Computation and Deduction 2016

Mini-project Report

Student
Dandan Xue

Supervisor
Andrzej Filinski

November 9, 2016

Contents

1 Introduction

We present a language Mini-List by showing its syntax and operational semantics. Then we introduce an equational system to this language. Finally, we show that this equational system is sound and complete for evaluation.

2 The Mini-List Language

2.1 Syntax

The syntax of Mini-List expressions e is as follows:

$$e ::= \mathbf{z} \mid \mathbf{s}\ e \mid \mathbf{nil} \mid e_1 : e_2 \mid e_1 +\!+ e_2 \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid x$$

nil stands for the empty list and $e_1 : e_2$ for a list with the first element e_1 and the others e_2. The operator $+\!+$ denotes the appending operation at the end of the list e_1 with the list e_2.

The syntax of values v is as follows:

$$v ::= \mathbf{z} \mid \mathbf{s}\ v \mid \mathbf{nil} \mid v_1 : v_2$$

To keep the paper proof simple and clear, we consider values are also expressions in Mini-List.

2.2 Type system

The types for Mini-List is given by the following grammar:

$$\tau ::= \mathbf{nat} \mid \mathbf{list}\ \tau_1$$

Here **nat** stands for the type of natural numbers, **list** τ_1 is the type of a list of elements that has the type τ_1.

We introduce *typing context* Γ to present the types of variables in the context:

$$\Gamma = [x_1 : \tau_1, ..., x_n : \tau_n]$$

where all the x_i are distinct. The judgment $\Gamma \vdash e : \tau$ means that in context Γ, e has type τ. We define the typing rules for this judgment in Figure 1.

Judgment $\boxed{\Gamma \vdash e : \tau}$:

T-Var: $\dfrac{}{\Gamma \vdash x : \tau}$ T-Z: $\dfrac{}{\Gamma \vdash \mathbf{z} : \mathbf{nat}}$ T-S: $\dfrac{e : \mathbf{nat}}{\Gamma \vdash \mathbf{s}\ e : \mathbf{nat}}$

T-Nil: $\dfrac{}{\Gamma \vdash \mathbf{nil} : \mathbf{list}\ \tau}$ T-Cons: $\dfrac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \mathbf{list}\ \tau}{\Gamma \vdash e_1 : e_2 : \mathbf{list}\ \tau}$

T-Append: $\dfrac{\Gamma \vdash e_1 : \mathbf{list}\ \tau \qquad \Gamma \vdash e_2 : \mathbf{list}\ \tau}{\Gamma \vdash e_1 : e_2 : \mathbf{list}\ \tau}$

T-Let: $\dfrac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x : \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2}$

Figure 1: Type system for Mini-List

2.3 Operational semantics

We define the evaluation context Δ to denote the evaluations of the variables in the context:

$$\Delta = [x_1 \downarrow v_1, ..., x_n \downarrow v_n]$$

where all the x_i are distinct.

The expression e that has typing context $\Gamma[x_i : \tau_i] \vdash e : \tau$ will evaluate to $\Delta[x_i \downarrow v_i] \vdash e \downarrow v$ for some $v : \tau$ and $v_i : \tau_i$. The evaluation rules appears in Figure 2.

Judgment $\boxed{\Delta \vdash e \downarrow v}$:

EVAL-VAR: $\dfrac{}{\Delta \vdash x \downarrow v}$ EVAL-Z : $\dfrac{}{\Delta \vdash \mathbf{z} \downarrow \mathbf{z}}$ EVAL-S : $\dfrac{\Delta \vdash e \downarrow v}{\Delta \vdash \mathbf{s}\ e \downarrow \mathbf{s}\ v}$

EVAL-NIL : $\dfrac{}{\Delta \vdash \mathbf{nil} \downarrow \mathbf{nil}}$ EVAL-CONS : $\dfrac{\Delta \vdash e_1 \downarrow v_1 \qquad \Delta \vdash e_2 \downarrow v_2}{\Delta \vdash e_1 : e_2 \downarrow v_1 : v_2}$

EVAL-APPEND : $\dfrac{\Delta \vdash e_1 \downarrow v_1 \qquad \Delta \vdash e_2 \downarrow v_2 \qquad \mathbf{fappend}\ (v_1, v_2, v)}{\Delta \vdash e_1 +\!+ e_2 \downarrow v}$

EVAL-LET : $\dfrac{\Delta \vdash e_1 \downarrow v_1 \qquad \Delta[x \downarrow v_1] \vdash e_2 \downarrow v}{\Delta \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \downarrow v}$

Figure 2: Operational Semantics of Mini-List

The function **fappend** (v_1, v_2, v) in EVAL-APPEND denotes the operation that appending the list v_2 to the end of the list v_1 obtains a new list v. We show its definition in the following figure.

Judgment $\boxed{\mathbf{fappend}\ (v_1, v_2, v)}$:

FAPPEND-NIL: $\dfrac{}{\mathbf{fappend}\ (\mathbf{nil}\ , v, v)}$ FAPPEND-CONS: $\dfrac{\mathbf{fappend}\ (v_2, v_3, v)}{\mathbf{fappend}\ ((v_1 : v_2), v_3, (v_1 : v))}$

Figure 3: The operation **fappend**

We can easily check that **fappend** has the following properties:

Lemma 1 (Fappend Determinacy)**.** *If* **fappend** (v_1, v_2, v_3)*, and* **fappend** (v_1, v_2, v_3')*, then* $v_3 = v_3'$*.*

Lemma 2 (Fappend Existance)**.** *If there ars two lists* v_1 *and* v_2*, then there must exist some list* v_3 *such that* **fappend** (v_1, v_2, v_3)*.*

Lemma 3 (Fappend Nil Right)**.** *For any list* v*, there must be* **fappend** $(v, \mathbf{nil}\ , v)$*.*

Lemma 4 (Fappend Associativity)**.** *If* **fappend** (v_1, v_2, v_3) *and* **fappend** (v_3, v_4, v_5)*, then* **fappend** (v_2, v_3, v_6) *and* **fappend** (v_1, v_6, v_5) *for some list* v_6*.*

Now with Lemma 2 and Lemma 1 we can easily check that any well-typed expression can be evaluated to some value and the evalution is deterministic.

Theorem 5 (Evaluation Existence)**.** *For any* e *that has type* $\Gamma[x_i : \tau_i] \vdash e : \tau$*, and* $\Delta[x_i \downarrow v_i]$ *for some* $v_i : \tau_i$*, then there must exist a derivation of* $\Delta[x_i \downarrow v_i] \vdash e \downarrow v$ *for some* $v : \tau$*.*

Theorem 6 (Evalutation Determinacy)**.** *For any evaluation* $\Delta[x \downarrow v_2] \vdash e \downarrow v_1$*, if there is another evalution* $\Delta[x \downarrow v_2'] \vdash e \downarrow v_1'$ *and* $v_2 = v_2'$*, then* $v_1 = v_1'$*.*

2.4 Equational system in Mini-List

We use the symbol "$\sim$" to denote that two expressions in Mini-List are *provably equal* or *convertible*. Two expressions e and e' that have the same type τ are considered *convertible* if they can be transformed to each other by the rules in Figure 4.

Judgement $\boxed{e \sim e'}$:

Eq-Ref : $\dfrac{}{e \sim e}$ Eq-Sym : $\dfrac{e' \sim e}{e \sim e'}$ Eq-Trans : $\dfrac{e_1 \sim e' \qquad e' \sim e_2}{e_1 \sim e_2}$

Eq-S : $\dfrac{e \sim e'}{\mathbf{s}\ e \sim \mathbf{s}\ e'}$ Eq-Cons : $\dfrac{e_1 \sim e_1' \qquad e_2 \sim e_2'}{e_1 : e_2 \sim e_1' : e_2'}$ Eq-Append : $\dfrac{e_1 \sim e_1' \qquad e_2 \sim e_2'}{e_1 +\!+ e_2 \sim e_1' +\!+ e_2'}$

Eq-Append-Nil-L : $\dfrac{}{\mathbf{nil} +\!+ e \sim e}$ Eq-Append-Nil-R : $\dfrac{}{e +\!+ \mathbf{nil} \sim e}$

Eq-Append-Cons : $\dfrac{}{(e_1 : e_2) +\!+ e_3 \sim e_1 : (e_2 +\!+ e_3)}$

Eq-Append-Asso : $\dfrac{}{(e_1 +\!+ e_2) +\!+ e_3 \sim e_1 +\!+ (e_2 +\!+ e_3)}$

Eq-Let : $\dfrac{e_1 \sim e_1' \qquad e_2 \sim e_2'}{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \sim \mathbf{let}\ x = e_1'\ \mathbf{in}\ e_2'}$

Eq-Let-Subst : $\dfrac{}{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \sim e_2[e_1/x]}$

Figure 4: Equational system of Mini-List

In the rule Eq-Let-Subst, $e_2[e_1/x]$ means we substitute e_1 for x in e_2.

3 Equation Soundness

We introduce the symbol "==" to denote that two expressions e and e' are *semantically equal*, i.e., $\mathbf{e == e'}$. Two expressions are semantically equal iff they evaluate to the same value.

Definition 7 (Semantical Equivalence $\Rightarrow$)**.** *For any two expressions e and e', if $\Delta \vdash e \downarrow v$ for some v, and $\Delta \vdash e' \downarrow v$, then $e == e'$.*

Definition 8 (Semantical Equivalence $\Leftarrow$)**.** *For any two expressions e and e', if $e == e'$ and $\Delta \vdash e \downarrow v$ for some v, then $\Delta \vdash e' \downarrow v$.*

Now we shall show that if two expressions are provably equal, then they are also semantically equal.

Theorem 9 (Equation Soundness)**.** *If $e \sim e'$, then $e == e'$.*

Proving this theorem is equivalent to prove the following two lemmas by Definition 7.

Lemma 10 (Equation Soundness Part 1)**.** *For any two expressions $\Gamma[x : \tau_2] \vdash e : \tau_1$, and $\Gamma[x : \tau_2] \vdash e' : \tau_1$, if $e \sim e'$, and $\Delta[x \downarrow v_2] \vdash e \downarrow v_1$, then $\Delta[x \downarrow v_2] \vdash e' \downarrow v_1$.*

Lemma 11 (Equation Soundness Part 2)**.** *For any two expressions $\Gamma[x : \tau_2] \vdash e' : \tau_1$, $\Gamma[x : \tau_2] \vdash e : \tau_1$, if $e \sim e'$, and $\Delta[x \downarrow v_2] \vdash e' \downarrow v_1$, then $\Delta[x \downarrow v_2] \vdash e \downarrow v_1$.*

We only need to prove one of these lemmas since the other one will also be provable by the symmetry rule of the equational system.

We now prove Equation Soundness Part 1.

Proof. Let $\mathcal{Q}$ be the derivation of $e \sim e'$, $\mathcal{E}$ of $\Delta[x \downarrow v_2] \vdash e \downarrow v$. We shall show that there is a deriation $\mathcal{E}'$ of $\Delta[x \downarrow v_2] \vdash e' \downarrow v$. By induction on $\mathcal{Q}$:

- Case $\mathcal{Q} = \dfrac{}{e \sim e}$, so $e' = e$. We immediately obtain that $\mathcal{E}'$ is the same with $\mathcal{E}$.

- Case $\mathcal{Q} = \dfrac{\begin{array}{c}\mathcal{Q}_0\\ e' \sim e\end{array}}{e \sim e'}$.

  By Theroem 5 Evalution Existence, there exists a derivation $\Delta \vdash \mathcal{E}_0$ of $e' \downarrow v'$. By IH on $\mathcal{Q}_0$ with $\mathcal{E}_0$, we get a derivation $\mathcal{E}_0'$ of $\Delta \vdash e \downarrow v'$.
  By the determinacy of evalutaion on $\mathcal{E}_0'$ with $\mathcal{E}$, we obtain $v' = v$.
  Then we obtain $\mathcal{E}'$ by replacing v' with v in $\mathcal{E}_0$ as required.

- Case $\mathcal{Q} = \dfrac{\begin{array}{cc}\mathcal{Q}_1 & \mathcal{Q}_2\\ e_1 \sim e_1' & e_1' \sim e_2\end{array}}{e_1 \sim e_2}$.

  By IH on $\mathcal{Q}_1$ with $\mathcal{E}$, we obtain the derivation $\mathcal{E}_1$ of $\Delta \vdash e_1' \downarrow v$.
  Then by IH on $\mathcal{Q}_2$ with $\mathcal{E}_1$, we obtain $\mathcal{E}'$ as required.

- Case $\mathcal{Q} = \dfrac{\begin{array}{c}\mathcal{Q}_0\\ e_0 \sim e_0'\end{array}}{\mathbf{s}\ e_0 \sim \mathbf{s}\ e_0'}$, so $e = \mathbf{s}\ e_0$ and $e' = \mathbf{s}\ e_0'$.

  $\mathcal{E}$ must have the shape:
  $$\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_0\\ \Delta \vdash e_0 \downarrow v_0\end{array}}{\Delta \vdash \mathbf{s}\ e_0 \downarrow \mathbf{s}\ v_0}$$
  By IH on $\mathcal{Q}_0$ with $\mathcal{E}_0$, we obtain the derivation $\mathcal{E}_0'$ of $\Delta \vdash e_0' \downarrow v_0$.
  Then we construct $\mathcal{E}'$ by Eval-S:
  $$\mathcal{E}' = \dfrac{\begin{array}{c}\mathcal{E}_0'\\ \Delta \vdash e_0' \downarrow v_0\end{array}}{\Delta \vdash \mathbf{s}\ e_0' \downarrow \mathbf{s}\ v_0}$$.

- Case $\mathcal{Q} = \dfrac{\begin{array}{cc}\mathcal{Q}_1 & \mathcal{Q}_2\\ e_1 \sim e_1' & e_2 \sim e_2'\end{array}}{e_1 : e_2 \sim e_1' : e_2'}$, so $e = e_1 : e_2$, and $e' = e_1' : e_2'$.

  $\mathcal{E}$ must have the shape:
  $$\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2\\ \Delta \vdash e_1 \downarrow v_1 & \Delta \vdash e_2 \downarrow v_2\end{array}}{\Delta \vdash e_1 : e_2 \downarrow v_1 : v_2}$$
  By IH on $\mathcal{Q}_1$ with $\mathcal{E}_1$, we get a derivation $\mathcal{E}_1'$ of $\Delta \vdash e_1' \downarrow v_1$.
  By IH on $\mathcal{Q}_2$ with $\mathcal{E}_2$, we get another derivaiton $\mathcal{E}_2'$ of $\Delta \vdash e_2' \downarrow v_2$.
  Then we construct $\mathcal{E}'$ by Eval-Cons:
  $$\mathcal{E}' = \dfrac{\begin{array}{cc}\mathcal{E}_1' & \mathcal{E}_2'\\ \Delta \vdash e_1' \downarrow v_1 & \Delta \vdash e_2' \downarrow v_2\end{array}}{\Delta \vdash e_1' : e_2' \downarrow v_1 : v_2}$$

- Case $\mathcal{Q} = \dfrac{\begin{array}{cc}\mathcal{Q}_1 & \mathcal{Q}_2\\ e_1 \sim e_1' & e_2 \sim e_2'\end{array}}{e_1 +\!+ e_2 \sim e_1' +\!+ e_2'}$, so $e = e_1 +\!+ e_2$, and $e' = e_1' +\!+ e_2'$.

  $\mathcal{E}$ must have the shape:
  $$\mathcal{E} = \dfrac{\begin{array}{ccc}\mathcal{E}_1 & \mathcal{E}_2 & \mathcal{F}\\ \Delta \vdash e_1 \downarrow v_1 & \Delta \vdash e_2 \downarrow v_2 & \mathbf{fappend}\ (v_1, v_2, v)\end{array}}{\Delta \vdash e_1 +\!+ e_2 \downarrow v}$$
  By IH on $\mathcal{Q}_1$ with $\mathcal{E}_1$, we get a derivation $\mathcal{E}_1'$ of $\Delta \vdash e_1' \downarrow v_1$.

By IH on $\mathcal{Q}_2$ with $\mathcal{E}_2$, we get another derivaiton $\mathcal{E}_2'$ of $\Delta \vdash e_2' \downarrow v_2$.
Then we construct $\mathcal{E}'$ by Eval-Append:

$$\mathcal{E}' = \dfrac{\begin{array}{ccc}\mathcal{E}_1' & \mathcal{E}_2' & \mathcal{F} \\ \Delta \vdash e_1' \downarrow v_1 & \Delta \vdash e_2' \downarrow v_2 & \textbf{fappend}\ (v_1, v_2, v)\end{array}}{\Delta \vdash e_1' +\!\!+ e_2' \downarrow v}$$

- Case $\mathcal{Q} = \dfrac{}{\textbf{nil} +\!\!+ e_1 \sim e_1}$, so $e = \textbf{nil} +\!\!+ e_1$, and $e' = e_1$.

  $\mathcal{E}$ must have the shape:

$$\mathcal{E} = \dfrac{\dfrac{}{\Delta \vdash \textbf{nil} \downarrow \textbf{nil}} \quad \begin{array}{cc}\mathcal{E}' & \mathcal{F} \\ \Delta \vdash e_1 \downarrow v & \textbf{fappend}\ (\textbf{nil}, v, v)\end{array}}{\Delta \vdash \textbf{nil} +\!\!+ e_1 \downarrow v}$$

  Thus $\mathcal{E}$ already includes $\mathcal{E}'$ and we are done.

- Case $\mathcal{Q} = \dfrac{}{e_1 +\!\!+ \textbf{nil} \sim e_1}$, so $e = e_1 +\!\!+ \textbf{nil}$, and $e' = e_1$.

  This case is analogous to Eq-Append-Nil-L (the last case) since the function **fappend** has the property **fappend** $(v, \textbf{nil}, v)$ by Lemma 3 Fappend Nil Right.

- Case $\mathcal{Q} = \dfrac{}{(e_1 : e_2) +\!\!+ e_3 \sim e_1 : (e_2 +\!\!+ e_3)}$, so $e = (e_1 : e_2) +\!\!+ e_3$, and $e' = e_1 : (e_2 +\!\!+ e_3)$.

  By the rules Eval-Append and Eval-Cons, $\mathcal{E}$ must look like:

$$\mathcal{E} = \dfrac{\dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \Delta \vdash e_1 \downarrow v_1 & e_2 \downarrow v_2\end{array}}{\Delta \vdash e_1 : e_2 \downarrow v_1 : v_2} \quad \begin{array}{cc}\mathcal{E}_3 & \mathcal{F} \\ \Delta \vdash e_3 \downarrow v_3 & \textbf{fappend}\ (v_2, v_3, v')\end{array}}{\textbf{fappend}\ (v_1 : v_2, v_3, v_1 : v')}$$
$$\Delta \vdash (e_1 : e_2) +\!\!+ e_3 \downarrow v_1 : v'$$

  Then we can construct $\mathcal{E}'$:

$$\mathcal{E}' = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \dfrac{\begin{array}{ccc}\mathcal{E}_2 & \mathcal{E}_3 & \mathcal{F} \\ \Delta \vdash e_2 \downarrow v_2 & \Delta \vdash e_3 \downarrow v_3 & \textbf{fappend}\ (v_2, v_3, v')\end{array}}{\Delta \vdash e_2 +\!\!+ e_3 \downarrow v'} \\ \Delta \vdash e_1 \downarrow v_1 & \end{array}}{\Delta \vdash e_1 : (e_2 +\!\!+ e_3) \downarrow v_1 : v'}$$

- Case $\mathcal{Q} = \dfrac{}{(e_1 +\!\!+ e_2) +\!\!+ e_3 \sim e_1 +\!\!+ (e_2 +\!\!+ e_3)}$, so $e = (e_1 +\!\!+ e_2) +\!\!+ e_3$, and $e' = e_1 +\!\!+ (e_2 +\!\!+ e_3)$.

  By the rule Eval-Append, $\mathcal{E}$ must look like:

$$\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \dfrac{\begin{array}{ccc}\mathcal{E}_2 & \mathcal{F}_1 & \\ \Delta \vdash e_1 \downarrow v_1 & \Delta \vdash e_2 \downarrow v_2 & \textbf{fappend}\ (v_1, v_2, v')\end{array}}{\Delta \vdash e_1 +\!\!+ e_2 \downarrow v'} & \begin{array}{cc}\mathcal{E}_3 & \mathcal{F}_2 \\ \Delta \vdash e_3 \downarrow v_3 & \textbf{fappend}\ (v', v_3, v)\end{array}\end{array}}{\Delta \vdash (e_1 +\!\!+ e_2) +\!\!+ e_3 \downarrow v}$$

  By Lemma 4 Fappend Associativity on $\mathcal{F}_1$ with $\mathcal{F}_2$, we get $\mathcal{F}_3$ and $\mathcal{F}_4$ for some list v''. Then by the rule Eval-Append we can construct $\mathcal{E}'$:

$$\mathcal{E}' = \dfrac{\dfrac{\mathcal{E}_1}{\Delta \vdash e_1 \downarrow v_1} \quad \dfrac{\dfrac{\mathcal{E}_2}{\Delta \vdash e_2 \downarrow v_2} \quad \dfrac{\mathcal{E}_3}{\Delta \vdash e_3 \downarrow v_3} \quad \dfrac{\mathcal{F}_3}{\textbf{fappend}\ (v_2, v_3, v'')}}{\Delta \vdash e_2 +\!\!+ e_3 \downarrow v''} \quad \dfrac{\mathcal{F}_4}{\textbf{fappend}\ (v_1, v'', v)}}{\Delta \vdash e_1 +\!\!+ (e_2 +\!\!+ e_3) \downarrow v}$$

- Case $\mathcal{Q} = \dfrac{\dfrac{\mathcal{Q}_1}{e_1 \sim e_1'} \quad \dfrac{\mathcal{Q}_2}{e_2 \sim e_2'}}{\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \sim \textbf{let}\ x = e_1'\ \textbf{in}\ e_2'}$, so e is **let** $x = e_1$ **in** e_2 and e' is **let** $x = e_1'$ **in** e_2'.

  Then $\mathcal{E}$ must have the shape: $\mathcal{E} = \dfrac{\dfrac{\mathcal{E}_1}{\Delta \vdash e_1 \downarrow v_1} \quad \dfrac{\mathcal{E}_2}{\Delta[x \downarrow v_1] \vdash e_2 \downarrow v}}{\Delta \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \downarrow v}$

  By IH on $\mathcal{Q}_1$ with $\mathcal{E}_1$ we obtain $\mathcal{E}_1'$ of $\Delta \vdash e_1' \downarrow v_1$.
  By IH on $\mathcal{Q}_2$ with $\mathcal{E}_2$,we get $\mathcal{E}_2'$ of $\Delta[x \downarrow v_1] \vdash e_2' \downarrow v$.

  Then we can construct $\mathcal{E}' = \dfrac{\dfrac{\mathcal{E}_1'}{\Delta \vdash e_1' \downarrow v_1} \quad \dfrac{\mathcal{E}_2'}{\Delta[x \downarrow v_1] \vdash e_2' \downarrow v}}{\Delta \vdash \textbf{let}\ x' = e_1'\ \textbf{in}\ e_2' \downarrow v}$

- Case $\mathcal{Q} = \dfrac{}{\Delta \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \sim e_2[e_1/x]}$, so e is **let** $x = e_1$ **in** e_2, e' is $e_2[e_1/x]$.
  This case is immediate. Because in the derivation $\mathcal{E}$, we get v by first replacing x with the value v_1 of e_1 in e_2, then evaluating e_2. While in $\mathcal{E}'$ we first replace x with e_1 in e_2, then evaluate e_2. So the evaluation results are the same.

■

4 Evaluation Completeness

The converse of Theorem 9 Equation Soundness also holds.

Theorem 12 (Equation Completeness)**.** *If $e == e'$, then $e \sim e'$.*

This theorem says that if both e and e' evaluate to the same value v, then there must exist a deriavation of $e \sim e'$. Since $e == v$ is evident by Definition 7, we only need to show the following Lemma 13. Because by this lemma we can also obtain $e' \sim v$, then by the symmetry and transitivity rules of the equational system, we will get $e \sim e'$.

Lemma 13. *For any evaluation $\Delta \vdash e \downarrow v$, there must have $e \sim v$.*

Before we prove Lemma 13, we should first prove the following lemma.

Lemma 14 (Fappend Completeness)**.** *If* **fappend** (v_1, v_2, v_3) *(by some derivation $\mathcal{F}$), then $(v_1 +\!\!+ v_2) \sim v_3$ (by some derivation $\mathcal{Q}$).*

Proof. By induction on $\mathcal{F}$:

- Case $\mathcal{F} = \dfrac{}{\textbf{fappend}\ (\textbf{nil}\ , v, v)}$, so $v_1 = \textbf{nil}, v_2 = v, v_3 = v$.

  By the rule Eq-Append-Nil-L, we immeditately get $v_1 +\!\!+ v_2 \sim v_3$.

- Case $\mathcal{F} = \dfrac{\dfrac{\mathcal{F}_0}{\textbf{fappend}\ (v_2', v_3', v)}}{\textbf{fappend}\ ((v_1' : v_2'), v_3', (v_1' : v))}$, so $v_1 = (v_1' : v_2'), v_2 = v_3'$, and $v_3 = (v_1' : v)$.

  By IH on $\mathcal{F}_0$, we obatin a derivation $\mathcal{Q}_0$ of $v_2' +\!\!+ v_3' \sim v$.
  Then by the rule Eq-Cons on $\mathcal{Q}_0$ and the equation $v_1' = v_1'$, we get $\mathcal{Q}'$ of $v_1' : (v_2' +\!\!+ v_3') \sim v_1' : v$.
  By the rule Eq-Append-Cons we also have $\mathcal{Q}''$ of $(v_1' : v_2') +\!\!+ v_3' \sim v_1' : (v_2' +\!\!+ v_3')$.
  Finally, by the transtivity of equations on $\mathcal{Q}''$ with $\mathcal{Q}'$ we get $\mathcal{Q}$ of $(v_1' : v_2') +\!\!+ v_3' \sim v_1' : v$.

■

Now we start to prove Lemma 13.

Proof. Let $\mathcal{E}$ be the derivation of $\Delta \vdash e \downarrow v$. We shall show that there is a deriation $\mathcal{Q}$ of $e \sim v$. By induction on $\mathcal{E}$:

- Case $\mathcal{E} = \dfrac{}{\Delta \vdash \mathbf{z} \downarrow \mathbf{z}}$, so both e and v are $\mathbf{z}$, then immediately $\mathbf{z} \sim \mathbf{z}$ by Eq-Ref.

- Case $\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_0 \\ \Delta \vdash e_0 \downarrow v_0\end{array}}{\Delta \vdash \mathbf{s}\ e_0 \downarrow \mathbf{s}\ v_0}$, so $e = \mathbf{s}\ e_0$ and $v = \mathbf{s}\ v_0$.

  By IH on $\mathcal{E}_0$, we obtain a derivation $\mathcal{Q}_0$ of $e_0 \sim v_0$. Then we can construct $\mathcal{Q}$ by Eq-S:

  $$\mathcal{Q} = \dfrac{\begin{array}{c}\mathcal{Q}_0 \\ e_0 \sim v_0\end{array}}{\mathbf{s}\ e_0 \sim \mathbf{s}\ v_0}$$

- The case for $\mathcal{E} =$ Eval-Nil is analogous to the case $\mathcal{E} =$ Eval-Z .

- Case $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \Delta \vdash e_1 \downarrow v_1 & \Delta \vdash e_2 \downarrow v_2\end{array}}{\Delta \vdash e_1 : e_2 \downarrow v_1 : v_2}$, so $e = e_1 : e_2$, $v = v_1 : v_2$.

  By IH on $\mathcal{E}_1$, we obtain a derivation $\mathcal{Q}_1$ of $e_1 \sim v_1$. By IH on $\mathcal{E}_2$, we obtain a derivation $\mathcal{Q}_2$ of $e_2 \sim v_2$. Then we construct $\mathcal{Q}$ by Eq-Cons: $\mathcal{Q} = \dfrac{\begin{array}{cc}\mathcal{Q}_1 & \mathcal{Q}_2 \\ e_1 \sim v_1 & e_2 \sim v_2\end{array}}{e_1 : e_2 \sim v_1 : v_2}$

- Case $\mathcal{E} = \dfrac{\begin{array}{ccc}\mathcal{E}_1 & \mathcal{E}_2 & \mathcal{F} \\ \Delta \vdash e_1 \downarrow v_1 & \Delta \vdash e_2 \downarrow v_2 & \mathbf{fappend}\ (v_1, v_2, v)\end{array}}{\Delta \vdash e_1 +\!\!+ e_2 \downarrow v}$, so $e = e_1 +\!\!+ e_2$.

  By IH on $\mathcal{E}_1$, we obtain a derivation $\mathcal{Q}_1$ of $e_1 \sim v_1$. By IH on $\mathcal{E}_2$, we obtain a derivation $\mathcal{Q}_2$ of $e_2 \sim v_2$. Then by Lemma 14 on $\mathcal{F}$, we obatin a derivation $\mathcal{Q}'$ of $v_1 +\!\!+ v_2 \sim v$. By the rule Eq-Append, we obtain another derivation $\mathcal{Q}''$ of $e_1 +\!\!+ e_2 \sim v_1 +\!\!+ v_2$:

  $$\mathcal{Q}'' = \dfrac{\begin{array}{cc}\mathcal{Q}_1 & \mathcal{Q}_2 \\ e_1 \sim v_1 & e_2 \sim v_2\end{array}}{e_1 +\!\!+ e_2 \sim v_1 +\!\!+ v_2}$$

  Now we get $\mathcal{Q}$ by Eq-Trans: $\mathcal{Q} = \dfrac{\begin{array}{cc}\mathcal{Q}'' & \mathcal{Q}' \\ e_1 +\!\!+ e_2 \sim v_1 +\!\!+ v_2 & v_1 +\!\!+ v_2 \sim v\end{array}}{e_1 +\!\!+ e_2 \sim v}$

- Case $\mathcal{E} = \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \Delta \vdash e_1 \downarrow v_1 & \Delta[x \downarrow v_1] \vdash e_2 \downarrow v\end{array}}{\Delta \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \downarrow v}$, so e is $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$.

  By the rule Eq-Let-Subst, we have $\mathcal{Q}'$ of $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \sim e_2[e_1/x]$. So we only need to show that there is a derivation $\mathcal{Q}''$ of $e_2[e_1/x] \sim v$, then by Eq-Trans on $\mathcal{Q}'$ with $\mathcal{Q}''$ we will get $\mathcal{Q}$.
  By IH on $\mathcal{E}_1$ we get a derivation $\mathcal{Q}_1$ of $e_1 \sim v_1$.
  Then by IH on $\mathcal{E}_2$, we get $\mathcal{Q}_2$ of $e_2 \sim v$ if $x \sim v_1$. Since we already have Q_1 that says $e_1 \sim v_1$, substituting e_1 for x in e_2 will give us $e_2 \sim v$ as required.

■

5 Implementation in Twelf

The implemenation of Mini-List and the proof of its properties in Twelf are consistent to our definitions and paper proof in previous sections. Besides, there are also some additional definitions or lemmas that we may think trivial on paper proof needed to be formalized in Twelf.

Firstly, in Mini-List we consider values also expressions to keep the paper proof clear and simple. But in Twelf since we use different type families for expressions and values, we should define an extra judgment (such `val2exp` in the code) to convert values to expressions, which is used in the proof of equational completeness.

Secondly, in Twelf we also need to formalize the identical relation between free variables, such as `eq-val` to denote two values are identical. Besides, lemmas about the symmetry, transtivity or replacement of identical free variables are also required.

See Appendix A for the source code in Twelf.

Appendices

Appendix A Source code

```
% Computation and Deduction (2016)
% Mini-project


% --- type system
tp : type.   %name tp T.

nat : tp.
list : tp -> tp.



% --- expressions
exp : tp -> type. %name exp E.

ez : exp nat.
es : exp nat -> exp nat.
enil : exp (list T).
econs : exp T -> exp (list T)-> exp (list T).
eappend : exp (list T) -> exp (list T) -> exp (list T).
elet : exp T1 -> (exp T1 -> exp T2) -> exp T2.



% --- values
val : tp -> type.  %name val V.

vz : val nat.
vs : val nat -> val nat.
vnil : val (list T).
vcons : val T -> val (list T) -> val (list T).



% V1 ++ V2 = V3
fappend : val (list T) -> val (list T) -> val (list T) -> type. %name fappend F.
%mode fappend +V1 +V2 -V3.

fappend/nil : fappend vnil V V.

fappend/cons : fappend (vcons V1 V2) V3 (vcons V1 V)
            <- fappend V2 V3 V.

%worlds () (fappend _ _ _).
%total V (fappend V _ _).
```

```
%% --- convert values to expressions

val2exp : val T -> exp T -> type.  %name val2exp VE.
%mode val2exp +V -E.

vz2ez : val2exp vz ez.

vs2es : val2exp (vs V) (es E)
      <- val2exp V E.

vnil2enil : val2exp vnil enil.

vcons2econs : val2exp (vcons V1 V2) (econs E1 E2)
         <- val2exp V1 E1
         <- val2exp V2 E2.

%worlds () (val2exp _ _).
%total V (val2exp V _).



% -- operational semantics

eval : exp T -> val T -> type.  %name eval EP.
%mode eval +E -V.

eval/z : eval ez vz.

eval/s : eval (es E) (vs V)
      <- eval E V.

eval/nil : eval enil vnil.

eval/cons : eval (econs E1 E2) (vcons V1 V2)
         <- eval E1 V1
         <- eval E2 V2.

eval/append : eval (eappend E1 E2)  V
         <- eval E1 V1
         <- eval E2 V2
         <- fappend V1 V2 V.

%%% -- let case ---
eval/let : eval (elet E1 E2) V
         <- eval E1 V1
         <- {x} eval x V1 ->  eval (E2 x) V.

%block eval-block1 : some {T:tp} {V:val T} block {x: exp T} {ex : eval x V}.
%worlds (eval-block1) (eval _ _).
%total E (eval E _).



%% --- e ~ e', equational system

eq : exp T -> exp T -> type.  %name eq Q.

eq/ : eq E E.

eq-sym :  eq E' E -> eq E E'.

eq-trans : eq E1 E' -> eq E' E2 -> eq E1 E2.

eq-s : eq E E' -> eq (es E) (es E').
```

```
eq-cons : eq E1 E1' -> eq E2 E2' -> eq (econs E1 E2) (econs E1' E2').

eq-append : eq E1 E1' -> eq E2 E2' -> eq (eappend E1 E2) (eappend E1' E2').


% -- []++ e ~ e
eq-append-nil-l : eq (eappend enil E) E.

% -- e ++ [] ~ e
eq-append-nil-r : eq (eappend E enil) E.

% -- (e1:e2)++e3 ~ e1:(e2++e3)
eq-append-cons : eq (eappend (econs E1 E2) E3) (econs E1 (eappend E2 E3)).

% -- (e1++e2)++e3 ~ e1++(e2++e3)
eq-append-asso : eq (eappend (eappend E1 E2) E3) (eappend E1 (eappend E2 E3)).


%% -- let
eq-let : eq E1 E1' -> ({x} {x'} eq x x' -> eq (E2 x) (E2' x'))
         -> eq (elet E1 E2) (elet E1' E2').

eq-let-subst : eq (elet E1 E2) (E2 E1).




%% --- values identity
eq-val : val T -> val T -> type. %name eq-val QV.

eq-val/: eq-val V V.


eq-val-sym : eq-val V V' -> eq-val V' V -> type.
%mode eq-val-sym +QV -QV'.

- : eq-val-sym eq-val/ eq-val/.

%worlds () (eq-val-sym _ _).
%total {} (eq-val-sym _ _).


eq-val-s : eq-val V V' -> eq-val (vs V) (vs V') -> type.
%mode eq-val-s +QV -QV'.

- : eq-val-s eq-val/ eq-val/.

%worlds () (eq-val-s _ _).
%total {} (eq-val-s _ _).


eq-val-cons : eq-val V1 V1' -> eq-val V2 V2'
-> eq-val (vcons V1 V2) (vcons V1' V2') -> type.
%mode eq-val-cons +QV1 +QV2 -QV.

- : eq-val-cons eq-val/ eq-val/ eq-val/.

%worlds () (eq-val-cons _ _ _).
%total {} (eq-val-cons _ _ _).



% --- some properties of fappend

% -- determinacy
fappend-determ : fappend V1 V2 V3 -> fappend V1 V2 V3' -> eq-val V3 V3' -> type.
%mode fappend-determ +F1 +F2 -QV.

fappend-determ/nil : fappend-determ fappend/nil fappend/nil eq-val/.
```

```
fappend-determ/cons : fappend-determ (fappend/cons F) (fappend/cons F') QV
         <- fappend-determ F F' QV'
         <- eq-val-cons eq-val/ QV' QV.

%worlds () (fappend-determ _ _ _).
%total F (fappend-determ F _ _).



%% -- fappend equal values will obtain equal values
eq-val-fappend : eq-val V1 V1'
               -> eq-val V2 V2'
               -> fappend V1 V2 V3
               -> fappend V1' V2' V3'
               -> eq-val V3 V3' -> type.
%mode eq-val-fappend +Q1 +Q2 +F1 +F2 -Q3.

- : eq-val-fappend eq-val/ eq-val/ F F' QV
   <- fappend-determ F F' QV.

%worlds () (eq-val-fappend _ _ _ _ _).
%total {} (eq-val-fappend _ _ _ _ _).



%% -- for any lists V1 and V2, there exists some V = V1 ++ V2
fappend-exists : {V1} {V2} fappend V1 V2 V -> type.
%mode fappend-exists +V1 +V2 -F.

fappend-exists/nil : fappend-exists vnil V fappend/nil.

fappend-exists/cons : fappend-exists (vcons V1 V2) V (fappend/cons F)
                    <- fappend-exists V2 V F.

%worlds () (fappend-exists _ _ _).
%total V (fappend-exists V _ _).



% --- V ++ nil = V
fappend-nil-r : fappend V vnil V' -> eq-val V V' -> type.
%mode fappend-nil-r +F -QV.

- : fappend-nil-r fappend/nil eq-val/.

- : fappend-nil-r (fappend/cons F) QV
      <- fappend-nil-r F QV1
      <- eq-val-cons eq-val/ QV1 QV.

%worlds () (fappend-nil-r _ _).
%total F (fappend-nil-r F _).



% --- (V1 ++ V2) ++ V3 = V1 ++ (V2++V3)
fappend-asso : fappend V1 V2 V3 -> fappend V3 V4 V5
            -> fappend V2 V4 V6 -> fappend V1 V6 V5 -> type.
%mode fappend-asso +F1 +F2 -F3 -F4.

- : fappend-asso fappend/nil F2 F2 fappend/nil.

- : fappend-asso (fappend/cons F1) (fappend/cons F2) F3 (fappend/cons F4)
      <- fappend-asso F1 F2 F3 F4.

%worlds () (fappend-asso _ _ _ _).
%total F (fappend-asso F _ _ _).
```

```
eq-eval-env : ({x} {ex: eval x V0} eval (E x) V)
            -> eq-val V0 V0' -> ({x} {ex': eval x V0'} eval (E x) V) -> type.
%mode eq-eval-env +EP +QV -EP'.

-: eq-eval-env EP eq-val/ EP.


%% -- evaluation determinacy

eval-determ : eval E V1 ->  eval E V2 -> eq-val V1 V2 -> type.
%mode eval-determ +EP1 +EP2 -QV.

eval-determ/z : eval-determ eval/z eval/z eq-val/.

eval-determ/s : eval-determ (eval/s EP) (eval/s EP') QV
               <- eval-determ EP EP' QV0
               <- eq-val-s QV0 QV.


eval-determ/nil : eval-determ eval/nil eval/nil eq-val/.

eval-determ/cons : eval-determ (eval/cons EP2 EP1) (eval/cons EP2' EP1') QV
                  <- eval-determ EP1 EP1' QV1
                  <- eval-determ EP2 EP2' QV2
                  <- eq-val-cons QV1 QV2 QV.


eval-determ/append : eval-determ
                  (eval/append F EP2 EP1)
                  (eval/append F' EP2' EP1')
                  QV
                  <- eval-determ EP1 EP1' QV1
                  <- eval-determ EP2 EP2' QV2
                  <- eq-val-fappend QV1 QV2 F F' QV.



% --- evaluation exist theorem

eval-exists : {E} eval E V -> type.
%mode eval-exists +E -EP.

eval-exists/z : eval-exists ez eval/z.

eval-exists/s : eval-exists (es E) (eval/s EP)
               <- eval-exists E EP.

eval-exists/nil : eval-exists enil eval/nil.

eval-exists/cons : eval-exists (econs E1 E2) (eval/cons EP2 EP1)
                  <- eval-exists E1 EP1
                  <- eval-exists E2 EP2.

eval-exists/append : eval-exists
(eappend E1 E2)
(eval/append (F: fappend V1 V2 V3) EP2 EP1)
               <- eval-exists E1 EP1
               <- eval-exists E2 EP2
               <- fappend-exists V1 V2 F.


%% --- let case
eval-determ/let : eval-determ
               (eval/let EP2 EP1)
               (eval/let EP2' EP1')
               QV
                  <- eval-determ EP1 EP1' QV1
                  <- eq-val-sym QV1 QV1'
                  <- eq-eval-env EP2' QV1' EP2''
```

```
                  <- {x} {ex: eval x V} {_ : eval-exists x ex}
                     eval-determ ex ex eq-val/
                     -> eval-determ (EP2 x ex)  (EP2'' x ex) QV.

eval-exists/let : eval-exists (elet E1 E2) (eval/let EP2 EP1)
               <- eval-exists E1 EP1
               <- {x} {ex: eval x V}  {_ : eval-determ ex ex eq-val/}
                  eval-exists x ex
                  -> eval-exists (E2 x) (EP2 x ex).


%block eval-determ-block : some {T: tp} {V: val T}
block {x: exp T} {ex: eval x V}
         {_ : eval-exists x ex}
         {_ : eval-determ ex ex eq-val/}.

%block eval-exists-block : some {T: tp} {V: val T}
block {x: exp T} {ex: eval x V}
         {_ : eval-determ ex ex eq-val/}
         {_ : eval-exists x ex}.


%% --- replace values in an evalution rule
eq-eval-val : eval E V -> eq-val V V' -> eval E V' -> type.
%mode eq-eval-val +EP +QV -EP'.
- : eq-eval-val EP eq-val/ EP.



%% --- soundness of equational system
%% if e~e' and e evaluates to v, then e' also evaluates to v

eq-sound : eq E E' -> eval E V -> eval E' V -> type.
%mode eq-sound +Q +EP -EP'.

eq-sound/ref : eq-sound eq/ EP EP.

eq-sound/sym : eq-sound (eq-sym Q) EP EP'
               <- eval-exists E' EP1'
               <- eq-sound Q EP1' EP1
               <- eval-determ EP1 EP QV
               <- eq-eval-val EP1' QV EP'.

eq-sound/trans : eq-sound (eq-trans Q1 Q2) EP EP'
                  <- eq-sound Q1 EP EP1'
                  <- eq-sound Q2 EP1' EP'.

eq-sound/s : eq-sound (eq-s Q1) (eval/s EP) (eval/s EP')
                  <- eq-sound Q1 EP EP'.

eq-sound/cons : eq-sound
               (eq-cons Q1 Q2)
               (eval/cons EP2 EP1)
               (eval/cons EP2' EP1')
               <- eq-sound Q1 EP1 EP1'
               <- eq-sound Q2 EP2 EP2'.

eq-sound/append : eq-sound
                  (eq-append Q1 Q2)
                  (eval/append F EP2 EP1)
                  (eval/append F EP2' EP1')
                  <- eq-sound Q1 EP1 EP1'
                  <- eq-sound Q2 EP2 EP2'.

eq-sound/append/nil-l : eq-sound
                     eq-append-nil-l
                     (eval/append fappend/nil EP eval/nil)
                     EP.

eq-sound/append/nil-r : eq-sound
```

```
                    eq-append-nil-r
                    (eval/append F eval/nil EP1)
                    EP
                    <- fappend-nil-r F QV
                    <- eq-eval-val EP1 QV EP.


eq-sound/append/cons : eq-sound
                    eq-append-cons
                    (eval/append (fappend/cons F) EP3 (eval/cons EP2 EP1))
                    (eval/cons (eval/append F EP3 EP2) EP1).


eq-sound/append/asso : eq-sound
                    eq-append-asso
                    (eval/append F2 EP3 (eval/append F1 EP2 EP1))
                    (eval/append F4 (eval/append F3 EP3 EP2) EP1)
                    <- fappend-asso F1 F2 F3 F4.


eq-sound/let : eq-sound
               (eq-let Q1 Q2)
               (eval/let EP2 EP1)
               (eval/let EP2' EP1')
               <- eq-sound Q1 EP1 EP1'
               <- {x} {ex: eval x V1}  {qx : eq x x}
                  {_ : eval-determ ex ex eq-val/}
                  {_ : eval-exists x ex}
                  eq-sound qx ex ex
                  -> eq-sound (Q2 x x qx) (EP2 x ex) (EP2' x ex).


eq-sound/let/subst : eq-sound
                   eq-let-subst
                   (eval/let EP2 EP1)
                   (EP2 _ EP1).


%block eq-sound-block : some {T: tp} {V: val T}
                   block {x: exp T} {ex: eval x V} {qx : eq x x}
                      {_ : eval-determ ex ex eq-val/}
                      {_ : eval-exists x ex}
                      {_ : eq-sound qx ex ex}.
%worlds (eval-determ-block | eval-exists-block | eq-sound-block)
(eval-exists _ _) (eq-sound _ _ _) (eq-eval-val _ _ _)
(eq-eval-env _ _ _) (eval-determ _ _ _).
%total {} (eq-eval-val _ _ _).
%total {} (eq-eval-env _ _ _).
%total EP (eval-determ EP _ _).
%total E (eval-exists E _).
%total Q (eq-sound Q _ _ ).



%% --fappend completeness

fappend-comp : fappend V1 V2 V3
            -> val2exp V1 E1
            -> val2exp V2 E2
            -> val2exp V3 E3
            -> eq (eappend E1 E2) E3 -> type.
%mode fappend-comp +F  +VE1 +VE2 -VE3 -Q .

fappend-comp/nil : fappend-comp fappend/nil vnil2enil VE VE eq-append-nil-l.

fappend-comp/cons : fappend-comp
                   (fappend/cons F)
                   (vcons2econs VE2 VE1)
                   VE3
```

```
                  (vcons2econs VE VE1)
                  (eq-trans eq-append-cons (eq-cons eq/ Q))
                  <- fappend-comp F VE2 VE3 VE Q.


%% --- completeness of equational system

eq-evcomp : eval E V -> val2exp V E' -> eq E E' -> type.
%mode eq-evcomp +EP -VE -Q.

eq-evcomp/z : eq-evcomp eval/z vz2ez eq/.

eq-evcomp/s : eq-evcomp (eval/s EP) (vs2es VE) (eq-s Q)
               <- eq-evcomp EP VE Q.

eq-evcomp/nil : eq-evcomp eval/nil vnil2enil eq/.

eq-evcomp/cons : eq-evcomp (eval/cons EP2 EP1)
               (vcons2econs VE2 VE1)
               (eq-cons Q1 Q2)
               <- eq-evcomp EP1 VE1 Q1
               <- eq-evcomp EP2 VE2 Q2.

eq-evcomp/append : eq-evcomp
                  (eval/append F EP2 EP1)
                  VE
                  (eq-trans
                  (eq-append Q1 Q2) Q')
                  <- eq-evcomp EP1 VE1 Q1
                  <- eq-evcomp EP2 VE2 Q2
                  <- fappend-comp F VE1 VE2 VE Q'.


eq-evcomp/let : eq-evcomp
               (eval/let EP2 EP1)
               (VE2 E1)
               (eq-trans eq-let-subst (Q2 E1 Q1))
               <- eq-evcomp EP1 VE1 Q1
               <- {x} {ex:eval x V1}
               {qx: eq x E1'}
               eq-evcomp ex VE1 qx
               -> eq-evcomp (EP2 x ex) (VE2 x) (Q2 x qx).


%block eq-evcomp-block : some {T: tp} {V: val T} {E: exp T} {VE: val2exp V E}
                     block {x: exp T} {ex: eval x V}
                     {qx: eq x E}
                     {_ : eq-evcomp ex VE qx } .
%worlds (eq-evcomp-block) (eq-evcomp _ _ _) (fappend-comp _ _ _ _ _).
%total F (fappend-comp F _ _ _ _).
%total EP (eq-evcomp EP _ _).
```