Act 1: Binary Session Types

With a deeply embedded binder representation.

Locally Nameless

with the Simply Typed Lambda-Calculus.

Terms $M, N ::= n \mid x \mid \lambda\,.\,M \mid M\,N$

Types $S, T ::= \texttt{base} \mid S \to T$

Locally Nameless

with the Simply Typed Lambda-Calculus.

Bound variables are represented by their De Bruijn index (i.e: a natural number).

$$\text{Terms } M, N ::= n \mid x \mid \lambda\, .\, M \mid M\, N$$

$$\text{Types } S, T ::= \texttt{base} \mid S \to T$$

Locally Nameless

with the Simply Typed Lambda-Calculus.

Bound variables are represented by their De Bruijn index (i.e: a natural number).

Free variables are represented by a name (i.e: an element of a nominal set).

$$\text{Terms } M, N ::= n \mid x \mid \lambda\, .\, M \mid M\, N$$

$$\text{Types } S, T ::= \texttt{base} \mid S \to T$$

Locally Nameless

with the Simply Typed Lambda-Calculus.

Bound variables are represented by their De Bruijn index (i.e: a natural number).

Free variables are represented by a name (i.e: an element of a nominal set).

Binders are anonymous (as with De Bruijn indices in general).

$$\text{Terms } M, N ::= n \mid x \mid \lambda \,.\, M \mid M\, N$$

$$\text{Types } S, T ::= \texttt{base} \mid S \to T$$

Locally Nameless

with the Simply Typed Lambda-Calculus.

Terms $M, N ::= n \mid x \mid \lambda \,.\, M \mid M\ N$

Types $S, T ::= \texttt{base} \mid S \to T$

$$M^x \equiv \{0 \to x\}M$$ Open a term.

$$\backslash^x M \equiv \{0 \leftarrow x\}M$$ Close a term.

$$\texttt{lc}(M)$$ A locally closed term.

Locally Nameless

with the Simply Typed Lambda-Calculus.

Terms $M, N ::= n \mid x \mid \lambda\,.\,M \mid M\;N$

Types $S, T ::= \texttt{base} \mid S \to T$

$$M^x \equiv \{0 \to x\}M$$

Open a term.

$$\backslash^x M \equiv \{0 \leftarrow x\}M$$

Close a term.

$$\texttt{lc}(M)$$

A locally closed term.

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}$$

$$\frac{\forall x \notin L \qquad \Gamma, x : S \vdash M^x : T}{\Gamma \vdash \lambda\,.\,M : S \to T}$$

$$\frac{\Gamma \vdash M : S \to T \qquad \Gamma \vdash N : S}{\Gamma \vdash M\;N : T}$$

Locally Nameless

with the Simply Typed Lambda-Calculus.

Terms $M, N ::= n \mid x \mid \lambda \,.\, M \mid M\ N$

Types $S, T ::= \texttt{base} \mid S \to T$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \frac{\forall x \notin L \qquad \Gamma, x : S \vdash M^x : T}{\Gamma \vdash \lambda \,.\, M : S \to T}$$

$$\frac{\Gamma \vdash M : S \to T \qquad \Gamma \vdash N : S}{\Gamma \vdash M\ N : T}$$

$M^x \equiv \{0 \to x\}M$ Open a term.

$\backslash^x M \equiv \{0 \leftarrow x\}M$ Close a term.

$\texttt{lc}(M)$ A locally closed term.

Locally Nameless

with the Simply Typed Lambda-Calculus.

Terms $M, N ::= n \mid x \mid \lambda\,.\,M \mid M\;N$

Types $S, T ::= \texttt{base} \mid S \to T$

$M^x \equiv \{0 \to x\}M$ Open a term.

$\backslash^x M \equiv \{0 \leftarrow x\}M$ Close a term.

$\texttt{lc}(M)$ A locally closed term.

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}$$

$$\frac{\forall x \notin L \qquad \Gamma, x : S \vdash M^x : T}{\Gamma \vdash \lambda\,.\,M : S \to T}$$

$$\frac{\Gamma \vdash M : S \to T \qquad \Gamma \vdash N : S}{\Gamma \vdash M\;N : T}$$

Locally Nameless

with the Simply Typed Lambda-Calculus.

Terms $M, N ::= n \mid x \mid \lambda\,.\,M \mid M\;N$

Types $S, T ::= \texttt{base} \mid S \to T$

$M^x \equiv \{0 \to x\}M$ Open a term.

$\backslash^x M \equiv \{0 \leftarrow x\}M$ Close a term.

$\texttt{lc}(M)$ A locally closed term.

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}$$

$$\frac{\forall x \notin L \qquad \Gamma, x : S \vdash M^x : T}{\Gamma \vdash \lambda\,.\,M : S \to T}$$

$$\frac{\Gamma \vdash M : S \to T \qquad \Gamma \vdash N : S}{\Gamma \vdash M\;N : T}$$

smolEMTST

A simple calculus with binary session types.

$$
\begin{array}{rrcl}
\text{Expressions} & e & ::= & \texttt{tt} \mid \texttt{ff} \mid () \mid x \\
\text{Sorts} & S & ::= & \mathit{bool} \mid \mathit{unit} \\
\text{Processes} & P, Q & ::= & k![e].P \mid k?().P \mid P \mid Q \\
& & & \mid \texttt{if}\ e\ \texttt{else}\ P\ \texttt{else}\ Q \\
& & & \mid \nu.P \mid !P \mid \texttt{inact} \\
\text{Types} & T & ::= & ?[S].T \mid ![S].T \mid \texttt{end} \mid \bot
\end{array}
$$

smolEMTST

A simple calculus with binary session types.

$$
\begin{aligned}
\text{Expressions} \quad & e ::= \texttt{tt} \mid \texttt{ff} \mid () \mid x \\
\text{Sorts} \quad & S ::= \mathit{bool} \mid \mathit{unit} \\
\text{Processes } P, Q \; & ::= k![e].P \mid k?().P \mid P \mid Q \\
& \mid \texttt{if}\ e\ \texttt{else}\ P\ \texttt{else}\ Q \\
& \mid \nu.P \mid !\,P \mid \texttt{inact} \\
\text{Types} \quad & T ::= ?[S].T \mid ![S].T \mid \texttt{end} \mid \bot
\end{aligned}
$$

smolEMTST

A simple calculus with binary session types.

$$
\begin{array}{rrcl}
\text{Expressions} & e & ::= & \texttt{tt} \mid \texttt{ff} \mid () \mid x \\
\text{Sorts} & S & ::= & \mathit{bool} \mid \mathit{unit} \\
\text{Processes} & P, Q & ::= & k![e].P \mid k?().P \mid P \mid Q \\
 & & & \mid \texttt{if}\ e\ \texttt{else}\ P\ \texttt{else}\ Q \\
 & & & \mid \nu.P \mid !\,P \mid \texttt{inact} \\
\text{Types} & T & ::= & ?[S].T \mid ![S].T \mid \texttt{end} \mid \bot
\end{array}
$$

smolEMTST

A simple calculus with binary session types.

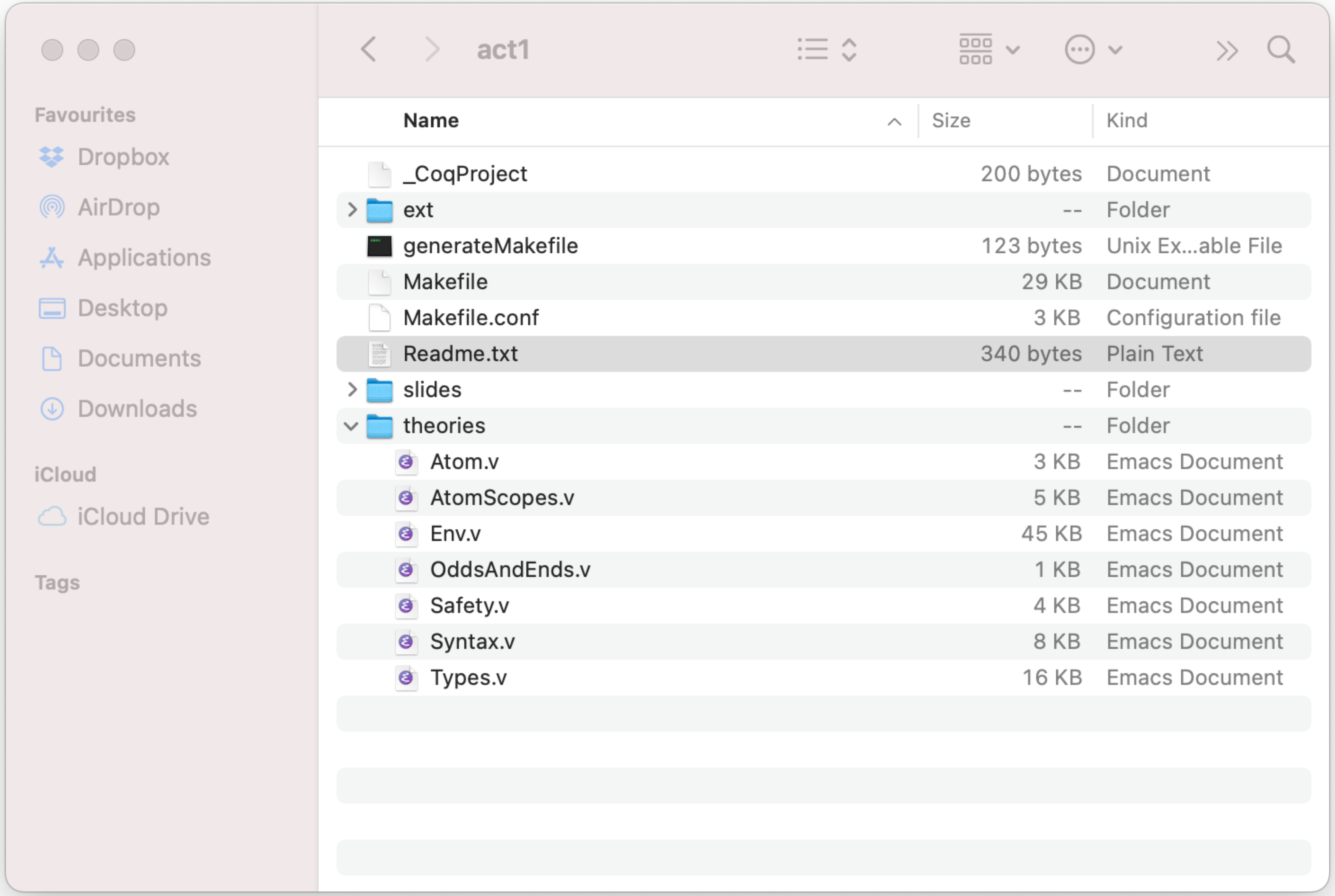
$$
\begin{array}{rrl}
\text{Expressions} & e & ::= \texttt{tt} \mid \texttt{ff} \mid () \mid x \\
\text{Sorts} & S & ::= \mathit{bool} \mid \mathit{unit} \\
\text{Processes} & P, Q & ::= k![e].P \mid k?().P \mid P \mid Q \\
 & & \quad \mid \texttt{if}\ e\ \texttt{else}\ P\ \texttt{else}\ Q \\
 & & \quad \mid \nu.P \mid\ !\,P \mid \texttt{inact} \\
\text{Types} & T & ::= ?[S].T \mid ![S].T \mid \texttt{end} \mid \bot
\end{array}
$$

expression variables

channel variables

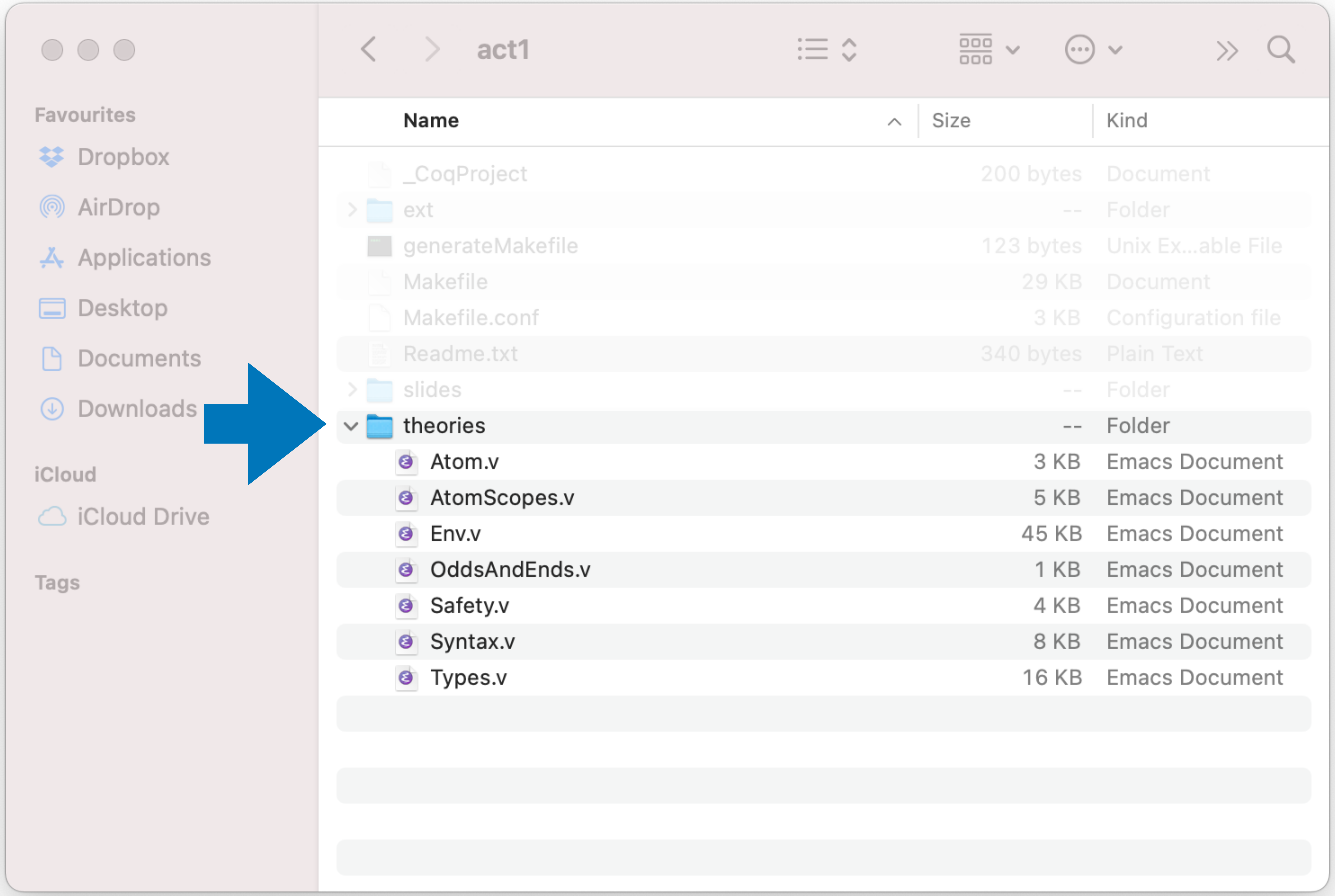
smolEMTST

At: http://github.com/emtst/gentleAdventure



smolEMTST

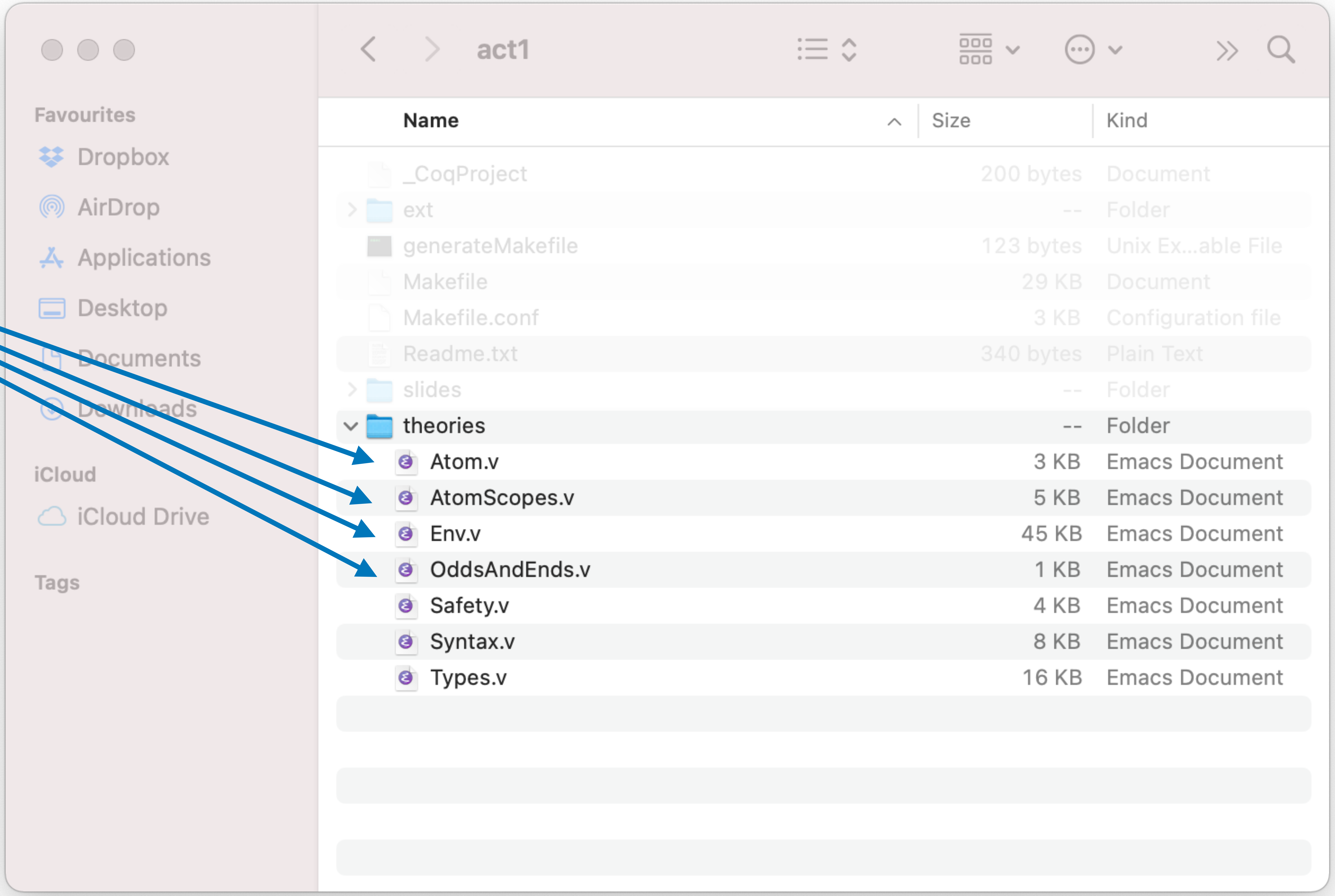
At: http://github.com/emtst/gentleAdventure



smolEMTST

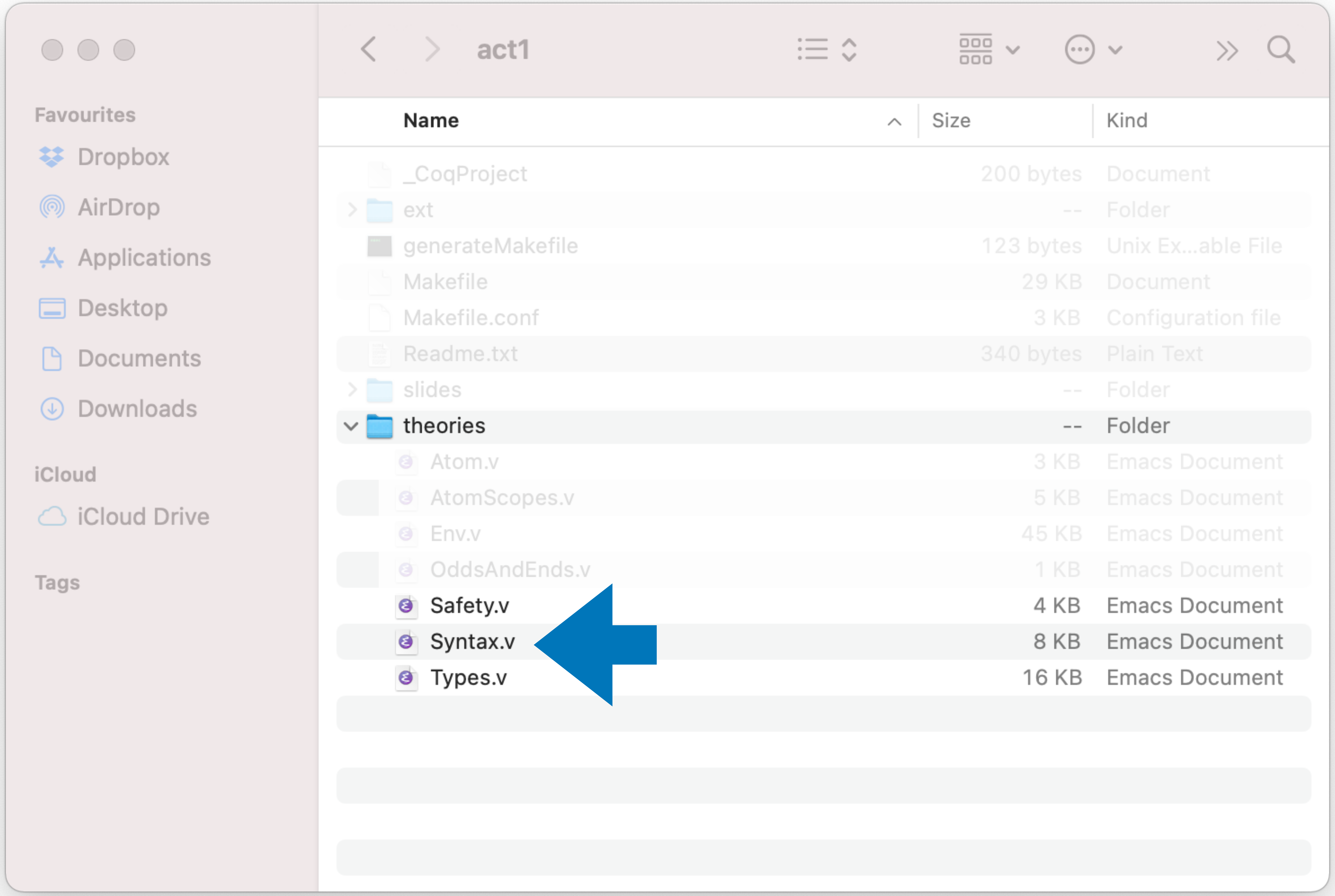
At: http://github.com/emtst/gentleAdventure

EMTST the tool:
- **multiple name scopes**
- **environment handling**



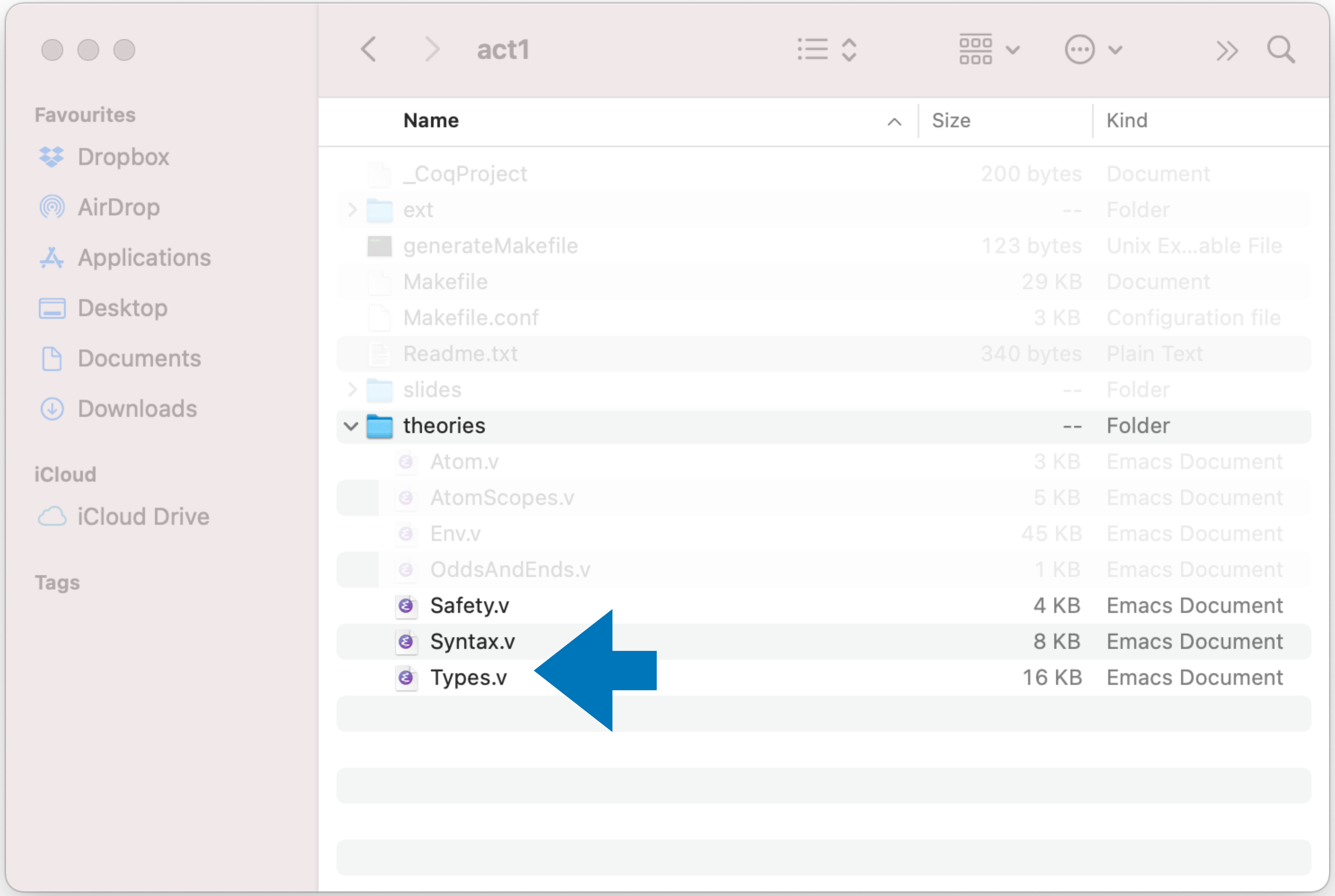
smolEMTST

At: http://github.com/emtst/gentleAdventure



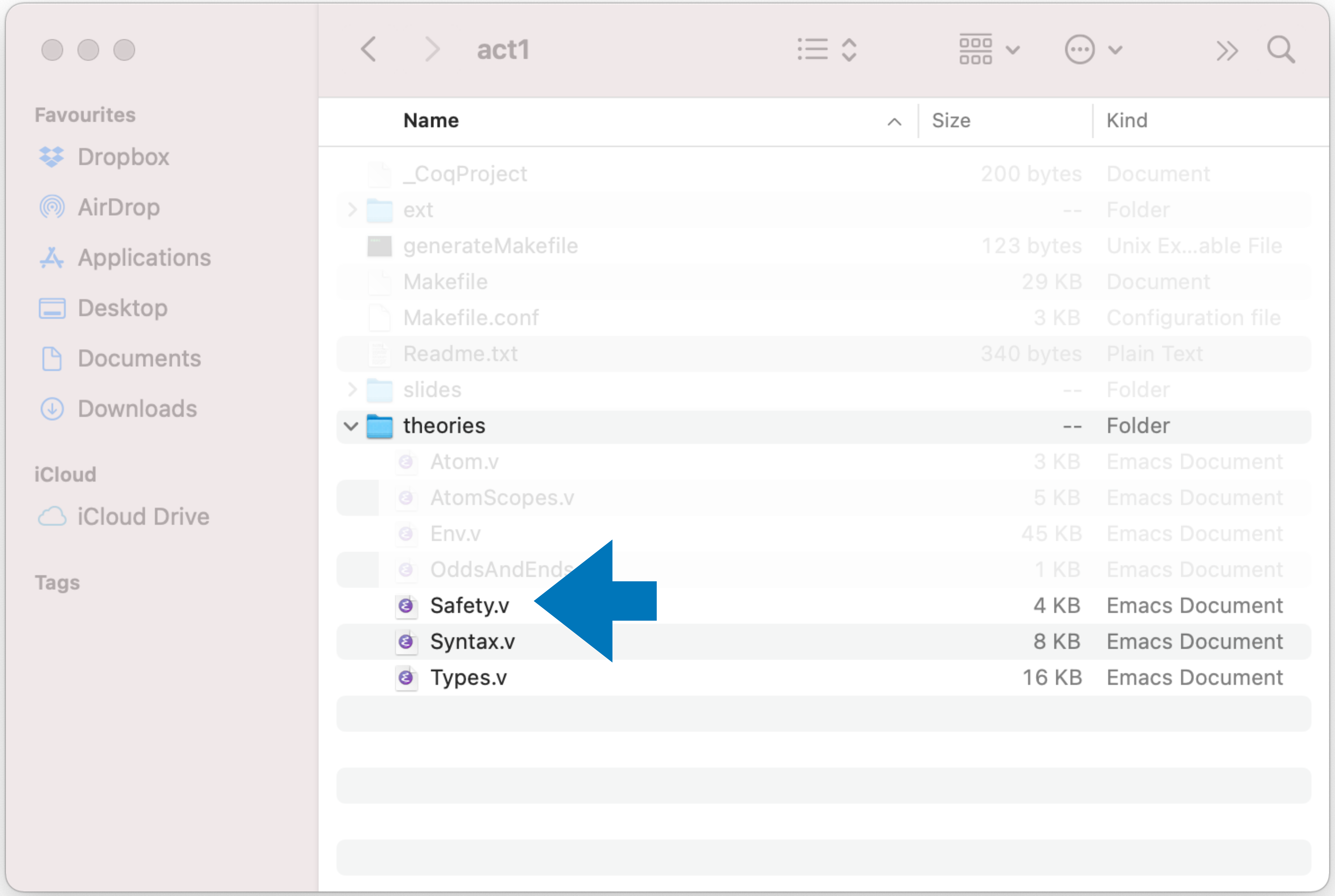
smolEMTST

At: http://github.com/emtst/gentleAdventure



smolEMTST

At: http://github.com/emtst/gentleAdventure



Representing the Syntax.

Locally closed brings the binding structure.

Expressions $e ::= \mathtt{tt} \mid \mathtt{ff} \mid () \mid x$

```
Inductive exp : Set :=
  | tt
  | ff
  | one
  | V of evar
.

Coercion V : evar >-> exp.
```

```
Inductive lc_exp : exp → Prop :=
  | lc_tt : lc_exp tt
  | lc_ff : lc_exp ff
  | lc_one : lc_exp one
  | lc_var a: lc_exp (V(EV.Free a))
.
```

```
(* Open a bound variable in an expression (original ope) *)
Definition open_exp (n : nat) (e' : exp) (e : exp) : exp :=
  match e with
  | V v ⇒ EV.open_var V n e' v
  | _ ⇒ e
  end.
```

Representing the Syntax.

Locally closed brings the binding structure.

Expressions $e ::= \mathtt{tt} \mid \mathtt{ff} \mid () \mid x$

```
Inductive exp : Set :=
  | tt
  | ff
  | one
  | V of evar
.

Coercion V : evar >-> exp.
```

```
Inductive lc_exp : exp -> Prop :=
  | lc_tt : lc_exp tt
  | lc_ff : lc_exp ff
  | lc_one : lc_exp one
  | lc_var a: lc_exp (V(EV.Free a))
.
```

```
(* Open a bound variable in an expression (original ope) *)
Definition open_exp (n : nat) (e' : exp) (e : exp) : exp :=
  match e with
  | V v => EV.open_var V n e' v
  | _ => e
  end.
```

Representing the Syntax.

Locally closed brings the binding structure.

Expressions $e ::= \texttt{tt} \mid \texttt{ff} \mid () \mid x$

```
Inductive exp : Set :=
  | tt
  | ff
  | one
  | V of evar
.

Coercion V : evar ⟶ exp.
```

```
Inductive lc_exp : exp → Prop :=
  | lc_tt : lc_exp tt
  | lc_ff : lc_exp ff
  | lc_one : lc_exp one
  | lc_var a: lc_exp (V(EV.Free a))
.
```

```
(* Open a bound variable in an expression (original ope) *)
Definition open_exp (n : nat) (e' : exp) (e : exp) : exp :=
  match e with
  | V v ⇒ EV.open_var V n e' v
  | _ ⇒ e
  end.
```

Representing the Syntax.

Locally closed brings the binding structure.

Expressions $e ::= \texttt{tt} \mid \texttt{ff} \mid () \mid x$

```
Inductive exp : Set :=
  | tt
  | ff
  | one
  | V of evar
.

Coercion V : evar >-> exp.
```

```
Inductive lc_exp : exp → Prop :=
  | lc_tt : lc_exp tt
  | lc_ff : lc_exp ff
  | lc_one : lc_exp one
  | lc_var a: lc_exp (V(EV.Free a))
.
```

```
(* Open a bound variable in an expression (original ope) *)
Definition open_exp (n : nat) (e' : exp) (e : exp) : exp :=
  match e with
  | V v ⇒ EV.open_var V n e' v
  | _ ⇒ e
  end.
```

Representing the Syntax.

Locally closed brings the binding structure.

Expressions $e ::= \texttt{tt} \mid \texttt{ff} \mid () \mid x$

```
Inductive exp : Set :=
  | tt
  | ff
  | one
  | V of evar
.

Coercion V : evar ↣ exp.
```

```
Inductive lc_exp : exp → Prop :=
  | lc_tt : lc_exp tt
  | lc_ff : lc_exp ff
  | lc_one : lc_exp one
  | lc_var a: lc_exp (V(EV.Free a))
.
```

```
(* Open a bound variable in an expression (original ope) *)
Definition open_exp (n : nat) (e' : exp) (e : exp) : exp :=
  match e with
  | V v ⇒ EV.open_var V n e' v
  | _ ⇒ e
  end.
```

The Syntax of Processes

Remember, locally closed brings the binding structure.

$$\text{Processes } P, Q ::= k![e].P \mid k?().P \mid P \mid Q$$
$$\mid \texttt{if}\ e\ \texttt{else}\ P\ \texttt{else}\ Q$$
$$\mid \nu.P \mid !\,P \mid \texttt{inact}$$

```
Inductive proc : Set :=
| send : CH.var → exp → proc → proc
| receive : CH.var → proc → proc
| ife : exp → proc → proc → proc
| par : proc → proc → proc
| inact : proc
| nu : proc → proc
| bang : proc → proc
.
```

The Syntax of Processes

Remember, locally closed brings the binding structure.

$$\text{Processes } P, Q ::= k![e].P \mid k?().P \mid P \mid Q$$
$$\mid \texttt{if } e \texttt{ else } P \texttt{ else } Q$$
$$\mid \nu.P \mid !P \mid \texttt{inact}$$

```
Inductive proc : Set :=
| send : CH.var → exp → proc → proc
| receive : CH.var → proc → proc
| ife : exp → proc → proc → proc
| par : proc → proc → proc
| inact : proc
| nu : proc → proc
| bang : proc → proc
.
```

The Syntax of Processes

Remember, locally closed brings the binding structure.

$$\text{Processes } P, Q ::= k![e].P \mid k?().P \mid P \mid Q$$
$$\mid \texttt{if } e \texttt{ else } P \texttt{ else } Q$$
$$\mid \nu.P \mid !\,P \mid \texttt{inact}$$

```
Inductive proc : Set :=
| send : CH.var → exp → proc → proc
| receive : CH.var → proc → proc
| ife : exp → proc → proc → proc
| par : proc → proc → proc
| inact : proc
| nu : proc → proc
| bang : proc → proc
.
```

The Syntax of Processes

Remember, locally closed brings the binding structure.

$$\text{Processes } P, Q ::= k![e].P \mid k?().P \mid P \mid Q \mid \texttt{if } e \texttt{ else } P \texttt{ else } Q \mid \nu.P \mid !P \mid \texttt{inact}$$

```
Inductive proc : Set :=
 | send : CH.var → exp → proc → proc
 | receive : CH.var → proc → proc
 | ife : exp → proc → proc → proc
 | par : proc → proc → proc
 | inact : proc
 | nu : proc → proc
 | bang : proc → proc
.
```

```
Fixpoint open_e (n : nat) (u : exp) (P : proc) : proc :=
  match P with
```

```
Fixpoint open_k (n : nat) (ko : CH.var) (P : proc) : proc :=
  match P with
```

```
Inductive lc : proc → Prop :=
| lc_send : forall k e P,
    CH.lc k →
    lc_exp e →
    lc P →
    lc (send k e P)

| lc_receive : forall (L : seq EV.atom) k P,
    CH.lc k →
    (forall x, x \notin L → lc (open_e0 P (V (EV.Free x)))) →
    lc (receive k P)

| lc_nu : forall (L : seq CH.atom) P,
    (forall k, k \notin L → lc (open_k0 P (CH.Free k))) →
    lc (nu P)

| lc_bang P: lc P → lc (bang P)

(* ... *)
```

The Syntax of Processes

Remember, locally closed brings the binding structure.

$$\text{Processes } P, Q ::= k![e].P \mid k?().P \mid P \mid Q$$
$$\mid \texttt{if } e \texttt{ else } P \texttt{ else } Q$$
$$\mid \nu.P \mid\, !P \mid \texttt{inact}$$

```
Inductive proc : Set :=
| send : CH.var → exp → proc → proc
| receive : CH.var → proc → proc
| ife : exp → proc → proc → proc
| par : proc → proc → proc
| inact : proc
| nu : proc → proc
| bang : proc → proc
.
```

```
Fixpoint open_e (n : nat) (u : exp) (P : proc) : proc :=
  match P with
```

```
Fixpoint open_k (n : nat) (ko : CH.var) (P : proc) : proc :=
  match P with
  | send k e P ⇒ send (opk n ko k) e (open_k n ko P)
```

```
Inductive lc : proc → Prop :=
| lc_send : forall k e P,
    CH.lc k →
    lc_exp e →
    lc P →
    lc (send k e P)

| lc_receive : forall (L : seq EV.atom) k P,
    CH.lc k →
    (forall x, x \notin L → lc (open_e0 P (V (EV.Free x)))) →
    lc (receive k P)

| lc_nu : forall (L : seq CH.atom) P,
    (forall k, k \notin L → lc (open_k0 P (CH.Free k))) →
    lc (nu P)

| lc_bang P: lc P → lc (bang P)

(* ... *)
```

Congruence and Reduction

Keeping an eye on locally nameless.

$$
\begin{array}{rl}
[\text{C-REFL}] & P \equiv P \\
[\text{C-INACT}] & P \mid \mathtt{inact} \equiv P \\
[\text{C-COMM}] & P \mid Q \equiv Q \mid P \\
[\text{C-ASSOC}] & (P \mid Q) \mid R \equiv P \mid (Q \mid R) \\
[\text{C-NU}] & \nu.(P \mid Q) \equiv \nu.P \mid Q \quad \text{if } \mathtt{lc}(Q) \\
[\text{C-NU'}] & \nu.\mathtt{inact} \equiv \mathtt{inact}
\end{array}
$$

Congruence and Reduction

Keeping an eye on locally nameless.

$$
\begin{array}{rl}
[\textsc{c-refl}] & P \equiv P \\
[\textsc{c-inact}] & P \mid \mathtt{inact} \equiv P \\
[\textsc{c-comm}] & P \mid Q \equiv Q \mid P \\
[\textsc{c-assoc}] & (P \mid Q) \mid R \equiv P \mid (Q \mid R) \\
[\textsc{c-nu}] & \nu.(P \mid Q) \equiv \nu.P \mid Q \quad \text{if } \mathtt{lc}(Q) \\
[\textsc{c-nu'}] & \nu.\mathtt{inact} \equiv \mathtt{inact}
\end{array}
$$

Congruence and Reduction

Keeping an eye on locally nameless.

[C-REFL]	$P \equiv P$
[C-INACT]	$P \mid \texttt{inact} \equiv P$
[C-COMM]	$P \mid Q \equiv Q \mid P$
[C-ASSOC]	$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$
[C-NU]	$\nu.(P \mid Q) \equiv \nu.P \mid Q$ if $\texttt{lc}(Q)$
[C-NU']	$\nu.\texttt{inact} \equiv \texttt{inact}$

[R-COM]	$k![e].P \mid k?().Q \to P \mid Q^e$
[R-CONG]	$P \equiv P'$ and $P' \to Q'$ and $Q' \equiv Q \Rightarrow P \to Q$
[R-SCOP]	$P \to Q \Rightarrow \nu.P \to \nu.Q$
[R-PAR]	$P \to P' \Rightarrow P \mid Q \to P' \mid Q$
[R-IF-TT]	$\texttt{if tt else } P \texttt{ else } Q \to P$
[R-IF-FF]	$\texttt{if ff else } P \texttt{ else } Q \to Q$

Congruence and Reduction

Keeping an eye on locally nameless.

$$[\textsc{C-Refl}] \quad P \equiv P$$

$$[\textsc{C-Inact}] \quad P \mid \texttt{inact} \equiv P$$

$$[\textsc{C-Comm}] \quad P \mid Q \equiv Q \mid P$$

$$[\textsc{C-Assoc}] \quad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$

$$[\textsc{C-Nu}] \quad \nu.(P \mid Q) \equiv \nu.P \mid Q \quad \text{if } \texttt{lc}(Q)$$

$$[\textsc{C-Nu'}] \quad \nu.\texttt{inact} \equiv \texttt{inact}$$

$$[\textsc{R-Com}] \quad k![e].P \mid k?().Q \to P \mid Q^{e}$$

$$[\textsc{R-Cong}] \quad P \equiv P' \text{and } P' \to Q' \text{and } Q' \equiv Q \Rightarrow P \to Q$$

$$[\textsc{R-Scop}] \quad P \to Q \Rightarrow \nu.P \to \nu.Q$$

$$[\textsc{R-Par}] \quad P \to P' \Rightarrow P \mid Q \to P' \mid Q$$

$$[\textsc{R-If-TT}] \quad \texttt{if}\ \texttt{tt}\ \texttt{else}\ P\ \texttt{else}\ Q \to P$$

$$[\textsc{R-If-FF}] \quad \texttt{if}\ \texttt{ff}\ \texttt{else}\ P\ \texttt{else}\ Q \to Q$$

Congruence and Reduction

Keeping an eye on locally nameless.

$$[\textsc{C-Refl}] \quad P \equiv P$$

$$[\textsc{C-Inact}] \quad P \mid \texttt{inact} \equiv P$$

$$[\textsc{C-Comm}] \quad P \mid Q \equiv Q \mid P$$

$$[\textsc{C-Assoc}] \quad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$

$$[\textsc{C-Nu}] \quad \nu.(P \mid Q) \equiv \nu.P \mid Q \quad \text{if } \texttt{lc}(Q)$$

$$[\textsc{C-Nu'}] \quad \nu.\texttt{inact} \equiv \texttt{inact}$$

$$[\textsc{R-Com}] \quad k![e].P \mid k?().Q \to P \mid Q^e$$

$$[\textsc{R-Cong}] \quad P \equiv P' \text{and } P' \to Q' \text{and } Q' \equiv Q \Rightarrow P \to Q$$

$$[\textsc{R-Scop}] \quad \forall k \notin L \quad P^k \to Q^k \Rightarrow \nu.P \to \nu.Q$$

$$[\textsc{R-Par}] \quad P \to P' \Rightarrow P \mid Q \to P' \mid Q$$

$$[\textsc{R-If-TT}] \quad \texttt{if tt else } P \texttt{ else } Q \to P$$

$$[\textsc{R-If-FF}] \quad \texttt{if ff else } P \texttt{ else } Q \to Q$$

Congruence and Reduction

Keeping an eye on locally nameless.

[C-REFL]	$P \equiv P$
[C-INACT]	$P \mid \texttt{inact} \equiv P$
[C-COMM]	$P \mid Q \equiv Q \mid P$
[C-ASSOC]	$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$
[C-NU]	$\nu.(P \mid Q) \equiv \nu.P \mid Q \quad \text{if } \texttt{lc}(Q)$
[C-NU']	$\nu.\texttt{inact} \equiv \texttt{inact}$

[R-COM]	$k![e].P \mid k?().Q \to P \mid Q^e$
[R-CONG]	$P \equiv P' \text{and } P' \to Q' \text{and } Q' \equiv Q \Rightarrow P \to Q$
[R-SCO	$\forall k \notin L \quad P^k \to Q^k \Rightarrow \nu.P \to \nu.Q$
[R-PAR]	$P \to P' \Rightarrow P \mid Q \to P' \mid Q$
[R-IF-TT]	$\texttt{if } \texttt{tt} \texttt{ else } P \texttt{ else } Q \to P$
[R-IF-FF]	$\texttt{if } \texttt{ff} \texttt{ else } P \texttt{ else } Q \to Q$

Representing Judgments

It’s really nothing new.

```
Reserved Notation "P ⟶ Q" (at level 70).
Inductive red : proc → proc → Prop :=
| r_com (k : CH.atom) e P Q:
    lc P →
    (par (send k e P) (receive k Q)) ⟶ (par P ({ope 0 ↝ e} Q))

| r_cong P P' Q Q' :
    lc P → lc Q →
    P ≡ P' →
    P' ⟶ Q' →
    Q' ≡ Q →
    P ⟶ Q

| r_scop P P':
    (forall (L : seq CH.atom) k,
        k \notin L → (open_k0 P (CH.Free k)) ⟶ (open_k0 P' (CH.Free k))) →
    nu P ⟶ nu P'

| r_par P P' Q:
    lc Q →
    P ⟶ P' →
    par P Q ⟶ par P' Q

| r_if_tt P Q: ife tt P Q ⟶ P
| r_if_ff P Q: ife ff P Q ⟶  Q
where "P ⟶ Q" := (red P Q).
```

Representing Judgments

It's really nothing new.

```
Reserved Notation "P ⟶ Q" (at level 70).
Inductive red : proc → proc → Prop :=
| r_com (k : CH.atom) e P Q:
    lc P →
    (par (send k e P) (receive k Q)) ⟶ (par P ({ope 0 ↝ e} Q))

| r_cong P P' Q Q' :
    lc P → lc Q →
    P ≡≡ P' →
    P' ⟶ Q' →
    Q' ≡≡ Q →
    P ⟶ Q

| r_scop P P':
    (forall (L : seq CH.atom) k,
        k \notin L → (open_k0 P (CH.Free k)) ⟶ (open_k0 P' (CH.Free k))) →
    nu P ⟶ nu P'

| r_par P P' Q:
    lc Q →
    P ⟶ P' →
    par P Q ⟶ par P' Q

| r_if_tt P Q: ife tt P Q ⟶ P
| r_if_ff P Q: ife ff P Q ⟶  Q
where "P ⟶ Q" := (red P Q).
```

[R-COM]	$k![e].P \mid k?().Q \to P \mid Q^e$
[R-CONG]	$P \equiv P' \text{and } P' \to Q' \text{and } Q' \equiv Q \Rightarrow P \to Q$
[R-SCOP]	$\forall k \notin L \quad P^k \to Q^k \Rightarrow \nu.P \to \nu.Q$
[R-PAR]	$P \to P' \Rightarrow P \mid Q \to P' \mid Q$
[R-IF-TT]	$\texttt{if tt else } P \texttt{ else } Q \to P$
[R-IF-FF]	$\texttt{if ff else } P \texttt{ else } Q \to Q$

Representing Judgments

It’s really nothing new.

```
Reserved Notation "P ⟶ Q" (at level 70).
Inductive red : proc → proc → Prop :=
| r_com (k : CH.atom) e P Q:
    lc P →
    (par (send k e P) (receive k Q)) ⟶ (par P ({ope 0 ↝ e} Q))

| r_cong P P' Q Q' :
    lc P → lc Q →
    P ≡ P' →
    P' ⟶ Q' →
    Q' ≡ Q →
    P ⟶ Q

| r_scop P P':
    (forall (L : seq CH.atom) k,
        k \notin L → (open_k0 P (CH.Free k)) ⟶ (open_k0 P' (CH.Free k))) →
    nu P ⟶ nu P'

| r_par P P' Q:
    lc Q →
    P ⟶ P' →
    par P Q ⟶ par P' Q

| r_if_tt P Q: ife tt P Q ⟶ P
| r_if_ff P Q: ife ff P Q ⟶  Q
where "P ⟶ Q" := (red P Q).
```

[R-COM]	$k![e].P \mid k?().Q \to P \mid Q^e$
[R-CONG]	$P \equiv P'$ and $P' \to Q'$ and $Q' \equiv Q \Rightarrow P \to Q$
[R-SCOP]	$\forall k \notin L \quad P^k \to Q^k \Rightarrow \nu.P \to \nu.Q$
[R-PAR]	$P \to P' \Rightarrow P \mid Q \to P' \mid Q$
[R-IF-TT]	if tt else P else $Q \to P$
[R-IF-FF]	if ff else P else $Q \to Q$

Representing Judgments

It's really nothing new.

```
Reserved Notation "P ⟶ Q" (at level 70).
Inductive red : proc → proc → Prop :=
| r_com (k : CH.atom) e P Q:
    lc P →
    (par (send k e P) (receive k Q)) ⟶ (par P ({ope 0 ↝ e} Q))

| r_cong P P' Q Q' :
    lc P → lc Q →
    P ≡≡ P' →
    P' ⟶ Q' →
    Q' ≡≡ Q →
    P ⟶ Q

| r_scop P P':
    (forall (L : seq CH.atom) k,
        k \notin L → (open_k0 P (CH.Free k)) ⟶ (open_k0 P' (CH.Free k))) →
    nu P ⟶ nu P'

| r_par P P' Q:
    lc Q →
    P ⟶ P' →
    par P Q ⟶ par P' Q

| r_if_tt P Q: ife tt P Q ⟶ P
| r_if_ff P Q: ife ff P Q ⟶  Q
where "P ⟶ Q" := (red P Q).
```

[R-COM]	$k![e].P \mid k?().Q \to P \mid Q^e$
[R-CONG]	$P \equiv P'$ and $P' \to Q'$ and $Q' \equiv Q \Rightarrow P \to Q$
[R-SCOP]	$\forall k \notin L \quad P^k \to Q^k \Rightarrow \nu.P \to \nu.Q$
[R-PAR]	$P \to P' \Rightarrow P \mid Q \to P' \mid Q$
[R-IF-TT]	$\texttt{if tt else } P \texttt{ else } Q \to P$
[R-IF-FF]	$\texttt{if ff else } P \texttt{ else } Q \to Q$

Typing smolEMTST

Locally nameless became easy by now.

[T-SEND]

$$\frac{\Gamma \vdash_e e : S \qquad \Gamma \vdash P : \Delta \cdot k : T}{\Gamma \vdash k![e].P : \Delta \cdot k : ![S].T}$$

[T-RECEIVE]

$$\frac{\forall x \notin L \qquad \Gamma \cdot x : S \vdash P^x : \Delta \cdot k : T}{\Gamma \vdash k?().P : \Delta \cdot k : ?[S].T}$$

[T-PAR]

$$\frac{\Gamma \vdash P : \Delta \qquad \Gamma \vdash P : \Delta' \qquad \Delta \asymp \Delta'}{\Gamma \vdash P \mid Q : \Delta \circ \Delta'}$$

[T-IFE]

$$\frac{\Gamma \vdash_e e : bool \qquad \Gamma \vdash P : \Delta \qquad \Gamma \vdash Q : \Delta}{\Gamma \vdash \texttt{if}\ e\ \texttt{else}\ P\ \texttt{else}\ Q : \Delta}$$

[T-INACT]

$$\frac{\texttt{completed}(\Delta)}{\Gamma \vdash \texttt{inact} : \Delta}$$

[T-BANG]

$$\frac{\texttt{completed}(D) \qquad \Gamma \vdash P : \cdot}{\Gamma \vdash\ !P : \Delta}$$

[T-NU]

$$\frac{\forall k \notin L \qquad \Gamma \vdash P^k : \Delta \cdot k : \bot}{\Gamma \vdash \nu.P : \Delta}$$

[T-NU']

$$\frac{\Gamma \vdash P : \Delta}{\Gamma \vdash \nu.P : \Delta}$$

Typing smolEMTST

Locally nameless became easy by now.

[T-SEND]
$$\frac{\Gamma \vdash_e e : S \qquad \Gamma \vdash P : \Delta \cdot k : T}{\Gamma \vdash k![e].P : \Delta \cdot k : ![S].T}$$

[T-RECEIVE]
$$\frac{\forall x \notin L \qquad \Gamma \cdot x : S \vdash P^x : \Delta \cdot k : T}{\Gamma \vdash k?().P : \Delta \cdot k : ?[S].T}$$

[T-PAR]
$$\frac{\Gamma \vdash P : \Delta \qquad \Gamma \vdash P : \Delta' \qquad \Delta \asymp \Delta'}{\Gamma \vdash P \mid Q : \Delta \circ \Delta'}$$

[T-IFE]
$$\frac{\Gamma \vdash_e e : bool \qquad \Gamma \vdash P : \Delta \qquad \Gamma \vdash Q : \Delta}{\Gamma \vdash \texttt{if}\ e\ \texttt{else}\ P\ \texttt{else}\ Q : \Delta}$$

[T-INACT]
$$\frac{\texttt{completed}(\Delta)}{\Gamma \vdash \texttt{inact} : \Delta}$$

[T-BANG]
$$\frac{\texttt{completed}(D) \qquad \Gamma \vdash P : \cdot}{\Gamma \vdash\ !P : \Delta}$$

[T-NU]
$$\frac{\forall k \notin L \qquad \Gamma \vdash P^k : \Delta \cdot k : \bot}{\Gamma \vdash \nu.P : \Delta}$$

[T-NU']
$$\frac{\Gamma \vdash P : \Delta}{\Gamma \vdash \nu.P : \Delta}$$

Typing smolEMTST

Locally nameless became easy by now.

[T-SEND]

$$\frac{\Gamma \vdash_e e : S \qquad \Gamma \vdash P : \Delta \cdot k : T}{\Gamma \vdash k![e].P : \Delta \cdot k : ![S].T}$$

[T-RECEIVE]

$$\frac{\forall x \notin L \qquad \Gamma \cdot x : S \vdash P^x : \Delta \cdot k : T}{\Gamma \vdash k?().P : \Delta \cdot k : ?[S].T}$$

[T-PAR]

$$\frac{\Gamma \vdash P : \Delta \qquad \Gamma \vdash P : \Delta' \qquad \Delta \asymp \Delta'}{\Gamma \vdash P \mid Q : \Delta \circ \Delta'}$$

[T-IFE]

$$\frac{\Gamma \vdash_e e : bool \qquad \Gamma \vdash P : \Delta \qquad \Gamma \vdash Q : \Delta}{\Gamma \vdash \texttt{if}\ e\ \texttt{else}\ P\ \texttt{else}\ Q : \Delta}$$

[T-INACT]

$$\frac{\texttt{completed}(\Delta)}{\Gamma \vdash \texttt{inact} : \Delta}$$

[T-BANG]

$$\frac{\texttt{completed}(D) \qquad \Gamma \vdash P : \cdot}{\Gamma \vdash !P : \Delta}$$

[T-NU]

$$\frac{\forall k \notin L \qquad \Gamma \vdash P^k : \Delta \cdot k : \bot}{\Gamma \vdash \nu.P : \Delta}$$

[T-NU']

$$\frac{\Gamma \vdash P : \Delta}{\Gamma \vdash \nu.P : \Delta}$$

Typing smolEMTST

Locally nameless became easy by now.

[T-SEND]

$$\frac{\Gamma \vdash_e e : S \qquad \Gamma \vdash P : \Delta \cdot k : T}{\Gamma \vdash k![e].P : \Delta \cdot k : ![S].T}$$

[T-RECEIVE]

$$\frac{\forall x \notin L \qquad \Gamma \cdot x : S \vdash P^x : \Delta \cdot k : T}{\Gamma \vdash k?().P : \Delta \cdot k : ?[S].T}$$

[T-PAR]

$$\frac{\Gamma \vdash P : \Delta \qquad \Gamma \vdash P : \Delta' \qquad \Delta \asymp \Delta'}{\Gamma \vdash P \mid Q : \Delta \circ \Delta'}$$

[T-IFE]

$$\frac{\Gamma \vdash_e e : bool \qquad \Gamma \vdash P : \Delta \qquad \Gamma \vdash Q : \Delta}{\Gamma \vdash \texttt{if}\ e\ \texttt{else}\ P\ \texttt{else}\ Q : \Delta}$$

[T-INACT]

$$\frac{\texttt{completed}(\Delta)}{\Gamma \vdash \texttt{inact} : \Delta}$$

[T-BANG]

$$\frac{\texttt{completed}(D) \qquad \Gamma \vdash P : \cdot}{\Gamma \vdash !P : \Delta}$$

[T-NU]

$$\frac{\forall k \notin L \qquad \Gamma \vdash P^k : \Delta \cdot k : \bot}{\Gamma \vdash \nu.P : \Delta}$$

[T-NU']

$$\frac{\Gamma \vdash P : \Delta}{\Gamma \vdash \nu.P : \Delta}$$

Substitution Lemmata

Because one lemma is not enough.

```
Lemma SubstitutionLemmaExp G x S S' e e':
  binds x S' G →
  oft_exp G e' S' →
  oft_exp G e S → oft_exp G (s[ x ↝ e']e e) S.
Proof.
  move⇒Hbind Hde' Hde.
  move:Hde'.
  elim Hde ; try constructor ; try assumption.
  intros.
  case: (EV.eq_reflect x x0).
  move⇒Sub.
  subst.
  simpl.
  rewrite eq_refl.
  have Heq : S' = S0 by apply: UniquenessBind ; [apply: Hbind | apply: H].
  rewrite-Heq.
  assumption.

  case/eqP⇒Hdiff⇒ //=.
  rewrite ifN_eq ; try assumption.
  by constructor.
Qed.
```

Substitution Lemmata

Because one lemma is not enough.

```
Lemma SubstitutionLemma_exp G x S S' e e':
  binds x S' G →
  oft_exp G e' S' →
  oft_exp G e S → oft_exp G (s[ x ↝ e']e e) S.
Proof.
  move⇒Hbind Hde' Hde.
  move:Hde'.
  elim Hde ; try constructor ; try assumption.
  intros.
  case: (EV.eq_reflect x x0).
  move⇒Sub.
  subst.
  simpl.
  rewrite eq_refl.
  have Heq : S' = S0 by apply: UniquenessBind ; [apply: Hbind | apply: H].
  rewrite-Heq.
  assumption.

  case/eqP⇒Hdiff⇒ //⇐.
  rewrite ifN_eq ; try assumption.
  by constructor.
Qed.
```

Substitution Lemmata

Because one lemma is not enough.

```
Lemma SubstitutionLemmaExp G x S S' e e':
  binds x S' G →
  oft_exp G e' S' →
  oft_exp G e S → oft_exp G (s[ x ↝ e']e e) S.
Proof.
  move⇒Hbind Hde' Hde.
  move:Hde'.
  elim Hde ; try constructor ; try assumption.
  intros.
  case: (EV.eq_reflect x x0).
  move⇒Sub.
  subst.
  simpl.
  rewrite eq_refl.
  have Heq : S' = S0 by apply: UniquenessBind ; [apply: Hbind | apply: H].
  rewrite-Heq.
  assumption.

  case/eqP⇒Hdiff⇒ //=.
  rewrite ifN_eq ; try assumption.
  by constructor.
Qed.
```

Substitution Lemmata

Because one lemma is not enough.

```
Lemma SubstitutionLemmaExp G x S S' e e':
  binds x S' G →
  oft_exp G e' S' →
  oft_exp G e S → oft_exp G (s[ x ⇝ e']e e) S.
Proof.
  move⇒Hbind Hde' Hde.
  move:Hde'
  elim Hde ; try constructor ; try assumption.
  intros.
  case: (EV.eq_reflect x x0).
  move⇒Sub.
  subst.
  simpl.
  rewrite eq_refl.
  have Heq : S' = S0 by apply: UniquenessBind ; [apply: Hbind | apply: H].
  rewrite-Heq.
  assumption.

  case/eqP⇒Hdiff⇒ //=.
  rewrite ifN_eq ; try assumption.
  by constructor.
Qed.
```

Substitution Lemmata

Because one lemma is not enough.

```
Lemma SubstitutionLemmaExp G x S S' e e':
  binds x S' G →
  oft_exp G e' S' →
  oft_exp G e S → oft_exp G (s[ x ⇝ e']e e) S.
Proof.
  move⇒Hbind Hde' Hde.
  move:Hde'.
  elim Hde ; try constructor ; try assumption.
  intros.
  case: (EV.eq_reflect x x0).
  move⇒Sub.
  subst.
  simpl.
  rewrite eq_refl.
  have Heq : S' = S0 by apply: UniquenessBind ; [apply: Hbind | apply: H].
  rewrite-Heq.
  assumption.

  case/eqP⇒Hdiff⇒//⇐.
  rewrite ifN_eq ; try assumption.
  by constructor.
Qed.
```

Substitution Lemmata

Because one lemma is not enough.

```
Lemma SubstitutionLemmaExp G x S S' e e':
  binds x S' G →
  oft_exp G e' S' →
  oft_exp G e S → oft_exp G (s[ x ↝ e']e e) S.
Proof.
  move⇒Hbind Hde' Hde.
  move:Hde'.
  elim Hde ; try constructor ; try assumption.
  intros.
  case: (EV.eq_reflect x x0).
  move⇒Sub.
  subst.
  simpl.
  rewrite eq_refl.
  have Heq : S' = S0 by apply: UniquenessBind ; [apply: Hbind | apply: H].
  rewrite-Heq.
  assumption.

  case/eqP⇒Hdiff⇒ //⇐.
  rewrite ifN_eq ; try assumption.
  by constructor.
Qed.
```

Substitution Lemmata

Because one lemma is not enough.

```
Lemma SubstitutionLemmaExp G x S S' e e':
  binds x S' G →
  oft_exp G e' S' →
  oft_exp G e S → oft_exp G (s[ x ↝ e']e e) S.
Proof.
  move⇒Hbind Hde' Hde.
  move:Hde'.
  elim Hde ; try constructor ; try assumption.
  intros.
  case: (EV.eq_reflect x x0).
  move⇒Sub.
  subst.
  simpl.
  rewrite eq_refl.
  have Heq : S' = S0 by apply: UniquenessBind ; [apply: Hbind | apply: H].
  rewrite-Heq.
  assumption.

  case/eqP⇒Hdiff⇒//⇐.
  rewrite ifN_eq ; try assumption.
  by constructor.
Qed.
```

```
Theorem ExpressionReplacement G P x E S D:
  binds x S G →
  oft_exp G E S →
  oft G P D →
  oft G (s[ x ↝ E]pe P) D.
Proof.
Admitted.
```

Substitution Lemmata

Because one lemma is not enough.

```
Lemma SubstitutionLemmaExp G x S S' e e':
  binds x S' G →
  oft_exp G e' S' →
  oft_exp G e S → oft_exp G (s[ x ↝ e']e e) S.
Proof.
  move⇒Hbind Hde' Hde.
  move:Hde'.
  elim Hde ; try constructor ; try assumption.
  intros.
  case: (EV.eq_reflect x x0).
  move⇒Sub.
  subst.
  simpl.
  rewrite eq_refl.
  have Heq : S' = S0 by apply: UniquenessBind ; [apply: Hbind | apply: H].
  rewrite-Heq.
  assumption.

  case/eqP⇒Hdiff⇒//=.
  rewrite ifN_eq ; try assumption.
  by constructor.
Qed.
```

```
Theorem ExpressionReplacement G P x E S D:
  binds x S G →
  oft_exp G E S →
  oft G P D →
  oft G (s[ x ↝ E]pe P) D.
Proof.
Admitted.
```

Substitutes expressions in processes

Substitution Lemmata

Because one lemma is not enough.

```
Lemma SubstitutionLemmaExp G x S S' e e':
  binds x S' G →
  oft_exp G e' S' →
  oft_exp G e S → oft_exp G (s[ x ⇝ e']e e) S.
Proof.
  move⇒Hbind Hde' Hde.
  move:Hde'.
  elim Hde ; try constructor ; try assumption.
  intros.
  case: (EV.eq_reflect x x0).
  move⇒Sub.
  subst.
  simpl.
  rewrite eq_refl.
  have Heq : S' = S0 by apply: UniquenessBind ; [apply: Hbind | apply: H].
  rewrite-Heq.
  assumption.

  case/eqP⇒Hdiff⇒ //⇐.
  rewrite ifN_eq ; try assumption.
  by constructor.
Qed.
```

```
Theorem ExpressionReplacement G P x E S D:
  binds x S G →
  oft_exp G E S →
  oft G P D →
  oft G (s[ x ⇝ E]pe P) D.
Proof.
Admitted.
```

Substitution Lemmata

Because one lemma is not enough.

```
Lemma SubstitutionLemmaExp G x S S' e e':
  binds x S' G →
  oft_exp G e' S' →
  oft_exp G e S → oft_exp G (s[ x ↝ e']e e) S.
Proof.
  move⇒Hbind Hde' Hde.
  move:Hde'.
  elim Hde ; try constructor ; try assumption.
  intros.
  case: (EV.eq_reflect x x0).
  move⇒Sub.
  subst.
  simpl.
  rewrite eq_refl.
  have Heq : S' = S0 by apply: UniquenessBind ; [apply: Hbind | apply: H].
  rewrite-Heq.
  assumption.

  case/eqP⇒Hdiff⇒ //=.
  rewrite ifN_eq ; try assumption.
  by constructor.
Qed.
```

```
Theorem ExpressionReplacement G P x E S D:
  binds x S G →
  oft_exp G E S →
  oft G P D →
  oft G (s[ x ↝ E]pe P) D.
Proof.
Admitted.
```

```
Lemma ChannelReplacement G P c c' D :
  oft G P D →
  def (subst_env c c' D) →
  oft G (s[ c ↝ chan_of_entry c' ]p P) (subst_env c c' D).
Proof.
  case: (boolP (c' == c)) ⇒[/eqP→/⟸|c_neq_c']; (* ... *)
```

Subject Reduction

These proofs will motivate some other lemmas (e.g: weakening).

```
Theorem CongruencePreservesOft G P Q D :
  P ≡ Q → oft G P D → oft G Q D.
Proof.
  elim⇒{P}{Q}//. (* ... *)
```

Subject Reduction

These proofs will motivate some other lemmas (e.g: weakening).

```
Theorem CongruencePreservesOft G P Q D :
  P ≡ Q → oft G P D → oft G Q D.
Proof.
  elim⇒{P}{Q}//. (* ... *)
```

Subject Reduction

These proofs will motivate some other lemmas (e.g: weakening).

```
Theorem CongruencePreservesOft G P Q D :
  P ≡ Q → oft G P D → oft G Q D.
Proof.
  elim⇒{P}{Q}//. (* ... *)
```

Subject Reduction

These proofs will motivate some other lemmas (e.g: weakening).

```
Theorem CongruencePreservesOft G P Q D :
  P ≡ Q → oft G P D → oft G Q D.
Proof.
  elim⇒{P}{Q}//. (* ... *)
```

Subject Reduction

These proofs will motivate some other lemmas (e.g: weakening).

```
Theorem CongruencePreservesOft G P Q D :
  P ≡ Q → oft G P D → oft G Q D.
Proof.
  elim⇒{P}{Q}//. (* ... *)
```

```
Theorem SubjectReductionStep G P Q D:
  oft G P D → P ⟶ Q → exists D', D ⇝ D' ∧ oft G Q D'.
Proof.
  move⇒ Op PQ.  (* ... *)
```

```
Theorem SubjectReduction G P Q D:
  oft G P D → P ⟶* Q → exists D', oft G Q D'.
Proof.
  move ⇒ Hoft PQ; elim: PQ D Hoft ⇒ {P} {Q} P.
  + by move⇒ D Hoft; exists D.
  + move⇒ Q R Step QR IH D Hoft.
    move: (SubjectReductionStep Hoft Step) ⇒ []D' []bD' Hoft'.
    by move: (IH D' Hoft').
Qed.
```

Subject Reduction

These proofs will motivate some other lemmas (e.g: weakening).

```
Theorem CongruencePreservesOft G P Q D :
  P ≡ Q → oft G P D → oft G Q D.
Proof.
  elim⇒{P}{Q}//. (* ... *)
```

```
Theorem SubjectReductionStep G P Q D:
  oft G P D → P ⟶ Q → exists D', D ⤳ D' ∧ oft G Q D'.
Proof.
  move⇒ Op PQ.  (* ... *)
```

```
Theorem SubjectReduction G P Q D:
  oft G P D → P ⟶* Q → exists D', oft G Q D'.
Proof.
  move ⇒ Hoft PQ; elim: PQ D Hoft ⇒ {P} {Q} P.
  + by move⇒ D Hoft; exists D.
  + move⇒ Q R Step QR IH D Hoft.
    move: (SubjectReductionStep Hoft Step) ⇒ []D' []bD' Hoft'.
    by move: (IH D' Hoft').
Qed.
```

Subject Reduction

These proofs will motivate some other lemmas (e.g: weakening).

```
Theorem CongruencePreservesOft G P Q D :
  P ≡ Q → oft G P D → oft G Q D.
Proof.
  elim⇒{P}{Q}//. (* … *)
```

```
Theorem SubjectReductionStep G P Q D:
  oft G P D → P ⟶ Q → exists D', D ⇝ D' ∧ oft G Q D'.
Proof.
  move⇒ Op PQ.  (* … *)
```

```
Theorem SubjectReduction G P Q D:
  oft G P D → P ⟶* Q → exists D', oft G Q D'.
Proof.
  move ⇒ Hoft PQ; elim: PQ D Hoft ⇒ {P} {Q} P.
  + by move⇒ D Hoft; exists D.
  + move⇒ Q R Step QR IH D Hoft.
    move: (SubjectReductionStep Hoft Step) ⇒ []D' []bD' Hoft'.
    by move: (IH D' Hoft').
Qed.
```

Subject Reduction

These proofs will motivate some other lemmas (e.g: weakening).

```
Theorem CongruencePreservesOft G P Q D :
  P ≡ Q → oft G P D → oft G Q D.
Proof.
  elim⇒{P}{Q}//. (* … *)
```

```
Theorem SubjectReductionStep G P Q D:
  oft G P D → P ⟶ Q → exists D', D ⇝ D' ∧ oft G Q D'.
Proof.
  move⇒ Op PQ.  (* … *)
```

```
Theorem SubjectReduction G P Q D:
  oft G P D → P ⟶* Q → exists D', oft G Q D'.
Proof.
  move ⇒ Hoft PQ; elim: PQ D Hoft ⇒ {P} {Q} P.
  + by move⇒ D Hoft; exists D.
  + move⇒ Q R Step QR IH D Hoft.
    move: (SubjectReductionStep Hoft Step) ⇒ []D' []bD' Hoft'.
    by move: (IH D' Hoft').
Qed.
```

Conclusion of the First Act.

No intermediate and no tiny ice cream like at the theatre.

- Deep embedding (LN) binders allows us to fully control the calculus.
- LN demands tribute for that control (in the shape of theorems).
- EMTST (the tool) helps with nominal sets and environments.
- **In the next act we explore what do we get if we give up control (using shallow embeddings).**