove_nested(current.tipe.clone(), lval.derefs.clone())?;
 if let (tipe) = result{

  let  Some(current) = self.0.get(&lval.ident) else {return Err(Error::UnknownVar(lval.ident.clone()))};
  let slot = Slot{tipe:tipe, lifetime:current.lifetime.clone()};
  self.0.insert(lval.ident.clone(), slot);
  return Ok(())

 }
return  Err(Error::Dummy);

}
```

$$\frac{\Gamma_1 \vdash w : \langle T\rangle^m \quad \neg\text{writeProhibited}(\Gamma_1, w) \quad \Gamma_2 = \text{move}(\Gamma_1, w)}{\Gamma_1 \vdash \langle w : T\rangle^1_\sigma \triangleright \Gamma_2}$$

Essentially if the Lval w is write prohibited then we can perform a move. Where a Move is defined as:

$$\frac{read(S1, w) = <v>^m \quad S_2 = write(S_1, w, \bot)}{S_1B\ w \to S_2Bv}$$

Although this is a Sementic Rule, We can infer that we have to walk down the path of dereferences and perform a destructive read in our Lval, like so:

```
pub fn moove_nested(&mut self, tipe:Type, i:usize) ->  Result<Type, Error> {
  if i ==0{
  return Ok(Type::Undefined(Box::new(tipe)))}

 match tipe{

  Type::TBox(x) =>{
  let (rest) = self.moove_nested(*x,i-1)?;

  return Ok(Type::TBox(Box::new(rest)))},
  Type::Ref(lval, _) =>{ return Err(Error::MoveBehindRef(lval.clone()))},
  Type::Undefined(x)=>{return Err(Error::Dummy)},
  x=>{return Err(Error::CannotDeref(x))}
  }


}
```

muut

```
pub fn muut(&self, lval: &Lval) -> bool {

let  Some(tipe) = self.contained(&lval.ident) else{return false};

let mut current_type = tipe;
let mut i = lval.derefs;
while i >0{

 match current_type{
 Type::Ref(lv, Mutable::Yes) =>{
   i-=1;
   let  Some(tipe) = self.contained(&lv.ident) else {return true;};

   current_type = tipe;},
 Type::Ref(lv, Mutable::No) =>{return false;},

 Type::TBox(x) =>{
   current_type= &*x},
 Type::Undefined(x)=>{
   current_type = &*x},
 _ =>{return true;}


 }

}
return true;
}
```

$$\begin{aligned}\text{mutable}(\Gamma, \epsilon \mid \mathbb{T}) &= \text{true}\\ \text{mutable}(\Gamma, (\pi \cdot *) \mid \mathbb{T}) &= \text{mutable}(\Gamma, \pi \mid \mathbb{T})\\ \text{mutable}(\Gamma, (\pi \cdot *) \mid \&\texttt{mut}\ w) &= \bigwedge_i \text{mut}(\Gamma, \pi \cdot w_i)\end{aligned}$$

All we have to do is recursively go down our path of derefences until we hit a mutable reference. If we don't hit one, then we simply return false

compatible

```
    pub fn compatible(&self, t1: &Type, t2: &Type) -> bool {

            match (t1,t2){
            (Type::Int, Type::Int) => {return true;},
            (Type::Unit, Type::Unit) => {return true;},
            (Type::TBox(x), Type::TBox(y))=>{self.compatible(x,y)},
            (Type::Ref(x,Mutable::Yes), Type::Ref(y, Mutable::Yes))=>{return true;},
            (Type::Ref(x,Mutable::No), Type::Ref(y, Mutable::No))=>{return true;},
            (Type::Undefined(x), y) =>{self.compatible(x, y)},
            (y,Type::Undefined(x)) =>{self.compatible(x, y)},
            _=>{return false;}


            }

    }
```

Although I could not find a compatibility rule within our paper, I went ahead and assumed that two types are compatible if they have the same nested definition

write and update

```
    pub fn write(&mut self, w: &Lval, tipe: Type) -> TypeResult<()> {
        //self.0.insert(&w.ident, tipe);
        let current = self.0.remove(&w.ident)
            .ok_or(Error::UnknownVar(w.ident.clone()))?;


         let (rest) = self.update(current.tipe.clone(), tipe, w.derefs.try_into().unwrap())?;

        let slot = Slot{tipe:rest, lifetime:current.lifetime.clone()};
        self.0.insert(w.ident.clone(), slot);
        Ok(())
    }
```

We simply get the type of our `lval` here ,the real magic happens in our update function

```
pub fn update(&mut self, old: Type, new: Type, i: i32) -> Result<Type, Error>{
 if i ==0{
    return Ok(new);
 }


 match old{
 Type::TBox(x) =>{
 let (replaced)= self.update(*x,new,i-1)?;
 return Ok(Type::TBox(Box::new(replaced)))
   },
 Type::Ref(lv, Mutable::Yes) =>{

     self.write(&lv, new.clone())?;

   Ok(Type::Ref(lv,Mutable::Yes))
  },

 Type::Ref(lv, Mutable::No) =>{
   return Err(Error::UpdateBehindImmRef(lv.clone()));

  },
 Type::Undefined(x)=>{

 let (replaced)= self.update(*x,new,i-1)?;
 return Ok(Type::Undefined(Box::new(replaced)))
```

```
  }
```

```
 x=>{Ok(new)}}}
```

According to rule in the Final Project Landing Page: $$

$$\begin{aligned}
\text{update}(\Gamma, \epsilon, \tau_1, \tau_2) &= (\Gamma, \tau_2)\\
\text{update}(\Gamma, *\pi, \Box\tau_1, \tau_2) &= (\Gamma', \Box\tau_1') \quad \text{where } (\Gamma', \tau_1') = \text{update}(\Gamma, \pi, \tau_1, \tau_2)\\
\text{update}(\Gamma, *\pi, \&\texttt{mut}\ u, \tau_2) &= (\Gamma', \&\texttt{mut}\ u) \quad \text{where } \Gamma' = \text{write}(\Gamma, \pi u, \tau_2)
\end{aligned}$$

$w =$

write$(\Gamma, w, \tau) = \Gamma'[x \mapsto \langle\sigma'\rangle^l]$ where
$\pi \mid x$
$\Gamma(x) = \langle\sigma\rangle^l$
$(\Gamma', \sigma') = \text{update}(\Gamma, \pi, \sigma, \tau)$

$$

We Recursively walk down the path of each Lval, if we encounter another Lval along the way we nest our selves deeper into the mix, Although The rule doesn't mention immutable references. I decided that it was appropriate to throw an error if we come across an immutable reference. Since we cannot write to an immutable reference

drop

```
pub fn drop(&mut self, l: Lifetime) {
 let mut to_drop = Vec::new();
 for value in self.0.keys(){
 let Some(slot) = self.0.get(value) else{panic!("Impossible");};
 if slot.lifetime==l{
 to_drop.push(value.clone());}}

 for value in to_drop{
  self.0.remove(&value);

  }}}
```

This is essentially the same definition as our semantic one with only slight modifications

$$\begin{aligned}
\text{drop}(S, \overrightarrow{\ }) &= S \\
\text{drop}(S, \psi \cup \{v^{\perp}\}) &= \text{drop}(S, \psi) \quad \text{if } v^{\perp} \neq \ell^{\bullet} \\
\text{drop}(S, \psi \cup \{\ell^{\bullet}\}) &= \text{drop}(S - \{\ell \mapsto \langle v^{\perp}\rangle^{*}\},\ \psi \cup \{v^{\perp}\}) \quad \text{where } S(\ell) = \langle v^{\perp}\rangle^{*}
\end{aligned}$$

The only deference now being we are dropping by lifetime not by the `lval` thus we iterate over each value in our environment and if it matches the given life time we remove it from our environment!

Type Checking Part 2

Basic Definitions

```
#[derive(PartialEq)]
#[derive(Debug)]
   pub enum Error {
    Dummy,
    UnknownVar(String),
    CannotDeref(Type),
    MovedOut(Lval),
    MoveBehindRef(Lval),
    UpdateBehindImmRef(Lval),
    CopyAfterMutBorrow(Lval),
    MoveAfterBorrow(Lval),
    MutBorrowBehindImmRef(Lval),
    MutBorrowAfterBorrow(Lval),
    BorrowAfterMutBorrow(Lval),
    Shadowing(String),
    IncompatibleTypes(Type, Type),
    LifetimeTooShort(Expr),
    AssignAfterBorrow(Lval),
}

type TypeResult<T> = Result<T, Error>;
#[derive(Debug)]
#[derive(PartialEq)]
#[derive(Clone)]
pub struct Context {
    pub env: Env,
    // TODO: anything else you need
}
```

Implementing the Context

`well_formed`

```
pub fn well_formed(&self, tipe: &Type, l: Lifetime) -> bool {
 match tipe{
 Type::Unit=>{true},
 Type::Int=>{true},
 Type::TBox(x)=> {self.well_formed(x,l)},
 Type::Ref(lv, Mutable)=>{
 let Some(slot) = self.env.0.get(&lv.ident) else{return false};
 self.lifetime_contains(slot.lifetime.clone(), l)}
 Type::Undefined(x)=>{return false;}

  }
}
```

We know that a type is well formed if it respects the following rules :

This is an axiom, an int always lives as long as a given lifetime

$$\frac{}{\Gamma \vdash \texttt{int} \geq l} \quad \text{(L-Int)}$$

Essentially if the Reference lives at least as long as our Borrow

$$\frac{\Gamma \vdash u : \langle T \rangle^m \quad m \geq l}{\Gamma \vdash \&[\texttt{mut}]\ u \geq l} \quad \text{(L-Borrow)}$$

If it is a Box we recourse within $$ $\Gamma \vdash T \geq l_{\overline{\Gamma \vdash \Box T \geq l \quad \text{(L-Box)}}}$

$$ For an Undefined, I decided that it would be appropriate to always return false, since we essentially are ‘pretending’ that the type no longer exists

`type_expr`

Base Cases:

```
  Expr::Unit => Ok(Type::Unit),
  Expr::Int(_) => Ok(Type::Int),
```

`Expr::Lv(x, Copyable::Yes)=>`

```
Expr::Lv(lv, Copyable::Yes) => {

 let slot = self.env.type_lval(lv)?;
 Ok(slot.tipe)
}
```

Because the function is Copyable, there is really not much to check

`Expr::Lv(x, Copyable::No)=>`

```
Expr::Lv(lv, Copyable::No) => {


 let slot = self.env.type_lval(lv)?;
 let ty   = slot.tipe.clone();
 let contained = self.env.contained(&lv.ident);

  if None== contained {
   return Err(Error::MovedOut(lv.clone()));
  }
 if let Some(Type::Ref(x, _))= contained{

  return Err(Error::MoveBehindRef(lv.clone()));}
 if Self::is_copyable(&ty) {
  if self.env.read_prohibited(lv) {
   return Err(Error::CopyAfterMutBorrow(lv.clone()));
  }
  *expr = Expr::Lv(lv.clone(), Copyable::Yes);
  return Ok(ty);
 }

 if self.env.write_prohibited(lv) {
    return Err(Error::MoveAfterBorrow(lv.clone()));
 }
 self.env.moove(lv)?;
 Ok(ty)

}
```

Because this Lval is not Copyable we must perform a move:

T-Move

$$\frac{\Gamma_1 \vdash w : \langle T \rangle^m \quad \neg\text{writeProhibited}(\Gamma_1, w) \quad \Gamma_2 = \text{move}(\Gamma_1, w)}{\Gamma_1 \vdash \langle w : T \rangle^1_\sigma \triangleright \Gamma_2}$$

Here we ensure many more things one is that the `lval` is not write prohibited, we have to ensure that it has not already been moved out, that we are not moving behind a reference and finally that the inner type is indeed copyable and not Not Write prohibited

`Expr::Borrow(x, Mutable::Yes)`

```
Expr::Borrow(lv, Mutable::Yes) => {
 let slot = self.env.type_lval(lv)?;
 let ty   = slot.tipe.clone();
 let contained = self.env.contained(&lv.ident);
 if self.env.write_prohibited(lv) {
  return Err(Error::MutBorrowAfterBorrow(lv.clone()));
 }
 if None== contained {
   return Err(Error::MovedOut(lv.clone())); }

 if !self.env.muut(lv) {
  return Err(Error::MutBorrowBehindImmRef(lv.clone()));}
  Ok(Type::Ref(lv.clone(), Mutable::Yes))
}
```

$$\frac{\Gamma \vdash w : \langle T \rangle^m \quad \text{mut}(\Gamma_1, w) \quad \neg\text{writeProhibited}(\Gamma, w)}{\Gamma \vdash \langle \&\text{mut } w : \&\text{mut } w \rangle^1_\sigma \triangleright \Gamma}(\text{T-MutBorrow})$$

We have to ensure that the value is not write prohibited, that it has not already been moved out AND that the inner reference is not immutable

`Expr::Borrow(x, Mutable::No)`

```
Expr::Borrow(lv, Mutable::No) => {
 let slot = self.env.type_lval(lv)?;
 let contained = self.env.contained(&lv.ident);
 let ty   = slot.tipe.clone();
 if self.env.read_prohibited(lv) {
  return Err(Error::BorrowAfterMutBorrow(lv.clone()));
 }
  if None== contained {
   return Err(Error::MovedOut(lv.clone()));
  }

 Ok(Type::Ref(lv.clone(), Mutable::No))
}
```

$$\frac{\Gamma \vdash w : \langle T \rangle^m \quad \neg\text{readProhibited}(\Gamma, w)}{\Gamma \vdash \langle \&w : \&w \rangle^1_\sigma \triangleright \Gamma}\text{(T-ImmBorrow)}$$

We have to ensure that the value is not read prohibited AND that it has not already been moved out

`Expr:: OBox(e1)`

```
Expr::OBox(e) => {
 let inner_type = self.type_expr(e)?;
 Ok(Type::TBox(Box::new(inner_type)))
}
```

$$\frac{\Gamma_1 \vdash \langle t : T \rangle^l_\sigma \triangleright \Gamma_2}{\Gamma_1 \vdash \langle \mathsf{box}\ t : \Box T \rangle^l_\sigma \triangleright \Gamma_2}$$

Essentially, according to the rule, we evaluate the type within our Box

`Expr::Block(statements, e1, l1)`

```
Expr::Block(stmts, result_expr, lifetime) => {
 for stmt in stmts {
  self.type_stmt(stmt)?;
  if let Stmt::LetMut(x, _) = stmt{
    let Some(ty_of) = self.env.0.get(x) else {panic!("Impossible");};;
    let slot = Slot{tipe:ty_of.tipe.clone(), lifetime:lifetime.clone()};
    self.env.0.insert(x.to_string(), slot);}

  if let Stmt::Assign(lv, expr) = stmt{
   let slot = self.env.type_lval(&lv)?;
   if slot.lifetime < *lifetime{
   return Err(Error::LifetimeTooShort(expr.clone()));
   }
   }

  }
 let t = self.type_expr(result_expr)?;


 self.env.drop(lifetime.clone());
 Ok(t)
}
```

Unfortunately Due to the Structure of the Code we Need to Employ Some Work arounds, however the Type Checking for each block remains the same

$$\frac{\Gamma_1 \vdash \langle t : T\rangle^m_\sigma \triangleright \Gamma_2 \quad \Gamma_2 \vdash T \geq l \quad \Gamma_3 = \text{drop}(\Gamma_2, m)}{\Gamma_1 \vdash \langle \{t\}^m : T\rangle^l_\sigma \triangleright \Gamma_3}$$

We type-check each expression within our block and then drop the lifetime. One of the Main differences is the 'sub evaluation' of the Let Rule:

$$\frac{x \notin \text{dom}(\Gamma_1) \quad \Gamma_1 \vdash \langle t : T\rangle^1_\sigma \triangleright \Gamma_2 \quad \Gamma_3 = \Gamma_2[x \mapsto \langle T\rangle^1]}{\Gamma_1 \vdash \langle \texttt{let mut } x = t : \epsilon\rangle^1_\sigma \triangleright \Gamma_3}\text{(T-Declare)}$$

Because our `type_stmt` does not take in a life time parameter, each time we encounter a `LetMut` in our block, we manually have to go back and set the lifetime after calling `type_stmt`

We also have to ensure that any time that we assign in our block that the lifetime of the assigned `lval` is greater than or equal to the lifetime of our hole block. Since otherwise the `lval` would not live until the entire execution of our program

`type_stmt`

`Stmt::Assign`

```
Stmt::Assign(lv, expr) =>{
let old_type = self.env.type_lval(lv)?;
println!("{:?}",old_type);
let new_type = self.type_expr(expr)?;
if !self.env.compatible(&new_type,&old_type.tipe){
 return Err(Error::IncompatibleTypes(old_type.tipe, new_type)); }
```

```
else if self.env.write_prohibited(lv){
 return Err(Error::AssignAfterBorrow(lv.clone()));}
self.env.write(lv, new_type)?;
return Ok(());
```

$$\frac{\Gamma_1 \vdash w : \langle \widetilde{T_1} \rangle^m \quad \Gamma_1 \vdash \langle t : T_2 \rangle^1_\sigma \triangleright \Gamma_2 \quad \Gamma_2 \vdash \widetilde{T_1} \approx T_2 \quad \Gamma_2 \vdash T_2 \succeq m \quad \Gamma_3 = \text{write}^\theta(\Gamma_2, w, T_2) \quad \neg\text{writeProhibited}(\Gamma_3, w)}{\Gamma_1 \vdash \langle w = t : \epsilon \rangle^1_\sigma \triangleright \Gamma_3}\text{(T-Assign)}$$

For assignments we have to ensure several things, we first use the context to get the old type then we evaluate the new type. Then we have to ensure that the new type T_2 and the old type T_1 are compatible, we also have to ensure that the lifetime of the new old type m is atleast as long as the lifetime of our new type (we perform this check within `type_expr`) and finally we ensure that our `lval` is not write prohibited. If all conditions are met, we can write to our context #### `Stmt::LetMut`

```
Stmt::LetMut(x, expr) =>{
if self.env.0.contains_key(x) {
 return Err(Error::Shadowing(x.to_string()));}

let new_type = self.type_expr(expr)?;
self.env.insert(x,new_type, Lifetime::global());
```

$$\frac{x \notin \text{dom}(\Gamma_1) \quad \Gamma_1 \vdash \langle t : T \rangle^1_\sigma \triangleright \Gamma_2 \quad \Gamma_3 = \Gamma_2[x \mapsto \langle T \rangle^1]}{\Gamma_1 \vdash \langle \texttt{let mut } x = t : \epsilon \rangle^1_\sigma \triangleright \Gamma_3}\text{(T-Declare)}$$

We have to ensure that the variable x is fresh, denoted by $x \notin dom(\Gamma_1)$, then we evaluate the expression within x and map it within our context!

Lexicalization

Admittedly my least favorite part of the project. I simply dislike string processing.

Basic Definitions

```
use std::str::Lines;
#[derive(Clone)]
#[derive(Debug)]
#[derive(PartialEq)]
pub enum Token {
    Lparen,
    Rparen,
    Lbracket,
    Rbracket,
    Eq,
    Ampersand,
    Star,
    Comma,
    Semicolon,
    Fn,
    Let,
    Mut,
    Box,
    Int(i32),
    Var(String),
}
```

```
pub struct Lexer<'a> {
    contents: Lines<'a>,
    curr_line_num: usize,
    curr_col_num: usize,
    curr_line: &'a str,
}

const LEXEMES : [(&str, Token); 13] = [
    ("(", Token::Lparen),
    (")", Token::Rparen),
    ("{", Token::Lbracket),
    ("}", Token::Rbracket),
    ("=", Token::Eq),
    ("&", Token::Ampersand),
    ("*", Token::Star),
    (",", Token::Comma),
    (";", Token::Semicolon),
    ("fn", Token::Fn),
    ("let", Token::Let),
    ("mut", Token::Mut),
    ("Box::new", Token::Box),
];

#[derive(Debug)]
pub enum Error {
    Unknown(usize, usize),
}
```

```
type LexResult = Result<Token, Error>;
```

variable

```
fn variable(&mut self) -> LexResult {
 let mut len =0;
 for character in self.curr_line.chars(){
   if character.is_ascii_alphanumeric() || character == '_' {
    len += 1;
   } else {
    break;
   }
  }
 if len ==0{
    return Err(self.unknown());
   }
 let var = &self.curr_line[..len];
 self.consume(len);
 return Ok(Token::Var(var.to_string()));

 }
```

Essentially we loop through each character in our current line, if we do not encounter a white space we simply add to `len` and slice it then consume that part of the current line.

int

```
fn int(&mut self) -> LexResult {
let mut len =0;
 for character in self.curr_line.chars(){
  if character.is_ascii_digit(){
     len+=1;
    }
  else{
     break;
    }

   }
 if len ==0{
    return Err(self.unknown());
   }
 let var = &self.curr_line[..len];
 let v =  var.parse::<i32>().unwrap();
  self.consume(len);
 return Ok(Token::Int(v));

  }
```

Same idea as before, only difference is that we break is we see something that isn't a digit!

skip_whitespace

```
fn skip_whitespace(&mut self){
 loop{
  if self.curr_line.is_empty() || self.curr_line.starts_with("//"){
   let Some(temp) = self.contents.next() else {return};

   self.curr_line= temp;
   self.curr_line_num += 1;
   self.curr_col_num = 1;
   continue;
  }

  let Some(ch) = self.curr_line.chars().next() else{return;};

  if ch.is_whitespace(){
  self.consume(1);}
  else {
   break;
  }
   } }
```

This is a function I incorporated that simply skips white spaces and and empty lines!

Parsing

`parse_block`

```
pub fn parse_block(&mut self) -> ParseResult<Expr> {
eprintln!("[parse_block] entering block parse");
self.next_token_match(Token::Lbracket)?;
let mut statements = Vec::new();
while *self.peek_token()? != Token::Rbracket {
 eprintln!("[parse_block] parsing statement, peek: {:?}", self.peek_token());
 let temp = self.parse_stmt()?;
 eprintln!("[parse_block] parsed statement: {:?}", temp);
 statements.push(temp);
}
eprintln!("[parse_block] parsing tail expression");
if let (Stmt::Expr(last)) = &statements[statements.len()-1]{

 let Some(Stmt::Expr(last)) = statements.pop() else {panic!("Impossible");};
 self.next_token_match(Token::Rbracket)?;
 eprintln!("[parse_block] exiting block parse");

 self.new_lifetime();
 return Ok(Expr::Block(statements, Box::new(last), Lifetime(self.lifetime)));

}
//All Functions return Unit;
self.next_token_match(Token::Rbracket)?;
eprintln!("[parse_block] exiting block parse");
self.new_lifetime();
return   Ok(Expr::Block(statements, Box::new(Expr::Unit), Lifetime(self.lifetime)));
}
```

A block is essentially a scope, Thus we loop through our code and we keep calling `parse_stmt` until we see a `}`. In which case we break out of the while loop and we check the last item within our vector of ‘statements’ to see if it is a expression or a statement. If it is a ‘expression’, we assign it to be the last expression within our Block, otherwise we assign a `Unit` to be the last expression within our block. (Since if a function does not return an explicit value it will always return a Unit). Any time we return a block, we must ensure a new unique lifetime, hence we call our `self.new_lifetime` function to ensure this!

`parse_box`

```
fn parse_box(&mut self) -> ParseResult<Expr> {
 eprintln!("[parse_box] parsing box expression");
 self.next_token_match(Token::Box)?;
 self.next_token_match(Token::Lparen)?;
 let e = self.parse_expr()?;
 self.next_token_match(Token::Rparen)?;
 eprintln!("[parse_box] parsed inner expr: {:?}", e);
 Ok(Expr::OBox(Box::new(e)))
}
```

This one is Pretty simple, essentially we check if the first Token is a `Box` and the very next token is a `)` if so, we parse the inner expression and ensure that the last token is a `)` then we simply return the expression!

`parse_stmt`

`Token::Star | Token::Var(_)`

```
Token::Star | Token::Var(_) => {
 eprintln!("[parse_stmt] star found");
 let lv = self.parse_lval()?;
 if *self.peek_token()? == Token::Eq {
  eprintln!("[parse_stmt] assignment detected");
  self.next_token()?;
  let next_expr = self.parse_expr()?;

  self.next_token_match(Token::Semicolon)?;
  Stmt::Assign(lv, next_expr)
 } else {
  self.next_token_match(Token::Semicolon)?;
  Stmt::Expr(Expr::Lv(lv, Copyable::No))
 }
}
```

Whether we have a `*` or a `Var(x)` we know we are dealing with an lval, thus we call our `parse_lval()` function. If the very next token is a "=" then we know we have an assignment in which case we parse the very next thing after the equal sign by calling `self.parse_expr()` . If we do not see an equal sign then we know we are dealing with a simple `lval` in which case we return it and assign it to not be copy-able.

`Token::Let =>`

```
Token::Let => {
 eprintln!("[parse_stmt] let found");
 self.next_token()?;
 self.next_token_match(Token::Mut)?;
 let var = self.next_token_var()?;
 self.next_token_match(Token::Eq)?;
 let next_expr = self.parse_expr()?;
 self.next_token_match(Token::Semicolon)?;
 Stmt::LetMut(var, next_expr)
},
```

When we encounter a `Let` , we assert that the next token is a `mut` (I do this because we did not define not mutable declarations within our spec nor did I see a definition within the paper). Afterwards we assert that we see a variable next. Then we parse the inner expressions by calling `self.parse_expr()`

Else Case

```
_ => {
 eprintln!("[parse_stmt] fallback expr stmt");
 let next_expr = self.parse_expr()?;
 self.next_token_match(Token::Semicolon)?;
 Stmt::Expr(next_expr)
}
```

If we do not see any of the previous cases then we know we are dealing with a expression not a statement!

`parse_expr`

Base Cases

```
  Token::Int(x) => {
  eprintln!("[parse_expr] int literal: {}", x);
  let Token::Int(v) = self.next_token()? else { unreachable!() };
  Expr::Int(v)
 }
 Token::Var(_) | Token::Star => {
  eprintln!("[parse_expr] lval expression");
  let lv = self.parse_lval()?;
  Expr::Lv(lv, Copyable::No)
 }
 Token::Box => {
  return self.parse_box();
 }
```

Pretty Self Explanatory, If we see an int we return an int and so on.

`Token::Ampersand`

```
Token::Ampersand => {
 eprintln!("[parse_expr] borrow expression");
 self.next_token()?;
 let mut muta = Mutable::No;
 if *self.peek_token()? == Token::Mut {
  self.next_token()?;
  muta = Mutable::Yes;
 }
 let inner_expr = self.parse_lval()?;
 Expr::Borrow(inner_expr, muta)
}
```

If we see an Ampersand then we know we are dealing with a borrow, then if next thing we see is a mute token then we can confirm it is a mutable borrow and set `muta = Mutable::Yes` otherwise `muta = Mutable::No` . Then we parse the inner l-val by calling `self.parse_lval()`

`Token::Lbracket`

```
 Token::Lbracket => {
  return self.parse_block();
 },
```

If we see a `{` we simply have a Block so we fallback to `self.parse_block()`

`Token::SemiColon`

```
Token::Semicolon => Expr::Unit,
```

I allowed a simple empty semicolon to be of type unit, since all items in rust must have a type !

`parse_lval`

```
pub fn parse_lval(&mut self) -> ParseResult<Lval> {
 eprintln!("[parse_lval] entering lval parse, peek: {:?}", self.peek_token());
 let mut derefs = 0;
 while *self.peek_token()? == Token::Star {
  self.next_token()?;
  derefs += 1;
 }

 let var = self.next_token()?;
 if let Token::Var(x) = var {
  eprintln!("[parse_lval] exiting lval parse: {} with {} derefs", x, derefs);
  Ok(Lval { ident: x, derefs })
 } else {
  eprintln!("Got unexpected in LVAl");
  Err(Error::Unexpected(var))
 }
}
```

This function somewhat simple, all we do is each time we see a * we add to the total number of deferences then we assert that we see a variable and return the Lval with proper number of recorded dereferences!

Finally

Our Parser Should be able to Parse the following Expression

```
fn main () {
  let mut y =6;
  y =8888;
  //This is a comment;


  let mut x = &y;
  let mut z = &x;
  {let mut i = 9;
  };
  }
```

Evaluation

For evaluation I chose to use the test that you gave me, with a few minor tweaks to the definition to fit my implementation - For example you used `Type::boxx` which I changed to `Type:TBox` - Overall the Logic Should be the exact same, I double checked of this but apologies if I missed anything peculiar

Here is the output on all the tests by running `cargo test`

```
test tests::tests1::tests::drop_owned ... ok
test tests::tests1::tests::drop_unowned ... ok
test tests::tests1::tests::drop_larger_example ... ok
test tests::tests1::tests::eval_block ... ok
test tests::tests1::tests::eval_block_mut_ref ... ok
test tests::tests1::tests::eval_block_ref ... ok
test tests::tests1::tests::eval_box ... ok
test tests::tests1::tests::eval_box_box ... ok
test tests::tests1::tests::eval_assign_copy ... ok
test tests::tests1::tests::eval_assign_move ... ok
test tests::tests1::tests::eval_copy ... ok
test tests::tests1::tests::eval_assign_move_deref ... ok
test tests::tests1::tests::eval_expr_stmt ... ok
test tests::tests1::tests::eval_assign_replace ... ok
test tests::tests1::tests::eval_let_mut ... ok
test tests::tests1::tests::eval_lits ... ok
test tests::tests1::tests::eval_move ... ok
test tests::tests1::tests::locate_ref ... ok
test tests::tests1::tests::locate_panic - should panic ... ok
test tests::tests1::tests::locate_var ... ok
test tests::tests1::tests::read_panic - should panic ... ok
test tests::tests1::tests::read_ref_owned ... ok
test tests::tests1::tests::read_var ... ok
test tests::tests1::tests::write_deref_read_diff ... ok
test tests::tests1::tests::write_panic - should panic ... ok
test tests::tests2::tests::basic_write_prohibited ... ok
test tests::tests2::tests::basic_write_prohibited_2 ... ok
test tests::tests2::tests::compatible_basic ... ok
test tests::tests2::tests::compatible_basic_fail ... ok
test tests::tests2::tests::compatible_refs ... ok
test tests::tests2::tests::env_contained ... ok
test tests::tests2::tests::drop_basic ... ok
test tests::tests1::tests::write_two_deref ... ok
test tests::tests3::type_tests::assign_err_borrow ... ok
test tests::tests2::tests::mut_succ ... ok
test tests::tests3::type_tests::assign_err_update_imm ... FAILED
test tests::tests3::type_tests::assign_ok_ref ... ok
test tests::tests2::tests::env_lval_box ... ok
test tests::tests2::tests::env_var ... ok
test tests::tests2::tests::basic_read_prohibited ... ok
test tests::tests2::tests::env_lval_ref ... ok
test tests::tests2::tests::env_contained_undefined ... ok
test tests::tests3::type_tests::cannot_copy ... ok
test tests::tests2::tests::move_under_ref ... ok
test tests::tests3::type_tests::cannot_move ... ok
test tests::tests3::type_tests::declare_ok ... ok
test tests::tests3::type_tests::declare_shadow ... ok
test tests::tests3::type_tests::imm_borrow_err ... ok
test tests::tests3::type_tests::assign_err_incompat ... ok
test tests::tests3::type_tests::imm_borrow_ok ... ok
test tests::tests3::type_tests::invalid_lval ... ok
test tests::tests3::type_tests::keep_move ... ok
test tests::tests3::type_tests::imm_borrow_err_moved_out ... ok
test tests::tests3::type_tests::mut_borrow_err_through_imm_ref ... ok
test tests::tests3::type_tests::mut_borrow_ok ... ok
test tests::tests3::type_tests::still_moved_out ... ok
test tests::tests2::tests::mut_fail ... ok
test tests::tests3::type_tests::assign_err_moved_out ... ok
test tests::tests3::type_tests::type_box ... ok
test tests::tests3::type_tests::assign_move_in ... ok
test tests::tests3::type_tests::assign_err_unknown ... ok
test tests::tests3::type_tests::assign_ok ... ok
test tests::tests3::type_tests::block_err_lifetime ... ok
test tests::tests2::tests::write_basic ... ok
test tests::tests3::type_tests::block_ok ... ok
test tests::tests3::type_tests::declare_moved_out ... ok
test tests::tests3::type_tests::copied ... ok
test tests::tests2::tests::write_ref ... ok
test tests::tests3::type_tests::type_value ... ok
test tests::tests2::tests::move_under_box ... ok
test tests::tests3::type_tests::make_copy ... ok
test tests::tests3::type_tests::move_behind_ref ... ok
test tests::tests3::type_tests::moved_out ... ok
test tests::tests3::type_tests::mut_borrow_err_already_borrowed ... ok
test tests::tests3::type_tests::mut_borrow_through_ref ... ok
test tests::tests3::type_tests::mut_borrow_err_moved_out ... ok

failures:

---- tests::tests3::type_tests::assign_err_update_imm stdout ----
Slot { tipe: Int, lifetime: Lifetime(0) }
```

```
thread 'tests::tests3::type_tests::assign_err_update_imm' panicked at src/tests/tests3.rs:459:9:
assertion `left == right` failed
  left: Ok(())
 right: Err(UpdateBehindImmRef(Lval { ident: "z", derefs: 2 }))


failures:
    tests::tests3::type_tests::assign_err_update_imm

test result: FAILED. 75 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.01s
```

As you can see I am failing one test, Which I could not for the life of me get it to pass. When I did, I would proceed to fail some other test