Please do not turn the page until everyone is ready.

Rules:

- The exam is closed-book, closed-note, except for one 8.5x11in piece of paper.
- Please stop promptly at 17:15.
- You can rip apart the pages, but please write your name on each page.
- There are 162 points total, distributed unevenly among 8 questions. The maximum number of points you can receive is 100 (if your total score exceeds this amount, you will receive 100).
- Most questions have multiple parts. You will receive points for any parts you complete.
- You are not expected to complete all questions.

Advice:

- Read questions carefully. Understand a question before you start writing.
- Write down thoughts and intermediate steps so you can get partial credit.
- The questions are not necessarily in order of difficulty. Skip questions you are not confident about and if you have time, come back to them later. Remember you only need 100 points total.
- If you have questions, ask.
- Relax. You are here to learn.
For your reference (page 1 of 2):

\[
\begin{align*}
\text{e} & ::= \lambda x. \text{e} \mid x \mid \text{e} \mid c \mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid e.l_i \mid \text{fix } e \\
\text{v} & ::= \lambda x. e \mid c \mid \{l_1 = v_1, \ldots, l_n = v_n\} \\
\tau & ::= \text{int} \mid \tau \rightarrow \tau \mid \{l_1 : \tau_1, \ldots, l_n : \tau_n\}
\end{align*}
\]

\text{e} \rightarrow \text{e}' \text{ and } \Gamma \vdash e : \tau \text{ and } \tau_1 \leq \tau_2

\[
\begin{array}{l}
(\lambda x. e) v \rightarrow e[v/x] \\
e_1 \rightarrow e'_1 \\
e_2 \rightarrow e'_2 \\
\text{fix } e \rightarrow \text{fix } e' \\
\text{fix } \lambda x. e \rightarrow e[\text{fix } \lambda x. e/x]
\end{array}
\]

\[
\begin{align*}
\{l_1 = v_1, \ldots, l_n = v_n\}, l_i & \rightarrow v_i \\
\{l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, l_i = e_i, \ldots, l_n = e_n\} & \rightarrow \{l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, l_i = e'_i, \ldots, l_n = e_n\}
\end{align*}
\]

\[
\begin{array}{l}
\Gamma \vdash c : \text{int} \\
\Gamma \vdash x : \Gamma(x) \\
\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2 \\
\Gamma \vdash e_1 \vdash e_2 : \tau_1 \\
\Gamma \vdash \text{fix } e : \tau \\
\Gamma \vdash e_1 : \tau_1 \\
\vdots \\
\Gamma \vdash e_n : \tau_n \quad \text{labels distinct} \\
\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \\
\Gamma \vdash e : \tau \quad \tau \leq \tau' \\
\Gamma \vdash e : \tau'
\end{array}
\]

\[
\{l_1 : \tau_1, \ldots, l_n : \tau_n, l : \tau\} \leq \{l_1 : \tau_1, \ldots, l_n : \tau_n\}
\]

\[
\{l_1 : \tau_1, \ldots, l_i : \tau_i, l_{i+1} : \tau_{i+1}, \ldots, l_n : \tau_n\} \leq \{l_1 : \tau_1, \ldots, l_i : \tau_i, l_{i+1} : \tau_{i+1}, \ldots, l_n : \tau_n\}
\]

\[
\tau_i \leq \tau'_i \\
\{l_1 : \tau_1, \ldots, l_i : \tau_i, \ldots, l_n : \tau_n\} \leq \{l_1 : \tau_1, \ldots, l_i : \tau'_i, \ldots, l_n : \tau_n\}
\]

\[
\begin{align*}
\tau_3 \leq \tau_1 & \quad \tau_2 \leq \tau_4 \\
\tau_1 \rightarrow \tau_2 & \leq \tau_3 \rightarrow \tau_4 \\
\tau & \leq \tau \\
\tau_1 \leq \tau_2 & \quad \tau_2 \leq \tau_3 \\
\tau_1 \leq \tau_3 
\end{align*}
\]
### System F: $e \rightarrow e'$ and $\Delta; \Gamma \vdash e : \tau$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e \rightarrow e'$</td>
<td>$\Delta; \Gamma \vdash e : \tau$</td>
</tr>
<tr>
<td>$e e_2 \rightarrow e' e_2$</td>
<td>$\Delta; \Gamma \vdash e : \tau$</td>
</tr>
<tr>
<td>$v e \rightarrow v e'$</td>
<td>$\Delta; \Gamma \vdash e : \tau$</td>
</tr>
<tr>
<td>$e[\tau] \rightarrow e'[\tau]$</td>
<td>$\Delta; \Gamma \vdash \lambda x : \tau. e : \tau$</td>
</tr>
<tr>
<td>$(\lambda x : \tau. e) v \rightarrow e[v/x]$</td>
<td>$\Delta, \alpha; \Gamma \vdash e : \tau_1$</td>
</tr>
<tr>
<td>$(\Lambda e. e)[\tau] \rightarrow e[\tau/\alpha]$</td>
<td>$\Delta; \Gamma \vdash \Lambda e. e : \forall \alpha. \tau_1$</td>
</tr>
</tbody>
</table>

Simple System F examples: Let $\text{id} = \Lambda \alpha. \lambda x. e$. Then $\text{id}$ has type $\forall \alpha. \alpha \rightarrow \alpha$; $\text{id}$ has type $\text{int} \rightarrow \text{int}$; and $\text{id}$ has type $\text{int} * \text{int}$ has type $(\text{int} * \text{int}) \rightarrow (\text{int} * \text{int})$.

### Sum types, iso-recursive types

<table>
<thead>
<tr>
<th>Rule</th>
<th>Derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e \ ::= \ldots</td>
<td>A(e)</td>
</tr>
<tr>
<td>$\tau \ ::= \ldots</td>
<td>\tau_1 + \tau_2</td>
</tr>
<tr>
<td>$v \ ::= \ldots</td>
<td>A(v)</td>
</tr>
</tbody>
</table>

### Module Thread:

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>type t</code></td>
<td><code>type t</code></td>
</tr>
<tr>
<td><code>val create : ('a -&gt; 'b) -&gt; 'a -&gt; t</code></td>
<td><code>val send : 'a channel -&gt; 'a -&gt; unit event</code></td>
</tr>
<tr>
<td><code>val join : t -&gt; unit</code></td>
<td><code>val receive : 'a channel -&gt; 'a event</code></td>
</tr>
</tbody>
</table>

### Module Mutex:

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>type t</code></td>
<td><code>type t</code></td>
</tr>
<tr>
<td><code>val create : unit -&gt; t</code></td>
<td><code>val send : 'a channel -&gt; 'a -&gt; unit event</code></td>
</tr>
<tr>
<td><code>val lock : t -&gt; unit</code></td>
<td><code>val receive : 'a channel -&gt; 'a event</code></td>
</tr>
<tr>
<td><code>val unlock : t -&gt; unit</code></td>
<td><code>val choose : 'a event list -&gt; 'a event</code></td>
</tr>
</tbody>
</table>
1. (12 points) For each of the following Caml definitions, does it type-check in Caml? If so, what type does it have? If not, why not?

   (a) let a = (fun f -> (fun x y -> x) (f 0) (f 10))
   (b) let b = (fun f -> (fun x y -> x) (f 0) (f true))
   (c) let c = (fun f -> (fun x y -> x) (f 0) (f (f 10)))
   (d) let d = (fun f -> (fun x y -> x) (f 0) (f 5 10))

**Solution:**

(a) Type-checks: (int -> 'a) -> 'a

(b) Does not type-check: The type-inferencer will conclude that g must be a function takes an int and a function that takes a bool, and these cannot both hold.

(c) Type-checks: (int -> int) -> int

(d) Type-checks: (int -> int -> 'a) -> int -> 'a
2. (12 points) Consider a typed λ-calculus with iso-recursive types where we use explicit expressions of the form \( \text{fold}_\tau e \) and \( \text{unfold} e \) (as opposed to subtyping). For each of the following typing rules, explain why it makes little if any sense to add the rule to our type system.

(a) 

\[
\Delta; \Gamma \vdash e : \mu\alpha.\tau \\
\Delta; \Gamma \vdash \text{unfold} e : \tau 
\]

(b) Let \( FTV(\tau) \) mean the free type variables in \( \tau \). Assume it has been defined correctly.

\[
\Delta; \Gamma \vdash e : \mu\alpha.\tau \quad \alpha \notin FTV(\tau) \\
\Delta; \Gamma \vdash \text{unfold} e : \tau 
\]

Solution:

(a) This rule is unsound. The result type could have free type variables \( \alpha \) which could then be captured by some outer binding, such as type-abstraction. It is enough to say that the type \( \tau \) may make no sense without \( \alpha \) being properly bound. For example, if in some context \( f \) has type \( \alpha \to \beta \) and \( x \) has type \( \mu\alpha.\alpha \), then \( f \text{ unfold} x \) would type-check, but \( x \) does not have the type \( \tau \) expects.

(b) This rule is trivially admissible. The existing typing rule for \( \text{unfold} e \) already gives the result type \( \tau \) when \( \alpha \notin FTV(\tau) \) by the definition of type substitution.
Name: 

3. (24 points) Assume we have a typed programming language formally defined by a small-step operational semantics and a typing judgment. Assume the appropriate Preservation and Progress Theorems hold for this language. Consider each question below separately and explain your answers briefly.

(a) Suppose we change the operational semantics by adding a new inference rule.
i. Is it possible that the Preservation Theorem is now false? Why?
ii. Is it possible that the Progress Theorem is now false? Why?

(b) Suppose we change the type system by adding a new inference rule.
ii. Is it possible that the Preservation Theorem is now false? Why?
ii. Is it possible that the Progress Theorem is now false? Why?

(c) Suppose we change the operational semantics by replacing one of the inference rules with a rule that is just like it except it has some additional hypothesis.
ii. Is it possible that the Preservation Theorem is now false? Why?
ii. Is it possible that the Progress Theorem is now false? Why?

(d) Suppose we change the type system by replacing one of the inference rules with a rule that is just like it except it has some additional hypothesis.
ii. Is it possible that the Preservation Theorem is now false? Why?
ii. Is it possible that the Progress Theorem is now false? Why?

Optionally, you can use the following abbreviated notation in your answers.

- **Progress:** If \( \vdash e : \tau \), then \( e \) is a value or there exists \( e' \) such that \( e \rightarrow e' \).
- **Preservation:** If \( (\vdash e : \tau \text{ and } e \rightarrow e') \), then \( \vdash e' : \tau \).

**Solution:**

(a) i. Yes, the new rule might produce an ill-typed term. \( D \) is easier to satisfy, so \( C \) and \( D \) may no longer imply \( E \).
ii. No, more operational rules cannot make it harder to step. \( B \) is easier to satisfy so \( A \) still implies \( B \).

(b) i. Yes, the new rule might let some term type-check that can take a step to produce a term that doesn’t type-check. \( C \) is easier to satisfy, so \( C \) and \( D \) may no longer imply \( E \) even though \( E \) is also easier to satisfy. As an example, suppose we have a rule give \((3 + 4) + ()\) type \textbf{int}. It can step to \(7 + ()\), which does not type-check.
ii. Yes, the new rules could allow a stuck term to type-check. \( A \) is easier to satisfy, so \( A \) may no longer imply \( B \). For example, have a rule that gives \(3 + ()\) type \textbf{int}.

(c) i. No, the changed rule is now harder to use, but any term allowed after a step was also allowed before the change. \( D \) is harder to satisfy, so \( C \) and \( D \) still imply \( E \).
ii. Yes, the changed rule might no longer apply, which could cause some expression to be stuck that was not stuck in the old language. \( B \) is harder to satisfy so \( A \) may no longer imply \( B \).

(d) i. Yes, the changed rule is now harder to use, so some term that used to type-check might no longer type-check, so if that term can be produced by a well-typed expression taking a step, Preservation no longer holds. \( E \) is harder to satisfy, so \( C \) and \( D \) may no longer imply \( E \) even though \( C \) is also harder to satisfy.
ii. No, only fewer terms type-check when we add hypotheses, so the theorem still applies to all the terms that type-check after the change. $A$ is harder to satisfy, so $A$ still implies $B$. 
4. (20 points) This problem uses System F extended with addition. Note that the answers to all parts should be brief.

(a) Give the appropriate System F typing rule for addition expressions of the form $e_1 + e_2$. (This should be easy and is unrelated to the other subproblems.)

T-Add

(b) Consider a typing context where:

- There are no type variables in scope.
- $x$ is the only term variable in scope and it has type $\forall \alpha. \alpha \to \alpha$.

i. What does $\tau$ need to be for the program fragment $x \ [\tau] \ (\lambda y : \text{int}. \ y - 3) \ 12$ to typecheck? (Recall application — of types or terms — associates to the left.)

ii. Given your choice for $\tau$, what type does $x \ [\tau] \ (\lambda y : \text{int}. \ y - 3) \ 12$ have?

(c) If $v$ is an arbitrary value of type $\forall \alpha. \alpha \to \alpha$, then what might $v \ [\tau] \ (\lambda y : \text{int}. \ y - 3) \ 12$ evaluate to?

(d) If $v$ is an arbitrary value such that $v \ (\lambda y : \text{int}. \ y - 3) \ 12$ type-checks (notice $v$ is a value and no longer polymorphic), then:

i. What type does $v$ have? (Hint: it’s different from the answer to part (b.i)).

ii. What might $v \ (\lambda y : \text{int}. \ y - 3) \ 12$ evaluate to? (Hint: it’s different from the answer to part (b.ii)).
Solution:

(a) \[
\Delta; \Gamma \vdash e_1 : \text{int} \quad \Delta; \Gamma \vdash e_2 : \text{int} \\
\Delta; \Gamma \vdash e_1 + e_2 : \text{int}
\]

(b) i. \( \tau \) must be \( \text{int} \rightarrow \text{int} \)
    ii. \( \text{int} \)

(c) It will always evaluate to 9 due to parametricity. In System F, any value of type \( \forall \alpha. \alpha \rightarrow \alpha \) is totally equivalent to the identity function.

(d) i. \((\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \tau_1)\) for any \( \tau_1 \).
    ii. It could produce any value whatsoever.
5. (16 points)
Consider a typed \( \lambda \)-calculus with sum types, pair types, recursive types, unit, and int.

(a) Define a type \( t_1 \) for a binary tree of integers where:
   - Each interior node has two children and no data.
   - Each leaf node has one integer and no children.
   - Your type definition should have the form \( \mu \alpha. \cdots \). 

Solution:

(a) \( \mu \alpha. \text{int} + (\text{unit} * \alpha * \alpha) \)

(b) Give a type \( t_2 \) for a binary tree of integers where:
   - Each node has no data and two \textit{optional} children (meaning each child may or may not be another binary tree, e.g., some nodes may have a single child). Again, each leaf node has one integer and no children.
   - Your type definition should have the form \( \mu \alpha. \cdots \). 

Solution:

(b) \( \mu \alpha. \text{int} + \text{unit} * (\text{int} + \alpha) + \text{unit} * (\text{int} + \alpha) * (\text{int} + \alpha) \)

(c) Can all possible values of type \( t_2 \) be translated to equivalent values of type \( t_1 \)? If yes, briefly explain your answer in plain English (no formal proof needed). If not, give an example.

Solution:

(c) No, for example the tree \texttt{unit->unit->5} would type-check as a value of type \( t_2 \) but not with \( t_1 \) because every \( t_1 \) is either an int node or has exactly two children.
6. Continuation passing style in OCaml.

(a) (12 points) Assume that the `eqk`, `addk`, `timesk`, `divk` functions are defined as follows.

```ocaml
let eqk a b k = k (a = b);;
let addk a b k = k ( a + b );;
let times a b k = k ( a * b );;
let divk a b k = k ( a / b );;
```

Using only the above functions, implement a CPS function `abcdk` that takes four integer arguments `a, b, c, d`, a regular continuation `k`, and an exception continuation `xk`, to compute the following integer expression: `a * ((b/c) + d)`. If `c` is 0, call the exception continuation `xk` and pass the offending value to it.

```ocaml
# let abcdk a b c d k xk = ...;;
val abcdk : int -> int -> int -> int -> (int -> 'a) -> (int -> 'a) -> 'a = <fun>
```
Name: 

(b) (20 points) Consider the direct style function that given a list of integers, returns a list where each value has been doubled (multiplied by 2).

```ocaml
let rec double l = 
    match l with 
    | [] -> [] 
    | h::tl -> (2*h) :: (double tl)
```

i. What is the type of `double` above?

ii. For a given call to `double` above, approximately how deep would the call-stack grow in terms of the function arguments?

iii. Write a version of `double` called `doublek` in continuation-passing style (i.e., it should take as arguments a list of integers and a continuation function: `let rec doublek l k = ...`), which uses a small constant amount of stack space. You can assume that the following CPS functions are defined.

```ocaml
open List;;
let eqk arg1 arg2 k = k (arg1 == arg2);;
let timesk arg1 arg2 k = k (arg1 * arg2);;
let hdk lst k = k (hd lst);;
let tlk lst k = k (tl lst);;
let consk h t k = k (h::t);;
```

iv. What is the type of the `doublek` function you wrote in part b.iii?

Solution:

(a) let abcdk a b c d k xk = 
    eqk c 0 
    (fun ex -> if ex then xk c 
        else divk b c 
            (fun bc -> addk bc d 
                (fun bcd -> timesk a bcd k))));;

(b) Double values in list.

i. int list -> int list
ii. Its depth will be proportional to the length of the list l.
iii. let rec doublek l k = 
    eqk l [] 
    (fun empty -> if empty then k [] 
        else hdk l 
            (fun h -> timesk h 2 
                (fun h2 -> tlk l 
                    (fun ltail -> doublek ltail 
                        (fun t -> consk h2 t k)))))
    iv. doublek has type int list -> (int list -> 'a) -> 'a
This problem considers static overloading and multimethods (dynamic dispatch). Consider this OO code skeleton:

class Food {}
class Plant extends Food {}
class Animal extends Food {}
class Tiger extends Animal {}
class Goat extends Animal {}

class Farmer {
    void feed(Animal a, Food f) { ... } // 1
    void feed(Animal a, Animal b) { ... } // 2
    void feed(Animal a, Plant p) { ... } // 3
    void main() {
        ...
    }
}

Replace the ... with a few lines of code such that:

- If the language uses static overloading then each of the three feed methods is called exactly once. Indicate which line of code calls which method using the line numbers specified in the code segment above.
- If the languages uses multimethods (dynamic dispatch), then every method call may be ambiguous because there is “no single best match” (where we assume the “goodness” of a match is the number of subsumptions to immediate supertypes and fewer is better). Indicate for each call, which method or methods correspond to the best match (if more than one, specify the methods tied by fewest required subsumptions).

**Solution:**
One possible answer below

```java
Animal g = new Goat();
Animal t = new Tiger();
Food p = new Plant();
Plant p2 = new Plant();

feed(g, p);    // (static: 1, multi: 1 and 3)
feed(t, g);    // (static: 2, multi: 2)
feed(g, p2);   // (static: 3, multi: 1 and 3)
```
8. (30 points) Consider STLC with integer constants with the addition of pairs as described below.

Syntax for pairs:

\[
\begin{align*}
    e & ::= \ldots | (e, e) | e.1 | e.2 \\
v & ::= \ldots | (v, v) \\
\tau & ::= \ldots | \tau * \tau
\end{align*}
\]

Evaluation rules (dynamic semantics) for pairs:

\[
\begin{align*}
    e & \rightarrow e' \\
    e_1 & \rightarrow e'_1 \\
    (e_1, e_2) & \rightarrow (e'_1, e_2) \\
    (v_1, e_2) & \rightarrow (v'_1, e'_2) \\
    e.1 & \rightarrow e'.1 \\
    e.2 & \rightarrow e'.2
\end{align*}
\]

Typing rules (static semantics) for pairs:

\[
\begin{align*}
    \Gamma \vdash e : \tau \\
    \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2 \\
    \Gamma \vdash e : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e.1 : \tau_1 \quad \Gamma \vdash e : \tau_1 * \tau_2 \quad \Gamma \vdash e.2 : \tau_2
\end{align*}
\]

**Theorem** (Type Soundness). If \( \cdot \vdash e : \tau \) and \( e \rightarrow^* e' \), then either \( e' \) is a value or there exists an \( e'' \) such that \( e' \rightarrow e'' \).

**Lemma** (Weakening). If \( \Gamma \vdash e : \tau \) and \( x \not\in \text{Dom}(\Gamma) \), then \( \Gamma, x : \tau' \vdash e : \tau \).

In the questions below, you will prove the progress and preservation theorems for this language. Only include the new cases for pairs (you do not need to include the rest of the STLC cases).

(a) Extend the following Canonical Forms lemma for pairs.

**Lemma** (Canonical Forms). If \( \cdot \vdash v : \tau \), then

i. If \( \tau \) is int, then \( v \) is a constant, i.e., some \( c \).

ii. If \( \tau \) is \( \tau_1 \rightarrow \tau_2 \), then \( v \) is a lambda, i.e., \( \lambda x. e \) for some \( x \) and \( e \).

iii. If \( \tau \) is \( \tau_1 \ast \tau_2 \), then ...

(b) State and prove the progress theorem for STLC with pairs (include only pair-relevant cases, i.e., T-Pair, T-Proj1, T-Proj2).

(c) State and prove the preservation theorem for STLC with pairs (include only pair-relevant cases, i.e., T-Pair, T-Proj1, T-Proj2).
Name:

Solution:

(a) If $\tau$ is $\tau_1 * \tau_2$, then $v$ is a pair, i.e., $(v_1, v_2)$ for some $v_1$ and $v_2$.

(b)  

**Theorem** (Progress). If $\cdot \vdash e : \tau$, then either $e$ is a value or there exists some $e'$ such that $e \rightarrow e'$.

**Progress.** The proof is by induction on (the height of) the derivation of $\cdot \vdash e : \tau$, proceeding by cases on the bottommost rule used in the derivation.

T-Proj1 $e$ is $e.1$ which is some $e_1$ smaller than $e$, which by induction is either a value or takes a step.  
If $e_1$ is a value, then by E-Pair1, $e$ is also a value.  
If $e_1$ is not a value, then by E-Proj1, $e$ takes a step.

T-Proj2 $e$ is $e.2$ which is some $e_2$ smaller than $e$, which by induction is either a value or takes a step.  
If $e_2$ is a value, then by E-Pair2, $e$ is also a value.  
If $e_2$ is not a value, then by E-Proj2, $e$ takes a step.

T-Pair $e$ is $(e_1, e_2)$ for some $e_1$ and $e_2$.  
By induction $e_1$ and $e_2$ are either values or take a step to $e'_1$ and $e'_2$, respectively.  
If both $e_1$ and $e_2$ are values, then by Canonical forms, $e$ is also a value.  
If $e_1$ is a value and $e_2$ is not a value, then by E-PairBeta2, $e$ takes a step.  
If $e_1$ is not a value, then by E-PairBeta1, $e$ takes a step.

\( \square \)

(c)  

**Theorem** (Preservation). If $\cdot \vdash e : \tau$ and $e \rightarrow e'$, then $\cdot \vdash e' : \tau$.

T-Proj1 $e$ is $\hat{e}.1$ with type $\tau_1$.  
By inversion on T-Proj1, we know that $\Gamma \vdash \hat{e} : \tau_1 * \tau_2$ for some $\tau_1$ and $\tau_2$.  
There are two cases for deriving $\hat{e}.1 \rightarrow e'$:

i. If $\hat{e}$ is not a value, then by E-Proj1 we have $\hat{e}.1 \rightarrow \hat{e}'.1$.  
By induction, given $\Gamma \vdash \hat{e}.1 : \tau_1$, we have $\Gamma \vdash \hat{e}'.1 : \tau_1$.  
Hence, $e' = \hat{e}'.1$ has type $\tau_1$.

ii. If $\hat{e}$ is a value, i.e., $(v_1, v_2)$ for some $v_1$ and $v_2$, then by E-Pair1 we have $(v_1, v_2).1 \rightarrow v_1$.  
By $\Gamma \vdash (v_1, v_2) : \tau_1 * \tau_2$ and induction, $\Gamma \vdash v_1 : \tau_1$.  
Hence, $e' = v_1$ has type $\tau_1$.

T-Proj2 $e$ is $\hat{e}.2$ with type $\tau_2$.  
By inversion on T-Proj2, we know that $\Gamma \vdash \hat{e} : \tau_1 * \tau_2$ for some $\tau_1$ and $\tau_2$.  
There are two cases for deriving $\hat{e}.2 \rightarrow e'$:

i. If $\hat{e}$ is not a value, then by E-Proj2 we have $\hat{e}.2 \rightarrow \hat{e}'.2$.  
By induction, given $\Gamma \vdash \hat{e}.2 : \tau_2$, we have $\Gamma \vdash \hat{e}'.2 : \tau_2$.  
Hence, $e' = \hat{e}'.2$ has type $\tau_2$.

ii. If $\hat{e}$ is a value, i.e., $(v_1, v_2)$ for some $v_1$ and $v_2$, then by E-Pair2 we have $(v_1, v_2).2 \rightarrow v_2$.  
By $\Gamma \vdash (v_1, v_2) : \tau_1 * \tau_2$ and induction, $\Gamma \vdash v_2 : \tau_2$.  
Hence, $e' = v_2$ has type $\tau_2$.

T-Pair $e$ is $e_1, e_2$ with type $\tau_1 * \tau_2$ for some $e_1$ with type $\tau_1$ and $e_2$ with type $\tau_2$.  
Then we can derive $(e_1, e_2) \rightarrow e'$ in the following ways.

i. E-PairBeta1: Then $e' = (e'_1, e_2)$ and $e_1 \rightarrow e'_1$.  
By induction, $e'_1$ has type $\tau_1$, then by T-Pair, $e'$ has type $\tau_1 * \tau_2$.

ii. E-PairBeta2: Then $e' = (v_1, e'_2)$ and $e_1$ is $v_1$ and $e_2 \rightarrow e'_2$.  
By induction, $e'_2$ has type $\tau_2$, then by T-Pair, $e'$ has type $\tau_1 * \tau_2$. 