CS 476 Final Exam

Sandeep Joshi

TOTAL POINTS

74.5 / 100

QUESTION 1

1 1 **7 / 7**

- ✓ + **1 pts** **pattern-match on l1**
- ✓ + **1 pts** **pattern-match on l2**
- ✓ + **1.5 pts** **base case**
- ✓ + **1.5 pts** **add pair of first elements**
- ✓ + **2 pts** **recursive call**
- \- **1 pts** Significant syntax issues
- \+ **0 pts** graded
- \+ **1.5 pts** recursive call missing argument
- \+ **1 pts** pair with wrong elements
- \+ **1.5 pts** recursive call with extra argument
- \+ **1 pts** recursive call to wrong function
- \- **0.5 pts** extra function call
- \- **0.5 pts** missing connector
- \- **1 pts** add elements to end of list

QUESTION 2

2 2 **9 / 12**

- ✓ + **2 pts** **overall structure**
- ✓ + **2 pts** **sequence rule**
- ✓ + **1.5 pts** **new rule**
- ✓ + **1 pts** **correct two fields**
- \+ **2 pts** int rules
- ✓ + **2.5 pts** **method rule**
- \+ **1 pts** var rule for c
- \- **0.5 pts** fields instead of methods

QUESTION 3

3 3 **6 / 11**

- ✓ + **1 pts** **a1**
- ✓ + **0.5 pts** **a2: x**
- ✓ + **1.5 pts** **a2: y's**
- \+ **0.5 pts** a2: one y
- ✓ + **1 pts** **a3: d's**
- \+ **1 pts** a3: c's
- \+ **0.5 pts** a3: one d
- \+ **0.5 pts** a3: one c
- \+ **1 pts** b1: LHS
- \+ **1 pts** b1: call
- \+ **1 pts** b1: result
- \+ **0.5 pts** b1: all but last step
- \+ **1 pts** b2: LHS
- ✓ + **1 pts** **b2: RHS**
- ✓ + **1 pts** **b2: call and result**

1 This is two steps.

QUESTION 4

4 4 **10 / 10**

- ✓ + **4 pts** **a: right answer**
- \+ **2 pts** a: sensible wrong answer
- ✓ + **2 pts** **b: is a closure**
- ✓ + **2 pts** **b: function as-is**
- ✓ + **2 pts** **b: environment**
- \- **1 pts** Incorrect environment values
- \+ **0 pts** graded
- \- **0.5 pts** extra variables in closure environment
- \+ **2 pts** gave type of g instead of value
- \+ **1 pts** gave partial type of g instead of value

QUESTION 5

5 5 **6.5 / 12**

- \+ **1 pts** type inference on e1
- \+ **1 pts** type inference on e2
- \+ **1.5 pts** types of e1 and e2 are t1 and t2, not complex types
- ✓ + **3 pts** **correct type for e1 . e2**
- ✓ + **1 pts** **keep constraints C1 and C2**
- \+ **3 pts** add correct constraints
- ✓ + **1.5 pts** **fresh variables where necessary**

+ **0 pts** graded
- **1 pts** Added extra terms
- **3 pts** No meaningful change from function application rule
- **1 pts** Constraints formatted incorrectly
+ **1.5 pts** Added constraints somewhat correctly
+ **1 pts** Partially correct fresh variables
+ **2.5 pts** almost correct type for e1 . e2
✓ + **1 pts** some type inference above the line
+ **1.5 pts** correct type checking rule
- **1 pts** missing constraints below the line

2 You needed to come up with a new rule, not build a proof tree with the given rule.

QUESTION 6

6 6 **10.5 / 12**

✓ + **1 pts** find first goal
✓ + **1 pts** choose a rule
+ **0.5 pts** choose a rule from rs0 instead of rs
✓ + **1 pts** call to make_fresh
+ **0.5 pts** call to make_fresh with no arguments
+ **1 pts** extract conclusion and premises of rule
✓ + **1.5 pts** call to unify
✓ + **1 pts** new goals include premises and old goals
+ **1 pts** apply s1 to new goals
✓ + **0.5 pts** apply s1 to half of new goals
✓ + **1.5 pts** reset rules to rs0
✓ + **1.5 pts** compose s1 and s
✓ + **1.5 pts** new stack frame
+ **1 pts** new stack frame with wrong goals
+ **1 pts** new stack frame made with rs - r instead of list
+ **1 pts** new stack frame with missing rules
- **2 pts** missing connectors
+ **0 pts** graded

QUESTION 7

7 7 **5 / 12**

✓ + **1 pts** a: right general structure
+ **1 pts** a: precondition is sane
✓ + **1 pts** a: postcondition is true
✓ + **1 pts** a: postcondition is informative
✓ + **2 pts** b: if-then-else rule
+ **3 pts** b: rules of consequence for assignments
+ **2 pts** b: checked implications
+ **1 pts** b: assignment rules
+ **1.5 pts** one rule of consequence
+ **1 pts** one correct implication
+ **1 pts** if-then-else rule without specific condition

QUESTION 8

8 8 **10 / 12**

+ **4 pts** a
✓ + **3 pts** b: synchronization rule
✓ + **2 pts** b: labels on steps
+ **1.5 pts** labels with inconsistent values
+ **1 pts** labels with "v" instead of real value
✓ + **1 pts** b: send rule
+ **1 pts** b: var rule
+ **0.5 pts** number rule instead of variable rule
✓ + **1 pts** b: recv rule
- **1 pts** Extra rules
- **1 pts** missing or incorrect values in environment
+ **0 pts** graded
+ **3 pts** a: stepped one side
+ **3.5 pts** a: almost correct
✓ + **3 pts** a: wrong value sent
- **0.5 pts** primed variables
+ **1.5 pts** labels on steps not quite right

3 y =2 here

4 x is not in this environment

5 need to apply variable rule here

QUESTION 9

9 9 **10.5 / 12**

✓ + **1 pts** look up command in C
✓ + **1 pts** evaluate e3
✓ + **1 pts** evaluate e1
✓ + **1 pts** look up value at l
✓ + **2 pts** comparison - two cases
+ **1.5 pts** true case: evaluate e2

✓ + **1 pts** true case: update x

✓ + **1.5 pts** true case: update store

✓ + **1 pts** false case: update x

✓ + **1 pts** step p to p+1

+ **0.5 pts** lookup in C partially correct

+ **0 pts** graded

+ **2 pts** cmpxchg is just load

+ **1 pts** sigma(e) instead of evaluating e

+ **1 pts** comparison below the line

+ **1 pts** return values instead of new state

- **0.5 pts** made e1 a memory location

CS 476 Fall 2018 Final

Name:	SANDEEP JOSHI
NetID:	659263861

- You have **2 hours** to complete this exam.
- This is a **closed-book** exam.
- Do not share anything with other students. Do not talk to other students. Do not look other students' exams. Do not expose your exam to easy viewing by other students. Violation of any of these rules will be considered cheating.
- If you believe there is an error or an ambiguous question, you may seek clarification from the instructor. Please speak quietly or write your question out.
- Including this cover sheet and rules at the end, there are 13 pages to the exam, including one blank page for workspace. Once the exam begins (and not before!), please verify that you have all 13 pages.
- Please write your name and NetID in the spaces above, and also in the provided space at the top of every sheet.
- Show your work. Partial credit will be given for incomplete answers.
- The pages at the end of the exam contain inference rules for various systems. You may detach these pages. If you do, please turn them in with the rest of your exam.
- If you finish with time remaining, check your work!

Question	Points	Score
1	7	
2	12	
3	11	
4	10	
5	12	
6	12	
7	12	
8	12	
9	12	
Total:	100	

Problem 1. (7 points)

Write an OCaml function `zip : int list -> bool list -> (int * bool) list` such that each element in the list returned by `zip l1 l2` is a pair of an element from `l1` and the corresponding element from `l2`. For instance, `zip [1; 2; 3] [true; false; true]` should return `[(1, true); (2, false); (3, true)]`. You may assume that the two inputs are always the same length. For full credit, do **not** use built-in library functions like `map`.

```
let rec zip (l1 : int list) (l2 : bool list) : (int * bool) list =
```

```
match l1, l2 with
| l :: l-rest , v :: v-rest ⟶ [(l, v)] @ zip l-rest v-rest]
| [], [] ⟶ []
```

Problem 2. (12 points)

The typing rules for a simple object-oriented language are given in Appendix A. Suppose the following classes are declared in the class table CT:

```
class Square extends Object {
  int side;

  int area(){
    return this.side * this.side;
  }
}
```

```
class ColorSquare extends Square {
  int color;

  int getColor(){
    return this.color;
  }
}
```

Write a proof tree for the judgment

$$\Gamma \vdash \texttt{c := new ColorSquare(2, 1); x := c.area()} : \mathrm{ok}$$

given that $\Gamma(\texttt{c}) = \texttt{ColorSquare}$ and $\Gamma(\texttt{x}) = \text{int}$.

Methods(CT, Colorspace) = {.. int area(){...} ...}

$$\dfrac{\dfrac{\Gamma(c) = \text{ColorSquare} \quad (\text{fields}(CT, \text{ColorSpace}) = \text{int side, int color}) \quad \Gamma \vdash 2 : \text{int} \quad \Gamma \vdash 1 : \text{int}}{\Gamma \vdash c := \text{new ColorSquare}(2,1) : ok} \quad \cancel{\Gamma \vdash x := c.area(); ok} \quad \dfrac{\Gamma(c) = \text{Colorspace} \quad \Gamma(x) = \text{int}}{\Gamma \vdash x = c.area() : ok}}{\Gamma \vdash c := \text{new ColorSquare}(2,1);\ x := c.area() : ok}$$

Problem 3. (11 points)

(a) (5 points) For each of the following lambda-terms, draw a line from each variable occurrence to the place where it is bound.

- $\lambda a.\lambda b.\ a$
- $\lambda y.\ \lambda x.\ \lambda y.\ y\ x\ y$
- $\lambda c.\ (\lambda d.\ c\ d)\ (\lambda d.\ d\ c)$

(b) (6 points) Consider the lambda-term $((\lambda x.\ x)\ (\lambda y.\ y))\ ((\lambda x.\ x\ x)\ (\lambda y.\ y))$.

- Evaluate the term according to call-by-name semantics, in which the argument to a function is not evaluated before the function is applied. Show each step.

$((\lambda x. x)(\lambda y. y))((\lambda x. x\ x)(\lambda y. y))$

$(\lambda x. \lambda x. x)$

$\cancel{(\lambda x. (\lambda y. y))}\ (\lambda x. \lambda x. x.\ \lambda x. x)$

$(\lambda x. \lambda x.\ \lambda x. \lambda x. x.\ \lambda x. x)$

- Evaluate the term according to call-by-value semantics, in which the argument to a function is fully evaluated before the function is applied. Show each step.

$((\lambda x. x)(\lambda y. y))\ ((\lambda x. x\ x)(\lambda y. y))$

$(\lambda y. y)\ ((\lambda y. y)(\lambda y. y))$

$(\lambda y. y)\ (\lambda y. y)$

$\lambda y. y$

Problem 4. (10 points)

Consider the following OCaml program:

```
let y = 3;;
let z = true;;
let g x = if z then x else y;;
let z = false;;
g 1;;
```

(a) (4 points) What value does the program return?

1

(b) (6 points) What is the value of g?

< fun g x = if z then x else y {z = true, y = 3} >

Problem 5. (12 points)

In a simple functional language, the type inference rule for function application is:

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \vdash e_2 : \tau_2 \mid C_2 \quad \tau \text{ fresh}}{\Gamma \vdash e_1\ e_2 : \tau \mid \{\tau_1 = \tau_2 \to \tau\} \cup C_1 \cup C_2}$$

Suppose we wanted to add an operator . to the language such that $f\ .\ g$ composes the functions f and g, that is, $(f\ .\ g)\ x$ is equal to $f\ (g\ x)$. If f if of type $b \to c$ and g is of type $a \to b$, then $f\ .\ g$ is of type $a \to c$. Write the type inference rule for the expression $e_1\ .\ e_2$. Remember that the types assigned to expressions in the premises must be unknown type variables (τ_1, τ_2, etc.) and not specific types (int, $a \to b$, etc.).

2

$$\frac{\Gamma \vdash e_1 : \tau_5 \mid C_3 \quad \Gamma \vdash e_4 : \tau_3 \mid C_4 \cup C_1 \cup C_2}{\Gamma \vdash e_1\ e_4 : \tau_2 \mid \{\tau_5 = \tau_3 \to \tau_2\}} \text{ fresh } \tau_2 \qquad \frac{\Gamma \vdash e_2 : \tau_4 \mid C_1 \quad \Gamma \vdash e_3 : \tau_1 \mid C_2}{\Gamma \vdash e_2\ e_3 : \tau_3 \mid \{\tau_4 = \tau_1 \to \tau_3\} \cup C_1 \cup C_2} \text{ fresh } \tau_3$$

$$\frac{}{\Gamma \vdash e_1 . e_2 : \tau \mid \{\tau = \tau_1 \to \tau_2\} \cup C_1 \cup C_2 \cup C_3 \cup C_4} \text{ fresh } \tau$$

Problem 6. (12 points)

The core algorithm of logic programming maintains a state with four components: a list gs of goals to prove, a set R of rules that have not yet been tried on the current goal, a substitution σ that holds the solution to the query, and a backtracking stack k. The algorithm proceeds as follows:

1. Let the current goal be the first goal of gs, which we will call g.
2. Choose a rule r from the set R to apply to g.
3. Make a fresh copy of r so that none of its variables clash with the variables of gs. Call the conclusion of this fresh copy t and its premises $t_1, ..., t_n$.
4. Unify the conclusion t with the goal g, obtaining a substitution σ_1.
5. Make the new state of the algorithm as follows:
   - Make the new list of goals by applying σ_1 to the list $t_1, ..., t_n$ followed by the rest of gs.
   - Set the set of rules not yet tried to R_0, the full set of rules from the original program.
   - Make the new substitution by composing σ_1 with the old substitution σ.
   - Make the new stack by adding the frame $(gs, R - \{r\}, \sigma)$ to the top of the old stack k.

Suppose you were implementing an interpreter for a logic language by writing a function `logic_step` that takes arguments `rs0`, `gs`, `rs`, `s`, and `k`, corresponding to R_0, gs, R, σ, and k respectively. Fill in the case of `logic_step` that executes the above algorithm. When choosing a rule to apply, you can simply choose the first element of `rs`. You do not need to provide code for the cases in which this process fails, for instance because there are no goals left or because unification fails. You may assume the existence of the following helper functions: `make_fresh`, which makes a fresh copy of a rule; `unify`, which performs unification; `apply_subst`, which applies a substitution to a list of goals; and `compose_subst`, which takes two substitutions and composes them.

```
let logic_step (rs0 : rule list) (gs : term list) (rs : rule list) (s : substitution) (k : stack)
  : term list * rule list * substitution * stack =
```

```
match gs , rs with
| g :: gs-rest , r :: rs-rest -> let t = make-fresh (r) in
                                 let sig' = unify t g in
                                 let new-gs = apply-subst sig' t :: gs-rest in
                                 let new-subst = compose-subst s sig' in

                                 (new-gs, rs0, new-subst, (gs, rs-rest, s) :: k)
```

Problem 7. (12 points)

(a) (4 points) Give an informative precondition and postcondition for the following program:

```
if x > y then z := x else z := y
```

~~$\{true \wedge x \text{ is int} \wedge y \text{ is int}\}$ if $x > y$ then $z := x$ else $z := y$ $\{z = x \vee z = y\}$~~

(b) (8 points) The proof rules for Floyd-Hoare Logic are given in Appendix B. Build a proof tree showing that the program satisfies the precondition and postcondition you gave in part (a). Make sure to check any necessary implications.

Answer to (a)

$\{true \wedge \exists a \wedge a < x \wedge a < y\}$ if $x > y$ then $z := x$ else $z := y$ $\{z > a \wedge (z = x \vee z = y) \wedge (z = Max(x, y))\}$

Answer to (b)

$$\dfrac{\{true \wedge \exists a \wedge a<x \wedge a<y \wedge (x>y) \Downarrow true\}\ z := x\ \left\{\begin{array}{l} z > a \wedge \\ (z = x \vee z = y) \wedge \\ (z = Max(x,y)) \end{array}\right\} \qquad \left\{\begin{array}{l} true \wedge \exists a \wedge \\ a<x \wedge a<y \wedge \\ (x>y) \Downarrow false \end{array}\right\}\ z := y\ \left\{\begin{array}{l} z > a \wedge \\ (z = x \vee z = y) \wedge \\ (z = Max(x,y)) \end{array}\right\}}{\{true \wedge \exists a \wedge (a<x) \wedge (a<y)\}\ \text{if } x > y \text{ then } z := x \text{ else } z := y\ \{z > a \wedge (z = x \vee z = y) \wedge (z = Max(x,y))\}}$$

Problem 8. (12 points)

The operational semantics for a simple concurrent programming language with synchronous communication are given in Appendix C.

(a) (4 points) What is the next step that the configuration $(\text{send}(x), \{x = 2\}) \mid\mid (y = \text{recv}(), \{y = 1\})$ takes?

$$(\text{Send}(x), \{x=2\}) \mid\mid (y = \text{recv}(), \{y=1\}) \longrightarrow (\text{Skip}, \{x=2\}) \mid\mid (\text{Skip}, \{y=1, x=2\})$$

(b) (8 points) Construct a proof tree justifying your answer to (a).

$$\dfrac{\dfrac{(\{x=2\}, x) \Downarrow 2}{(\text{Send}(x), \{x=2\}) \xrightarrow{out\ 2} (\text{Skip}, \{x=2\})} \qquad \dfrac{}{(y := \text{recv}(), \{y=1\}) \xrightarrow{in\ 2} (\text{Skip}, \{y=1, x=2\})}}{(\text{Send}(x), \{x=2\}) \mid\mid (y := \text{recv}(), \{y=1\}) \longrightarrow (\text{Skip}, \{x=2\}) \mid\mid (\text{Skip}, \{y=1, x=2\})}$$

Problem 9. (12 points)

The semantic rules for memory load and store operations in a simple assembly language are as follows:

$$\frac{C(p) = (x = \texttt{load}\ e) \quad (e, \rho) \Downarrow \ell \quad \sigma(\ell) = v}{C, L \vdash (p, \rho, \sigma) \to (p + 1, \rho[x \mapsto v], \sigma)} \qquad \frac{C(p) = (\texttt{store}\ e_1, e_2) \quad (e_1, \rho) \Downarrow v \quad (e_2, \rho) \Downarrow \ell}{C, L \vdash (p, \rho, \sigma) \to (p + 1, \rho, \sigma[\ell \mapsto v])}$$

Suppose we wanted to add a compare-and-exchange command `cmpxchg` to the language, such that $x = \texttt{cmpxchg}\ e_1, e_2, e_3$ does the following:

1. Evaluates e_3 to a memory location ℓ.
2. Compares the value of e_1 with the value in memory at ℓ.
3. If the two values are equal, stores the value of e_2 at ℓ and returns 1. If the two values are not equal, leaves the memory unchanged and returns 0.

Write one or more inference rules giving the semantics of `cmpxchg`. (Hint: it may be easiest to give two rules.)

$$\frac{C(p) = (x = \text{cmpxchg}\ e_1, e_2, e_3) \quad (e_3, \rho) \Downarrow l \quad \sigma(l) = v \quad Eq(e_1, v) \Downarrow true}{C, L \vdash (p, \rho, \sigma) \longrightarrow (p+1, \rho[x \to 1], \sigma[l \to e_2])} \qquad \frac{C(p) = (x = \text{cmpxchg}\ e_1, e_2, e_3) \quad (e_3, \rho) \Downarrow l \quad \sigma(l) = v \quad Eq(e_1, v) \Downarrow false}{C, L \vdash (p, \rho, \sigma) \longrightarrow (p+1, \rho[x \to 0], \sigma)}$$

A Typing Rules for a Simple Object-Oriented Language

$$\frac{}{C <: C} \qquad \frac{CT(C) = \text{class } C \text{ extends } D\{...\}}{C <: D} \qquad \frac{C <: D \quad D <: E}{C <: E}$$

$$\frac{(n \text{ is a number})}{\Gamma \vdash n : \text{int}} \qquad \frac{(b \text{ is a boolean})}{\Gamma \vdash b : \text{bool}} \qquad \frac{(\Gamma(x) = \tau)}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e : \tau_1 \quad \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2} \qquad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \oplus e_2 : \text{int}} \text{ where } \oplus \text{ is an arithmetic operator}$$

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \otimes e_2 : \text{bool}} \text{ where } \otimes \text{ is a boolean operator} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 = e_2 : \text{bool}} \qquad \frac{\Gamma \vdash e : C \quad (\text{fields}(CT, C) = ..., \tau f, ...)}{\Gamma \vdash e.f : \tau}$$

$$\frac{(\Gamma(x) = \tau) \quad \Gamma \vdash e : \tau}{\Gamma \vdash x := e : \text{ok}} \qquad \frac{\Gamma \vdash c_1 : \text{ok} \quad \Gamma \vdash c_2 : \text{ok}}{\Gamma \vdash c_1;\ c_2 : \text{ok}} \qquad \frac{(\Gamma(x) = C) \quad (\text{fields}(CT, C) = \tau_1\ f_1, ..., \tau_n\ f_n) \quad \Gamma \vdash e_1 : \tau_1\ ...\ \Gamma \vdash e_n : \tau_n}{\Gamma \vdash x := \texttt{new}(e_1, ..., e_n) : \text{ok}}$$

$$\frac{(\Gamma(x) = \tau) \quad \Gamma \vdash e : C \quad (\text{methods}(CT, C) = ..., \tau\ m(\tau_1\ x_1, ..., \tau_n\ x_n)\{...\}, ...) \quad \Gamma \vdash e_1 : \tau_1\ ...\ \Gamma \vdash e_n : \tau_n}{\Gamma \vdash x := e.m(e_1, ..., e_n) : \text{ok}}$$

B Floyd-Hoare Logic for a Simple Imperative Language

$$\frac{}{\{P\}\ \text{skip}\ \{P\}} \qquad \frac{\{P\}\ c_1\ \{Q\} \quad \{Q\}\ c_2\ \{R\}}{\{P\}\ c_1; c_2\ \{R\}} \qquad \frac{}{\{[x \mapsto e]P\}\ x := e\ \{P\}} \qquad \frac{P_1 \Rightarrow P_2 \quad \{P_2\}\ c\ \{Q_2\} \quad Q_2 \Rightarrow Q_1}{\{P_1\}\ c\ \{Q_1\}}$$

$$\frac{\{P \wedge (e = \text{true})\}\ c_1\ \{Q\} \quad \{P \wedge (e = \text{false})\}\ c_2\ \{Q\}}{\{P\}\ \text{if } e \text{ then } c_1 \text{ else } c_2\ \{Q\}} \qquad \frac{\{P \wedge (e = \text{true})\}\ c\ \{P\}}{\{P\}\ \text{while } e \text{ do } c\ \{P \wedge (e = \text{false})\}}$$

C Operational Semantics for a Simple Concurrent Language

$$\frac{(n \text{ is a number})}{(n, \sigma) \Downarrow n} \qquad \frac{(b \text{ is a boolean})}{(b, \sigma) \Downarrow b} \qquad \frac{(\sigma(x) = v)}{(x, \sigma) \Downarrow v}$$

$$\frac{(e_1, \sigma) \Downarrow v_1 \qquad (e_2, \sigma) \Downarrow v_2 \qquad (v_1 \oplus v_2 = v)}{(e_1 \oplus e_2, \sigma) \Downarrow v} \text{ where } \oplus \text{ is an arithmetic, comparison, or boolean operator}$$

$$\frac{(e, \sigma) \Downarrow v}{(x := e, \sigma) \to (\text{skip}, \sigma[x \mapsto v])} \qquad \frac{(c_1, \sigma) \to (c_1', \sigma')}{(c_1; c_2, \sigma) \to (c_1'; c_2, \sigma')} \qquad \frac{}{(\text{skip}; c_2, \sigma) \to (c_2, \sigma)} \qquad \frac{(e, \sigma) \Downarrow v}{(\text{send}(e), \sigma) \xrightarrow{\text{out}(v)} (\text{skip}, \sigma)} \qquad \frac{}{(x := \text{recv}(), \sigma) \xrightarrow{\text{in}(v)} (\text{skip}, \sigma)}$$

$\sigma[x \mapsto v]$

$$\frac{(e_1, \sigma_1) \to (e_1', \sigma_1')}{(e_1, \sigma_1) \mid\mid (e_2, \sigma_2) \to (e_1', \sigma_1') \mid\mid (e_2, \sigma_2)} \qquad \frac{(e_2, \sigma_2) \to (e_2', \sigma_2')}{(e_1, \sigma_1) \mid\mid (e_2, \sigma_2) \to (e_1, \sigma_1) \mid\mid (e_2', \sigma_2')}$$

$$\frac{(e_1, \sigma_1) \xrightarrow{\text{out}(v)} (e_1', \sigma_1') \qquad (e_2, \sigma_2) \xrightarrow{\text{in}(v)} (e_2', \sigma_2')}{(e_1, \sigma_1) \mid\mid (e_2, \sigma_2) \to (e_1', \sigma_1') \mid\mid (e_2', \sigma_2')} \qquad \frac{(e_1, \sigma_1) \xrightarrow{\text{in}(v)} (e_1', \sigma_1') \qquad (e_2, \sigma_2) \xrightarrow{\text{out}(v)} (e_2', \sigma_2')}{(e_1, \sigma_1) \mid\mid (e_2, \sigma_2) \to (e_1', \sigma_1') \mid\mid (e_2', \sigma_2')}$$

D Scratch Space