Final Exam

15-814 Types and Programming Languages
Frank Pfenning

December 13, 2018

Name: Andrew ID:

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.

	Parametric Polymorphism	Data Abstraction	Exceptions	Quotation	Session Types	
	Prob 1	Prob 2	Prob 3	Prob 4	Prob 5	Total
Score						
Max	50	55	50	45	50	250

1 Parametric Polymorphism (50 pts)

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

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

We define the a family of types **only** τ by

$$\mathbf{only}\ \tau = \forall \gamma.\, (\tau \to \gamma) \to \gamma$$

Task 1 (10 pts). Define

in : $\forall \alpha.\, \alpha \to \mathbf{only}\ \alpha$

Task 2 (10 pts). Define

out : $\forall \alpha.\, \mathbf{only}\ \alpha \to \alpha$

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

Task 4 (10 pts). Evaluate *in* $(\mathit{out}\ w)$ for a closed value $w : \mathbf{only}\ \tau$

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 *out* and *in* that

$$(\mathit{out} \circ \mathit{in}) \sim (\lambda x.\, x) : \forall\alpha.\, \alpha \to \alpha$$

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

$$\mathit{out}\ (\mathit{in}\, v) \mapsto^* v$$

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

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 $0 = \mathit{false}$ and $n = \mathit{true}$ 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 $e \sim e' : \tau$ inductively on the structure of τ, assuming e and e' are closed and of type τ. 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, $\mathit{bool} = (\mathsf{false} : 1) + (\mathsf{true} : 1)$. Give a necessary and sufficient condition for

$$v \sim v' : \mathit{bool}$$

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

$v \sim v' : \mathit{bool}$ iff

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

$$v \sim v' : \mathit{nat}$$

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

$v \sim v' : \mathit{nat}$ iff

Now we consider the type

$$\mathsf{BOOL} = \exists \alpha.\, (\mathit{bool} \to \alpha) \otimes (\alpha \to \alpha) \otimes (\alpha \to \mathit{bool})$$

which represents a module with hidden implementation type τ for α and functions

to	:	$\mathit{bool} \to \tau$	map a boolean to its representation
neg	:	$\tau \to \tau$	negate the representation
from	:	$\tau \to \mathit{bool}$	map a representation back to a boolean

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

$\mathsf{Impl}_1 : \mathsf{BOOL}$
$\mathsf{Impl}_1 = \langle \mathit{bool}, \mathit{not}, \mathit{not}, \mathit{not} \rangle$

In the second implementation, booleans are represented with type *nat* where *zero* represents false and all non-zero numbers represent true.

$\mathsf{Impl}_2 : \mathsf{BOOL}$
$\mathsf{Impl}_2 = \langle \mathit{nat}, \mathit{rep}, \mathit{neg}, \mathit{unrep} \rangle$

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

$\mathsf{False} = \mathsf{false} \cdot \langle\,\rangle$
$\mathsf{True} = \mathsf{true} \cdot \langle\,\rangle$

$\mathsf{Z} = \mathbf{fold}\,(\mathsf{z} \cdot \langle\,\rangle)$
$\mathsf{S}\,x = \mathbf{fold}\,(\mathsf{s} \cdot x)$

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

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 $R : \textit{bool} \leftrightarrow \textit{nat}$ between the representations.

Task 5 (10 pts). Prove that $\textit{not} \sim \textit{rep} : \textit{bool} \rightarrow R$.

Task 6 (10 pts). Proof that *not* $\sim$ *neg* $: R \to R$.

It should also be true that *not* $\sim$ *unrep* $: R \to$ *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:

$$\text{Expressions} \quad e \quad ::= \quad \ldots \mid \mathbf{fail} \mid \mathbf{try}\ e\ \mathbf{catch}\ e'$$

The intended semantics is as follows.

- $\mathbf{try}\ e\ \mathbf{catch}\ e'$ 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'.
- **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:

$$\begin{array}{l}
\mathbf{try}\ v_1\ \mathbf{catch}\ v_2 \mapsto^* v_1 \\
\mathbf{try}\ \mathbf{fail}\ \mathbf{catch}\ v_2 \mapsto^* v_2 \\
\mathbf{try}\ (\mathbf{try}\ \mathbf{fail}\ \mathbf{catch}\ v_1)\ \mathbf{catch}\ v_2 \mapsto^* v_1 \\
\mathbf{try}\ \mathbf{fail}\ \mathbf{catch}\ \mathbf{fail} \mapsto^* \mathbf{fail} \\
(\mathbf{try}\ (\lambda x.\, \mathbf{fail})\ \mathbf{catch}\ v_2)\, v_1 \mapsto^* \mathbf{fail}
\end{array}$$

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

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

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
- $k \blacktriangleleft \textbf{fail}$: signal an exception to continuation k

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

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

Task 4 (10 pts). We write $s : \sigma$ if state s returns a final answer of type σ 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.

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

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

4 Quotation (45 points)

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

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

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

$\mathit{not} : \mathit{bool} \to \mathit{bool}$
$\mathit{not}\; \mathsf{False} = \mathsf{True}$
$\mathit{not}\; \mathsf{True} = \mathsf{False}$

$\mathit{and} : \mathit{bool} \to \mathit{bool} \to \mathit{bool}$
$\mathit{or} : \mathit{bool} \to \mathit{bool} \to \mathit{bool}$

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

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

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

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

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

$\mathit{equiv}'\; x = \mathbf{box}\; (\lambda y.\, \mathit{or}\; (\mathit{and}\; x\; y)\; (\mathit{and}\; (\mathit{not}\; x)\; (\mathit{not}\; y)))$

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

Task 3 (10 pts). Restage the definition of *equiv′* so it is correctly typed, using *and′* from Task 1 wherever possible.

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

$equiv'' : bool \to \Box(bool \to bool)$

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

(i) We can define a function $bool \to \Box\, bool$.

(ii) We can define a function $\forall\alpha.\, \alpha \to \Box\, \alpha$.

(iii) We can define a function $\forall\alpha.\, \Box\, \alpha \to \alpha$.

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

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

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

$$bin = \oplus\{\mathsf{b0} : bin, \mathsf{b1} : bin, \epsilon : 1\}$$

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

$\Vdash zero :: (z : bin)$
$z \leftarrow zero =$

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

$x : bin \Vdash succ :: (y : bin)$
$y \leftarrow succ \leftarrow x =$
  case x ($\mathsf{b0} \Rightarrow$

  $\mid \mathsf{b1} \Rightarrow$

  $\mid \epsilon \Rightarrow$

)

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

$pos = \oplus\{\mathsf{b0} : pos, \mathsf{b1} : bin\}$

$x : pos \Vdash pred :: (y : bin)$

$y \leftarrow pred \leftarrow x =$

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 ϵ. You may use *zero*, *succ*, and *pred* as needed, at the indicated types.

$x : \mathit{bin} \Vdash \mathit{numbits} :: (y : \mathit{bin})$

$y \leftarrow \mathit{numbits} \leftarrow x =$

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 $\mathit{numbits}(n) = \lfloor \mathit{log}_2(n) \rfloor + 1$ 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).

Appendix: Some Inference Rules

A Syntax

Types τ 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.

$$
\begin{array}{lcll}
\tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \tau_1 \otimes \tau_2 \mid 1 \mid \sum_{i\in I}(i : \tau_i) \mid \tau_1 \,\&\, \tau_2 \mid \rho(\alpha.\tau) & \\
e & ::= & x & \text{(variables)} \\
 & \mid & \lambda x.\, e \mid e_1\, e_2 & (\to) \\
 & \mid & i \cdot e \mid \textbf{case}\ e\ \{i \cdot x_i \Rightarrow e_i\}_{i\in I} & (+) \\
 & \mid & \langle e_1, e_2\rangle \mid \textbf{case}\ e_0\ \{\langle x_1, x_2\rangle \Rightarrow e'\} & (\otimes) \\
 & \mid & \langle\,\rangle \mid \textbf{case}\ e_0\ \{\langle\,\rangle \Rightarrow e'\} & (1) \\
 & \mid & \langle\!|e_1, e_2|\!\rangle \mid e \cdot l \mid e \cdot r & (\&) \\
 & \mid & \textbf{fold}(e) \mid \textbf{unfold}(e) & (\rho) \\
 & \mid & \textbf{fix}(x.e) & \text{(recursion)}
\end{array}
$$

B Statics, Expressions: $\Gamma \vdash e : \tau$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ (\textsc{Var}) \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'}\ (\text{I-}\to)$$

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\, e_2 : \tau'}\ (\text{E-}\to)$$

$$\frac{\Gamma \vdash e : \tau_j \quad (j \in I)}{\Gamma \vdash j \cdot e : \sum_{i \in I}(i : \tau_i)}\ (\text{I-+})$$

$$\frac{\Gamma \vdash e : \sum_{i \in I}(i : \tau_i) \quad \Gamma, x_i : \tau_i \vdash e_i : \tau \quad (\forall i \in I)}{\Gamma \vdash \textbf{case}\ e\ \{i \cdot x_i \Rightarrow e_i\}_{i \in I} : \tau}\ (\text{E-+})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \otimes \tau_2}\ (\text{I-}\otimes) \qquad \frac{\Gamma \vdash e_0 : \tau_1 \otimes \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e' : \tau}{\Gamma \vdash \textbf{case}\ e_0\ \{\langle x_1, x_2 \rangle \Rightarrow e'\} : \tau}\ (\text{E-}\otimes)$$

$$\frac{}{\Gamma \vdash \langle\,\rangle : 1}\ (\text{I-1}) \qquad \frac{\Gamma \vdash e_0 : 1 \quad \Gamma \vdash e' : \tau}{\Gamma \vdash \textbf{case}\ e_0\ \{\langle\,\rangle \Rightarrow e'\} : \tau}\ (\text{E-1})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle\!| e_1, e_2 |\!\rangle : \tau_1 \,\&\, \tau_2}\ (\text{I-}\&) \qquad \frac{\Gamma \vdash e : \tau_1 \,\&\, \tau_2}{\Gamma \vdash e \cdot l : \tau_1}\ (\text{E-}\&_l) \frac{\Gamma \vdash e : \tau_1 \,\&\, \tau_2}{\Gamma \vdash e \cdot r : \tau_2}\ (\text{E-}\&_r)$$

$$\frac{\Gamma \vdash e : [\rho(\alpha.\tau)/\alpha]\tau}{\Gamma \vdash \textbf{fold}(e) : \rho(\alpha.\tau)}\ (\text{I-}\rho) \qquad \frac{\Gamma \vdash e : \rho(\alpha.\tau)}{\Gamma \vdash \textbf{unfold}(e) : [\rho(\alpha.\tau)/\alpha]\tau}\ (\text{E-}\rho)$$

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \textbf{fix}(x.e) : \tau}\ (\textsc{Fix})$$

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

$$\frac{x : \tau \vdash e : \tau'}{\lambda x.e :: \tau \to \tau'}\ (\text{IV-}\to) \qquad \frac{v :: \tau_j \quad (j \in I)}{j \cdot v :: \sum_{i \in I}(i : \tau_i)}\ (\text{IV-+})$$

$$\frac{v_1 :: \tau_1 \quad v_2 :: \tau_2}{\langle v_1, v_2 \rangle :: \tau_1 \otimes \tau_2}\ (\text{IV-}\otimes) \qquad \frac{}{\langle\,\rangle :: 1}\ (\text{IV-1})$$

$$\frac{\cdot \vdash e_1 : \tau_1 \quad \cdot \vdash e_2 : \tau_2}{\langle\!| e_1, e_2 |\!\rangle :: \tau_1 \,\&\, \tau_2}\ (\text{IV-}\&) \qquad \frac{v :: [\rho(\alpha.\tau)/\alpha]\tau}{\textbf{fold}(v) :: \rho(\alpha.\tau)}\ (\text{IV-}\rho)$$

D Dynamics: $e \mapsto e'$ and v *val*

$$\frac{}{\lambda x.e \text{ val}} \text{ (V-}\to\text{)} \qquad \frac{v_2 \text{ val}}{(\lambda x.\, e_1)\, v_2 \mapsto [v_2/x]e_1} \text{ (R-}\to\text{)}$$

$$\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2} \text{ (CE-}\to_1\text{)} \qquad \frac{v_1 \text{ val} \quad e_2 \mapsto e_2'}{v_1\, e_2 \mapsto e_1\, e_2'} \text{ (CE-}\to_2\text{)}$$

$$\frac{v \text{ val}}{i \cdot v \text{ val}} \text{ (V-+)} \quad \frac{e \mapsto e'}{i \cdot e \mapsto i \cdot e'} \text{ (CI-+)} \quad \frac{e \mapsto e'}{\textbf{case } e\ \{i \cdot x_i \Rightarrow e_i\}_{i \in I} \mapsto \textbf{case } e'\ \{i \cdot x_i \Rightarrow e_i\}_{i \in I}} \text{ (CE-+)}$$

$$\frac{v_j \text{ val}}{\textbf{case } (j \cdot v_j)\ \{i \cdot x_i \Rightarrow e_i\}_{i \in I} \mapsto [v_j/x_j]e_j} \text{ (R-+)}$$

$$\frac{v_1 \text{ val} \quad v_2 \text{ val}}{\langle v_1, v_2 \rangle \text{ val}} \text{ (V-}\otimes\text{)} \qquad \frac{e_1 \mapsto e_1'}{\langle e_1, e_2 \rangle \mapsto \langle e_1', e_2 \rangle} \text{ (CI-}\otimes_1\text{)} \qquad \frac{v_1 \text{ val} \quad e_2 \mapsto e_2'}{\langle v_1, e_2 \rangle \mapsto \langle v_1, e_2' \rangle} \text{ (CI-}\otimes_2\text{)}$$

$$\frac{e_0 \mapsto e_0'}{\textbf{case } e_0\ \{\langle x_1, x_2 \rangle \Rightarrow e'\} \mapsto \textbf{case } e_0'\ \{\langle x_1, x_2 \rangle \Rightarrow e'\}} \text{ (CE-}\otimes\text{)}$$

$$\frac{v_1 \text{ val} \quad v_2 \text{ val}}{\textbf{case } \langle v_1, v_2 \rangle\ \{\langle x_1, x_2 \rangle \Rightarrow e'\} \mapsto [v_1/x_1, v_2/x_2]e'} \text{ (R-}\otimes\text{)}$$

$$\frac{}{\langle\, \rangle \text{ val}} \text{ (V-1)} \qquad \frac{e_0 \mapsto e_0'}{\textbf{case } e_0\ \{\langle\, \rangle \Rightarrow e'\} \mapsto \textbf{case } e_0'\ \{\langle\, \rangle \Rightarrow e'\}} \text{ (CE-1)} \qquad \frac{}{\textbf{case } \langle\, \rangle\ \{\langle\, \rangle \Rightarrow e'\} \mapsto e'} \text{ (R-1)}$$

$$\frac{}{\langle\!|e_1, e_2|\!\rangle \text{ val}} \text{ (V-\&)} \qquad \frac{e \mapsto e'}{e \cdot l \mapsto e' \cdot l} \text{ (CI-\&}_l\text{)} \qquad \frac{e \mapsto e'}{e \cdot r \mapsto e' \cdot r} \text{ (CI-\&)}_r$$

$$\frac{}{\langle\!|e_1, e_2|\!\rangle \cdot l \mapsto e_1} \text{ (R-\&}_l\text{)} \qquad \frac{}{\langle\!|e_1, e_2|\!\rangle \cdot r \mapsto e_2} \text{ (R-\&}_r\text{)}$$

$$\frac{v \text{ val}}{\textbf{fold}(v) \text{ val}} \text{ (V-}\rho\text{)} \qquad \frac{e \mapsto e'}{\textbf{fold}(e) \mapsto \textbf{fold}(e')} \text{ (CI-}\rho\text{)}$$

$$\frac{e \mapsto e'}{\textbf{unfold}(e) \mapsto \textbf{unfold}(e')} \text{ (CE-}\rho\text{)} \qquad \frac{v \text{ val}}{\textbf{unfold}(\textbf{fold}(v)) \mapsto v} \text{ (R-}\rho\text{)}$$

$$\frac{}{\textbf{fix}(x.e) \mapsto [\textbf{fix}(x.e)/x]e} \text{ (R-F}\textsc{ix}\text{)}$$

Session Types

Process expressions: forward, spawn, and tail-call

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

Session types and process expressions: message passing

Type	**Provider**	**Client**	**Continuation Type**
$c : \oplus\{\ell : A_\ell\}_{\ell \in L}$	$(c.k \; ; P)$	$\mathbf{case}\; c \; \{\ell \Rightarrow Q_\ell\}_{\ell \in L}$	$c : A_k$
$c : \&\{\ell : A_\ell\}_{\ell \in L}$	$\mathbf{case}\; c \; \{\ell \Rightarrow P_\ell\}_{\ell \in L}$	$(c.k \; ; Q)$	$c : A_k$
$c : 1$	$\mathbf{close}\; c$	$\mathbf{wait}\; c \; ; Q$	*(none)*

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

$$\frac{}{y : A \Vdash (x \leftarrow y) :: (x : A)} \;\mathsf{id}$$

$$\frac{\Delta_1 \Vdash f :: (x : A) \quad \Delta_2, x : A \Vdash Q :: (z : C)}{\Delta_1, \Delta_2 \Vdash (x \leftarrow f \leftarrow |\Delta_1| \; ; Q) :: (z : C)} \;\mathsf{spawn} \qquad \frac{\Delta \Vdash f :: (x : A)}{\Delta \Vdash (x \leftarrow f \leftarrow |\Delta|) :: (x : A)} \;\mathsf{tail}$$

$$\frac{k \in L \quad \Delta \Vdash P :: (x : A_k)}{\Delta \Vdash (x.k \; ; P) :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})} \;\oplus R \qquad \frac{(\text{for all } \ell \in L) \quad \Delta, x : A_\ell \Vdash Q_\ell :: (z : C)}{\Delta, x : \oplus\{\ell : A_\ell\}_{\ell \in L} \Vdash (\mathbf{case}\; x \; \{\ell \Rightarrow Q_\ell\}_{\ell \in L}) :: (z : C)} \;\oplus L$$

$$\frac{(\text{for all } \ell \in L) \quad \Delta \Vdash P_\ell :: (x : A_\ell)}{\Delta \Vdash (\mathbf{case}\; x \; \{\ell \Rightarrow P_\ell\}_{\ell \in L}) :: (x : \&\{\ell : A_\ell\})} \;\& R \qquad \frac{k \in L \quad \Delta, x : A_k \Vdash Q :: (z : C)}{\Delta, x : \&\{\ell : A_\ell\}_{\ell \in L} \Vdash (x.k \; ; Q) :: (z : C)} \;\& L$$

$$\frac{}{\cdot \Vdash \mathbf{close}\, x :: (x : 1)} \;1R \qquad \frac{\Delta \Vdash Q :: (z : C)}{\Delta, x : 1 \Vdash (\mathbf{wait}\; x \; ; Q) :: (z : C)} \;1L$$

Dynamics

$(\mathsf{id}C)$	$\mathsf{proc}\; P\; d, \mathsf{proc}\; (c \leftarrow d)\; c \;\mapsto\; \mathsf{proc}\; ([c/d]P)\; c$
$(\mathsf{spawn}C)$	$\mathsf{proc}\; (x \leftarrow f \leftarrow \overline{d} \; ; Q)\; c \;\mapsto\; \mathsf{proc}\; ([\overline{d}/\overline{y}, a/x]P)\; a, \mathsf{proc}\; ([a/x]Q)\; c$ $\quad$ (a fresh) where $x \leftarrow f \leftarrow \overline{y} = P$
$(\mathsf{tail}C)$	$\mathsf{proc}\; (c \leftarrow f \leftarrow \overline{d})\; c \;\mapsto\; \mathsf{proc}\; ([\overline{d}/\overline{y}, c/x]P)\; c$ $\quad$ where $x \leftarrow f \leftarrow \overline{y} = P$
$(\oplus C)$	$\mathsf{proc}\; (c.k \; ; P)\; c, \mathsf{proc}\; (\mathbf{case}\; c \; \{\ell \Rightarrow Q_\ell\}_{\ell \in L})\; d \;\mapsto\; \mathsf{proc}\; P\; c, \mathsf{proc}\; Q_k\; d$
$(\& C)$	$\mathsf{proc}\; (\mathbf{case}\; c \; \{\ell \Rightarrow P_\ell\}_{\ell \in L})\; c, \mathsf{proc}\; (c.k \; ; Q)\; d \;\mapsto\; \mathsf{proc}\; P_k\; c, \mathsf{proc}\; Q\; d$
$(1C)$	$\mathsf{proc}\; (\mathbf{close}\, c)\; c, \mathsf{proc}\; (\mathbf{wait}\; c \; ; Q)\; d \;\mapsto\; \mathsf{proc}\; Q\; d$