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.

<table>
<thead>
<tr>
<th>Prob 1</th>
<th>Prob 2</th>
<th>Prob 3</th>
<th>Prob 4</th>
<th>Prob 5</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max</td>
<td>50</td>
<td>55</td>
<td>50</td>
<td>45</td>
<td>50</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:

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

We define the a family of types \(\text{only } \tau\) by

\(\text{only } \tau = \forall \gamma. (\tau \to \gamma) \to \gamma\)

**Task 1** (10 pts). Define

\(\text{in} : \forall \alpha. \alpha \to \text{only } \alpha\)

**Task 2** (10 pts). Define

\(\text{out} : \forall \alpha. \text{only } \alpha \to \alpha\)

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

\[
\text{out} (\text{in } v) \mapsto^* v
\]

(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 \(0 = \text{false}\) and \(n = \text{true}\) 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 \(\tau\), 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:

\[
(\to) \quad e \sim e' : \tau_1 \to \tau_2 \text{ iff for all } v_1 \sim v'_1 : \tau_1 \text{ we have } e v_1 \sim e' v'_1 : \tau_2
\]

\[
(+) \quad v \sim v' : \tau_1 + \tau_2 \text{ iff either } v = l \cdot v_1, v' = l \cdot v'_1, \text{ and } v_1 \sim v'_1 : \tau_1 \text{ or } v = r \cdot v_2, v' = r \cdot v'_2, \text{ and } v_2 \sim v'_2 : \tau_2
\]

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

\[
v \sim v' : \text{bool}
\]

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

\[
v \sim v' : \text{bool} \text{ iff } ...
\]

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

\[
v \sim v' : \text{nat}
\]

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

\[
v \sim v' : \text{nat} \text{ iff } ...
\]
Now we consider the type

\[ \text{BOOL} = \exists \alpha. (\text{bool} \to \alpha) \otimes (\alpha \to \alpha) \otimes (\alpha \to \text{bool}) \]

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

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

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

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

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

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

\textbf{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 \\
\text{Z} &= \text{fold} (z \cdot \langle \rangle) \\
\text{S} x &= \text{fold} (s \cdot x)
\end{align*}

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

**Task 5** (10 pts). Prove that $\text{not} \sim \text{rep} : \text{bool} \rightarrow R$. 
Task 6 (10 pts). Proof that $\neg \sim \text{neg} : R \to R$.

It should also be true that $\neg \sim \text{unrep} : R \to \text{bool}$ 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:

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

The intended semantics is as follows.

- **try e 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:

- $\text{try } v_1 \text{ catch } v_2 \mapsto^* v_1$
- $\text{try fail catch } v_2 \mapsto^* v_2$
- $\text{try (try fail catch } v_1) \text{ catch } v_2 \mapsto^* v_1$
- $\text{try fail catch fail} \mapsto^* \text{fail}$
- $(\text{try } (\lambda x. \text{fail} \text{ catch } v_2) \text{ v_1} \mapsto^* \text{fail}$

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 $k$
- $k \triangleleft \text{fail}$: 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 $\tau$ to a final answer of type $\sigma$. 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 $\sigma$ 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.

\[
\frac{k \downarrow \tau \Rightarrow \sigma \vdash e : \tau}{k \triangleright e : \sigma}
\]

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 $\Psi$ 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:

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

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:

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

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

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

\[\text{and}' : \text{bool} \rightarrow \Box (\text{bool} \rightarrow \text{bool})\]

**Task 2** (5 pts). The proposed staged definition for equivalence of booleans, $\text{equiv}' : \text{bool} \rightarrow \Box (\text{bool} \rightarrow \text{bool})$

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

is not well-typed. Explain where and why typing fails.
**Task 3** (10 pts). Restage the definition of $\text{equiv}'$ so it is correctly typed, using $\text{and}'$ from Task 1 wherever possible.

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

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

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

(i) We can define a function $\text{bool} \to \Box \text{bool}$.

(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

\[ bin = \oplus\{b0 : bin, b1 : bin, \epsilon : 1\} \]

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.

\[ x : bin \vdash succ :: (y : bin) \]
\[ y \leftarrow succ \leftarrow x = \]
\[ \text{case } x (b0 \Rightarrow \]  
\[ | b1 \Rightarrow \]
\[ | \epsilon \Rightarrow \]
\[ ) \]

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

\[ pos = \oplus\{b0 : pos, b1 : bin\} \]

\[ x : pos \vdash pred :: (y : bin) \]
\[ y \leftarrow pred \leftarrow x = \]
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 $\epsilon$. You may use zero, succ, and pred as needed, at the indicated types.

$x : \text{bin} \vdash \text{nubits} :: (y : \text{bin})$

$y \leftarrow \text{nubits} \leftarrow x =$

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 $\text{nubits}(n) = \lfloor \log_2(n) \rfloor + 1$ 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).
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.

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

$$
e ::= x \quad \text{(variables)}
| \lambda x. e \quad \text{($\to$)}
| i \cdot e \quad \text{($\otimes$)}
| \langle e_1, e_2 \rangle \quad \text{($\&$)}
| \text{case } e_0 \{ (x_1, x_2) \Rightarrow e' \}
| \text{fold}(e) \quad \text{(recursion)}
| \text{fix}(x.e) \quad \text{(recursion)}
$$
B Statics, Expressions: $\Gamma \vdash e : \tau$

\[
\begin{align*}
\Gamma & \vdash x : \tau \quad \text{(VAR)} \\
\Gamma & \vdash \lambda x.e : \tau \rightarrow \tau' \quad \text{(I-\rightarrow)} \\
\Gamma & \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \tau \\
\Gamma & \vdash e_1 \cdot e_2 : \tau' \quad \text{(E-\rightarrow)} \\
\Gamma & \vdash e : \tau_j \quad (j \in I) \\
\Gamma & \vdash \sum_{i \in I}(i : \tau_i) \quad \text{(I+)} \\
\Gamma & \vdash \sum_{i \in I}(i : \tau_i) \quad \Gamma, x_i : \tau_i \vdash e_i : \tau \quad (\forall i \in I) \\
\Gamma & \vdash \text{case } e \{ i \cdot x_i \Rightarrow e_i \}_{i \in I} : \tau \quad \text{(E+)} \\
\Gamma & \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \\
\Gamma & \vdash \langle e_1, e_2 \rangle : \tau_1 \otimes \tau_2 \quad \text{(I-\otimes)} \\
\Gamma & \vdash \text{case } e_0 \{ \langle \rangle \Rightarrow e' \} : \tau \quad \text{(E-\otimes)} \\
\Gamma & \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \\
\Gamma & \vdash \langle e_1, e_2 \rangle : \tau_1 \& \tau_2 \quad \text{(I-\&)} \\
\Gamma & \vdash \text{case } e_0 \{ \langle \rangle \Rightarrow e' \} : \tau \quad \text{(E-\&)} \\
\Gamma & \vdash e : \rho(\alpha.\tau) \\
\Gamma & \vdash \text{fold}(e) : \rho(\alpha.\tau) \quad \text{(I-\rho)} \\
\Gamma, x : \tau & \vdash e : \tau \\
\Gamma & \vdash \text{fix}(x.e) : \tau \quad \text{(FIX)} \\
\end{align*}
\]

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

\[
\begin{align*}
\Gamma & \vdash x : \tau \vdash e : \tau' \quad \text{(IV-\rightarrow)} \\
\Gamma & \vdash \langle \rangle : 1 \quad \text{(IV-1)} \\
\Gamma & \vdash \lambda x.e :: \tau \rightarrow \tau' \quad \text{(IV-\rightarrow)} \\
\Gamma & \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \\
\Gamma & \vdash \langle e_1, e_2 \rangle : \tau_1 \& \tau_2 \quad \text{(IV-\&)} \\
\Gamma & \vdash v : [\rho(\alpha.\tau)/\alpha]_\tau \\
\Gamma & \vdash \text{fold}(v) : \rho(\alpha.\tau) \quad \text{(IV-\rho)} \\
\end{align*}
\]

16
D Dynamics: $e \mapsto e'$ and $v \ val$

\[
\begin{align*}
\frac{\lambda x.e \ val}{(V\rightarrow)} & \quad \frac{v_2 \ val}{(R\rightarrow)} \\
\frac{e_1 \mapsto e'_1}{(CE\rightarrow_1)} & \quad \frac{v_1 \ val \ e \mapsto e'_2}{(CE\rightarrow_2)} \\
\frac{i \cdot e \mapsto i \cdot e'}{(CI\rightarrow_1)} & \quad \frac{e \mapsto e'}{(CI\rightarrow_2)} \\
\frac{\text{case } e \{i \cdot x_i \Rightarrow e_i\}_{i \in I} \mapsto \text{case } e' \{i \cdot x_i \Rightarrow e_i\}_{i \in I}}{(CE\rightarrow_+)}
\end{align*}
\]

\[
\begin{align*}
\frac{v_1 \ val \ v_2 \ val}{(V\otimes)} & \quad \frac{e_1 \mapsto e'_1}{(CI\otimes_1)} & \quad \frac{v_1 \ val \ e_2 \mapsto e'_2}{(CI\otimes_2)} \\
\frac{\text{case } \langle v_1, v_2 \rangle \{x_1, x_2 \Rightarrow e'\} \mapsto \text{case } \langle v_1, v_2 \rangle \{x_1, x_2 \Rightarrow e'\}}{(CE\otimes)}
\end{align*}
\]

\[
\begin{align*}
\frac{e_0 \mapsto e'_0}{(R+)} & \\
\frac{\text{case } e_0 \{\langle x_1, x_2 \rangle \Rightarrow e'\} \mapsto \text{case } e'_0 \{\langle x_1, x_2 \rangle \Rightarrow e'\}}{(R-)}
\end{align*}
\]

\[
\begin{align*}
\frac{e_0 \mapsto e'_0}{(V\otimes)} & \quad \frac{e \mapsto e'}{(CI\&)} \\
\frac{e \cdot l \mapsto e' \cdot l}{(CI\&_l)} & \quad \frac{e \cdot r \mapsto e' \cdot r}{(CI\&_r)} \\
\frac{\langle e_1, e_2 \rangle \cdot l \mapsto e_1}{(R\& \_l)} & \quad \frac{\langle e_1, e_2 \rangle \cdot r \mapsto e_2}{(R\& \_r)}
\end{align*}
\]

\[
\begin{align*}
\frac{v \ val}{(V\rho)} & \quad \frac{\text{fold}(v) \ val}{(CI\rho)} & \quad \frac{e \mapsto e'}{(CI\rho)} \\
\frac{\text{fold}(e) \mapsto \text{fold}(e')}{(CI\rho)} & \quad \frac{\text{fold}(e) \mapsto \text{fold}(e')}{(CI\rho)} \\
\frac{\text{unfold}(e) \mapsto \text{unfold}(e')}{(CE\rho)} & \quad \frac{v \ val}{(R\rho)} \\
\frac{\text{fix}(x.e) \mapsto [\text{fix}(x.e)/x]e}{(R-FIX)}
\end{align*}
\]
Session Types

Process expressions: forward, spawn, and tail-call

\[ c \leftarrow d \]
implies \( c \) 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

<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} ) &amp; (c.k ; P)</td>
<td>case c {( \ell \Rightarrow Q_\ell )}_{\ell \in L}</td>
<td>c : A_k</td>
<td></td>
</tr>
<tr>
<td>( c : &amp; { \ell : A_\ell }<em>{\ell \in L} ) case c {( \ell \Rightarrow P</em>\ell )}_{\ell \in L}</td>
<td>(c.k ; Q)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( c : 1 ) close c</td>
<td>wait c ; Q</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

\[ y : A \vdash (x \leftarrow y) :: (x : A) \quad \text{id} \]

\[ \Delta_1 \vdash f :: (x : A) \quad \Delta_2, x : A \vdash Q :: (z : C) \quad \text{spawn} \]

\[ \Delta_1, \Delta_2 \vdash (x \leftarrow f ; \Delta_1 ; Q) :: (z : C) \quad \text{tail} \]

\[ k \in L \quad \Delta \vdash P :: (x : A_k) \]

\[ \Delta \vdash (x.k ; P) :: (x : \oplus \{ \ell : A_\ell \}_{\ell \in L}) \quad \oplus R \]

\[ (\text{for all } \ell \in L) \quad \Delta \vdash P_\ell :: (x : A_\ell) \quad \oplus L \]

\[ \Delta \vdash \text{case } x \{ \ell \Rightarrow P_\ell \}_{\ell \in L} :: (x : \& \{ \ell : A_\ell \}) \quad \& R \]

\[ (\text{for all } \ell \in L) \quad \Delta \vdash x : A_\ell \vdash Q :: (z : C) \quad \& L \]

\[ \Delta \vdash Q :: (z : C) \quad \Delta, x : 1 \vdash (\text{wait } x ; Q) :: (z : C) \quad 1L \]

Dynamics

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

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

\[ \text{where } x \leftarrow f \leftarrow \overline{y} = P \]

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

\[ \text{where } x \leftarrow f \leftarrow \overline{y} = P \]

\[ \text{(⊕C)} \quad \text{proc } (c.k ; P) \ c, \text{proc } (\text{case } c \{ \ell \Rightarrow Q_\ell \}_{\ell \in L}) \ d \rightarrow \text{proc } P \ c, \text{proc } Q_k \ d \]

\[ \text{(＆C)} \quad \text{proc } (\text{case } c \{ \ell \Rightarrow P_\ell \}_{\ell \in L}) \ c, \text{proc } (c.k ; Q) \ d \rightarrow \text{proc } P_k \ c, \text{proc } Q \ d \]

\[ \text{(1C)} \quad \text{proc } (\text{close } c) \ c, \text{proc } (\text{wait } c ; Q) \ d \rightarrow \text{proc } Q \ d \]