1 Types

The language we are giving semantics to is functional; if you don’t have some background in a functional language, understanding what certain features mean—especially types—will take time. Even if you do have such background (for example, Haskell from CISC 260/360), some language features are “distilled”: describing their realistic form (e.g. Haskell data declarations) is more work than describing their simplified form.

The language features we will add are:

- A “unit” expression (), which is the only value of type unit. (In Haskell, both the value and the type are written ()
- Pairs (Pair e₁ e₂); the components of the pair can be extracted with (Proj₁ e) and (Proj₂ e). For example, (Proj₂ (Pair 3 4)) should step to 4. (In Haskell, pairs are written (e₁, e₂). The components can be extracted using fst and snd, but Haskell programmers tend to use pattern matching instead. Pattern matching is a little too complicated to describe here.)
- Sums. These provide one instance of Haskell data declarations, namely, the Either type. Roughly, the type int + bool includes both integers and booleans. However, an integer by itself—say 3—does not have type int + bool; the integer must be “injected” into the sum, by writing
  \((\text{Inj}_1 3)\)
  Similarly, a Boolean by itself must be injected by writing
  \((\text{Inj}_2 \text{False})\)
  Sums are “eliminated” (this may make more sense once we see the typing rule) by
  \((\text{Case } e (x_1 \Rightarrow e_1) (x_2 \Rightarrow e_2))\)
  For example,
  \((\text{Case } (\text{Inj}_1 3) (x_1 \Rightarrow (+ x_1 x_1)) (x_2 \Rightarrow 0))\)
  should step to \([3/x_1](+ x_1 x_1) = (+ 3 3)\).
§1 Types

2 Typing for Lλ

Expressions $e ::= ()$
   $| n | (+ e e) | (- e e) | (Abs e)$
   $| True | False | (Ite e e e)$
   $| (= e e) | (< e e)$
   $| x | (Lam x e) | (Call e e)$
   $| (Pair e e) | (Proj_1 e) | (Proj_2 e)$
   $| (Inj_1 e) | (Inj_2 e) | (Case e (x => e) (x => e))$

Values $v ::= ()$
   $| n$
   $| True | False$
   $| x | (Lam x e)$
   $| (Pair v v)$
   $| (Inj_1 v) | (Inj_2 v)$

Types $S, T ::= unit$ unit type
   $| int$ type of integers
   $| bool$ type of booleans
   $| S \rightarrow T$ type of functions on $S$ that produce $T$
   $| S \times T$ type of pairs of one $S$ and one $T$
   $| S + T$ disjoint union or sum type: contains either an $S$ or a $T$

Typing contexts $\Gamma ::= \emptyset$ empty context
   $| \Gamma, x : S$ $x$ has type $S$
§2 Typing for $\text{L}\lambda$

$\Gamma \vdash e : T$ Under assumptions $\Gamma$, expression $e$ has type $T$

$\Gamma \vdash (\text{Lam} \; x \; e) : (S \rightarrow T)$ →Intro

$\Gamma \vdash \text{Call}_1 \; e_1 \; e_2 : T$ →Elim

$\Gamma \vdash \langle \text{Abs} \; e_1 \rangle : \text{int}$ type-abs

$\Gamma \vdash \langle \text{Inj}_1 \; e_1 \rangle : (S_1 + S_2)$ +Intro1

$\Gamma \vdash \langle \text{Inj}_2 \; e_2 \rangle : (S_1 + S_2)$ +Intro2

$\Gamma \vdash \langle \text{Case} \; e \; (x_1 \Rightarrow e_1) \; (x_2 \Rightarrow e_2) \rangle : T$ +Elim

$\Gamma \vdash \langle \text{Pair} \; e_1 \; e_2 \rangle : (S_1 \times S_2)$ ×Intro

$\Gamma \vdash \langle \text{Proj}_1 \; e \rangle : S_1$ ×Elim1

$\Gamma \vdash \langle \text{Proj}_2 \; e \rangle : S_2$ ×Elim2

Figure 1 Typing with functions, integers, booleans, sums (unions), and pairs (structs)
### Small-step semantics for Lλ

**Contexts**

\[ C ::= [] \ nakedright{\ |} (\ + C e) \ nakedright{\ |} (\ - C e) \ nakedright{\ |} (Abs C) \ nakedright{\ |} (Ite C e e) \ nakedright{\ |} (\ = C e) \ nakedright{\ |} (\ < C e) \ nakedright{\ |} (\ Call C e) \ nakedright{\ |} (\ Pair C e) \ nakedright{\ |} (Abs C) \ nakedright{\ |} (Ite C e e) \ nakedright{\ |} (\ = C e) \ nakedright{\ |} (\ < C e) \ nakedright{\ |} (\ Call C e) \ nakedright{\ |} (\ Pair C e) \ nakedright{\ |} (Abs C) \]

**Expression** \( e \rightarrow_R e' \)

- **Expression** \( e \rightarrow_R e' \) reduces to \( e' \)

  \[
  \begin{align*}
  (\ + n_1 n_2) & \rightarrow_R (n_1 + n_2) & \text{red-add} \\
  (\ - n_1 n_2) & \rightarrow_R (n_1 - n_2) & \text{red-sub} \\
  (Abs n) & \rightarrow_R |n| & \text{red-abs} \\
  (n_1 n_2) & \rightarrow_R (n_1 = n_2) & \text{red-equals} \\
  (n_1 n_2) & \rightarrow_R (n_1 < n_2) & \text{red-lessthan} \\
  (Ite True e_{\text{then}} e_{\text{else}}) & \rightarrow_R e_{\text{then}} & \text{red-ite-then} \\
  (Ite False e_{\text{then}} e_{\text{else}}) & \rightarrow_R e_{\text{else}} & \text{red-ite-else} \\
  (Call (Lam x e) v) & \rightarrow_R [v/x]e & \text{red-beta} \\
  (Proj_1 (Pair v_1 v_2)) & \rightarrow_R v_1 & \text{red-proj1} \\
  (Proj_2 (Pair v_1 v_2)) & \rightarrow_R v_2 & \text{red-proj2} \\
  (Case (Inj_1 v_1) (x_1 \rightarrow e_1) (x_2 \rightarrow e_2)) & \rightarrow_R [v_1/x_1]e_1 & \text{red-case1} \\
  (Case (Inj_2 v_2) (x_1 \rightarrow e_1) (x_2 \rightarrow e_2)) & \rightarrow_R [v_2/x_2]e_2 & \text{red-case2}
  \end{align*}
  
\]

**Expression** \( e \rightarrow e' \) takes one step to \( e' \)

\[
\frac{e \rightarrow_R e'}{C[e] \rightarrow C[e'] \quad \text{step-context}}
\]
§3 Small-step semantics for $\lambda$

3.1 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 $\rightarrow$Intro. That case comes up when a $\text{Lam}$ is the body of a $\text{Lam}$.)

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 $\text{Call}$ 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 $D_1$ derives $\emptyset \vdash e_1 : S_1$ and $D_1 \prec D$ ($D_1$ is a subderivation of $D$) 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:
  
  $\emptyset \vdash e : S$  
  
  $e = x$  
  
  $(x : S) \in \emptyset$  
  
  $(x : S) \in \emptyset$  
  
  Not derivable, so this case is impossible

- unitIntro:
  
  $\emptyset \vdash e : S$  
  
  $e = ()$  
  
  By inversion on rule unitIntro  
  
  $S = \text{unit}$  
  
  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.
\[\begin{array}{ll}

type-equals: & \\
\emptyset \vdash e : S & \text{Given} \\
e = ( = e_1 e_2 ) & \text{By inversion on type-equals} \\
n \vdash e_1 : \text{int} & " \\
n \vdash e_2 : \text{int} & " \\
e \rightarrow e' & \text{Given} \\
( = e_1 e_2 ) \rightarrow e' & \text{By above equation} \\
\end{array}\]

By inversion (rule step-context) on \(( = e_1 e_2 ) \rightarrow e'\), there exist \(C, e_0, e_0'\) such that \(( = e_1 e_2 ) = C[e_0] \) and \(e' = C[e_0']\) and \(e_0 \rightarrow_R e_0'\).

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

1. \(C = []\)

\(\begin{array}{ll}
\vdash e_1 : \text{int} & \text{By rule type-int} \\
\vdash e_2 : \text{int} & \text{By above equations} [e' = (n_1 = n_2)] \\
\vdash e_0 : \text{int} & \text{By IH with } e_1 \text{ as } e_1 \text{ and } C[e_0'] \text{ as } e_0' \\
\vdash e' : \text{bool} & \text{By type-equals} \\
\end{array}\)

2. \(C = (C_1 e_2)\)

\[\begin{array}{ll}
(C_1 e_0) \rightarrow (C_1 e_0') & \text{By step-context} \\
( = C_1[e_0] e_2 ) \rightarrow ( = C_1[e_0'] e_2 ) & \text{By above equation } [C = \ldots ] \\
\end{array}\]
3. $C = (\equiv v_1 C_2)$, where $v_1 = e_1$

Similar to subcase 2, with $e = (\equiv v_1 C_2[e_0])$ and $e' = (\equiv v_1 C_2[e_0'])$ and the IH on $e_2$ (which is $C_2[e_0]$).

- **type-ite:**

$$
\begin{align*}
\varnothing \vdash e : S & \quad \text{Given} \\
 e \mapsto e' & \quad \text{Given} \\
 e = (\text{Ite} \: e_0 \: e_1 \: e_2) & \quad \text{By inversion on type-ite} \\
 \varnothing \vdash e_0 : \text{bool} & \quad "" \\
 \varnothing \vdash e_1 : S & \quad "" \\
 \varnothing \vdash e_2 : S & \quad ""
\end{align*}
$$

$(\text{Ite} \: e_0 \: e_1 \: e_2) \mapsto e'$  

By above equation

Consider cases of $C$:

1. $C = []$

$(\text{Ite} \: e_0 \: e_1 \: e_2) \mapsto_R e'$  

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:

$e_0 = \text{True}$  

By inversion on rule red-ite-then

$e_1 = e'$  

""

$\varnothing \vdash e_1 : S$  

Above

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

$e_0 = \text{False}$  

By inversion on rule red-ite-then

$e_2 = e'$  

""

$\varnothing \vdash e_2 : S$  

Above

2. $C = (\text{Ite} \: C_1 \: e_1 \: e_2)$

(I was persuaded to suddenly use $f$ for expressions. This is temporary.)

$$
\begin{align*}
e_0 = C_1[f] & \quad \text{By inversion on step-context} \\
e = (\text{Ite} \: C_1[f] \: e_1 \: e_2) & \quad "" \\
(\text{Ite} \: e_0 \: e_1 \: e_2) = (\text{Ite} \: C_1 \: e_1 \: e_2) & \quad \text{By above equation} \\
f \mapsto_R f' & \quad "" \\
C_1[f] \mapsto C_1[f'] & \quad \text{By step-context} \\
e_0 \mapsto C_1[f'] & \quad \text{By above equation}
\end{align*}
$$

$D_1$ derives

$\varnothing \vdash e_0 : \text{bool}$  

Above

$D_1$ is a subderivation of $D$

$\varnothing \vdash C_1[f'] : \text{bool}$  

By IH [with $e_0$ as $e_1$ and bool as $S_1$ and $C_1[f']$ as $e_1'$]

$e' = (\text{Ite} \: C_1[f'] \: e_1 \: e_2)$  

By above equations $C = (\text{Ite} \: C_1 \: e_1 \: e_2)$

$\varnothing \vdash C_1[f'] : \text{bool}$  

Above

$\varnothing \vdash e_1 : S$  

Above

$\varnothing \vdash e_2 : S$  

Above

$\varnothing \vdash (\text{Ite} \: C_1[f'] \: e_1 \: e_2) : S$  

By type-ite
§3 Small-step semantics for L\(\lambda\)

- \(\rightarrow\)Intro:
  Impossible.
- \(\rightarrow\)Elim:
  [First, use inversion.]

\[
\text{eqn-a } e = \text{Call } e_1 e_2 \quad \text{By inversion on rule } \rightarrow\text{Elim}
\]
\[
\emptyset \vdash e_1 : T \rightarrow S \quad ''
\]
\[
\emptyset \vdash e_2 : T \quad ''
\]

[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.]

\[
e \mapsto e' \quad \text{Given}
\]
\[
e = C[e_0] \quad \text{By inversion on rule step-context}
\]
\[
e' = C[e_0'] \quad ''
\]
\[
e_0 \mapsto_R e_0' \quad ''
\]

\((\text{Call } e_1 e_2) = C[e_0]\) \quad \text{By above equation “eqn-a”}

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

Consider cases of \(C\).

1. \(C = []\):

\[
e = e_0 \quad \text{By above equations}
\]
\[
e' = e_0' \quad \text{By above equations}
\]
\[
(\text{Call } e_1 e_2) \mapsto_R e_0' \quad \text{By above equations}
\]

[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 \(e_0 = (\text{Call } e_1 e_2)\).]

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

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

\[
\emptyset \vdash e_1 : T \rightarrow S
\]
§3 Small-step semantics for L

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

\[ \emptyset \vdash (\text{Lam } x e_{\text{body}}) : T \rightarrow S \]

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 \( \ldots \vdash (\text{Lam } x e_{\text{body}}) : \ldots \), namely \( \rightarrow \text{Intro} \).

\[ x : T \vdash e_{\text{body}} : S \quad \text{By inversion on rule } \rightarrow \text{Intro} \]

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

2. \( C = (\text{Call } C_1 e_2) \):

   This case is similar to the \( (= C_1 e_2) \) 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 \( \text{Call} \).

3. \( C = (\text{Call } v_1 C_2) \):

   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 \( e = (\text{Abs } e_1) \).

Exercise 1. Do this case. There should be two subcases, one for \( C = [\ ] \) and one for \( C = (\text{Abs } C_1) \).

- \( +\text{Intro1} \):

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

As in some earlier cases, we need to think about what \( C \) is.

1. \( C = [\ ] \)

   We have \( e = (\text{Inj}_1 e_1) \) and \( e = C[e_0] \) above, so if \( C = [\ ] \) then \( e = e_0 = (\text{Inj}_1 e_1) \) and we have

   \[ (\text{Inj}_1 e_1) \mapsto_R e_0' \]

   Fortunately, there is no reduction rule that can derive this—an \( \text{Inj} \) by itself doesn’t reduce. (It only reduces within a \( \text{Case} \), similar to how a \( \text{Lam} \) only reduces within a \( \text{Call} \).)

   So this subcase is impossible.
§3 Small-step semantics for L\(\lambda\)

2. \(C = (\text{Inj}_1 C_1)\)
   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 \((-\ \text{Abs}\ -3\ 1)\) 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 \((\text{Call} \ e_1 \ e_2)\), or (b) knowing that the conclusion of a derivation is by some particular rule, say \(-\text{Elim}\). 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 \(\emptyset \vdash v : \text{unit}\) then \(v = ()\).

2. If \(\emptyset \vdash v : \text{bool}\) then either \(v = \text{True}\) or \(v = \text{False}\).

3. If \(\emptyset \vdash v : \text{int}\) 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 \(v = (\text{Pair} v_1 \ v_2)\) 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 \(v = (\text{Lam} \ x \ e)\) and \(x : S_1 \vdash e : S_2\).

6. If \(\emptyset \vdash v : (S_1 + S_2)\) then either (1) there exists \(v_1\) such that \(v = (\text{Inj}_1 \ v_1)\) and \(\emptyset \vdash v_1 : S_1\) or (2) there exists \(v_2\) such that \(v = (\text{Inj}_2 \ v_2)\) and \(\emptyset \vdash v_2 : S_2\).

Proof. [See assignment 5.]
Conjecture 2 (Progress).
For all e and S such that D derives ∅ ⊢ e : S,
either (1) e is a value, or (2) there exists e’ such that e → e’.

Proof. By induction on the derivation of ∅ ⊢ e : S.

Induction hypothesis (IH): For all e_0 and S_0 such that D_0 derives ∅ ⊢ e_0 : S_0 and D_0 is a subderivation of D, either (1) e_0 is a value, or (2) there exists e_0’ such that e_0 → e_0’.

Consider cases of the rule concluding ∅ ⊢ e : S.

• type-assum: By inversion, we have (a) e = x and (b) (e : S) ∈ ∅. But (b) is impossible, so this case is impossible.

• →Intro: By inversion, e = (Lam x e_0). By the grammar of values, (Lam x e_0) 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 →Intro case, we know by inversion that e is a value, which is part (1) of the goal.

• →Elim:
  e = (Call e_1 e_2)  By inversion on rule →Elim
  ∅ ⊢ e_1 : (T → S) "
  ∅ ⊢ e_2 : T "

[Since we know that e = (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 → e’, that is, (Call e_1 e_2) → 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.]

∅ ⊢ e_1 : (T → S)  Above
  either (e1.1) e_1 is a value, or
  (e1.2) e_1 → e_1’  By IH [with e_1 as e_0 and T → 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 → e_1’.
  e_1 → e_1’  Above [given for case (e1.2)]
  e_1 = C_1[e_3]  By inversion on rule step-context
  e_1’ = C_1[e_3’] "
  e_3 →_R e_3’ "
Let $C = \langle \text{Call } C_1 e_2 \rangle$. [We need to apply rule step-context, so we need a $C$. But we get to choose the $C$.]

\[
e_3 \mapsto_R e'_3 \quad \text{Above}
\]

$C[e_3] \mapsto C'[e'_3]$ By rule step-context

$C_1[e_3] = e_1$ Above

$C[e_3] = (\text{Call } e_1 e_2)$ By above equations $C = \langle \text{Call } C_1 e_2 \rangle$ and $C_1[e_3] = e_1$

$C[e_3] = e$ By above equation

Let $e' = C[e'_3]$. [We get to choose $e'$: The statement we are trying to prove says: \ldots there exists $e'$ such that $e \mapsto e'$. Now our goal is to prove $e \mapsto e'$.]

\[
C[e_3] \mapsto C[e'_3] \quad \text{Above}
\]

\[
e \mapsto e' \quad \text{By above equations } C[e_3] = e \text{ and } e' = C[e'_3]
\]

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 $e_2$.]

\[
\emptyset \vdash e_2 : T \quad \text{Above}
\]

either (e2.1) $e_2$ is a value, or

\[
(e2.2) \quad e_2 \mapsto e'_2 \quad \text{By IH [with } e_2 \text{ as } e_0 \text{ and } T \text{ as } S_0 \text{ and } e'_2 \text{ 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 $e_2 \mapsto e'_2$.

\[
e_2 \mapsto e'_2 \quad \text{Above [given for case (e2.2)]}
\]

$e_2 = C_2[e_4]$ By inversion on rule step-context

\[
e'_2 = C_2[e'_4] \quad \text{''}
\]

\[
e_4 \mapsto_R e'_4 \quad \text{''}
\]

**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 $C$; having $C_2$ helps (as having $C_1$ 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 $e = \langle \text{Call } e_1 e_2 \rangle$—we got that way back at the beginning of the \rightarrow Elim case—we know $e = \langle \text{Call } v_1 v_2 \rangle$. 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 $C = [\text{}]:$ we need to show that entire expression $e$ reduces, that is, we need to show $\langle \text{Call } v_1 v_2 \rangle \mapsto_R e'$. The only rule that can potentially derive that is red-beta, which requires that $v_1$ have the form Lam. ]
§3 Small-step semantics for $\text{L}$

\[ \emptyset \vdash e_1 : (T \rightarrow S) \quad \text{Above} \]
\[ e_1 \text{ is a value, that is, } e_1 = v_1 \quad \text{Above (e1.1)} \]
\[ e_2 \text{ is a value, that is, } e_2 = v_2 \quad \text{Above (e2.1)} \]
\[ e_1 = (\text{Lam } x \ e_{\text{body}}) \quad \text{By Lemma 3 (Value Inversion), part 5} \]
\[ x : T \vdash e_{\text{body}} : S \quad " " \]
\[ (\text{Call } (\text{Lam } x \ e_{\text{body}}) \ v_2) \mapsto_R [v_2/x]e_{\text{body}} \quad \text{By red-beta} \]
\[ (\text{Call } (\text{Lam } x \ e_{\text{body}}) \ v_2) \mapsto [v_2/x]e_{\text{body}} \quad \text{By step-context [with } C = []]) \]
\[ (\text{Call } (\text{Lam } x \ e_{\text{body}}) \ v_2) = e \quad \text{By above equations} \]

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

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

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