15–312: Principles of Programming Languages

Practice Midterm Examination
Sample Solutions

February 26, 2014

- There are 14 pages in this examination, comprising 4 questions worth a total of 100 points.
- You have 80 minutes to complete this examination.
- Please answer all questions in the space provided with the question.
- There are scratch pages at the end for your use.
- You may refer to your personal notes and to the text, but to no other person or source, during the examination.

Full Name: ______________________________

Andrew ID: ______________________________

Question:	Short Answer	Classes and Casts	Databases	Type Safety	Total
Points:	25	25	25	25	100
Score:					

Question 1 [25]: Short Answer

Answer the following questions in the space provided.

α-equivalence

(a) (5 points) Rewrite the following expression so that variables that refer to different binding sites have different names.

$$\begin{aligned}&\lambda x : \mathsf{nat}.\\ &\quad \mathsf{let}\ w\ \mathsf{be}\ \lambda x : \mathsf{nat}.\,\mathsf{s}\ x\\ &\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ \lambda y : \mathsf{nat}.\,\lambda x : \mathsf{nat}.\,\mathsf{rec}(y, x, y.x.w\ x)\\ &\ \mathsf{in}\ y\ x\ x\end{aligned}$$

Solution:

$$\begin{aligned}&\lambda x : \mathsf{nat}.\\ &\quad \mathsf{let}\ succ\ \mathsf{be}\ \lambda n : \mathsf{nat}.\,\mathsf{s}\ n\\ &\ \mathsf{in}\ \mathsf{let}\ plus\ \mathsf{be}\ \lambda n : \mathsf{nat}.\,\lambda m : \mathsf{nat}.\,\mathsf{rec}(n, m, n'.y.succ\ y)\\ &\ \mathsf{in}\ plus\ x\ x\end{aligned}$$

Gödel's T

(a) (4 points) The number of moves to solve the Tower of Hanoi puzzle is given by the following recurrence relation, where n is the number of disks:

$$\begin{cases} H(0) & = 1 \\ H(n+1) & = 2H(n) + 1 \end{cases}$$

Define an expression $\mathsf{hanoi} : \mathsf{nat} \to \mathsf{nat}$ in Gödel's T such that $\mathsf{hanoi}\ \overline{n} \mapsto^* \overline{H(n)}$ for any $n \in \mathbb{N}$. You may assume that the function $\mathsf{plus} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$ implementing addition has been defined for you.

Solution:

$$\begin{aligned}\mathsf{hanoi} \triangleq\ &\lambda n : \mathsf{nat}.\\ &\quad \mathsf{rec}(n,\\ &\qquad \mathsf{s}\ \mathsf{z},\\ &\qquad x.y.\,\mathsf{s}\ (\mathsf{plus}\ y\ y))\end{aligned}$$

PCF

(a) (6 points) Recall the definition of the binomial coefficient as read off Pascal's triangle:

$$\begin{cases} P(n,0) & = 1 \\ P(0,k+1) & = 0 \\ P(n+1,k+1) & = P(n,k) + P(n,k+1) \end{cases}$$

Define an expression $\mathsf{coef} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$ in PCF such that $\mathsf{coef}\ \overline{n}\ \overline{k} \mapsto^* \overline{P(n,k)}$ for any $n, k \in \mathbb{N}$. You may assume that the function $\mathsf{plus} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$ implementing addition has been defined for you.

Solution:

$$\begin{array}{lll} \mathsf{coef} & \triangleq & \mathsf{fix}[\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}]\mathsf{coef}. \\ & & \lambda n : \mathsf{nat}.\ \lambda k : \mathsf{nat}. \\ & & \quad \mathsf{ifz}(k, \mathsf{s\ z}, \\ & & \qquad\quad k'.\mathsf{ifz}(n, \mathsf{z}, \\ & & \qquad\qquad\qquad n'.\mathsf{plus}\ (\mathsf{coef}\ n'\ k')\ (\mathsf{coef}\ n'\ k))) \end{array}$$

Recursive Types

In this following exercises, consider the extension of PCF with sums, products and recursive types (and nothing else).

The type of a *stream processor* would be written as follows in an SML-like language:

```
datatype SP = Get of nat -> SP
            | Put of nat * SP
```

(a) (2 points) Define the type SP using sums, products and recursive types.

Solution:

$$\mathsf{SP} \triangleq \mu t.(\mathsf{nat} \to t) + (\mathsf{nat} \times t)$$

(b) (3 points) The following stream processor, written in SML-like syntax, increments every element of a stream of natural numbers by one:[1]

```
val rec adds : SP = Get (fn x => Put (s x, adds))
```

It reads an element, `x`, outputs `s x` (the successor of `x`), and then calls itself recursively. Define `adds` in PCF using the recursive type you defined in the previous exercise. Please use the abstract syntax forms for sums in your answer, writing, for example, $\texttt{in}[\tau_1][\tau_2][l](e_1)$ for an injection of e_1 into the left half of the sum type $\tau_1 + \tau_2$.

Solution:

$$\begin{aligned}\mathsf{adds} \triangleq\ & \mathsf{fix}[\mathsf{SP}](\mathsf{adds}.\\ & \mathsf{fold}(\texttt{in}[\mathsf{nat} \to \mathsf{SP}][\mathsf{nat} \times \mathsf{SP}][l](\lambda x : \mathsf{nat}.\\ & \mathsf{fold}(\texttt{in}[\mathsf{nat} \to \mathsf{SP}][\mathsf{nat} \times \mathsf{SP}][r](\mathsf{s}\ x, \mathsf{adds})))))\end{aligned}$$

Modernized Algol

(a) (5 points) Define the commands $\mathsf{newcounter}(a.m)$ and $\mathsf{getcounter}[a]$ such that the first time $\mathsf{getcounter}[a]$ is encountered, it evaluates to $\mathsf{ret}(z)$. The second time, $\mathsf{ret}(s\ z)$. Etc...

Solution:

$$\begin{aligned}\mathsf{newcounter}(a.m) &\triangleq \mathsf{decl}(\mathsf{z}; a.m)\\ \mathsf{setcounter}[a] &\triangleq \{x \leftarrow \mathsf{get}[a]; \mathsf{set}[a](s(x)); ret(x)\}\end{aligned}$$

[1]In SML, the keywords `val rec` allow defining recursive functions without using `fun`. SML would however reject this example because `val rec` can be used *only* with functions.

Question 2 [25]: Classes and Casts

Let us consider the following variation on typing for objects that is found in common object-oriented languages. The focus of attention in this question is on the types of objects created using the `new` construct; it does *not* concern the material in Chapter 25 of PFPL.

$$\begin{array}{llcl} \mathsf{Typ} & \tau & ::= & \texttt{obj} \\ & & \mid & \texttt{obj}\,[c] \\ \mathsf{Exp} & e & ::= & \texttt{new}\,[c]\,(e) \\ & & \mid & \texttt{data}\,[c]\,(e) \end{array}$$

There are two classes, `cart` and `pol`. Associated with each class, c, is a type, τ_c, of the instance data for that class. For example, with the two classes just mentioned, we have

$$\tau_{\texttt{cart}} = \langle x \hookrightarrow \texttt{real}, y \hookrightarrow \texttt{real} \rangle$$
$$\tau_{\texttt{pol}} = \langle \rho \hookrightarrow \texttt{real}, \theta \hookrightarrow \texttt{real} \rangle$$

The type $\texttt{obj}\,[c]$ represents the type of objects whose class is known to be c, and whose instance data is therefore known to be of type τ_c. Thus we have the following two typing rules for introducing and eliminating objects:

$$\frac{\Gamma \vdash e : \tau_c}{\Gamma \vdash \texttt{new}\,[c]\,(e) : \texttt{obj}\,[c]} \qquad \frac{\Gamma \vdash e : \texttt{obj}\,[c]}{\Gamma \vdash \texttt{data}\,[c]\,(e) : \tau_c}$$

Notice that, so far, we have not made use of the type `obj`, but only of the types $\texttt{obj}\,[c]$ for a known class c.

The dynamics is completely straightforward, with the key rule expressing the inverse relation between the introduction and elimination forms:

$$\frac{e\ \mathsf{val}}{\texttt{new}\,[c]\,(e)\ \mathsf{val}} \qquad \frac{e\ \mathsf{val}}{\texttt{data}\,[c]\,(\texttt{new}\,[c]\,(e)) \mapsto e}$$

(a) (5 points) State the canonical forms theorem for the type $\texttt{obj}\,[c]$:

Solution: If $e\ \mathsf{val}$ and $e : \texttt{obj}\,[c]$, then $e = \texttt{new}\,[c]\,(e')$ for some class c and some $e' : \tau_c$ such that $e'\ \mathsf{val}$.

(b) (5 points) The above type system has a fundamental problem that arises when we consider conditional expressions of the form $\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, where $e : \texttt{bool}$, e_1 is a polar point, and e_2 is a Cartesian point. If e_1 and e_2 are of different classes, the conditional is ill-typed. GIve a one-line example of such an expression:

Solution:

$$\begin{array}{l} \texttt{if}\ e \\ \quad \texttt{then}\ \texttt{new}\,[\texttt{cart}]\,(\langle x \hookrightarrow 1.0, y \hookrightarrow 1.0 \rangle) \\ \quad \texttt{else}\ \texttt{new}\,[\texttt{pol}]\,(\langle \rho \hookrightarrow 1.0, \theta \hookrightarrow 0.0 \rangle) \end{array}$$

(c) (5 points) The usual solution to this problem is to introduce an *up-cast* that allows us to weaken the type of an object by "forgetting" its class:

$$\frac{\Gamma \vdash e : \texttt{obj}\,[c]}{\Gamma \vdash \texttt{up}\,[c]\,(e) : \texttt{obj}}$$

The type `obj` represents the type of any object at all, regardless of its class. Rewrite the preceding example using upcasts so that the overall type of the conditional is `obj`:

Solution:

$$\begin{array}{l}\texttt{if}\,e\\ \quad\texttt{then}\,\texttt{up[cart]}\,\texttt{(new[cart]}\,\texttt{(}\langle x \hookrightarrow 1.0, y \hookrightarrow 1.0\rangle\texttt{))}\\ \quad\texttt{else}\,\texttt{up[pol]}\,\texttt{(new[pol]}\,\texttt{(}\langle \rho \hookrightarrow 1.0, \theta \hookrightarrow 0.0\rangle\texttt{))}\end{array}$$

(d) (5 points) The dynamics of up-casting states that it forms a value of type `obj`:

$$\frac{e\ \mathsf{val}}{\texttt{up}\,[c]\,(e)\ \mathsf{val}}$$

State the canonical forms theorem for the type `obj`:

Solution: if $e\ \mathsf{val}$ and $e : \texttt{obj}$, then e is of the form $\texttt{up}\,[c]\,(e)$ for some class c and some e such that $e\ \mathsf{val}$ and $e : \texttt{obj}\,[c]$.

(e) (5 points) Up-casting solves the problem of the conditional, but it also introduces another problem: if you know only that an object has type `obj`, you cannot recover its instance data. Specifically, if $e : \texttt{obj}$, then $\texttt{data}\,[c]\,(e)$ is ill-typed. The usual solution to this problem is to introduce a *down-cast* that allows us to recover the class of an object:

$$\frac{\Gamma \vdash e : \texttt{obj}}{\Gamma \vdash \texttt{down}\,[c]\,(e) : \texttt{obj}\,[c]}$$

Give a dynamics for down-cast that preserves (does not disrupt) the canonical forms property for the types $\texttt{obj}\,[c]$ stated earlier. Be sure to take account of the possibility of down-casting an object of the wrong class!

Solution:

$$\frac{e \longmapsto e'}{\texttt{down}\,[c]\,(e) \longmapsto \texttt{down}\,[c]\,(e')}$$

$$\frac{}{\texttt{down}\,[c]\,(\texttt{up}\,[c]\,(e)) \longmapsto e}$$

$$\frac{c \neq c'}{\texttt{down}\,[c]\,(\texttt{up}\,[c']\,(e'))\ \mathsf{err}}$$

Question 3 [25]: Databases

We consider a simple abstraction of a relational database in which the type $\mathsf{db}(\tau)$ represents a database whose *schema* is given by the type τ. The schema may be any type at all, but in practice it is often a labeled product type of the form

$$\langle l_1 \hookrightarrow \tau_1, \ldots, l_n \hookrightarrow \tau_n \rangle.$$

The labels are the *columns*, or *attributes*, of the database, and the corresponding types describe the data in each column. For example, $\langle \mathsf{id} \hookrightarrow \mathsf{nat}, \mathsf{salary} \hookrightarrow \mathsf{nat} \rangle$ might be the schema of a database with two columns.

Many programming languages provide access to a database through the three constructs specified by the following grammar snippet:

$$e \ ::= \ \mathsf{empty}_\tau \ \mid \ \mathsf{stash}(e_0, e_1) \ \mid \ \mathsf{dbrec}(e, e_0, x.y.e_1) \ \mid \ \ldots$$

empty_τ creates a new database with schema τ without anything in it. $\mathsf{stash}(e_0, e_1)$ adds the value of e_0 as an *entry* in the database e_1. Last, the iterator, $\mathsf{dbrec}(e, e_0, x.y.e_1)$ permits computing with the entries in the database e: here e_0 is the computation to be performed when this database is empty while $x.y.e_1$ describe what to do when it is not. In this case, x will be bound to some row in the database, y will be the result of processing the rest of the database, and e_1 is an expression that carries out this computation when the database is not empty.

The typing rules for empty_τ and stash are as follows:

$$\frac{}{\Gamma \vdash \mathsf{empty}_\tau : \mathsf{db}(\tau)}\ \mathsf{empty} \qquad\qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \mathsf{db}(\tau)}{\Gamma \vdash \mathsf{stash}(e_1, e_2) : \mathsf{db}(\tau)}\ \mathsf{stash}$$

You will be asked to write the typing rule for dbrec, but first read on.

The dynamic semantics of the database operators are given by the following rules:

$$\frac{}{\mathsf{empty}_\tau\ \mathsf{val}}\ \mathsf{v\_empty} \qquad\qquad \frac{e_0\ \mathsf{val}}{\mathsf{stash}(e_0, e_1)\ \mathsf{val}}\ \mathsf{v\_stash}$$

$$\frac{e_0 \mapsto e_0'}{\mathsf{stash}(e_0, e_1) \mapsto \mathsf{stash}(e_0', e_1)}\ \mathsf{s\_stash}$$

$$\frac{e \mapsto e'}{\mathsf{dbrec}(e, e_0, x.y.e_1) \mapsto \mathsf{dbrec}(e', e_0, x.y.e_1)}\ \mathsf{s\_dbrec_1}$$

$$\frac{}{\mathsf{dbrec}(\mathsf{empty}_\tau, e_0, x.y.e_1) \mapsto e_0}\ \mathsf{s\_dbrec_2}$$

$$\frac{e\ \mathsf{val}}{\mathsf{dbrec}(\mathsf{stash}(e, e'), e_0, x.y.e_1) \mapsto \left[\begin{array}{r} e/x \\ \mathsf{dbrec}(e', e_0, x.y.e_1)/y \end{array}\right] e_1}\ \mathsf{s\_dbrec_3}$$

(a) (5 points) Write the typing rule for dbrec.

Solution:

$$\frac{\Gamma \vdash e : \mathsf{db}(\tau) \quad \Gamma \vdash e_0 : \tau' \quad \Gamma, x : \tau, y : \tau' \vdash e_1 : \tau'}{\Gamma \vdash \mathsf{dbrec}(e, e_0, x.y.e_1) : \tau'}\ \mathsf{dbrec}$$

(b) (5 points) Define a function *union* : $\mathsf{db}(\tau) \to \mathsf{db}(\tau) \to \mathsf{db}(\tau)$ that, given two databases with the same schema, creates a database containing the entries of both.

Solution:

$$\begin{array}{l} \lambda d_1 : \mathsf{db}(\tau). \\ \lambda d_2 : \mathsf{db}(\tau). \\ \quad \mathsf{dbrec}(d_1, d_2, x.y.\mathsf{stash}(x, y)) \end{array}$$

(c) (5 points) Give a definition for the function $\mathit{project}[l_i]$ of type

$$\mathsf{db}(\langle l_1 \hookrightarrow \tau_1, \ldots, l_n \hookrightarrow \tau_n \rangle) \to \mathsf{db}(\langle l_i \hookrightarrow \tau_i \rangle)$$

(where $1 \le i \le n$) that computes the projection of a database on the column labeled l_i.

Solution:

$$\begin{array}{l} \lambda d : \mathsf{db}(\langle l_1 \hookrightarrow \tau_1, \ldots, l_n \hookrightarrow \tau_n \rangle). \\ \quad \mathsf{dbrec}(d, \mathsf{empty}_{\langle l_i \hookrightarrow \tau_i \rangle}, x.y.\mathsf{stash}(\langle l_i \hookrightarrow x \cdot l_i \rangle, y)) \end{array}$$

(d) (5 points) Give a definition of the function *select* of type

$$(\tau \to \mathsf{bool}) \to \mathsf{db}(\tau) \to \mathsf{db}(\tau)$$

that selects those entries of a database that satisfy the given predicate. You can use the standard elimination form for bool's given by $\mathsf{if}(e, e_{\text{true}}, e_{\text{false}})$.

Solution:

$$\begin{array}{l} \lambda p : \tau \to \mathsf{bool}. \\ \lambda d : \mathsf{db}(\tau_n). \\ \quad \mathsf{dbrec}(d, \mathsf{empty}_\tau, x.y.\mathsf{if}(p\ x, \mathsf{stash}(x, y), y)) \end{array}$$

(e) (5 points) The above definition of dbrec does not give the programmer access to the rest of the current database in the recursive case. Define a variant of this construct that permits that, give its typing rule, and the one evaluation rule that is most affected by this change.

Solution: This variant has the form $\mathsf{dbrec}'(e, e_0, x.x'.y.e_1)$, which differs from dbrec by the presence of the additional abstractor x'.

The updated typing rule is:

$$\frac{\Gamma \vdash e : \mathsf{db}(\tau) \quad \Gamma \vdash e_0 : \tau' \quad \Gamma, x : \tau, x' : \mathsf{db}(\tau), y : \tau' \vdash e_1 : \tau'}{\Gamma \vdash \mathsf{dbrec}'(e, e_0, x.x'.y.e_1) : \tau'}\;\mathsf{dbrec}'$$

The evaluation rule most affected is *s_dbrec*$_3$, which becomes:

$$\frac{e\ \mathsf{val}}{\mathsf{dbrec}'(\mathsf{stash}(e, e'), e_0, x.x'.y.e_1) \mapsto \left[\begin{array}{r} e/x \\ e'/x' \\ \mathsf{dbrec}(e', e_0, x.y.e_1)/y \end{array}\right] e_1}\;\mathsf{s\_dbrec'}_3$$

Question 4 [25]: Type Safety

In this problem, we will extend PCF with binary trees, and prove safety theorems for this new extension. Of course, PCF can already encode binary trees, but for this problem, we will treat them as a primitive concept.

Syntax

We begin by giving the syntax of our language.

$$
\begin{array}{rcll}
\text{Types} & \tau & ::= & \mathsf{Tree}[\tau] \\
\\
\text{Expressions} & e & ::= & \mathsf{nil}[\tau] \\
 & & & \mathsf{tree}[\tau](e_1; e_2; e_3) \\
 & & & \mathsf{fold}(e_1; e_2; e_3)
\end{array}
$$

The type $\mathsf{Tree}[\tau]$ is a binary tree in which each node has a piece of associated data of type τ. Introductory expressions nil and tree are used to create trees. The eliminatory expression is fold. This has nothing to do with the fold of recursive types, but rather is analagous to the fold operation of a list.

Static Semantics

$$
\frac{}{\Gamma \vdash \mathsf{nil}[\tau] : \mathsf{Tree}[\tau]}\ (\textsc{Nil})
\qquad
\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash e_1 : \mathsf{Tree}[\tau] \qquad \Gamma \vdash e_2 : \mathsf{Tree}[\tau]}{\Gamma \vdash \mathsf{tree}[\tau](e; e_1; e_2) : \mathsf{Tree}[\tau]}\ (\textsc{Tree})
$$

$$
\frac{\Gamma \vdash e : \mathsf{Tree}[\tau_1] \qquad \Gamma \vdash f : \tau_1 \to \tau_2 \to \tau_2 \to \tau_2 \qquad \Gamma \vdash b : \tau_2}{\Gamma \vdash \mathsf{fold}(e; f; b) : \tau_2}\ (\textsc{Fold})
$$

(Nil) and (Tree) are straight-forward constructors of type $\mathsf{Tree}[\tau]$. We can use an expression e of type $\mathsf{Tree}[\tau]$ with $\mathsf{fold}(e; f; b)$. The argument b is the base case, or what the expression will evaluate to if $e = \mathsf{nil}[\tau]$. The argument f is the aggregation function, which takes the value of the node along with two intermediate results (one for the left subtree and one for the right), and computes the result.

Dynamic Semantics

We will use a lazy semantics for trees. Any tree is a value. Node values and subtrees will only be evaluated when necessary. Therefore, the value judgement for this language is defined as follows:

$$(\textsc{NilVal})\ \frac{}{\mathsf{nil}[\tau]\ \mathsf{value}} \qquad\qquad \frac{}{\mathsf{tree}[\tau](e; t_1; t_2)\ \mathsf{value}}\ (\textsc{TreeVal})$$

(a) (5 points) : A canonical forms lemma tells us what we can assume about well-typed values. Complete the following statement (but do not prove):

Lemma 1. *(Canonical Forms) If $e : \mathsf{Tree}[\tau]$ and e value then...*

Solution: *either $e = \mathsf{nil}[\tau]$ or $e = \mathsf{tree}[\tau](e_1; e_2; e_3)$ for some $e_1 : \tau$ and $e_2 : \mathsf{Tree}[\tau]$ and $e_3 : \mathsf{Tree}[\tau]$.*

There are three evaluation steps for fold. The first evaluates the first argument until it is determined if we are trying to use an empty tree or not. The second transition works on the empty tree by returning the argument b. The third transition works on the non-empty tree by first performing a fold on the two subtrees, and then combining the results, along with the value of the node, e by the function f.

$$(\textsc{FoldEval}_1)\ \frac{e \mapsto e'}{\mathsf{fold}(e; f; b) \mapsto \mathsf{fold}(e'; f; b)} \qquad\qquad \frac{}{\mathsf{fold}(\mathsf{nil}[\tau]; f; b) \mapsto b}\ (\textsc{FoldEval}_2)$$

$$\frac{}{\mathsf{fold}(\mathsf{tree}[\tau](e; e_1; e_2); f; b) \mapsto f\ e\ (\mathsf{fold}(e_1; f; b))\ (\mathsf{fold}(e_2; f; b))}\ (\textsc{FoldEval}_3)$$

Working with Trees

The elimination form for trees, fold, is analogous to the fold operation of a list in ML.
Example: If $e : \mathsf{Tree}[nat]$, then
$sum(e) = \mathsf{fold}(e; (\lambda x : \mathsf{nat}.\lambda y : \mathsf{nat}.\lambda z : \mathsf{nat}.x + y + z); 0)$
will compute the sum of all nodes in the tree.

(b) (5 points) : Write the expression $\mathsf{map}[\tau_1][\tau_2](f)$, where $f : \tau_1 \to \tau_2$ maps one element of the tree to another, and $\mathsf{map}[\tau_1][\tau_2](f) : \mathsf{Tree}[\tau_1] \to \mathsf{Tree}[\tau_2]$ lifts the operation to act on trees.

Solution: $\mathsf{map}[\tau_1][\tau_2](f) =$
$\lambda e : \mathsf{Tree}[\tau_1].\mathsf{fold}(e; (\lambda x : \tau_1.\lambda y : \mathsf{Tree}[\tau_2].\lambda z : \mathsf{Tree}[\tau_2].\mathsf{tree}[\tau_2](f\ x; y; z)); \mathsf{nil}[\tau_2])$

Proving Progress

Theorem 1. *(Progress) If $e : \tau$ then either e* value *or $e \mapsto e'$ for some e'.*

(c) (3 points) case (Nil):

Solution: $\mathsf{nil}[\tau]$ value

(d) (2 points) case (Tree):

Solution: $\mathsf{tree}[\tau](e; e_1; e_2)$ value

(e) (10 points) case (Fold):
(Hint: When appealing to the inductive hypothesis, there are two subcases to consider.)

Solution: We assume that (i) $e : \tau_1$ and (ii) $f : \tau_1 \rightarrow \tau_2 \rightarrow \tau_2 \rightarrow \tau_2$ and (iii) $b : \tau$ and need to show that either $\mathsf{fold}(e; f; b)$ value or $\mathsf{fold}(e; f; b) \mapsto e'$ for some e'.

1. e value or $e \mapsto e''$ for some e'', by IH on i
2. Assume e value
   (a) $e = \mathsf{nil}[\tau_1]$ or $e = \mathsf{tree}[\tau_1](e_1; e_2; e_3)$ for some e_1, e_2, e_3 by CFL, i, 2
   (b) Assume $e = \mathsf{nil}[\tau_1]$
      i. $\mathsf{fold}(e; f; b) \mapsto b$, by (FoldEval$_2$) and 2b
   (c) Assume $e = \mathsf{tree}[\tau_1](e_1; e_2; e_3)$ for some e_1, e_2, e_3
      i. $\mathsf{fold}(e; f; b) \mapsto f\ e_1\ (\mathsf{fold}(e_2; f; b))\ (\mathsf{fold}(e_3; f; b))$, by (FoldEval$_3$) and 2c
3. Assume $e \mapsto e''$, for some e''
   (a) $\mathsf{fold}(e; f; b) \mapsto \mathsf{fold}(e'; f; b)$, by (FoldEval$_1$) and 3

Scratch Work:

Scratch Work: