lec14: Typing for Lλ

Joshua Dunfield

March 19, 2018

Added missing type-abs rule in Figure 1; fixed details in preservation proof; added →Elim case of preservation that we did on 13 March; progress to be added

1 Typing for Lλ

Expressions	e	$::=$	`()`
			$\mid$ n $\mid$ `(+` e e`)` $\mid$ `(-` e e`)` $\mid$ `(Abs` e`)`
			$\mid$ `True` $\mid$ `False` $\mid$ `(Ite` e e e`)`
			$\mid$ `(=` e e`)` $\mid$ `(<` e e`)`
			$\mid$ x $\mid$ `(Lam` x e`)` $\mid$ `(Call` e e`)`
			$\mid$ `(Pair` e e`)` $\mid$ $(\texttt{Proj}_1\ e)$ $\mid$ $(\texttt{Proj}_2\ e)$
			$\mid$ $(\texttt{Inj}_1\ e)$ $\mid$ $(\texttt{Inj}_2\ e)$ $\mid$ `(Case` e `(`x `=>` e`)` `(`x `=>` e`))`
Values	v	$::=$	`()`
			$\mid$ n
			$\mid$ `True` $\mid$ `False`
			$\mid$ x $\mid$ `(Lam` x e`)`
			$\mid$ `(Pair` v v`)`
			$\mid$ $(\texttt{Inj}_1\ v)$ $\mid$ $(\texttt{Inj}_2\ v)$

Types	S, T	$::=$	unit	unit type
			$\mid$ int	type of integers
			$\mid$ bool	type of booleans
			$\mid$ $S \to T$	type of functions on S that produce T
			$\mid$ $S \times T$	type of pairs of one S and one T
			$\mid$ $S + T$	*disjoint union* or *sum* type: contains either an S or a T
Typing contexts	Γ	$::=$	$\emptyset$	empty context
			$\mid$ $\Gamma, x : S$	x has type S

$\boxed{\Gamma \vdash e : T}$ Under assumptions Γ, expression e has type T

$$\frac{(x : S) \in \Gamma}{\Gamma \vdash x : S}\ \text{type-assum} \qquad \frac{\Gamma, x : S \vdash e : T}{\Gamma \vdash (\mathtt{Lam}\ x\ e) : (S \to T)}\ {\to}\text{Intro} \qquad \frac{\Gamma \vdash e_1 : (S \to T) \qquad \Gamma \vdash e_2 : S}{\Gamma \vdash (\mathtt{Call}\ e_1\ e_2) : T}\ {\to}\text{Elim}$$

$$\frac{}{\Gamma \vdash \mathtt{()} : \mathsf{unit}}\ \text{unitIntro} \qquad \frac{}{\Gamma \vdash \mathtt{True} : \mathsf{bool}}\ \text{type-true} \qquad \frac{}{\Gamma \vdash \mathtt{False} : \mathsf{bool}}\ \text{type-false}$$

$$\frac{}{\Gamma \vdash n : \mathsf{int}}\ \text{intIntro} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash (\mathtt{+}\ e_1\ e_2) : \mathsf{int}}\ \text{type-add} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash (\mathtt{-}\ e_1\ e_2) : \mathsf{int}}\ \text{type-sub}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{int}}{\Gamma \vdash (\mathtt{Abs}\ e_1) : \mathsf{int}}\ \text{type-abs} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash (\mathtt{=}\ e_1\ e_2) : \mathsf{bool}}\ \text{type-equals} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash (\mathtt{<}\ e_1\ e_2) : \mathsf{bool}}\ \text{type-lt}$$

$$\frac{\Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash e_{\text{then}} : T \qquad \Gamma \vdash e_{\text{else}} : T}{\Gamma \vdash (\mathtt{Ite}\ e\ e_{\text{then}}\ e_{\text{else}}) : T}\ \text{type-ite}$$

$$\frac{\Gamma \vdash e_1 : S_1}{\Gamma \vdash (\mathtt{Inj}_1\ e_1) : (S_1 + S_2)}\ {+}\text{Intro1} \qquad \frac{\Gamma \vdash e_2 : S_2}{\Gamma \vdash (\mathtt{Inj}_2\ e_2) : (S_1 + S_2)}\ {+}\text{Intro2}$$

$$\frac{\Gamma \vdash e : (S_1 + S_2) \qquad \Gamma, x_1 : S_1 \vdash e_1 : T \qquad \Gamma, x_2 : S_2 \vdash e_2 : T}{\Gamma \vdash (\mathtt{Case}\ e\ (x_1\ \mathtt{=>}\ e_1)\ (x_2\ \mathtt{=>}\ e_2)) : T}\ {+}\text{Elim}$$

$$\frac{\Gamma \vdash e_1 : S_1 \qquad \Gamma \vdash e_2 : S_2}{\Gamma \vdash (\mathtt{Pair}\ e_1\ e_2) : (S_1 \times S_2)}\ {\times}\text{Intro} \qquad \frac{\Gamma \vdash e : (S_1 \times S_2)}{\Gamma \vdash (\mathtt{Proj}_1\ e) : S_1}\ {\times}\text{Elim1} \qquad \frac{\Gamma \vdash e : (S_1 \times S_2)}{\Gamma \vdash (\mathtt{Proj}_2\ e) : S_2}\ {\times}\text{Elim2}$$

Figure 1 **Typing with functions, integers, booleans, sums (unions), and pairs (structs)**

2 Small-step semantics for Lλ

$$
\begin{array}{lrl}
\text{Contexts} & \mathcal{C} ::= & [\,] \\
& & \mid (\texttt{+}\ \mathcal{C}\ e) \mid (\texttt{+}\ v\ \mathcal{C}) \\
& & \mid (\texttt{-}\ \mathcal{C}\ e) \mid (\texttt{-}\ v\ \mathcal{C}) \\
& & \mid (\texttt{Abs}\ \mathcal{C}) \\
& & \mid (\texttt{Ite}\ \mathcal{C}\ e\ e) \\
& & \mid (\texttt{=}\ \mathcal{C}\ e) \mid (\texttt{=}\ v\ \mathcal{C}) \\
& & \mid (\texttt{<}\ \mathcal{C}\ e) \mid (\texttt{<}\ v\ \mathcal{C}) \\
& & \mid (\texttt{Call}\ \mathcal{C}\ e) \\
& & \mid (\texttt{Call}\ v\ \mathcal{C}) \\
& & \mid (\texttt{Pair}\ \mathcal{C}\ e) \mid (\texttt{Pair}\ v\ \mathcal{C}) \\
& & \mid (\texttt{Proj}_1\ \mathcal{C}) \mid (\texttt{Proj}_2\ \mathcal{C}) \\
& & \mid (\texttt{Inj}_1\ \mathcal{C}) \mid (\texttt{Inj}_2\ \mathcal{C}) \mid (\texttt{Case}\ \mathcal{C}\ (x\ \texttt{=>}\ e)\ (x\ \texttt{=>}\ e))
\end{array}
$$

$\boxed{e \mapsto_R e'}$ Expression e reduces to e'

$$\frac{}{(\texttt{+}\ n_1\ n_2) \mapsto_R (n_1 + n_2)}\ \text{red-add} \qquad \frac{}{(\texttt{-}\ n_1\ n_2) \mapsto_R (n_1 - n_2)}\ \text{red-sub} \qquad \frac{}{(\texttt{Abs}\ n) \mapsto_R |n|}\ \text{red-abs}$$

$$\frac{}{(\texttt{=}\ n_1\ n_2) \mapsto_R (n_1 = n_2)}\ \text{red-equals} \qquad \frac{}{(\texttt{<}\ n_1\ n_2) \mapsto_R (n_1 < n_2)}\ \text{red-lessthan}$$

$$\frac{}{(\texttt{Ite True}\ e_{\text{then}}\ e_{\text{else}}) \mapsto_R e_{\text{then}}}\ \text{red-ite-then} \qquad \frac{}{(\texttt{Ite True}\ e_{\text{then}}\ e_{\text{else}}) \mapsto_R e_{\text{then}}}\ \text{red-ite-then}$$

$$\frac{}{(\texttt{Call}\ (\texttt{Lam}\ x\ e)\ v) \mapsto_R [v/x]e}\ \text{red-beta}$$

$$\frac{}{(\texttt{Proj}_1\ (\texttt{Pair}\ v_1\ v_2)) \mapsto_R v_1}\ \text{red-proj1} \qquad \frac{}{(\texttt{Proj}_2\ (\texttt{Pair}\ v_1\ v_2)) \mapsto_R v_2}\ \text{red-proj2}$$

$$\frac{}{(\texttt{Case}\ (\texttt{Inj}_1\ v_1)\ (x_1\ \texttt{=>}\ e_1)\ (x_2\ \texttt{=>}\ e_2)) \mapsto_R [v_1/x_1]e_1}\ \text{red-case1}$$

$$\frac{}{(\texttt{Case}\ (\texttt{Inj}_2\ v_2)\ (x_1\ \texttt{=>}\ e_1)\ (x_2\ \texttt{=>}\ e_2)) \mapsto_R [v_2/x_2]e_2}\ \text{red-case2}$$

$\boxed{e \mapsto e'}$ expression e takes one step to e'

$$\frac{e \mapsto_R e'}{\mathcal{C}[e] \mapsto \mathcal{C}[e']}\ \text{step-context}$$

2.1 Preservation and Progress

Lemma 1 (Substitution). *If* $x : T \vdash e_1 : S$ *and* $\emptyset \vdash v_2 : T$ *then* $\emptyset \vdash [v_2/x]e_1 : S$.

Proving the Substitution Lemma is very tedious. It's not entirely straightforward, because—while the above form is sufficient for Type Preservation—the proof of the Substitution Lemma doesn't work unless we *generalize* the induction hypothesis. (To see why, try to prove the case of the above substitution lemma when $x : T \vdash e_1 : S$ is derived by →Intro. That case comes up when a `Lam` is the body of a `Lam`.)

Lemma 2 (Substitution (Generalized)). *If* $\Gamma_L, x : T, \Gamma_R \vdash e_1 : S$ *and* $\emptyset \vdash v_2 : T$ *then* $\Gamma_L, \Gamma_R \vdash [v_2/x]e_1 : S$.

If we did prove this, we would see that we can generalize the above result further: none of the steps of the proof actually use the fact that v_2 is a value, so it could be generalized to show $\Gamma_L, \Gamma_R \vdash [e_2/x]e_1 : S$. That generalization would be very useful if we were proving type preservation for a call-by-name language, because then the reduction rule for a `Call` would substitute e_2, not v_2, for x.

Conjecture 1 (Type Preservation).
If $\emptyset \vdash e : S$
and $e \mapsto e'$
then $\emptyset \vdash e' : S$.

Proof. By induction on the derivation of $\emptyset \vdash e : S$. [Should also be possible to induct on e, since in our current typing rules, the expressions in the premises are always subexpressions of the expression in the conclusion. However, some type systems do not have that property, so inducting on the derivation is a good habit.]

I.H.: If $\mathcal{D}_1$ derives $\emptyset \vdash e_1 : S_1$ and $\mathcal{D}_1 \prec \mathcal{D}$ ($\mathcal{D}_1$ is a subderivation of $\mathcal{D}$) and $e_1 \mapsto e_1'$ then $\emptyset \vdash e_1' : S_1$.

Consider cases of the rule concluding $\emptyset \vdash e : S$.

- type-assum:

$\emptyset \vdash e : S$	Given
$e = x$	By inversion on type-assum
$(x : S) \in \emptyset$	″
$(x : S) \in \emptyset$	Not derivable, so this case is impossible

- unitIntro:

$\emptyset \vdash e : S$	Given
$e = ()$	By inversion on rule unitIntro
$S = \text{unit}$	″
$e \mapsto e'$	Given
$() \mapsto e'$	By above equation
$() \mapsto e'$	Not derivable, so this case is impossible

- type-true, type-false, intIntro: impossible for reasons similar to unitIntro: based on what is known about e, the judgment $e \mapsto e'$ is not derivable.

- type-equals:

$$\begin{array}{rl}
\emptyset \vdash e : S & \text{Given} \\
e = (\texttt{=}\ e_1\ e_2) & \text{By inversion on type-equals} \\
S = \mathsf{bool} & '' \\
\emptyset \vdash e_1 : \mathsf{int} & '' \\
\emptyset \vdash e_2 : \mathsf{int} & '' \\
e \mapsto e' & \text{Given} \\
(\texttt{=}\ e_1\ e_2) \mapsto e' & \text{By above equation}
\end{array}$$

By inversion (rule step-context) on $(\texttt{=}\ e_1\ e_2) \mapsto e'$, there exist $\mathcal{C}$, e_0, e'_0 such that $(\texttt{=}\ e_1\ e_2) = \mathcal{C}[e_0]$ and $e' = \mathcal{C}[e'_0]$ and $e_0 \mapsto_R e'_0$.

Since $(\texttt{=}\ e_1\ e_2) = \mathcal{C}[e_0]$, there are three possible shapes of $\mathcal{C}$ based on the grammar:

1. $\mathcal{C} = []$

   $(\texttt{=}\ e_1\ e_2) \mapsto_R e'$

   $$\begin{array}{rl}
   e_1 = n_1 \text{ and } e_2 = n_2 & \text{By inversion on red-equals} \\
   e' = (n_1 = n_2) & ''
   \end{array}$$

   Either: $(n_1 = n_2) = \mathtt{True}$

   $$\begin{array}{ll}
   \emptyset \vdash \mathtt{True} : \mathsf{bool} & \text{By rule type-true} \\
   \emptyset \vdash e' : \mathsf{bool} & \text{By above equations } [e' = (n_1 = n_2) = \mathtt{True}]
   \end{array}$$

   Or: $(n_1 = n_2) = \mathtt{False}$

   $$\begin{array}{ll}
   \emptyset \vdash \mathtt{False} : \mathsf{bool} & \text{By rule type-false} \\
   \emptyset \vdash e' : \mathsf{bool} & \text{By above equations } [e' = (n_1 = n_2) = \mathtt{False}]
   \end{array}$$

2. $\mathcal{C} = (\texttt{=}\ \mathcal{C}_1\ e_2)$

   [To follow this case, it may be helpful to draw the syntax tree for e and draw $\mathcal{C}$ as a path from the root of e to e_0. Then $\mathcal{C}_1$ is the path from the root of e_1 to e_0.]

   Since $\mathcal{C}[e_0] = (\texttt{=}\ e_1\ e_2)$ and $\mathcal{C} = (\texttt{=}\ \mathcal{C}_1\ e_2)$, we have $e_1 = \mathcal{C}_1[e_0]$.

   $$\begin{array}{rl}
   \mathcal{C}[e_0] \mapsto \mathcal{C}[e'_0] & \text{Above} \\
   (\texttt{=}\ \mathcal{C}_1[e_0]\ e_2) \mapsto (\texttt{=}\ \mathcal{C}_1[e'_0]\ e_2) & \text{By above equation } [\mathcal{C} = \dots] \\
   e_0 \mapsto_R e'_0 & \text{Above} \\
   \mathcal{C}_1[e_0] \mapsto \mathcal{C}_1[e'_0] & \text{By step-context} \\
   e_1 \mapsto \mathcal{C}_1[e'_0] & \text{By above equation } e_1 = \mathcal{C}_1[e_0]
   \end{array}$$

   $$\begin{array}{ll}
   \emptyset \vdash e_1 : \mathsf{int} & \text{Above} \\
   \\
   \emptyset \vdash \mathcal{C}_1[e'_0] : \mathsf{int} & \text{By IH with } e_1 \text{ as } e_1 \text{ and } \mathcal{C}_1[e'_0] \text{ as } e'_1 \\
   \emptyset \vdash e_2 : \mathsf{int} & \text{Above} \\
   \\
   \emptyset \vdash (\texttt{=}\ \mathcal{C}_1[e'_0]\ e_2) : \mathsf{int} & \text{By type-equals} \\
   e' = \mathcal{C}[e'_0] & \text{Above} \\
   \quad = (\texttt{=}\ \mathcal{C}_1[e'_0]\ e_2) & \text{By above equation} \\
   \emptyset \vdash e' : \mathsf{int} & \text{By above equation}
   \end{array}$$

3. $\mathcal{C} = (= v_1\ \mathcal{C}_2)$, where $v_1 = e_1$
   Similar to subcase 2, with $e = (= v_1\ \mathcal{C}_2[e_0])$ and $e' = (= v_1\ \mathcal{C}_2[e_0'])$ and the IH on e_2 (which is $\mathcal{C}_2[e_0]$).

- type-ite:

$\emptyset \vdash e : S$	Given
$e \mapsto e'$	Given
$e = (\texttt{Ite}\ e_0\ e_1\ e_2)$	By inversion on type-ite
$\emptyset \vdash e_0 : \mathsf{bool}$	″
$\emptyset \vdash e_1 : S$	″
$\emptyset \vdash e_2 : S$	″
$(\texttt{Ite}\ e_0\ e_1\ e_2) \mapsto e'$	By above equation

Consider cases of $\mathcal{C}$:

1. $\mathcal{C} = [\,]$

   $(\texttt{Ite}\ e_0\ e_1\ e_2) \mapsto_R e'$ By inversion on step-context

   The above judgment could have been derived by either red-ite-then, or red-ite-else.

   – Above $\mapsto_R$ judgment was derived by red-ite-then:

$e_0 = \texttt{True}$	By inversion on rule red-ite-then
$e_1 = e'$	″
$\emptyset \vdash e_1 : S$	Above

   – Above $\mapsto_R$ judgment was derived by red-ite-else:

$e_0 = \texttt{False}$	By inversion on rule red-ite-then
$e_2 = e'$	″
$\emptyset \vdash e_2 : S$	Above

2. $\mathcal{C} = (\texttt{Ite}\ \mathcal{C}_1\ e_1\ e_2)$
   (I was persuaded to suddenly use f for expressions. This is temporary.)

	$e_0 = \mathcal{C}_1[f]$	By inversion on step-context
	$e = (\texttt{Ite}\ \mathcal{C}_1[f]\ e_1\ e_2)$	″
	$(\texttt{Ite}\ e_0\ e_1\ e_2) = (\texttt{Ite}\ \mathcal{C}_1\ e_1\ e_2)$	By above equation
	$f \mapsto_R f'$	″
	$\mathcal{C}_1[f] \mapsto \mathcal{C}_1[f']$	By step-context
	$e_0 \mapsto \mathcal{C}_1[f']$	By above equation
$\mathcal{D}_1$ derives	$\emptyset \vdash e_0 : \mathsf{bool}$	Above
	$\mathcal{D}_1$ is a subderivation of $\mathcal{D}$	
	$\emptyset \vdash \mathcal{C}_1[f'] : \mathsf{bool}$	By IH [with e_0 as e_1 and bool as S_1 and $\mathcal{C}_1[f']$ as e_1']
	$e' = (\texttt{Ite}\ \mathcal{C}_1[f']\ e_1\ e_2)$	By above equations $\mathcal{C} = (\texttt{Ite}\ \mathcal{C}_1\ e_1\ e_2)$
	$\emptyset \vdash \mathcal{C}_1[f'] : \mathsf{bool}$	Above
	$\emptyset \vdash e_1 : S$	Above
	$\emptyset \vdash e_2 : S$	Above
	$\emptyset \vdash (\texttt{Ite}\ \mathcal{C}_1[f']\ e_1\ e_2) : S$	By type-ite

- →Intro:

  Impossible.

- →Elim:

  [First, use inversion.]

eqn-a	$e = (\texttt{Call}\ e_1\ e_2)$	By inversion on rule →Elim
	$\emptyset \vdash e_1 : T \to S$	"
	$\emptyset \vdash e_2 : T$	"

  [Our goal is to show that e' has type S. Currently, we don't know anything about e'. We have used inversion on the given typing derivation we have, so we look to the second given derivation, of $e \mapsto e'$. Because there is only one rule, step-context, that can derive $\mapsto$ judgments, we can use inversion on that rule.]

$e \mapsto e'$	Given
$e = \mathcal{C}[e_0]$	By inversion on rule step-context
$e' = \mathcal{C}[e_0']$	"
$e_0 \mapsto_R e_0'$	"
$(\texttt{Call}\ e_1\ e_2) = \mathcal{C}[e_0]$	By above equation "eqn-a"

  [Since $e' = \mathcal{C}[e_0']$, we want to show that $\mathcal{C}[e_0']$ has type S. But we don't know what $\mathcal{C}$ is; there are three possible cases.]

  Consider cases of $\mathcal{C}$.

  1. $\mathcal{C} = []$:

$e = e_0$	By above equations
$e' = e_0'$	By above equations
$(\texttt{Call}\ e_1\ e_2) \mapsto_R e_0'$	By above equations

     [Whenever you learn something new about an expression, you should probably try using inversion. We learned $e_0 \mapsto_R e_0'$ a little while ago, but we couldn't use inversion because we knew nothing about e_0—we didn't know which reduction rule concluded $e_0 \mapsto_R e_0'$. Now we know that $e_0 = (\texttt{Call}\ e_1\ e_2)$.]

$(\texttt{Call}\ e_1\ e_2) \mapsto_R e_0'$	Above
$e_1 = (\texttt{Lam}\ x\ e_{\text{body}})$	By inversion on rule red-beta
$e_2 = v_2$	"
$e_0' = [v_2/x]e_{\text{body}}$	"

     Since we also have $e' = e_0'$, we now know $e' = [v_2/x]e_{\text{body}}$.
     So our goal is to show $\emptyset \vdash [v_2/x]e_{\text{body}} : S$.
     To get there, we need to do two things that we didn't need to do in previous cases. The first is to recall (way up above) that

     $$\emptyset \vdash e_1 : T \to S$$

Combined with $e_1 = (\texttt{Lam}\ x\ e_{body})$, we have

$$\emptyset \vdash (\texttt{Lam}\ x\ e_{body}) : T \rightarrow S$$

Having learned something about the e_1 in this judgment, this is a spot where we should try using inversion. Only one typing rule can derive $\ldots \vdash (\texttt{Lam}\ x\ e_{body}) : \ldots$, namely $\rightarrow$Intro.

$x : T \vdash e_{body} : S$ By inversion on rule $\rightarrow$Intro

But we still haven't reached our goal, because $x : T \vdash e_{body} : S$ talks about the expression e_{body}, not about $[v_2/x]e_{body}$. The second new thing is to use a *substitution lemma*.

2. $\mathcal{C} = (\texttt{Call}\ \mathcal{C}_1\ e_2)$:
   This case is similar to the $(= \mathcal{C}_1\ e_2)$ subcase of the type-equals case: in both, the reduction is inside the first subexpression. The reasoning is essentially the same, whether the first subexpression is inside an = or a Call.
3. $\mathcal{C} = (\texttt{Call}\ v_1\ \mathcal{C}_2)$:
   This case is also similar to the corresponding case for type-equals—which I didn't write out.

- type-add, type-sub, type-lt: Similar to the type-equals case.
- type-abs: Similar to the type-equals case, but somewhat easier because there's only one subexpression of $e = (\texttt{Abs}\ e_1)$.

■ **Exercise 1.** Do this case. There should be two subcases, one for $\mathcal{C} = []$ and one for $\mathcal{C} = (\texttt{Abs}\ \mathcal{C}_1)$.

- $+$Intro1:

$e = (\texttt{Inj}_1\ e_1)$	By inversion on rule $+$Intro1
$S_1 = (S_1 + S_2)$	″
$\emptyset \vdash e_1 : S_1$	″
$e \mapsto e'$	Given
$e = \mathcal{C}[e_0]$	By inversion on rule step-context
$e' = \mathcal{C}[e_0']$	″
$e_0 \mapsto_R e_0'$	″

As in some earlier cases, we need to think about what $\mathcal{C}$ is.

1. $\mathcal{C} = []$
   We have $e = (\texttt{Inj}_1\ e_1)$ and $e = \mathcal{C}[e_0]$ above, so if $\mathcal{C} = []$ then $e = e_0 = (\texttt{Inj}_1\ e_1)$ and we have

$$(\texttt{Inj}_1\ e_1) \mapsto_R e_0'$$

   Fortunately, there is no reduction rule that can derive this—an Inj by itself doesn't reduce. (It only reduces within a Case, similar to how a Lam only reduces within a Call.) So this subcase is impossible.

2. $\mathcal{C} = (\texttt{Inj}_1\ \mathcal{C}_1)$
   Similar to one of the subcases of the type-abs case.

- +Intro2: similar to the +Intro1 case.
- +Elim: ...
- ×Intro: ...
- ×Elim1: ...
- ×Elim2: ... □

[Following 3 paragraphs copied from a5]
For most languages, including ours, it is impossible to prove progress without first proving a lemma known as *canonical forms* or *value inversion*.

The first name, canonical forms, comes from the idea that the values of a given type—as opposed to expressions that are not values—are the original or canonical forms of that type. For example, while (+ 1 1) and (- 5 3) and (- (Abs −3) 1) are all *expressions* of type int—and, in a sense, represent the same integer 2 since they all eventually step to 2—we would not consider these expressions as defining the set of integers. But we can say that the *values* of type int—which are the integer constants n—define the integers.

The second name, value inversion, comes from the fact that the lemma uses inversion on a given derivation—but not the inversion we have often used, where we reason either from (a) knowing that we have an expression e of a particular form, say $(\texttt{Call}\ e_1\ e_2)$, or (b) knowing that the conclusion of a derivation is by some particular rule, say →Elim. Instead, the inversion is based on the combination of two facts:

- We know that the expression is a value.
- We know something about the expression's type.

Lemma 3 (Value Inversion)**.**

1. *If* $\emptyset \vdash v : \mathsf{unit}$ *then* $v = ()$.
2. *If* $\emptyset \vdash v : \mathsf{bool}$ *then either* $v = \texttt{True}$ *or* $v = \texttt{False}$.
3. *If* $\emptyset \vdash v : \mathsf{int}$ *then there exists* n *such that* $v = n$.
4. *If* $\emptyset \vdash v : (S_1 \times S_2)$
   then there exist v_1 *and* v_2 *such that* $v = (\texttt{Pair}\ v_1\ v_2)$ *and* $\emptyset \vdash v_1 : S_1$ *and* $\emptyset \vdash v_2 : S_2$.
5. *If* $\emptyset \vdash v : (S_1 \rightarrow S_2)$ *then there exist* x *and* e *such that* $v = (\texttt{Lam}\ x\ S_1\ e)$ *and* $x : S_1 \vdash e : S_2$.

Proof. [See assignment 5.] □

Conjecture 2 (Progress)**.**
For all e *and* S *such that* $\mathcal{D}$ *derives* $\emptyset \vdash e : S$,
either (1) e *is a value, or (2) there exists* e' *such that* $e \mapsto e'$.

Proof. By induction on the derivation of $\emptyset \vdash e : S$.

Induction hypothesis (IH): For all e_0 and S_0 such that $\mathcal{D}_0$ derives $\emptyset \vdash e_0 : S_0$ and $\mathcal{D}_0$ is a subderivation of $\mathcal{D}$, either (1) e_0 is a value, or (2) there exists e_0' such that $e_0 \mapsto e_0'$.

Consider cases of the rule concluding $\emptyset \vdash e : S$.

- type-assum: By inversion, we have (a) $e = x$ and (b) $(e : S) \in \emptyset$. But (b) is impossible, so this case is impossible.

- $\rightarrow$Intro: By inversion, $e = (\texttt{Lam}\ x\ e_{\text{body}})$. By the grammar of values, $(\texttt{Lam}\ x\ e_{\text{body}})$ is a value. Therefore e is a value, which is part (1) of our goal "either (1) e is a value, or (2) ...", so this case is done.

- unitIntro, type-true, type-false, intIntro: As in the $\rightarrow$Intro case, we know by inversion that e is a value, which is part (1) of the goal.

- $\rightarrow$Elim:

$e = (\texttt{Call}\ e_1\ e_2)$	By inversion on rule $\rightarrow$Elim
$\emptyset \vdash e_1 : (T \rightarrow S)$	"
$\emptyset \vdash e_2 : T$	"

  [Since we know that $e = (\texttt{Call}\ e_1\ e_2)$, which is not a value according to the grammar of values, we have no hope of proving part (1) of the goal: e is not a value. So we need to prove part (2): there exists some e' such that $e \mapsto e'$, that is, $(\texttt{Call}\ e_1\ e_2) \mapsto e'$.]

  [Inversion has carried us as far as it can. Fortunately, it has given us some smaller derivations, which means we are allowed to use the IH on them. In a proof, it's often helpful to use the IH "speculatively": you might not immediately see how the IH will bring you closer to the goal, but it often does. Speculatively or not, you should make sure you use the IH where it is allowed, that is, on smaller things. This proof is by induction, so we can use the IH on smaller derivations.]

$\emptyset \vdash e_1 : (T \rightarrow S)$	Above
either (e1.1) e_1 is a value, or	
(e1.2) $e_1 \mapsto e_1'$	By IH [with e_1 as e_0 and $T \rightarrow S$ as S_0 and e_1' as e_0']

  [We could have (e1.1), or (e1.2); we don't know which. So we have to consider both of those cases. The (e1.2) case turns out to be easier so I'll do it first; it doesn't matter in what order we write the cases.]

  - Subcase (e1.2): there exists e_1' such that $e_1 \mapsto e_1'$.

$e_1 \mapsto e_1'$	Above [given for case (e1.2)]
$e_1 = \mathcal{C}_1[e_3]$	By inversion on rule step-context
$e_1' = \mathcal{C}_1[e_3']$	"
$e_3 \mapsto_R e_3'$	"

    Let $\mathcal{C} = (\texttt{Call}\ \mathcal{C}_1\ e_2)$. [We need to apply rule step-context, so we need a $\mathcal{C}$. But we get to choose the $\mathcal{C}$.]

$e_3 \mapsto_R e_3'$	Above
$\mathcal{C}[e_3] \mapsto \mathcal{C}[e_3']$	By rule step-context
$\mathcal{C}_1[e_3] = e_1$	Above
$\mathcal{C}[e_3] = (\texttt{Call}\ e_1\ e_2)$	By above equations $\mathcal{C} = (\texttt{Call}\ \mathcal{C}_1\ e_2)$ and $\mathcal{C}_1[e_3] = e_1$
$\mathcal{C}[e_3] = e$	By above equation

Let $e' = \mathcal{C}[e_3']$. [We get to choose e': The statement we are trying to prove says: ... *there exists* e' *such that* $e \mapsto e'$. Now our goal is to prove $e \mapsto e'$.]

$\mathcal{C}[e_3] \mapsto \mathcal{C}[e_3']$	Above
$e \mapsto e'$	By above equations $\mathcal{C}[e_3] = e$ and $e' = \mathcal{C}[e_3']$

Goal (2) is $e \mapsto e'$, so we're done with this subcase.

– Subcase (e1.1): e_1 is a value.

[Unfortunately, this subcase is longer. We used the IH on the derivation for e_1; let's try using the IH on the derivation for e_2.]

$\emptyset \vdash e_2 : T$	Above
either (e2.1) e_2 is a value, or	
(e2.2) $e_2 \mapsto e_2'$	By IH [with e_2 as e_0 and T as S_0 and e_2' as e_0']

[Again we have to split into cases, because either (e2.1) or (e2.2), and we must handle both possibilities. Here, too, I choose to write the cases in the opposite order.]

* Sub-subcase (e2.2), inside subcase (e1.1): (e2.2) There exists e_2' such that $e_2 \mapsto e_2'$.

$e_2 \mapsto e_2'$	Above [given for case (e2.2)]
$e_2 = \mathcal{C}_2[e_4]$	By inversion on rule step-context
$e_2' = \mathcal{C}_2[e_4']$	″
$e_4 \mapsto_R e_4'$	″

Remainder left as an exercise: The idea is the same as subcase (e1.2): we need to reach goal (2), $e \mapsto e'$. To show that $e \mapsto e'$, we need to apply rule step-context. To apply rule step-context, we need to find an appropriate $\mathcal{C}$; having $\mathcal{C}_2$ helps (as having $\mathcal{C}_1$ helped in subcase (e1.2)).

* Sub-subcase (e2.1), inside subcase (e1.1): e_2 is a value.

[We are inside subcase (e1.1), so we know that e_1 is a value. We also know that e_2 is a value. Since $e = (\texttt{Call}\ e_1\ e_2)$—we got that way back at the beginning of the $\rightarrow$Elim case—we know $e = (\texttt{Call}\ v_1\ v_2)$. Our definition of evaluation contexts doesn't allow holes inside values, so trying to look inside v_1 or v_2 for a $[]$ isn't going to work. Instead, we will need to use step-context with $\mathcal{C} = []$: we need to show that *entire expression* e reduces, that is, we need to show $(\texttt{Call}\ v_1\ v_2) \mapsto_R e'$. The only rule that can potentially derive that is red-beta, which requires that v_1 have the form `Lam`.]

$\emptyset \vdash e_1 : (T \rightarrow S)$	Above
e_1 is a value, that is, $e_1 = v_1$	Above (e1.1)
e_2 is a value, that is, $e_2 = v_2$	Above (e2.1)
$e_1 = (\texttt{Lam}\ x\ e_{body})$	By Lemma 3 (Value Inversion), part 5
$x : T \vdash e_{body} : S$	″

$$\begin{array}{rcll} \texttt{(Call (Lam } x\ e_{\text{body}}\texttt{)}\ v_2\texttt{)} & \mapsto_{\mathsf{R}} & [v_2/x]e_{\text{body}} & \text{By red-beta} \\ \texttt{(Call (Lam } x\ e_{\text{body}}\texttt{)}\ v_2\texttt{)} & \mapsto & [v_2/x]e_{\text{body}} & \text{By step-context [with } \mathcal{C} = [\,]] \end{array}$$

$$\texttt{(Call (Lam } x\ e_{\text{body}}\texttt{)}\ v_2\texttt{)} = e \qquad \text{By above equations}$$

Let $e' = [v_2/x]e_{\text{body}}$.

$e \mapsto e'$ By above equations $e = \texttt{(Call (Lam } x\ e_{\text{body}}\texttt{)}\ v_2\texttt{)}$ and $e' = [v_2/x]e_{\text{body}}$

Goal (2) is $e \mapsto e'$, so we're done with this sub-subcase.

- type-add:
- type-sub:
- type-abs:
- type-equals:
- type-lt:
- type-ite:
- $+$Intro1:
- $+$Intro2:
- $+$Elim:
- $\times$Intro:
- $\times$Elim1:
- $\times$Elim2:

□