Final Exam

15-814 Types and Programming Languages
Frank Pfenning
December 13, 2018

Name: Sample Solution Andrew ID: fp

Instructions

• This exam is closed-book, closed-notes.
• You have 180 minutes to complete the exam.
• There are 5 problems.
• For reference, on pages 15–18 there is an appendix with sections on the syntax, statics, and dynamics.

<table>
<thead>
<tr>
<th></th>
<th>Parametric Polymorphism</th>
<th>Data Abstraction</th>
<th>Exceptions</th>
<th>Quotation</th>
<th>Session Types</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score</td>
<td>50</td>
<td>55</td>
<td>50</td>
<td>45</td>
<td>50</td>
<td>250</td>
</tr>
<tr>
<td>Max</td>
<td>50</td>
<td>55</td>
<td>50</td>
<td>45</td>
<td>50</td>
<td>250</td>
</tr>
</tbody>
</table>
1 Parametric Polymorphism (50 pts)

In this problem we use the implicit form of parametric polymorphism and we only allow pure
\( \lambda \)-expressions (in particular, we disallow fixed points \( \text{fix } x. e \)). As a reminder, we have the following
typing rules, with the usual provisos:

\[
\frac{\Delta, \alpha \text{ type } ; \Gamma \vdash e : \tau}{\Delta \vdash e : \forall \alpha. \tau} \quad (I-\forall) \\
\frac{\Delta ; \Gamma \vdash e : \forall \alpha. \tau \quad \Delta \vdash \sigma \text{ type}}{\Delta ; \Gamma \vdash e : \sigma/\alpha[\tau]} \quad (E-\forall)
\]

We define the a family of types only \( \tau \) by
\[
\text{only } \tau = \forall \gamma. (\tau \to \gamma) \to \gamma
\]

Task 1 (10 pts). Define
\[
in : \forall \alpha. \alpha \to \text{only } \alpha
\]

\[
in = \lambda x. \lambda f. f \, x
\]

Task 2 (10 pts). Define
\[
\text{out} : \forall \alpha. \text{only } \alpha \to \alpha
\]

\[
\text{out} = \lambda y. y \, (\lambda x. x)
\]

Task 3 (10 pts). Evaluate \( \text{out} \, (\text{in } v) \) for a closed value \( v : \tau \).

\[
\text{out} \, (\text{in } v) \\
\mapsto \text{out} \, (\lambda f. f \, v) \\
\mapsto (\lambda f. f \, v) \, (\lambda x. x) \\
\mapsto (\lambda x. x) \, v \\
\mapsto v
\]

Task 4 (10 pts). Evaluate \( \text{in} \, (\text{out } w) \) for a closed value \( w : \text{only } \tau \)
Task 5 (10 pts). Circle all statements that are true in the setting of this problem as explained at the beginning of this section.

(i) Any closed well-typed expression evaluates to a value.
(ii) There is no closed expression of type $\forall \alpha. \alpha$.
(iii) We can conclude without knowing the definitions of $\text{out}$ and $\text{in}$ that

$$(\text{out} \circ \text{in}) \sim (\lambda x. x) : \forall \alpha. \alpha \rightarrow \alpha$$

(iv) We can conclude without knowing the definitions of $\text{out}$ and $\text{in}$ that for any closed value $v : \tau$ we have

$$\text{out} \ (\text{in} \ v) \rightarrow^* v$$

(v) For any closed expression of type $e : \tau$ we have $e \sim e : \tau$.

(i)–(v) are all true.
2 Data Abstraction (55 points)

In this problem we explore data abstraction. More specifically, we consider whether the usual
convention in C-like languages that 0 = false and n = true for n > 0 is somehow defensible.

For this enterprise we use existential types to represent abstraction and logical equality to
reason about representation independence. Recall that the baseline for logical equality is Kleene
equality, $e \simeq e'$ which means that there is a value $v$ such that $e \mapsto^* v$ and $e' \mapsto^* v$. As during
lectures, we assume that all expressions we are concerned with terminate.

As a reminder, we define $e \sim e'$ inductively on the structure of $\tau$, assuming $e$ and $e'$ are
closed and of type $\tau$. We then close the relation on both sides under Kleene equality. Here are two
cases in the definition:

- $(\rightarrow) e \sim e' : \tau_1 \rightarrow \tau_2$ iff for all $v_1 \sim v_1' : \tau_1$ we have $e v_1 \sim e' v_1' : \tau_2$
- $(+) v \sim v' : \tau_1 + \tau_2$ iff either $v = l \cdot v_1$, $v' = l \cdot v_1'$, and $v_1 \sim v_1' : \tau_1$ or $v = r \cdot v_2$, $v' = r \cdot v_2'$, and $v_2 \sim v_2' : \tau_2$

Task 1 (5 pts). We define as usual, $bool = (false : 1) + (true : 1)$. Give a necessary and sufficient
condition for

$v \sim v' : bool$

for closed values $v$ and $v'$ of type $bool$ (which is then closed under Kleene equality to obtain
$e \sim e' : bool$).

Option 1: $v \sim v' : bool$ iff $v = v'$.
Option 2: $v \sim v' : bool$ iff either $v = true \cdot (\cdot) = v'$ or $v = false \cdot (\cdot) = v'$.

Task 2 (5 pts). We define as usual, $nat = \rho \alpha. (z : 1) + (s : \alpha)$. Give a necessary and sufficient
condition for

$v \sim v' : nat$

for closed values $v$ and $v'$ of type $nat$ (which is then closed under Kleene equality to obtain
$e \sim e' : nat$).

Option 1: $v \sim v' : nat$ iff $v = v'$.
Option 2: we define (inductively)

\[
\begin{align*}
 x \sim y : nat \\
 fold (z \cdot (\cdot)) & \sim fold (z \cdot (\cdot)) : nat \\
 fold (s \cdot x) & \sim fold (s \cdot y) : nat
\end{align*}
\]
Now we consider the type

$$BOOL = \exists \alpha. (bool \to \alpha) \otimes (\alpha \to \alpha) \otimes (\alpha \to bool)$$

which represents a module with hidden implementation type $\tau$ for $\alpha$ and functions

$$\begin{align*}
to & : bool \to \tau \quad \text{map a boolean to its representation} \\
neg & : \tau \to \tau \quad \text{negate the representation} \\
from & : \tau \to bool \quad \text{map a representation back to a boolean}
\end{align*}$$

In the first implementation, booleans are represented with type $bool$. For our own reasons, a Boolean value is internally represented by its negation.

$\begin{align*}
\text{Impl}_1 : BOOL \\
\text{Impl}_1 = \langle bool, \text{not}, \text{not}, \text{not} \rangle
\end{align*}$

In the second implementation, booleans are represented with type $\mathbb{N}$ where $0$ represents false and all non-zero numbers represent true.

$\begin{align*}
\text{Impl}_2 : BOOL \\
\text{Impl}_2 = \langle \mathbb{N}, \text{rep}, \text{neg}, \text{unrep} \rangle
\end{align*}$

**Task 3** (15 pts). Provide definitions for $\text{rep}$, $\text{neg}$ and $\text{unrep}$. You may use the following constructors and also pattern-match against them.

$$\begin{align*}
\text{False} &= \text{false} \cdot \langle \rangle \\
\text{True} &= \text{true} \cdot \langle \rangle \\
Z &= \text{fold} (z \cdot \langle \rangle) \\
S \; x &= \text{fold} (s \cdot x)
\end{align*}$$

Please make sure to explicitly state the type and the definition of each function.

$$\begin{align*}
\text{rep} &: bool \to \mathbb{N} \\
\text{rep} \; \text{False} &= Z \\
\text{rep} \; \text{True} &= S \; Z \\
\text{neg} &: \mathbb{N} \to \mathbb{N} \\
\text{neg} \; Z &= S \; Z \\
\text{neg} \; (S \; x) &= Z \\
\text{unrep} &: \mathbb{N} \to bool \\
\text{unrep} \; Z &= \text{false} \\
\text{unrep} \; (S \; x) &= \text{true}
\end{align*}$$

Now we want to prove that these two implementations are logically equivalent and therefore indistinguishable in a language satisfying parametricity.

**Task 4** (10 pts). Define an appropriate relation $R : bool \leftrightarrow \mathbb{N}$ between the representations.
Task 5 (10 pts). Prove that \(\text{not} \sim \text{rep} : \text{bool} \to R\).

\[
\begin{array}{c}
\text{True} R Z \\
\text{False} R S w
\end{array}
\]

\[
\begin{array}{c}
\text{val} \\
\text{w}
\end{array}
\]

\[
\begin{array}{c}
R_t \\
R_f
\end{array}
\]

\[
\begin{align*}
v & \sim v' : \text{bool} & \text{Assumption} \\
v = \text{False} & = v' \text{ or } v = \text{True} = v' & \text{By definition of } \sim \text{ at type } \text{bool} \\
v = \text{False} & = v' & \text{First subcase} \\
\text{True} R Z & & \text{By rule } R_t \\
(\text{not False}) R (\text{rep False}) & & \text{By closure under } \simeq \\
\text{not} & \sim \text{rep} & \text{By definition of } \sim \text{ at type } \text{bool} \to R \\
v = \text{True} & = v' & \text{Second subcase} \\
\text{False} R (S Z) & & \text{By rule } R_f \\
(\text{not True}) R (\text{rep True}) & & \text{By closure under } \simeq \\
\text{not} & \sim \text{rep} & \text{By definition of } \sim \text{ at type } \text{bool} \to R
\end{align*}
\]
Task 6 (10 pts). Proof that not $\sim$ neg : $R \rightarrow R$.

\[
\begin{array}{ll}
v \sim v' : R & \text{Assumption} \\
v R v' & \text{Assumption} \\
v = \text{True} \text{ and } v' = Z & \text{By definition of } \sim \text{ at } R \\
\text{False } R (S Z) & \text{First case (rule } R_t) \\
(not v) R (neg v') & \text{By rule } R_f \\
not \sim \neg & \text{By closure under } \simeq \\
v = \text{False and } v' = S v'' \text{ for some value } v'' & \text{Second case (rule } R_f) \\
\text{True } R Z & \text{By rule } R_t \\
(not v) R (neg v') & \text{By closure under } \simeq \\
not \sim \neg & \text{By definition of } \sim \\
\end{array}
\]

It should also be true that not $\sim$ unrep : $R \rightarrow \text{bool}$ but you do not have to prove this.
3 Exceptions in the K Machine (50 points)

In this problem we explore extending our functional language with exceptions. For simplicity, we have just two new forms of expressions:

\[
\begin{align*}
\text{Expressions} & \quad e & \ ::= & \quad \ldots \mid \text{fail} \mid \text{try } e \text{ catch } e'
\end{align*}
\]

The intended semantics is as follows.

- \textbf{try } e \textbf{ catch } e' \textbf{ evaluates } e. If it returns normally with value \( v \) we ignore the exception handler \( e' \) and return \( v \). If \( e \) raises an exception we handle this exception and continue evaluation with \( e' \).

- \textbf{fail} raises an exception instead of returning a value. The innermost enclosing handler (if there is one) will catch this exception; otherwise the whole computation will simply fail.

We do not formalize the usual dynamics, but here are some examples:

\[
\begin{align*}
\text{try } v_1 \text{ catch } v_2 & \mapsto^* v_1 \\
\text{try fail } v_2 & \mapsto^* v_2 \\
\text{try (try fail catch } v_1) \text{ catch } v_2 & \mapsto^* v_1 \\
\text{try fail catch fail} & \mapsto^* \text{fail} \\
\text{(try } \lambda x. \text{fail } \text{ catch } v_2) v_1 & \mapsto^* \text{fail}
\end{align*}
\]

The last example illustrates the scoping of the try/catch blocks.

\textbf{Task 1} (10 pts). Give typing rules for the new expressions such that type preservation holds.

\[
\begin{array}{c}
\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash e' : \tau}{\Gamma \vdash \text{try } e \text{ catch } e' : \tau} \\
\hline
\frac{\Gamma \vdash \text{fail} : \tau}{\Gamma \vdash \text{fail} : \tau}
\end{array}
\]
Task 2 (15 pts). Extend the K machine so that there are three possible forms of states $s$:

- $k \triangleright e$: evaluate $e$ with continuation $k$
- $k \triangleleft v$: return value $v$ to continuation $k$
- $k \triangledown$ fail: signal an exception to continuation $k$

In addition to the new rules, indicate if any of the existing rules need to be changed.

The existing rules remain unchanged.

$$
\begin{align*}
    k \triangleright \text{fail} & \quad \mapsto \quad k \triangleleft \text{fail} \\
    k \triangleright (\text{try } e \text{ catch } e') & \quad \mapsto \quad k \circ (\text{try } \_ \text{ catch } e') \triangleright e \\
    k \circ (\text{try } \_ \text{ catch } e') \triangleleft v & \quad \mapsto \quad k \triangleleft v \\
    k \circ (\text{try } \_ \text{ catch } e') \triangledown \text{fail} & \quad \mapsto \quad k \triangleright e' \\
    k \circ f \triangledown \text{fail} & \quad \mapsto \quad k \triangleleft \text{fail} \quad \text{for } f \neq (\text{try } \_ \text{ catch } e')
\end{align*}
$$

Task 3 (5 pts). Recall that we typed continuations as $k \div \tau \Rightarrow \sigma$, expressing that $k$ maps a value of type $\tau$ to a final answer of type $\sigma$. Provide the typing rules for all new forms of continuation from your answer in Task 2.

$$
\frac{ k \div \tau \Rightarrow \sigma \quad \vdash e' : \tau }{ k \circ (\text{try } \_ \text{ catch } e') \div \tau \Rightarrow \sigma }
$$
**Task 4** (10 pts). We write $s: \sigma$ if state $s$ returns a final answer of type $\sigma$ if it terminates. There are three typing rules, one for each kind of state. We have filled in one for you already supply the other two.

$$\frac{k \vdash \tau \Rightarrow \sigma \quad \vdash e : \tau}{k \triangleright e : \sigma}$$

\[
\begin{array}{ll}
k \vdash \tau \Rightarrow \sigma \quad \vdash v : \tau \quad v \text{val} & \quad k \vdash \tau \Rightarrow \sigma \\
\hline
k \lhd v : \sigma & k \triangleright \text{fail} : \sigma
\end{array}
\]

**Task 5** (10 pts). State the progress theorem for the extended K machine.

If $s: \sigma$ then either (i) $s \rightarrow s'$ for some $s'$ or (ii) $s = \epsilon \lhd v$ for some $v$ val or (iii) $s = \epsilon \triangleright \text{fail}$. 
4 Quotation (45 points)

In this problem we explore quotation and staged computation. Recall the judgment $\Psi ; \Gamma \vdash e : \tau$ where $\Psi$ contains expression variables $u : \tau$ and $\Gamma$ contains ordinary value variables $x : \tau$. We have one new type constructor $\Box \tau$ with the following statics:

$$\frac{\Psi ; \vdash e : \tau}{\Psi ; \Gamma \vdash \text{box } e : \Box \tau} \quad (\text{I-}\Box) \quad \frac{\Psi ; \Gamma \vdash e : \Box \tau \quad \Psi, u : \tau ; \Gamma \vdash e' : \tau'}{\Psi ; \Gamma \vdash \text{case } e \{ \text{box } u \Rightarrow e' \} : \tau'} \quad (\text{E-}\Box)$$

We define the booleans as usual as $\text{bool} = (\text{false} : 1) + (\text{true} : 1)$ and allow definitions by pattern matching that can be desugared into the usual case constructs as in Problem 2. In particular:

- $\text{not} : \text{bool} \to \text{bool}$
- $\text{not False} = \text{True}$
- $\text{not True} = \text{False}$
- $\text{and} : \text{bool} \to \text{bool} \to \text{bool}$
- $\text{or} : \text{bool} \to \text{bool} \to \text{bool}$

We have omitted the definitions of $\text{and}$ and $\text{or}$. We assume these three functions as well as the constructors False and True can be used freely, including inside quoted expressions $\text{box } e$.

**Task 1** (10 pts). Write a well-typed (that is, properly staged) function

$$\text{and}' : \text{bool} \to \Box(\text{bool} \to \text{bool})$$

$$\begin{align*}
\text{and}' \text{ True} &= \text{box } (\lambda y. y) \\
\text{and}' \text{ False} &= \text{box } (\lambda y. \text{False})
\end{align*}$$

**Task 2** (5 pts). The proposed staged definition for equivalence of booleans,

$$\text{equiv}' : \text{bool} \to \Box(\text{bool} \to \text{bool})$$

$$\text{equiv}' x = \text{box } (\lambda y. \text{or } (\text{and } x y) (\text{and } (\text{not } x) (\text{not } y)))$$

is not well-typed. Explain where and why typing fails.

The two occurrences of $x$ in the right-hand side are ordinary variables bound outside the $\text{box}$ but used inside, which is prohibited.
Task 3 (10 pts). Restage the definition of $equiv'$ so it is correctly typed, using $and'$ from Task 1 wherever possible.

\[
aux : \square(bool \rightarrow bool) \rightarrow \square(bool \rightarrow bool) \rightarrow \square(bool \rightarrow bool)
\]
\[
aux(box\,u)\,(box\,w) = box(\lambda y.\,or\,(u\,x)\,(w\,(not\,y)))
\]
\[
equiv'\,x = aux\,(and'\,x)\,(and'\,(not\,x))
\]

Task 4 (10 pts). Implement directly an even more streamlined staged version of equivalence.

\[
equiv'' : bool \rightarrow \square(bool \rightarrow bool)
\]

\[
equiv''\,False = box(\lambda y.\,not\,y)
\]
\[
equiv''\,True = box(\lambda y.y)
\]

Task 5 (10 pts). Circle all true statements.

(i) We can define a function $bool \rightarrow \square bool$.

(ii) We can define a function $\forall \alpha.\,\alpha \rightarrow \square \alpha$.

(iii) We can define a function $\forall \alpha.\,\square \alpha \rightarrow \alpha$.

(iv) We can define a function $\forall \alpha.\,\forall \beta.\,(\square \alpha) \otimes (\square \beta) \rightarrow \square (\alpha \otimes \beta)$.

(v) We can define a function $\forall \alpha.\,\forall \beta.\,\square (\alpha \otimes \beta) \rightarrow (\square \alpha) \otimes (\square \beta)$

(i), (iii), (iv), and (v) are true; (ii) is false.
5 Session Types (50 points)

For a quick reference on session types and processes, see page 18 in the appendix. As usual in this course, we define numbers in binary representation as

\[ \text{bin} = \oplus \{ b_0 : \text{bin}, b_1 : \text{bin}, \epsilon : 1 \} \]

Task 1 (10 pts). Complete the following definition of zero.

\[ \vdash \text{zero} :: (z : \text{bin}) \]
\[ z \leftarrow \text{zero} = \]

\[ z \leftarrow \text{zero} = \]
\[ z.\epsilon ; \text{close} \ z \]

Task 2 (10 pts). Complete the following definition of succ, which produces on y the sequence of bits representing the successor of x.

\[ x : \text{bin} \vdash \text{succ} :: (y : \text{bin}) \]
\[ y \leftarrow \text{succ} \leftarrow x = \]
\[ \text{case } x (b_0 \Rightarrow \]
\[ | b_1 \Rightarrow \]
\[ | \epsilon \Rightarrow \]
\[ ) \]

\[ x : \text{bin} \vdash \text{succ} :: (y : \text{pos}) \]
\[ y \leftarrow \text{succ} \leftarrow x = \]
\[ \text{case } x (b_0 \Rightarrow y.b_1 ; y \leftarrow x \]
\[ | b_1 \Rightarrow y.b_0 ; y \leftarrow \text{succ} \leftarrow x \]
\[ | \epsilon \Rightarrow y.b_1 ; y.\epsilon ; \text{wait } y ; \text{close } x ) \]

Task 3 (10 pts). Complete the following definition of the predecessor process pred. It produces on y a sequence of bits representing the predecessor of x, where x must represent a strictly positive number. This constraint is expressed by the type

\[ \text{pos} = \oplus \{ b_0 : \text{pos}, b_1 : \text{bin} \} \]
\[ \text{x : pos \vdash pred :: (y : bin)} \]

\[ \text{y \leftarrow pred \leftarrow x =} \]

\[
\begin{array}{l}
\text{y \leftarrow pred \leftarrow x =} \\
\text{\hspace{1em} case x (b0 \Rightarrow y.b1 ; y \leftarrow pred \leftarrow x} \\
\text{\hspace{2em} | b1 \Rightarrow y.b0 ; y \leftarrow x )} \\
\end{array}
\]

**Task 4 (15 pts).** Define the following process that calculates the number of bits in \( x \) and outputs that number along \( y \). We define this as the number of \( b0 \) and \( b1 \) labels, and not counting \( \epsilon \). You may use \( \text{zero} \), \( \text{succ} \), and \( \text{pred} \) as needed, at the indicated types.

\[ \text{x : bin \vdash numbits :: (y : bin)} \]

\[ \text{y \leftarrow numbits \leftarrow x =} \]

\[
\begin{array}{l}
\text{y \leftarrow numbits \leftarrow x =} \\
\text{\hspace{1em} case x (b0 \Rightarrow y' \leftarrow numbits \leftarrow x ;} \\
\text{\hspace{2em} y \leftarrow succ \leftarrow y'} \\
\text{\hspace{2em} | b1 \Rightarrow y' \leftarrow numbits \leftarrow x ;} \\
\text{\hspace{2em} y \leftarrow succ \leftarrow y'} \\
\text{\hspace{2em} | \epsilon \Rightarrow \text{wait} \ x \ ; \ y \leftarrow \text{zero} )} \\
\end{array}
\]

**Task 5 (5 pts).** We might conjecture that the number of bits in a strictly positive binary number is equal to the floor of the logarithm of that number plus one, that is \( \text{numbits}(n) = \lfloor \log_2(n) \rfloor + 1 \) provided \( n > 0 \). However, this is not the case. Explain briefly why, and how you might write the logarithm function (you do not need to write any code).

\[ \text{The number could have leading zeroes.} \]
\[ \text{We could fix this by ensuring in the representation that there are never any leading zeros.} \]
\[ \text{This could be enforced, for example, by changing the type \text{bin} to be more strict.} \]
\[ \text{Or we could write a new function \text{numbits}' that avoids counting leading zeroes.} \]
Appendix: Some Inference Rules

A Syntax

Types $\tau$ and terms $e$ are given by the following grammars, where $I$ ranges over finite index sets. We present disjoint sums in their $n$-ary form and lazy pairs in their binary form, because it is these forms we use in this exam.

$$
\tau ::= \alpha \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \otimes \tau_2 \mid 1 \mid \sum_{i \in I} (i : \tau_i) \mid \tau_1 \& \tau_2 \mid \rho(\alpha, \tau)
$$

$$
e ::= x \mid \lambda x. e \mid e_1 e_2 \mid i \cdot e \mid \text{case } e \{i \cdot x_i \Rightarrow e_i\}_{i \in I} \mid \langle e_1, e_2 \rangle \mid \text{case } e_0 \{\langle x_1, x_2 \rangle \Rightarrow e'\} \mid \langle\rangle \mid \text{case } e_0 \{\langle\rangle \Rightarrow e'\} \mid \langle e_1, e_2 \rangle \mid e \cdot l \mid e \cdot r \mid \text{fold}(e) \mid \text{unfold}(e) \mid \text{fix}(x. e) \mid\text{recursion}
$$
B  Statics, Expressions: $\Gamma \vdash e : \tau$

\[
\begin{array}{c}
x : \tau \in \Gamma \\
\Gamma \vdash x : \tau \quad (\text{VAR})
\end{array}
\]

\[
\begin{array}{c}
\Gamma, x : \tau \vdash e : \tau' \\
\Gamma \vdash \lambda x. e : \tau \to \tau' \quad (I \to)
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash e_1 : \tau \to \tau' \\
\Gamma \vdash e_2 : \tau \\
\Gamma \vdash e_1 e_2 : \tau'
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash e : \tau_j \\
\Gamma \vdash j \cdot e : \sum_{i \in I} (i : \tau_i) \\
(I +)
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash e : \sum_{i \in I} (i : \tau_i) \\
\Gamma, x_i : \tau_i \vdash e_i : \tau \quad (\forall i \in I)
\Gamma \vdash \text{case} \ e \{i \cdot x_i \Rightarrow e_i\}_{i \in I} : \tau \\
\text{(E +)}
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash e_1 : \tau_1 \\
\Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \otimes \tau_2 \\
\text{(I –)}
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash \text{fold}(e) : \rho(\alpha.\tau) \\
\text{(I -\rho)}
\end{array}
\]

\[
\begin{array}{c}
\Gamma, x : \tau \vdash e : \tau \\
\Gamma \vdash \text{fix}(x. e) : \tau \\
\text{(FI X)}
\end{array}
\]

C  Statics, Closed Values: $v :: \tau$

\[
\begin{array}{c}
x : \tau \vdash e : \tau' \\
\text{\lambda x. e :: \tau \to \tau'} \quad (\text{IV –})
\end{array}
\]

\[
\begin{array}{c}
v :: \tau_j \\
\text{\{ j \in I \}} \\
\text{(IV +)}
\end{array}
\]

\[
\begin{array}{c}
v_1 :: \tau_1 \\
v_2 :: \tau_2 \\
\langle v_1, v_2 \rangle :: \tau_1 \otimes \tau_2 \\
\text{(IV -\otimes)}
\end{array}
\]

\[
\begin{array}{c}
\text{\cdot \vdash e_1 : \tau_1} \\
\text{\cdot \vdash e_2 : \tau_2} \\
\langle e_1, e_2 \rangle :: \tau_1 \& \tau_2 \\
\text{(IV -\&)}
\end{array}
\]

\[
\begin{array}{c}
v :: \rho(\alpha.\tau) \\
\text{\{ \}} :: 1 \\
\text{(IV -\1)}
\end{array}
\]

\[
\begin{array}{c}
v :: [\rho(\alpha.\tau)/\alpha]_\tau \\
\text{fold}(v) :: \rho(\alpha.\tau) \\
\text{(IV -\rho)}
\end{array}
\]
D  Dynamics: $e \mapsto e'$ and $v$ \textit{val}

\[
\begin{align*}
\frac{\lambda x.e \ \text{val}}{\lambda x.e \val} & \quad \frac{v_1 \val}{(V\rightarrow)} \\
\frac{e_1 \mapsto e'_1}{e_1 \mapsto e'_1} & \quad \frac{v_1 \mapsto e'_1}{(CE\rightarrow_1)} \\
\frac{e_1 \mapsto e'_1}{e_1 \ e_2 \mapsto e'_1 \ e_2} & \quad \frac{v_1 \ e_2 \mapsto e'_1 \ e_2}{(CE\rightarrow_2)} \\
\frac{v \ \text{val}}{i \cdot v \ \text{val}} & \quad \frac{e \mapsto e'}{(CI\rightarrow)} \quad \frac{\case e \ \{i \cdot x_i \Rightarrow e_i\}_{i \in I} \mapsto e'}{\textit{case} e' \ \{i \cdot x_i \Rightarrow e_i\}_{i \in I}} \\
\frac{v_1 \ e_2 \mapsto e'_1 \ e_2}{(R\rightarrow)} & \quad \frac{v_1 \ e_2 \mapsto e'_1 \ e_2}{(CE\rightarrow_2)}
\end{align*}
\]
**Session Types**

Process expressions: forward, spawn, and tail-call

\[ c \leftarrow d \quad \text{implement } c \text{ by } d \text{ and terminate} \]
\[ x \leftarrow f \leftarrow d_1, \ldots, d_n ; Q \quad \text{spawn } f, \text{ passing } \text{it channels } d_1, \ldots, d_n \]
\[ f \text{ will provide a fresh channel } a \text{ to client } [a/x]Q \]
\[ c \leftarrow f \leftarrow d_1, \ldots, d_n \quad \text{tail call to } f \text{ providing } c \text{ and using } d_1, \ldots, d_n \]

Session types and process expressions: message passing

<table>
<thead>
<tr>
<th>Type</th>
<th>Provider</th>
<th>Client</th>
<th>Continuation Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c \oplus { \ell : A_\ell }_{\ell \in L} )</td>
<td>((c.k : P))</td>
<td>\text{case } c { \ell \Rightarrow Q_\ell }_{\ell \in L} \</td>
<td>( c : A_k )</td>
</tr>
<tr>
<td>( c : &amp; { \ell : A_\ell }_{\ell \in L} )</td>
<td>case ( c { \ell \Rightarrow P_\ell }_{\ell \in L} )</td>
<td>((c.k : Q))</td>
<td>( c : A_k )</td>
</tr>
<tr>
<td>( c : 1 )</td>
<td>\text{close } c</td>
<td>\text{wait } c ; Q</td>
<td>(none)</td>
</tr>
</tbody>
</table>

Statics (where \(|y_1 : A_1, \ldots, y_n : A_n| = y_1, \ldots, y_n|)

\[
\begin{align*}
\Delta_1 \vdash f :: (x : A) \quad \Delta_2, x : A \vdash Q :: (z : C) \\
\Delta_1, \Delta_2 \vdash (x \leftarrow f \leftarrow \Delta_1 ; Q) :: (z : C) & \quad \text{spawn} \\
\Delta \vdash (x \leftarrow f \leftarrow |\Delta|) :: (x : A) & \quad \text{tail call} \\
\end{align*}
\]

\[
\begin{align*}
k \in L \\
\Delta \vdash P :: (x : A_k) & \quad \oplus R \\
\Delta \vdash (x.k : P) :: (x : \oplus \{ \ell : A_\ell \}_{\ell \in L}) & \quad \oplus L \\
\text{(for all } \ell \in L) \quad \Delta \vdash P_\ell :: (x : A_\ell) & \quad \& R \\
\Delta \vdash (\text{case } x \{ \ell \Rightarrow P_\ell \}_{\ell \in L}) :: (x : \& \{ \ell : A_\ell \}_{\ell \in L}) & \quad \& L \\
\Delta \vdash Q :: (z : C) & \quad 1 R \\
\Delta, x : 1 \vdash (\text{wait } x ; Q) :: (z : C) & \quad 1 L
\end{align*}
\]

Dynamics

(idC) \( \text{proc } P \ c \leftarrow d, \text{proc } (c \leftarrow d) c \mapsto \text{proc } ([c/d]P) c \)

(spawnC) \( \text{proc } (x \leftarrow f \leftarrow \overline{d} ; Q) c \mapsto \text{proc } ([d/\overline{y}, a/x]P) a, \text{proc } ([a/x]Q) c \quad \text{(a fresh)} \)

where \( x \leftarrow f \leftarrow \overline{y} = P \)

(tailC) \( \text{proc } (c \leftarrow f \leftarrow \overline{d}) c \mapsto \text{proc } ([d/\overline{y}, c/x]P) c \quad \text{where } x \leftarrow f \leftarrow \overline{y} = P \)

(⊕C) \( \text{proc } (c.k : P) c, \text{proc } (\text{case } c \{ \ell \Rightarrow Q_\ell \}_{\ell \in L}) d \mapsto \text{proc } P c, \text{proc } Q_k d \)

(&C) \( \text{proc } (\text{case } c \{ \ell \Rightarrow P_\ell \}_{\ell \in L}) c, \text{proc } (c.k : Q) d \mapsto \text{proc } P_k c, \text{proc } Q d \)

(1C) \( \text{proc } (\text{close } c) c, \text{proc } (\text{wait } c ; Q) d \mapsto \text{proc } Q d \)