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 15:15.
- You can rip apart the pages, but please write your name on each page.
- There are 196 points total, distributed unevenly among 9 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):

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

\[
e \to e' \quad \text{and} \quad \Gamma \vdash e : \tau \quad \text{and} \quad \tau_1 \leq \tau_2
\]

\[
\begin{array}{cccccc}
\lambda x. e & \to & e[v/x] & \quad & e_1 & \to e'_1 & \quad & e_2 & \to e'_2 & \quad & e & \to e' \\
 e_1 e_2 & \to & e'_1 e_2 & \quad & v e_2 & \to & v e'_2 & \quad & \text{fix } e & \to & \text{fix } e' & \quad & \text{fix } \lambda x. e & \to & e[\text{fix } \lambda x. e/x]
\end{array}
\]

\[
\{l_1 = v_1, \ldots, l_n = v_n\}. l_i \to v_i
\]

\[
e_i \to e'_i
\]

\[
\{l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, l_i = e_i, \ldots, l_n = e_n\} \to \{l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, l_i = e'_i, \ldots, l_n = e_n\}
\]

\[
\begin{array}{cccccc}
\Gamma \vdash e : \text{int} & \quad & \Gamma, x : \tau_1 \vdash e : \tau_2 & \quad & \Gamma \vdash e_1 : \tau_2 \to \tau_1 & \quad & \Gamma \vdash e_2 : \tau_2 \\
\Gamma, x : \tau_1 \vdash e : \tau_2 & \quad & \Gamma \vdash e_1 e_2 : \tau_1 & \quad & \Gamma \vdash e : \tau \to \tau & \quad & \Gamma \vdash \text{fix } e : \tau
\end{array}
\]

\[
\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n \quad \text{labels distinct}
\]

\[
\begin{array}{cccccc}
\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} & \quad & \{l_1 : \tau_1, \ldots, l_n : \tau_n\} & \quad & 1 \leq i \leq n \\
\Gamma \vdash e : \{l_1 : \tau_1, \ldots, l_n : \tau_n\} & \quad & \Gamma \vdash e_i : \tau_i
\end{array}
\]

\[
\begin{array}{cccccc}
\Gamma \vdash e : \tau & \quad & \tau \leq \tau' & \quad & \Gamma \vdash e : \tau' & \quad & \{l_1 : \tau_1, \ldots, l_n : \tau_n, l : \tau\} \leq \{l_1 : \tau_1, \ldots, l_n : \tau_n\}
\end{array}
\]

\[
\begin{array}{cccccc}
\{l_1 : \tau_1, \ldots, l_{i-1} : \tau_{i-1}, l_i : \tau_i, \ldots, l_n : \tau_n\} & \leq & \{l_1 : \tau_1, \ldots, l_i : \tau_i, l_{i-1} : \tau_{i-1}, \ldots, l_n : \tau_n\} \\
\tau_i \leq \tau'_i & \quad & \{l_1 : \tau_1, \ldots, l_i : \tau_i, \ldots, l_n : \tau_n\} & \leq & \{l_1 : \tau_1, \ldots, l_i : \tau'_i, \ldots, l_n : \tau_n\}
\end{array}
\]

\[
\begin{array}{cccccc}
\tau_3 \leq \tau_1 & \quad & \tau_2 \leq \tau_3 & \quad & \tau_4 \leq \tau & \quad & \tau \leq \tau & \quad & \tau_1 \leq \tau_2 & \quad & \tau_2 \leq \tau_3 & \quad & \tau_1 \leq \tau_3
\end{array}
\]

\[
\begin{array}{cccccc}
\tau_1 \to \tau_2 \leq \tau_3 \to \tau_4 & \quad & \tau \leq \tau & \quad & \tau_1 \leq \tau_2 & \quad & \tau_2 \leq \tau_3 & \quad & \tau_1 \leq \tau_3
\end{array}
\]
System F (syntax)  
\[
\begin{align*}
e & ::= c | x | \lambda x : \tau. e | e \ e | \Delta \alpha. e | e[\tau] \\
\tau & ::= \text{int} | \tau \rightarrow \tau | \alpha | \forall \alpha. \tau \\
v & ::= c | \lambda x : \tau. e | \Delta \alpha. e
\end{align*}
\]
\[
\begin{align*}
\Gamma & ::= \cdot | \Gamma, x : \tau \\
\Delta & ::= \cdot | \Delta, \alpha
\end{align*}
\]

System F: \(e \rightarrow e'\) and \(\Delta; \Gamma \vdash e : \tau\)

\[
\begin{align*}
e \rightarrow e' & \quad \frac{e \rightarrow e'}{e_2 \rightarrow e'_2 e_2} \quad \frac{v \rightarrow v e'\ e' \rightarrow e'}{e[\tau] \rightarrow e'[\tau]} \quad (\lambda x : \tau. e) \rightarrow e[\tau/v] \quad (\Delta \alpha. e)[\tau] \rightarrow e[\tau/\alpha]
\end{align*}
\]

Simple System F examples: Let \(\text{id} = \Delta \alpha. \lambda x : \alpha. x\). Then \(\text{id}\) has type \(\forall \alpha. \alpha \rightarrow \alpha\); \(\text{id} [\text{int}]\) has type \(\text{int} \rightarrow \text{int}\); and \(\text{id} [\text{int} \ast \text{int}]\) has type \((\text{int} \ast \text{int}) \rightarrow (\text{int} \ast \text{int})\).

---

Module Thread:

\[
\begin{align*}
type & t \\
val create & : (\text{\texttt{\textquotesingle}a \rightarrow \text{\textquotesingle}b}) \rightarrow \text{\textquotesingle}a \rightarrow t \\
val join & : t \rightarrow \text{unit}
\end{align*}
\]

Module Mutex:

\[
\begin{align*}
type & t \\
val create & : \text{unit} \rightarrow t \\
val lock & : t \rightarrow \text{unit} \\
val unlock & : t \rightarrow \text{unit}
\end{align*}
\]

Module Event:

\[
\begin{align*}
type & \text{\textquotesingle}a \text{ channel} \\
type & \text{\textquotesingle}a \text{ event} \\
val new_channel & : \text{unit} \rightarrow \text{\textquotesingle}a \text{ channel} \\
val send & : \text{\textquotesingle}a \text{ channel} \rightarrow \text{\textquotesingle}a \rightarrow \text{unit \ event} \\
val receive & : \text{\textquotesingle}a \text{ channel} \rightarrow \text{\textquotesingle}a \text{ event} \\
val choose & : \text{\textquotesingle}a \text{ event list} \rightarrow \text{\textquotesingle}a \text{ event} \\
val wrap & : \text{\textquotesingle}a \text{ event} \rightarrow (\text{\textquotesingle}a \rightarrow \text{\textquotesingle}b) \rightarrow \text{\textquotesingle}b \text{ event} \\
val sync & : \text{\textquotesingle}a \text{ event} \rightarrow \text{\textquotesingle}a
\end{align*}
\]

3
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. (15 points) Consider a typed λ-calculus with iso-recursive types where we use explicit expressions of the form fold, e and 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) \[ \frac{\Delta; \Gamma \vdash e : \mu \alpha. \tau}{\Delta; \Gamma \vdash \text{unfold } e : \tau} \]

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

\[ \frac{\Delta; \Gamma \vdash e : \mu \alpha. \tau \quad \alpha \not\in \text{FTV(\tau)}}{\Delta; \Gamma \vdash \text{unfold } e : \tau} \]

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 \(\alpha \rightarrow \beta\) and \(x\) has type \(\mu \alpha. \alpha\), then \(f \text{ unfold } x\) would type-check, but \(x\) does not have the type \(f\) expects.

(b) This rule is trivially admissible. The existing typing rule for \text{unfold } e\ already gives the result type \tau when \(\alpha \not\in \text{FTV(\tau)}\) by the definition of type substitution.
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?
   ii. Is it possible that the Progress Theorem is now false?

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

(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.
   i. Is it possible that the Preservation Theorem is now false?
   ii. Is it possible that the Progress Theorem is now false?

(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.
   i. Is it possible that the Preservation Theorem is now false?
   ii. Is it possible that the Progress Theorem is now false?

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

- Preservation: If $\frac{\frac{\vdash e : \tau}{\vdash e' : \tau}}{\vdash e \rightarrow e'}$, then $\vdash e' : \tau$.
- Progress: If $\vdash e : \tau$, then $e$ is a value or there exists $e'$ such that $e \rightarrow e'$. 
For the benefit of full, precise explanations, let’s state Preservation and Progress this way:

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

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

(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. $A$ is easier to satisfy, so $A$ and $B$ may no longer imply $C$ even though $C$ is also easier to satisfy. As an example, suppose we have a rule give $(3 + 4) + ()$ type int. It can step to $7 + ()$, which does not type-check.
   ii. Yes, the new rules could allow a stuck term to type-check. $D$ is easier to satisfy, so $D$ may no longer imply $E$. For example, have a rule that gives $3 + ()$ type 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. $B$ is harder to satisfy, so $A$ and $B$ still imply $C$.
   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. $E$ is harder to satisfy so $D$ may no longer imply $E$.

(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. $C$ is harder to satisfy, so $A$ and $B$ may no longer imply $C$ even though $A$ 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. $D$ is harder to satisfy, so $D$ still implies $E$. 

7
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 unrelated to the other problems.) Hint: See p. 3 for existing typing rules for System F.

(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 $\forall \alpha. \alpha \rightarrow \alpha$.

   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 $\forall \alpha. \alpha \rightarrow \alpha$, 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) 

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

(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. (20 points)
Consider a typed λ-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 one integer and two children.
   - Each leaf node has no data.

(b) Give a type \( t_2 \) for a binary tree of integers where:
   - Each node has one integer and two \textit{optional} children (meaning each child may or may not be another binary tree).

(c) Explain in English how there is exactly one value of type \( t_1 \) that cannot be translated to an equivalent value of type \( t_2 \).

(d) Define the value you described in the previous problem using an actual λ-calculus expression. Make sure your value has type \( t_1 \).

\textbf{Solution:}

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

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

(c) The empty tree can be represented with a value of type \( t_1 \) but not with \( t_2 \) because every \( t_2 \) has at least one int.

(d) (This answer varies depending how \( t_1 \) is defined.) \( \text{fold}_{\mu \alpha. \text{unit} + (\text{int} \times \alpha \times \alpha)} A()() \)
6. (30 points) Continuation passing style in OCaml.

(a) 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: \(a \times (\frac{b}{c} + d)\). 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>
```
(b) Consider the direct style function for appending two lists.

```ml
let rec append xs ys =  
match xs with   
  | [] -> ys   
  | x::xs' -> x :: (append xs' ys)
```

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

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

iii. Use a helper function written in continuation-passing style to give a different version of `append` that uses a small constant amount of stack space. You are allowed to use '::' in your implementation.

iv. What is the type of the helper function you used in part (e)? (Note the type of `append` itself should still be the same as in part (b).)

**Solution:**

(a) let abckdk 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) Appending two lists.

i. `'a list -> 'a list -> 'a list`

ii. Its depth will be proportional to the length of `xs`. (The length of `ys` is irrelevant.)

iii. let append xs ys =  
```
  let rec f k xs =   
  match xs with   
    | [] -> k ys   
    | x::xs' -> f (fun zs -> k (x::zs)) xs'  
  in f (fun x -> x) xs
```

iv. `f` has type `(a list -> 'b) -> 'a list -> 'b` (Note if your helper function takes `ys` as an argument that is fine, but then the type has another `'a list`. You might also have arguments in a different order.)

v. let append xs ys =  
```
  let rec rev_append xs ys =   
  match xs with   
    | [] -> ys   
    | x::xs' -> rev_append xs' (x::ys)  
  in rev_append (rev_append xs [])) ys
```
7. (25 points)

Use Concurrent ML to complete an implementation of “infinite” arrays (in the sense that no index is out of bounds), without using Caml’s references or arrays. More specifically, implement the `new_array` function for this code:

\begin{verbatim}
(* Interface *)
type 'a myarray
val new_array : 'a -> 'a myarray (* Initially, every index maps to 'a *)
val set : 'a myarray -> int -> 'a -> unit (* change value in index *)
val get : 'a myarray -> int -> 'a (* return current value in index *)

(* Implementation *)
open Event
open Thread
type 'a myarray = ((int * ('a channel)) channel) * ((int * 'a) channel)

let new_array init = (* for you *)
let set (_,c) i v =  
  sync (send c (i,v))
let get (c,_) i =
  let ret = new_channel() in
  sync (send c (i,ret));
  sync (receive ret)
\end{verbatim}

Hints: Do not worry about being efficient. Have `set` work in \(O(1)\) time and `get` work in (worst-case) \(O(n)\) time where \(n\) is the number of `set` operations preceding it. Use an association list. Sample solution is about 15 lines, including a short helper function for traversing a list. Use `choose` and `wrap`.

**Solution:**

\begin{verbatim}
let rec assoc lst i default =
  match lst with
  | [] -> default
  | (j,v)::tl -> if i=j then v else assoc tl i default

let new_array init =
  let getter,setter = new_channel(), new_channel() in
  let rec server lst =
    sync (choose [  
      wrap (receive getter)
        (fun (i,c) -> (sync (send c (assoc lst i init))); server lst);
      wrap (receive setter)
        (fun pr -> server (pr::lst)) )
  in
  ignore (Thread.create server []); getter,setter
\end{verbatim}
8. (20 points)

Consider a class-based OOP language like we did in class, but suppose methods do not have implicit access to self (also known as this). More specifically:

- As usual, subclasses inherit the fields and methods of superclasses and can override methods. The method-lookup rules are the same.
- As usual, we “confuse” classes and types, so a subclass is a subtype.
- Unlike in OOP, a method in class C has to take an explicit argument of type C to access any fields/methods in its body. For example, instead of having a method like
  `int get_sum() { return self.x + self.y }`
  and calling it like
  `e.get_sum()` (assuming the method is defined in a class C with fields x and y), we instead would have to do something like
  `int get_sum(C obj) { return obj.x + obj.y }`
  and call it like
  `e.get_sum(e)`.

So, this is less convenient than OOP (notice how `e.get_sum(e)` has to repeat e), but more flexible (since callers are not required to use the same e in both places).

(a) In this strange language, give an example showing how you want covariant subtyping on method arguments in order for overriding methods to use the fact that they are defined in the subclass.

(b) Show how covariant subtyping on method arguments is unsound by giving a use of your example from part (a) in which the program gets stuck even though it typechecks with covariant subtyping.

(c) In 1–3 sentences, explain why normal OOP does not have this “covariant subtyping is unsound, but contravariant subtyping is not what you want” problem.

Solution:

(a) `class C {
    int m(C obj) { return 0; }
}
class D extends C {
    int x;
    int m(D obj) { return obj.x; }
}

When overriding m in class D we want the argument (which is playing the role of self) to have type D, else we cannot access the x field. Since D≤C, we need covariant subtyping.

(b) `C c = new D();
c.m(new C());`

We use subsumption to assign a D object to c. The class definition of D allows m to override since we have covariant subtyping and D≤C. But then we call c.m(new C()), which will get stuck because the argument has no x field.

(c) In normal OOP, callers do not choose which object is bound to self. In e.m(), it must be e that is bound to self in the body of the call, so if e’s class overrides m assuming self has a subtype of what it has in the method being overridden, that assumption will always hold.
9. (30 points) Consider STLC with integer constants with the addition of pairs as described below.

Syntax for pairs:

e ::= ... | (e,e) | e.1 | e.2  
v ::= ... | (v,v)  
\tau ::= ... | \tau \ast \tau

Evaluation rules (dynamic semantics) for pairs:

\[ e \rightarrow e' \]

\[ \text{E-PairBeta1} \quad e_1 \rightarrow e'_1 \quad (e_1,e_2) \rightarrow (e'_1,e_2) \]
\[ \text{E-PairBeta2} \quad e_2 \rightarrow e'_2 \quad (v_1,e_2) \rightarrow (v_1,e'_2) \]
\[ \text{E-Proj1} \quad e.1 \rightarrow e'.1 \quad \tau.1 \rightarrow \tau'.1 \]
\[ \text{E-Proj2} \quad e.2 \rightarrow e'.2 \quad \tau.2 \rightarrow \tau'.2 \]

Typing rules (static semantics) for pairs:

\[ \Gamma \vdash e : \tau \]

\[ \text{T-Pair} \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma \vdash (e_1,e_2) : \tau_1 \ast \tau_2 \]
\[ \text{T-Proj1} \quad \Gamma \vdash e : \tau_1 \ast \tau_2 \quad \Gamma \vdash e.1 : \tau_1 \quad \Gamma \vdash e.2 : \tau_2 \]

Theorem (Type Soundness). If \( \vdash e : \tau \) 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 \( x \notin \text{Dom}(\Gamma) \), 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 \( \vdash v : \tau \), 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: ________________________________