A verified abstract machine for functional coroutines − Coq formalization (companion technical report)

by Tristan Crolard

CNAM Paris, France

Email: tristan.crolard@cnam.fr

Abstract

Functional coroutines are a restricted form of control mechanism, where each continuation comes with its local environment. This restriction was originally obtained by considering a constructive version of Parigot's classical natural deduction which is sound and complete for the Constant Domain logic. In this article, we present a refinement of de Groote's abstract machine which is proved to be correct for functional coroutines. Therefore, this abstract machine also provides a direct computational interpretation of the Constant Domain logic.

This companion technical report contains the full Coq formalization (all the proofs are available in the Coq source file). The section numbering is the same as in the article for easy cross referencing.

1 Introduction

2 Dependency relations

3 Abstract machines

Require Import *Utf8*.
Add LoadPath "~/Sources/Coq/Tactics".
Require Import *CpdtTactics LibTactics TwelfTactics*.

Require Import *Arith List String*.
Require Import *Coq.extraction.ExtrOcamlString*.

Definition *ble_nat* (*n* : *nat*) (*m* : *nat*) : *bool* := *beq_nat* (*n* − *m*) 0.
Notation "*n* == *m*" := (*beq_nat n m*) (at level 70, no associativity).
Notation "*n* ≦ *m*" := (*ble_nat n m*) (at level 70, no associativity).

Reserved Notation "$m_1 - m_2 = m$" (at level 30).

Inductive *minus* : *nat* → *nat* → *nat* → *Prop* :=

| $Minus_1$: ∀ m_1,

$$m_1 - 0 = m_1$$

| $Minus_2$: ∀ m_1 m_2 m,

$$\frac{m_1 - m_2 = m}{S(m_1) - S(m_2) = m}$$

where "$m_1 - m_2 = m$" := (*minus* m_1 m_2 m).

Fixpoint f (m : *nat*) : ($m - m = 0$) :=
 match m **with**
 | 0 => $Minus_1$ 0
 | $S(n)$ => $Minus_2$ n n 0 (f n)
 end.

Lemma *minus_trivial* : ∀ n, *not* ($0 = Sn$).
Proof.
 auto.
Qed.

Lemma *minus_id* : ∀ m, $m - m = 0$.
Proof.
 exact f.
Qed.

Hint Resolve *minus_id*.

Lemma *minus_unique* : ∀ m_2 m_1 m m', $m_1 - m_2 = m \rightarrow m_1 - m_2 = m' \rightarrow m = m'$.
Proof.
 intros.
 induction H.
 - inversion H0.
  auto.
 - inversion H0.
  auto.
Qed.

Hint Resolve *minus_unique* : *gen_subst_db*.

Lemma *minus_succ* : ∀ n_3 n_2 n_1, $n_1 - (Sn_2) = n_3 \rightarrow n_1 - n_2 = (Sn_3)$.
Proof.
 intros n_3 n_2.
 induction n_2.
 - intros n_1 $\mathscr{D}$.
  inversion $\mathscr{D}$.
  inversion H1.
  apply Minus_1.
 - intros n_1 $\mathscr{D}$.
  inversion $\mathscr{D}$.
  apply Minus_2.
  apply IHn_2.
  exact H1.
Qed.

Lemma *minus_swap* : ∀ n_3 n_2 n_1, $n_1 - n_2 = n_3 \rightarrow n_1 - n_3 = n_2$.
Proof.
 intros n_3 n_2.
 induction n_2.
 - intros n_1 $\mathscr{D}$.
  inversion $\mathscr{D}$.
  apply minus_id.
 - intros n_1 $\mathscr{D}$.

inversion $\mathscr{D}$.
apply minus_succ.
apply Minus_2.
apply IHn_2.
exact H1.
Qed.

Definition *vector* := *list nat*.
Definition *table* := *list vector*.

Reserved Notation "$\mathscr{I}(n) = m$" (at level 30).

Inductive *fetch* {*A* : *Type*} : *list A* → *nat* → A → *Prop* :=

| Fetch_1 : ∀ $\mathscr{I}$ (*n*:*A*),

$$(n::\mathscr{I})(0)=n$$

| Fetch_2 : ∀ $\mathscr{I}$ (*n*:*A*) n_1 n_2,

$$\frac{\mathscr{I}(n_1)=n_2}{(n::\mathscr{I})(Sn_1)=n_2}$$

where "$\mathscr{I}(n) = m$" := (@*fetch* _ $\mathscr{I}$ *n m*).

Lemma *fetch_unique* : ∀ *A n* $\mathscr{I}$ (*y*:*A*) (*y*′:*A*), $\mathscr{I}(n)=y \to \mathscr{I}(n)=y' \to y=y'$.
Proof.
introv. revert_last.
intros y1 H H0. revert H0. revert y1.
induction H;
(intros y2 D2; inversion D2; repeat progress (eauto || econstructor || gen_subst || crush)).
Qed.

Hint Resolve *fetch_unique* : *gen_subst_db*.

Reserved Notation "$n - \mathscr{I}\ (\ l\ ) = g$" (at level 30).

Inductive *compute* : *list nat* → *nat* → *nat* → *nat* → *Prop* :=

| Compute_1 : ∀ $\mathscr{I}$ *n l g k*,

$$\frac{\mathscr{I}(l)=k \qquad n-k=g}{n-\mathscr{I}(l)=g}$$

where "$n - \mathscr{I}\ (\ l\ ) = g$" := (*compute* $\mathscr{I}$ *n l g*).

Lemma *compute_unique* : ∀ $\mathscr{I}$ *n l y y*′, $n-\mathscr{I}(l)=y \to n-\mathscr{I}(l)=y' \to y=y'$.
Proof.
introv. revert_last.
intros y1 H H0. revert H0. revert y1.
induction H;
(intros y2 D2; inversion D2; repeat progress (eauto || econstructor || gen_subst || crush)).
Qed.

Hint Resolve *compute_unique* : *gen_subst_db*.

Reserved Notation "$n \in \mathscr{I}$" (at level 30).

Inductive *member* : *list nat* → *nat* → *Prop* :=

| $Member_1$: ∀ $\mathcal{J}$ *n*,

$$n \in (n :: \mathcal{J})$$

| $Member_2$: ∀ $\mathcal{J}$ *n n′*,

$$\frac{n \in \mathcal{J}}{n \in (n' :: \mathcal{J})}$$

where "$n \in \mathcal{J}$" := (*member* $\mathcal{J}$ *n*).

Lemma *domain* : ∀ $\mathcal{J}$ *k*, $k \in \mathcal{J} \to \exists n, \mathcal{J}(n)=k$.
Proof.
intros $\mathcal{J}$ k D.
induction D.
- exists 0.
  apply $Fetch_1$.
- destruct IHD.
  exists (S x).
  apply $Fetch_2$.
  exact H.
Qed.

3.1 Safe λ_{ct}-terms

Inductive *term* : *Type* :=
| *Var* (*n* : *nat*)
| *App* (t_1 : *term*) (t_2 : *term*)
| *Lambda* (*t* : *term*)
| *Catch* (*t* : *term*)
| *Throw* (α : *nat*) (*t* : *term*).

Notation "'`' *x* '`'" := (*Var x*) (at level 80).
Notation "t_1 " t_2" := (*App* t_1 t_2) (at level 80).
Notation "'λ' *t*" := (*Lambda t*) (at level 80).

Notation "'**catch**'" := *Catch* (at level 80).
Notation "'**throw**'" := *Throw* (at level 80).
Notation "'**get-context**'" := *Catch* (at level 80).
Notation "'**set-context**'" := *Throw* (at level 80).

Reserved Notation "'*Safe*_' '(' *n* ')^(' $\mathcal{J}$, $\mathcal{J}_\mu$ ')' (*t*)" (at level 30).

Inductive *safe* : *term* → *list nat* → *list* (*list nat*) → *nat* → *Prop* :=

| $Safe_1$: ∀ $\mathcal{J}$ $\mathcal{J}_\mu$ *n g k*,

$$\frac{n - g = k \qquad k \in \mathcal{J}}{Safe_n^{\mathcal{J},\mathcal{J}_\mu}(\text{`}g\text{`})}$$

| $Safe_2$: ∀ $\mathcal{J}$ $\mathcal{J}_\mu$ *n t u*,

$$\frac{Safe_n^{\mathcal{J},\mathcal{J}_\mu}(t) \qquad Safe_n^{\mathcal{J},\mathcal{J}_\mu}(u)}{Safe_n^{\mathcal{J},\mathcal{J}_\mu}(t\,u)}$$

| $Safe_3$: ∀ $\mathcal{J}$ $\mathcal{J}_\mu$ *n t*,

$$\frac{Safe_{Sn}^{(Sn::\mathcal{J}),\mathcal{J}_\mu}(t)}{Safe_n^{\mathcal{J},\mathcal{J}_\mu}(\lambda t)}$$

| $Safe_4 : \forall\ \mathcal{J}\ \mathcal{J}_\mu\ n\ t,$

$$\frac{Safe_n^{\mathcal{J},(\mathcal{J}::\mathcal{J}_\mu)}(t)}{Safe_n^{\mathcal{J},\mathcal{J}_\mu}(\textbf{catch}\ t)}$$

| $Safe_5 : \forall\ \mathcal{J}\ \mathcal{J}'\ \mathcal{J}_\mu\ n\ t\ \alpha,$

$$\frac{\mathcal{J}_\mu(\alpha) = \mathcal{J}' \qquad Safe_n^{\mathcal{J}',\mathcal{J}_\mu}(t)}{Safe_n^{\mathcal{J},\mathcal{J}_\mu}(\textbf{throw}\ \alpha\ t)}$$

where "'*Safe_*' '(' *n* ')^(' $\mathcal{J}$, $\mathcal{J}_\mu$ ')' (*t*)" := (*safe t* $\mathcal{J}$ $\mathcal{J}_\mu$ *n*).

3.2 From local indices to global indices

Reserved Notation "'↓_' '(' *n* ')^(' $\mathcal{J}$, $\mathcal{J}\mu$ ')' (t_1) = t_2" (at level 30).

Inductive *litogi* : *term* → *list nat* → *list* (*list nat*) → *nat* → *term* → *Prop* :=

| $Litogi_1 : \forall\ \mathcal{J}\ \mathcal{J}_\mu\ n\ g\ l,$

$$\frac{n - \mathcal{J}(l) = g}{\downarrow_n^{\mathcal{J},\mathcal{J}_\mu}(`l`) = (`g`)}$$

| $Litogi_2 : \forall\ \mathcal{J}\ \mathcal{J}_\mu\ n\ t\ u\ t'\ u',$

$$\frac{\downarrow_n^{\mathcal{J},\mathcal{J}_\mu}(t) = t' \qquad \downarrow_n^{\mathcal{J},\mathcal{J}_\mu}(u) = u'}{\downarrow_n^{\mathcal{J},\mathcal{J}_\mu}(t\,u) = (t'u')}$$

| $Litogi_3 : \forall\ \mathcal{J}\ \mathcal{J}_\mu\ n\ t\ t',$

$$\frac{\downarrow_{Sn}^{(Sn::\mathcal{J}),\mathcal{J}_\mu}(t) = t'}{\downarrow_n^{\mathcal{J},\mathcal{J}_\mu}(\lambda t) = (\lambda t')}$$

| $Litogi_4 : \forall\ \mathcal{J}\ \mathcal{J}_\mu\ n\ t\ t',$

$$\frac{\downarrow_n^{\mathcal{J},(\mathcal{J}::\mathcal{J}_\mu)}(t) = t'}{\downarrow_n^{\mathcal{J},\mathcal{J}_\mu}(\textbf{get-context}\ t) = (\textbf{catch}\ t')}$$

| $Litogi_5 : \forall\ \mathcal{J}\ \mathcal{J}'\ \mathcal{J}_\mu\ n\ t\ t'\ \alpha,$

$$\frac{\mathcal{J}_\mu(\alpha) = \mathcal{J}' \qquad \downarrow_n^{\mathcal{J}',\mathcal{J}_\mu}(t) = t'}{\downarrow_n^{\mathcal{J},\mathcal{J}_\mu}(\textbf{set-context}\ \alpha\ t) = (\textbf{throw}\ \alpha\ t')}$$

where "'↓_' '(' *n* ')^(' $\mathcal{J}$, $\mathcal{J}_\mu$ ')' (t_1) = t_2" := (*litogi* t_1 $\mathcal{J}$ $\mathcal{J}_\mu$ *n* t_2).

Lemma *litogi_unique* : $\forall\ \mathcal{J}\ \mathcal{J}_\mu\ n\ x\ y\ y', \downarrow_n^{\mathcal{J},\mathcal{J}_\mu}(x) = y \to \downarrow_n^{\mathcal{J},\mathcal{J}_\mu}(x) = y' \to y = y'.$
Proof.
```
 introv. revert_last.
 intros y1 H H0. revert H0. revert y1.
 induction H;
   (intros y2 D2; inversion D2; repeat progress ( eauto || econstructor || gen_subst || crush )).
```
Qed.

Hint Resolve *litogi_unique* : *gen_subst_db*.

Lemma *safe_image* : $\forall\ \mathcal{J}\ \mathcal{J}_\mu\ n\ t', Safe_n^{\mathcal{J},\mathcal{J}_\mu}(t') \to \exists t, \downarrow_n^{\mathcal{J},\mathcal{J}_\mu}(t) = t'.$
Proof.
```
 intros J J_μ n t D.
 induction D.
 - apply domain in H0.
   apply minus_swap in H.
```

```
    destruct H0 as [ l D111 ].
    exists (`l`).
    apply Litogi_1.
    eapply Compute_1.
    exact D111.
    exact H.
  - destruct IHD1 as [ t' D11 ].
    destruct IHD2 as [ u' D12 ].
    exists (t' _*_ u').
    apply Litogi_2.
    exact D11.
    exact D12.
  - destruct IHD as [ t' D11 ].
    exists (_λ_ t').
    apply Litogi_3.
    exact D11.
  - destruct IHD as [ t' D11 ].
    exists (get-context t').
    apply Litogi_4.
    exact D11.
  - destruct IHD as [ t' D11 ].
    exists (set-context α t').
    eapply Litogi_5.
    exact H.
    exact D11.
Qed.
```

3.3 Abstract machine for λ_{ct}-terms

3.3.1 Closure, environment, stack and state

```
Inductive clos :=
| Cl (t : term) (ℰ : list clos) (ℰ_μ : list (list clos)).

Definition stack := list clos.
Definition c_env := list clos.
Definition k_env := list stack.

Inductive state :=
| St (t : term) (ℰ : c_env) (ℰ_μ : k_env) (𝒮 : stack).

Notation "⟨ t , ℰ , ℰ_μ , 𝒮 ⟩" := (St t ℰ ℰ_μ 𝒮) (at level 80).
Notation "[ t , ℰ , ℰ_μ ]" := (Cl t ℰ ℰ_μ) (at level 80).
```

3.3.2 Evaluation rules

```
Reserved Notation "σ_1 ⇝ σ_2" (at level 30).

Inductive step : state → state → Prop :=

| K_var : ∀ k t ℰ ℰ' ℰ_μ ℰ'_μ 𝒮,
```

$$\frac{\mathscr{E}(k) = [t, \mathscr{E}', \mathscr{E}'_\mu]}{\langle\, `k` , \mathscr{E}, \mathscr{E}_\mu, \mathcal{S}\rangle \rightsquigarrow \langle t, \mathscr{E}', \mathscr{E}'_\mu, \mathcal{S}\rangle}$$

| *K_app* : ∀ *t u* $\mathscr{E}$ $\mathscr{E}_\mu$ $\mathcal{S}$,

$$\langle (t\,u), \mathscr{E}, \mathscr{E}_\mu, \mathcal{S}\rangle \rightsquigarrow \langle t, \mathscr{E}, \mathscr{E}_\mu, [u, \mathscr{E}, \mathscr{E}_\mu] :: \mathcal{S}\rangle$$

| *K_abs* : ∀ *t c* $\mathscr{E}$ $\mathscr{E}_\mu$ $\mathcal{S}$,

$$\langle \lambda t, \mathscr{E}, \mathscr{E}_\mu, c :: \mathcal{S}\rangle \rightsquigarrow \langle t, (c :: \mathscr{E}), \mathscr{E}_\mu, \mathcal{S}\rangle$$

| *K_catch* : ∀ *t* $\mathscr{E}$ $\mathscr{E}_\mu$ $\mathcal{S}$,

$$\langle \textbf{catch}\ t, \mathscr{E}, \mathscr{E}_\mu, \mathcal{S}\rangle \rightsquigarrow \langle t, \mathscr{E}, (\mathcal{S} :: \mathscr{E}_\mu), \mathcal{S}\rangle$$

| *K_throw* : ∀ *t* α $\mathscr{E}$ $\mathscr{E}_\mu$ $\mathcal{S}$ $\mathcal{S}'$,

$$\frac{\mathscr{E}_\mu(\alpha) = \mathcal{S}'}{\langle \textbf{throw}\ \alpha\ t, \mathscr{E}, \mathscr{E}_\mu, \mathcal{S}\rangle \rightsquigarrow \langle t, \mathscr{E}, \mathscr{E}_\mu, \mathcal{S}'\rangle}$$

where "$\sigma_1 \rightsquigarrow \sigma_2$" := (*step* σ_1 σ_2).

Lemma *step_unique* : ∀ σ_1 σ_2 σ_2', $\sigma_1 \rightsquigarrow \sigma_2 \to \sigma_1 \rightsquigarrow \sigma_2' \to \sigma_2 = \sigma_2'$.
Proof.
```
 introv. revert_last.
 intros y1 H H0. revert H0. revert y1.
 induction H;
  (intros y2 D2; inversion D2; repeat progress ( eauto || econstructor || gen_subst || crush )).
```
Qed.

Hint Resolve *step_unique* : *gen_subst_db*.

3.4 Abstract machine for safe λ_{ct}-terms (with local environments)

3.4.1 Closure, environment, stack and state

Inductive *clos_l* :=
| *Cl_l* (*t* : *term*) ($\mathscr{L}$: *list clos_l*) ($\mathscr{L}_\mu$: *list* (*list clos_l*)) ($\mathscr{E}_\mu$: *list* (*list clos_l*)).

Definition *stack_l* := *list clos_l*.
Definition *l_env_l* := *list clos_l*.
Definition *m_env_l* := *list l_env_l*.
Definition *k_env_l* := *list stack_l*.

Inductive *state_l* :=
| *St_l* (*t* : *term*) ($\mathscr{L}$: *l_env_l*) ($\mathscr{L}_\mu$: *m_env_l*) ($\mathscr{E}_\mu$: *k_env_l*) ($\mathcal{S}$: *stack_l*).

Notation "⟨ *t* , $\mathscr{L}$, $\mathscr{L}_\mu$, $\mathscr{E}_\mu$, $\mathcal{S}$ ⟩" := (*St_l t* $\mathscr{L}$ $\mathscr{L}_\mu$ $\mathscr{E}_\mu$ $\mathcal{S}$) (at level 80).
Notation "[*t* , $\mathscr{L}$, $\mathscr{L}_\mu$, $\mathscr{E}_\mu$]" := (*Cl_l t* $\mathscr{L}$ $\mathscr{L}_\mu$ $\mathscr{E}_\mu$) (at level 80).

3.4.2 Evaluation rules

Reserved Notation "$\sigma_1 \rightsquigarrow$ '^*l*' σ_2" (at level 30).

Inductive *step_l* : *state_l* → *state_l* → *Prop* :=

| *L_var* : ∀ *k t* $\mathscr{L}$ $\mathscr{L}'$ $\mathscr{L}_\mu$ $\mathscr{L}'_\mu$ $\mathscr{E}_\mu$ $\mathscr{E}'_\mu$ $\mathcal{S}$,

$$\frac{\mathscr{L}(k) = [t, \mathscr{L}', \mathscr{L}'_\mu, \mathscr{E}'_\mu]}{\langle\, `k`, \mathscr{L}, \mathscr{L}_\mu, \mathscr{E}_\mu, \mathcal{S}\rangle \rightsquigarrow^{l} \langle t, \mathscr{L}', \mathscr{L}'_\mu, \mathscr{E}'_\mu, \mathcal{S}\rangle}$$

| *L_app* : $\forall\ t\ u\ \mathcal{L}\ \mathcal{L}_\mu\ \mathcal{E}_\mu\ \mathcal{S}$,

$$\langle (t\,u), \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow^l \langle t, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu, [u, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu] :: \mathcal{S}\rangle$$

| *L_abs* : $\forall\ t\ c\ \mathcal{L}\ \mathcal{L}_\mu\ \mathcal{E}_\mu\ S'$,

$$\langle \lambda t, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu, c :: S'\rangle \rightsquigarrow^l \langle t, (c :: \mathcal{L}), \mathcal{L}_\mu, \mathcal{E}_\mu, S'\rangle$$

| *L_catch* : $\forall\ t\ \mathcal{L}\ \mathcal{L}_\mu\ \mathcal{E}_\mu\ \mathcal{S}$,

$$\langle \textbf{get-context}\ t, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow^l \langle t, \mathcal{L}, (\mathcal{L} :: \mathcal{L}_\mu), (\mathcal{S} :: \mathcal{E}_\mu), \mathcal{S}\rangle$$

| *L_throw* : $\forall\ t\ \alpha\ \mathcal{L}\ \mathcal{L}'\ \mathcal{L}_\mu\ \mathcal{E}_\mu\ \mathcal{S}\ \mathcal{S}'$,

$$\frac{\mathcal{L}_\mu(\alpha) = \mathcal{L}' \qquad \mathcal{E}_\mu(\alpha) = \mathcal{S}'}{\langle \textbf{set-context}\ \alpha\ t, \mathcal{L}, \mathcal{L}_\mu, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow^l \langle t, \mathcal{L}', \mathcal{L}_\mu, \mathcal{E}_\mu, \mathcal{S}'\rangle}$$

where "$\sigma_1 \rightsquigarrow$ '^l' σ_2" := (*step_l* $\sigma_1\ \sigma_2$).

Lemma *step_l_unique* : $\forall\ \sigma_1\ \sigma_2\ \sigma_2'$, $\sigma_1 \rightsquigarrow^l \sigma_2 \to \sigma_1 \rightsquigarrow^l \sigma_2' \to \sigma_2 = \sigma_2'$.
Proof.
```
 introv. revert_last.
 intros y1 H H0. revert H0. revert y1.
 induction H;
   (intros y2 D2; inversion D2; repeat progress ( eauto || econstructor || gen_subst || crush )).
```
Qed.

Hint Resolve *step_l_unique* : *gen_subst_db*.

4 Bisimulations

4.1 Abstract machine for safe λ_{ct}-terms (with indirection tables)

4.1.1 Closure, environment, stack and state

Inductive *clos_i* :=
| *Cl_i* (*t* : *term*) (*n* : *nat*) ($\mathcal{I}$: *vector*) ($\mathcal{I}_\mu$: *table*) ($\mathcal{E}$: *list clos_i*) ($\mathcal{E}_\mu$: *list* (*list clos_i*)).

Definition *stack_i* := *list clos_i*.
Definition *c_env_i* := *list clos_i*.
Definition *k_env_i* := *list stack_i*.

Inductive *state_i* :=
| *St_i* (*t* : *term*) (*n* : *nat*) ($\mathcal{I}$: *vector*) ($\mathcal{I}_\mu$: *table*) ($\mathcal{E}$: *c_env_i*) ($\mathcal{E}_\mu$: *k_env_i*) ($\mathcal{S}$: *stack_i*).

Notation "$\langle$ *t* , *n* , $\mathcal{I}$, $\mathcal{I}_\mu$, $\mathcal{E}$, $\mathcal{E}_\mu$, $\mathcal{S}$ $\rangle$" := (*St_i t n* $\mathcal{I}$ $\mathcal{I}_\mu$ $\mathcal{E}$ $\mathcal{E}_\mu$ $\mathcal{S}$) (at level 80).
Notation "[*t* , *n* , $\mathcal{I}$, $\mathcal{I}_\mu$, $\mathcal{E}$, $\mathcal{E}_\mu$]" := (*Cl_i t n* $\mathcal{I}$ $\mathcal{I}_\mu$ $\mathcal{E}$ $\mathcal{E}_\mu$) (at level 80).

4.1.2 Evaluation rules

Reserved Notation "$\sigma_1 \rightsquigarrow$ '^i' σ_2" (at level 30).

Inductive *step_i* : *state_i* → *state_i* → *Prop* :=

| *I_var* : ∀ *n n′ l g t* $\mathcal{I}$ $\mathcal{I}'$ $\mathcal{I}_\mu$ $\mathcal{I}'_\mu$ $\mathcal{E}$ $\mathcal{E}'$ $\mathcal{E}_\mu$ $\mathcal{E}'_\mu$ $\mathcal{S}$,

$$\frac{n - \mathcal{I}(l) = g \qquad \mathcal{E}(g) = [t, n', \mathcal{I}', \mathcal{I}'_\mu, \mathcal{E}', \mathcal{E}'_\mu]}{\langle\, `l`, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow^i \langle t, n', \mathcal{I}', \mathcal{I}'_\mu, \mathcal{E}', \mathcal{E}'_\mu, \mathcal{S}\rangle}$$

| *I_app* : ∀ *n t u* $\mathcal{I}$ $\mathcal{I}_\mu$ $\mathcal{E}$ $\mathcal{E}_\mu$ $\mathcal{S}$,

$$\langle (t\,u), n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow^i \langle t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, [u, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu] :: \mathcal{S}\rangle$$

| *I_abs* : ∀ *n t c* $\mathcal{I}$ $\mathcal{I}_\mu$ $\mathcal{E}$ $\mathcal{E}_\mu$ *S′*,

$$\langle \lambda t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, c :: S'\rangle \rightsquigarrow^i \langle t, (Sn), ((Sn) :: \mathcal{I}), \mathcal{I}_\mu, c :: \mathcal{E}, \mathcal{E}_\mu, S'\rangle$$

| *I_catch* : ∀ *n t* $\mathcal{I}$ $\mathcal{I}_\mu$ $\mathcal{E}$ $\mathcal{E}_\mu$ $\mathcal{S}$,

$$\langle \textbf{get-context}\ t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow^i \langle t, n, \mathcal{I}, (\mathcal{I} :: \mathcal{I}_\mu), \mathcal{E}, (\mathcal{S} :: \mathcal{E}_\mu), \mathcal{S}\rangle$$

| *I_throw* : ∀ *n t α* $\mathcal{I}$ $\mathcal{I}'$ $\mathcal{I}_\mu$ $\mathcal{E}$ $\mathcal{E}_\mu$ $\mathcal{S}$ $\mathcal{S}'$,

$$\frac{\mathcal{I}_\mu(\alpha) = \mathcal{I}' \qquad \mathcal{E}_\mu(\alpha) = \mathcal{S}'}{\langle \textbf{set-context}\ \alpha\ t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}\rangle \rightsquigarrow^i \langle t, n, \mathcal{I}', \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}'\rangle}$$

where "$\sigma_1 \rightsquigarrow$ '^*i*' σ_2" := (*step_i* σ_1 σ_2).

Lemma *step_i_unique* : ∀ σ_1 σ_2 σ_2', $\sigma_1 \rightsquigarrow^i \sigma_2 \to \sigma_1 \rightsquigarrow^i \sigma_2' \to \sigma_2 = \sigma_2'$.
Proof.
```
 introv. revert_last.
 intros y1 H H0. revert H0. revert y1.
 induction H;
   (intros y2 D2; inversion D2; repeat progress ( eauto || econstructor || gen_subst || crush )).
```
Qed.

Hint Resolve *step_i_unique* : *gen_subst_db*.

4.2 Lock-step simulation (−)

Reserved Notation "*c* '^⋆=_*c*' *c′*" (at level 30).
Reserved Notation "$\mathcal{S}$ '^⋆=_*s*' $\mathcal{S}'$" (at level 30).
Reserved Notation "$\mathcal{E}$ '^⋆=_*e*' $\mathcal{E}'$" (at level 30).
Reserved Notation "$\mathcal{E}_\mu$ '^⋆=_*k*' $\mathcal{E}'_\mu$" (at level 30).

Inductive *translate_clos* : *clos_i* → *clos* → *Prop* :=
| *T_cl* : ∀ *n t u* $\mathcal{I}$ $\mathcal{I}_\mu$ $\mathcal{E}$ $\mathcal{E}'$ $\mathcal{E}_\mu$ $\mathcal{E}'_\mu$,

$$\frac{\downarrow_n^{\mathcal{I}, \mathcal{I}_\mu}(t) = u \qquad \mathcal{E}^\star =_e \mathcal{E}' \qquad \mathcal{E}_\mu^\star =_k \mathcal{E}'_\mu}{[t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu]^\star =_c [u, \mathcal{E}', \mathcal{E}'_\mu]}$$

where "*c* '^⋆=_*c*' *c′*" := (*translate_clos c c′*)

with *translate_stack* : *stack_i* → *stack* → *Prop* :=
| T_stack_1 :

$$nil^\star =_s nil$$

| T_stack_2 : ∀ *c c′* $\mathcal{S}$ $\mathcal{S}'$,

$$\frac{c^\star =_c c' \qquad \mathcal{S}^\star =_s \mathcal{S}'}{(c :: \mathcal{S})^\star =_s (c' :: \mathcal{S}')}$$

where "$\mathcal{S}$ '^$\star$=_s' $\mathcal{S}'$" := (*translate_stack* $\mathcal{S}$ $\mathcal{S}'$)

with *translate_c_env* : *c_env_i* → *c_env* → *Prop* :=
| *T_c_env*$_1$:

$$nil^\star =_e nil$$

| *T_c_env*$_2$: ∀ c c' $\mathcal{E}$ $\mathcal{E}'$,

$$\frac{c^\star =_c c' \qquad \mathcal{E}^\star =_e \mathcal{E}'}{(c :: \mathcal{E})^\star =_e (c' :: \mathcal{E}')}$$

where "$\mathcal{E}$ '^$\star$=_e' $\mathcal{E}'$" := (*translate_c_env* $\mathcal{E}$ $\mathcal{E}'$)

with *translate_k_env* : *k_env_i* → *k_env* → *Prop* :=
| *T_k_env*$_1$:

$$nil^\star =_k nil$$

| *T_k_env*$_2$: ∀ $\mathcal{S}$ $\mathcal{S}'$ $\mathcal{E}_\mu$ $\mathcal{E}'_\mu$,

$$\frac{\mathcal{S}^\star =_s \mathcal{S}' \qquad \mathcal{E}_\mu^\star =_k \mathcal{E}'_\mu}{(\mathcal{S} :: \mathcal{E}_\mu)^\star =_k (\mathcal{S}' :: \mathcal{E}'_\mu)}$$

where "$\mathcal{E}_\mu$ '^$\star$=_k' $\mathcal{E}'_\mu$" := (*translate_k_env* $\mathcal{E}_\mu$ $\mathcal{E}'_\mu$).

Reserved Notation "σ '^$\star$=_σ' σ'" (at level 30).

Inductive *translate_st* : *state_i* → *state* → *Prop* :=
| *T_st* : ∀ n t u $\mathcal{I}$ $\mathcal{I}_\mu$ $\mathcal{E}$ $\mathcal{E}'$ $\mathcal{E}_\mu$ $\mathcal{E}'_\mu$ $\mathcal{S}$ $\mathcal{S}'$,

$$\frac{[t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu]^\star =_c [u, \mathcal{E}', \mathcal{E}'_\mu] \qquad \mathcal{S}^\star =_s \mathcal{S}'}{\langle t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}\rangle^\star =_\sigma \langle u, \mathcal{E}', \mathcal{E}'_\mu, \mathcal{S}'\rangle}$$

where "σ '^$\star$=_σ' σ'" := (*translate_st* σ σ').

Scheme *translate_clos_mut* := **Induction for** *translate_clos* **Sort** *Prop*
  with *translate_stack_mut* := **Induction for** *translate_stack* **Sort** *Prop*
  with *translate_c_env_mut* := **Induction for** *translate_c_env* **Sort** *Prop*
  with *translate_k_env_mut* := **Induction for** *translate_k_env* **Sort** *Prop*.

Lemma *translate_clos_unique* : ∀ c c_2 c'_2, $c^\star =_c c_2 \to c^\star =_c c'_2 \to c_2 = c'_2$
  with *translate_stack_unique* : ∀ s s_2 s'_2, $s^\star =_s s_2 \to s^\star =_s s'_2 \to s_2 = s'_2$
  with *translate_c_env_unique* : ∀ e e_2 e'_2, $e^\star =_e e_2 \to e^\star =_e e'_2 \to e_2 = e'_2$
  with *translate_k_env_unique* : ∀ k k_2 k'_2, $k^\star =_k k_2 \to k^\star =_k k'_2 \to k_2 = k'_2$.
Proof.
 Tactic Notation "*apply_unique_tac*" tactic(l) :=
  (unique_tac
    (l (unique_spec_1 *translate_clos*)
      (unique_spec_1 *translate_stack*)
      (unique_spec_1 *translate_c_env*)
      (unique_spec_1 *translate_k_env*))).
 - *apply_unique_tac translate_clos_mut*.
 - *apply_unique_tac translate_stack_mut*.
 - *apply_unique_tac translate_c_env_mut*.
 - *apply_unique_tac translate_k_env_mut*.
Qed.

Hint Resolve *translate_clos_unique* : *gen_subst_db*.

Hint Resolve *translate_stack_unique* : *gen_subst_db*.
Hint Resolve *translate_c_env_unique* : *gen_subst_db*.
Hint Resolve *translate_k_env_unique* : *gen_subst_db*.

Lemma *translate_st_unique* : $\forall\ \sigma\ \sigma_2\ \sigma_2',\ \sigma^\star =_\sigma \sigma_2 \to \sigma^\star =_\sigma \sigma_2' \to \sigma_2 = \sigma_2'$.
Proof.
simple_unique_tac.
Qed.

Hint Resolve *translate_st_unique* : *gen_subst_db*.

4.2.1 Soundness

Lemma *fetch_sound* : $\forall\ \mathscr{E}\ \mathscr{E}',\ \mathscr{E}^\star =_e \mathscr{E}' \to \forall\ n\ c,\ \mathscr{E}(n) = c \to \exists c', c^\star =_c c' \wedge \mathscr{E}'(n) = c'$.
Proof.
```
intros ℰ ℰ' D1 n c D2.
revert ℰ' D1.
induction D2.
- intros ℰ' D1.
  inversion D1.
  exists c'.
  split.
  * assumption.
  * apply Fetch_1.
- intros ℰ' D1.
  inversion D1.
  apply IHD2 in H3.
  destruct H3.
  destruct H3.
  exists x.
  split.
  * exact H3.
  * apply Fetch_2.
    exact H4.
```
Qed.

Lemma *fetch_mu_sound* : $\forall\ n\ \mathcal{S}\ \mathscr{E}_\mu,\ \mathscr{E}_\mu(n) = \mathcal{S} \to \forall \mathscr{E}_\mu',\ \mathscr{E}_\mu^\star =_k \mathscr{E}_\mu' \to \exists \mathcal{S}',\ \mathcal{S}^\star =_s \mathcal{S}' \wedge \mathscr{E}_\mu'(n) = \mathcal{S}'$.
Proof.
```
intros n S ℰ_μ D2.
induction D2.
- intros ℰ'_μ D1.
  inversion D1.
  exists 𝒮'.
  {
    split.
    - assumption.
    - apply Fetch_1.
  }
- intros ℰ'_μ D1.
  inversion D1.
  apply IHD2 in H3.
  destruct H3.
  destruct H3.
  exists x.
  {
    split.
```

```
    - exact H3.
    - apply Fetch_2.
      exact H4.
  }
Qed.

Theorem soundness : ∀ σ_1 σ_2 σ_1', σ_1 ⇝^i σ_2 → σ_1^⋆ =_σ σ_1' → ∃σ_2', σ_1' ⇝ σ_2' ∧ σ_2^⋆ =_σ σ_2'.
Proof.
 intros σ_1 σ_2 σ_1' D1.
 induction D1.
 - intro D2.
   inversion D2.
   inversion H9.
   inversion H14.
   assert (H222 := (fetch_sound _ _ H21)).
   subst.
   apply H222 in H0.
   destruct H0.
   destruct x.
   exists (⟨t0, ℰ0, ℰ_μ0, 𝒮'⟩).
   {
    split.
   - apply K_var.
     inversion H24.
     inversion H.
     subst.
     assert (H71 := fetch_unique _ _ _ _ _ H7 H1).
     subst.
     assert (H82 := minus_unique _ _ _ _ H8 H2).
     subst.
     destruct H0.
     assumption.
   - subst.
     apply T_st.
     destruct H0.
     assumption.
     assumption.
   }
-  intro D2.
   inversion D2.
   inversion H7.
   inversion H12.
   subst.
   exists (⟨t', ℰ', ℰ_μ', [u', ℰ', ℰ_μ']::𝒮'⟩).
   {
    split.
   - apply K_app.
   - {
       apply T_st.
     - apply T_cl.
       assumption.
       assumption.
       assumption.
     - apply T_stack_2.
       apply T_cl.
       assumption.
```

```
      assumption.
      assumption.
      assumption.
    }
  }
- intro D2.
  inversion D2.
  inversion H7.
  inversion H12.
  inversion H8.
  subst.
  exists (⟨t', (c'::𝓔'), 𝓔_μ', 𝓢'0⟩).
  split.
  * apply K_abs.
  * apply T_st.
    + apply T_cl.
      assumption.
      apply T_c_env_2.
      assumption.
      assumption.
      assumption.
    + assumption.
- intro D2.
  inversion D2.
  inversion H7.
  inversion H12.
  subst.
  exists (⟨t', 𝓔', (𝓢'::𝓔_μ'), 𝓢'⟩).
  split.
  * apply K_catch.
  *  apply T_st.
    + apply T_cl.
      assumption.
      assumption.
      apply T_k_env_2.
      assumption.
      assumption.
    + assumption.
- intro D2.
  inversion D2.
  inversion H9.
  inversion H14.
  subst.
  assert (H333 := (fetch_mu_sound _ _ _ H0 _ H22)).
  destruct H333 as [ 𝓢2 H444 ].
  destruct H444.
  exists (⟨t', 𝓔', 𝓔_μ', 𝓢2⟩).
  split.
  * apply K_throw.
    assumption.
  * apply T_st.
    + apply T_cl.
      assert (V2 := fetch_unique _ _ _ _ _ H25 H).
      subst.
      assumption.
      assumption.
```

```
    assumption.
  + assumption.
Qed.
```

4.2.2 Completeness

Lemma *fetch_complete* : $\forall\ \mathscr{E}\ \mathscr{E}',\ \mathscr{E}^\star =_e \mathscr{E}' \to \forall\ n\ c',\ \mathscr{E}'(n) = c' \to \exists c, c^\star =_c c' \wedge \mathscr{E}(n) = c.$
Proof.

```
 intros E E' D1 n c D2.
 revert E D1.
 induction D2.
 - intros E D1.
   inversion D1.
   exists c.
   split.
   * assumption.
   * apply Fetch_1.
 - intros E D1.
   inversion D1.
   apply IHD2 in H3.
   destruct H3.
   destruct H3.
   exists x.
   split.
   * exact H3.
   * apply Fetch_2.
     exact H4.
Qed.
```

Lemma *fetch_mu_complete* : $\forall\ n\ \mathcal{S}'\ \mathscr{E}'_\mu,\ \mathscr{E}'_\mu(n) = \mathcal{S}' \to \forall \mathscr{E}_\mu, \mathscr{E}^\star_\mu =_k \mathscr{E}'_\mu \to \exists \mathcal{S}, \mathcal{S}^\star =_s \mathcal{S}' \wedge \mathscr{E}_\mu(n) = \mathcal{S}.$
Proof.

```
 intros n S E'_μ D2.
 induction D2.
 - intros E_μ D1.
   inversion D1.
   exists S.
   {
    split.
    - assumption.
    - apply Fetch_1.
   }
 - intros E_μ D1.
   inversion D1.
   apply IHD2 in H3.
   destruct H3.
   destruct H3.
   exists x.
   {
    split.
    - exact H3.
    - apply Fetch_2.
      exact H4.
   }
Qed.
```

Theorem *completeness* : $\forall\ \sigma_1'\ \sigma_2'\ \sigma_1,\ \sigma_1' \rightsquigarrow \sigma_2' \rightarrow\ \sigma_1^\star =_\sigma \sigma_1' \rightarrow\ \exists \sigma_2,\ \sigma_1 \rightsquigarrow^i \sigma_2 \wedge\ \sigma_2^\star =_\sigma \sigma_2'$.
Proof.

```
 intros σ_1' σ_2' σ_1 D1.
 induction D1.
 - intro D2.
   inversion D2.
   inversion H3.
   inversion H10.
   assert (H222 := (fetch_complete _ _ H17)).
   subst.
   apply H222 in H.
   destruct H.
   destruct x.
   exists (⟨t0, n0, ℐ0, ℐ_μ0, ℰ1, ℰ_μ1, 𝒮0⟩).
   {
    split.
   - eapply I_var.
     exact H24.
     inversion H.
     subst.
     exact H1.
   - subst.
     apply T_st.
     destruct H.
     assumption.
     assumption.
   }
 - intro D2.
   inversion D2.
   inversion H2.
   inversion H9.
   subst.
   exists (⟨t2, n, ℐ, ℐ_μ, ℰ0, ℰ_μ0, [u2, n, ℐ, ℐ_μ, ℰ0, ℰ_μ0]::𝒮0⟩).
   {
    split.
    - apply I_app.
    - {
        apply T_st.
        - apply T_cl.
          assumption.
          assumption.
          assumption.
        - apply T_stack_2.
          apply T_cl.
          assumption.
          assumption.
          assumption.
          assumption.
      }
   }
 - intro D2.
   inversion D2.
   inversion H2.
   inversion H9.
   inversion H5.
   subst.
```

```
  exists (⟨t2, (S n), (S n)::𝒥, 𝒥_μ, (c0::ℰ0), ℰ_μ0, 𝒮1⟩).
  {
   split.
   - apply I_abs.
   - {
      apply T_st.
      - apply T_cl.
        assumption.
        apply T_c_env_2.
        assumption.
        assumption.
        assumption.
      - assumption.
     }
  }
- intro D2.
  inversion D2.
  inversion H2.
  inversion H9.
  subst.
  exists (⟨t2, n, 𝒥, (𝒥::𝒥_μ), ℰ0, (𝒮0::ℰ_μ0), 𝒮0⟩).
  {
   split.
   - apply I_catch.
   - {
      apply T_st.
      - apply T_cl.
        assumption.
        assumption.
        apply T_k_env_2.
        assumption.
        assumption.
      - assumption.
     }
  }
- intro D2.
  inversion D2.
  inversion H3.
  inversion H10.
  subst.
  assert (H333 := (fetch_mu_complete _ _ _ H _ H18)).
  destruct H333 as [ 𝒮2 H444 ].
  destruct H444.
  exists (⟨t2, n, 𝒥', 𝒥_μ, ℰ0, ℰ_μ0, 𝒮2⟩).
  {
   split.
   - apply I_throw.
     assumption.
     assumption.
   - {
      apply T_st.
      - apply T_cl.
        subst.
        assumption.
        assumption.
        assumption.
```

```
      - assumption.
    }
  }
Qed.
```

4.3 Lock-step simulation $(-)^{\diamond}$

Reserved Notation "$\mathscr{E}$ (n − k) = c" (at level 30).

Inductive *compute_l* : *c_env_i* → *nat* → *nat* → *clos_i* → *Prop* :=

| *Compute_l*$_1$: ∀ $\mathscr{E}$ n g k c,

$$\frac{n-k=g \qquad \mathscr{E}(g)=c}{\mathscr{E}(n-k)=c}$$

where "$\mathscr{E}$ (n − k) = c" := (*compute_l* $\mathscr{E}$ n k c).

Lemma *compute_l_unique* : ∀ $\mathscr{E}$ n k c c', $\mathscr{E}(n-k)=c \to \mathscr{E}(n-k)=c' \to c=c'$.
Proof.
```
 introv. revert_last.
 intros y1 H H0. revert H0. revert y1.
 induction H;
  (intros y2 D2; inversion D2; repeat progress ( eauto || econstructor || gen_subst || crush )).
```
Qed.

Hint Resolve *compute_l_unique* : *gen_subst_db*.

Reserved Notation "c '^◇=_c' c''" (at level 30).
Reserved Notation "$\mathscr{S}$ '^◇=_s' $\mathscr{S}''$" (at level 30).
Reserved Notation "$\mathscr{E}_\mu$ '^◇=_k' $\mathscr{E}'_\mu$" (at level 30).

Inductive *flatten* : *nat* → *c_env_i* → *vector* → *l_env_l* →*Prop* :=
| *T_flatten*$_1$: ∀ n $\mathscr{E}$,

$$\mathit{flatten}\ n\ \mathscr{E}\ \mathit{nil}\ \mathit{nil}$$

| *T_flatten*$_2$: ∀ c c' n k $\mathscr{E}$ $\mathscr{I}$ $\mathscr{L}$,

$$\frac{\mathscr{E}(n-k)=c \qquad c^{\diamond}=_c c' \qquad \mathit{flatten}\ n\ \mathscr{E}\ \mathscr{I}\ \mathscr{L}}{\mathit{flatten}\ n\ \mathscr{E}\ (k::\mathscr{I})\ (c'::\mathscr{L})}$$

with *map_flatten* : *nat* → *c_env_i* → *table* → *m_env_l* →*Prop* :=
| *T_map_flatten*$_1$: ∀ n $\mathscr{E}$,

$$\mathit{map\_flatten}\ n\ \mathscr{E}\ \mathit{nil}\ \mathit{nil}$$

| *T_map_flatten*$_2$: ∀ n $\mathscr{E}$ $\mathscr{I}$ $\mathscr{L}$ $\mathscr{I}_\mu$ $\mathscr{L}_\mu$,

$$\frac{\mathit{flatten}\ n\ \mathscr{E}\ \mathscr{I}\ \mathscr{L} \qquad \mathit{map\_flatten}\ n\ \mathscr{E}\ \mathscr{I}_\mu\ \mathscr{L}_\mu}{\mathit{map\_flatten}\ n\ \mathscr{E}\ (\mathscr{I}::\mathscr{I}_\mu)\ (\mathscr{L}::\mathscr{L}_\mu)}$$

with *translate_clos_l* : *clos_i* → *clos_l* → *Prop* :=
| *T_cl_l* : ∀ n t $\mathscr{I}$ $\mathscr{I}_\mu$ $\mathscr{L}$ $\mathscr{L}_\mu$ $\mathscr{E}$ $\mathscr{E}_\mu$ $\mathscr{E}'_\mu$,

$$\frac{\mathit{flatten}\ n\ \mathscr{E}\ \mathscr{I}\ \mathscr{L} \qquad \mathit{map\_flatten}\ n\ \mathscr{E}\ \mathscr{I}_\mu\ \mathscr{L}_\mu \qquad \mathscr{E}^{\diamond}_\mu=_k \mathscr{E}'_\mu}{[t,n,\mathscr{I},\mathscr{I}_\mu,\mathscr{E},\mathscr{E}_\mu]^{\diamond}=_c [\,t,\mathscr{L},\mathscr{L}_\mu,\mathscr{E}'_\mu]}$$

where "c '^◇=_c' c''" := (*translate_clos_l* c c')

with *translate_stack_l* : *stack_i* → *stack_l* → *Prop* :=
| *T_stack_l*$_1$:

$$nil^\diamond =_s nil$$

| *T_stack_l*$_2$: ∀ *c c′ 𝒮 𝒮′*,

$$\frac{c^\diamond =_c c' \qquad \mathcal{S}^\diamond =_s \mathcal{S}'}{(c :: \mathcal{S})^\diamond =_s (c' :: \mathcal{S}')}$$

where "*𝒮* '^⋄=_s' *𝒮′*" := (*translate_stack_l 𝒮 𝒮′*)

with *translate_k_env_l* : *k_env_i* → *k_env_l* → *Prop* :=
| *T_k_env_l*$_1$:

$$nil^\diamond =_k nil$$

| *T_k_env_l*$_2$: ∀ *𝒮 𝒮′ ℰ*$_\mu$ *ℰ*$'_\mu$,

$$\frac{\mathcal{S}^\diamond =_s \mathcal{S}' \qquad \mathcal{E}_\mu^\diamond =_k \mathcal{E}'_\mu}{(\mathcal{S} :: \mathcal{E}_\mu)^\diamond =_k (\mathcal{S}' :: \mathcal{E}'_\mu)}$$

where "*ℰ*$_\mu$ '^⋄=_k' *ℰ*$'_\mu$" := (*translate_k_env_l ℰ*$_\mu$ *ℰ*$'_\mu$).

Reserved Notation "*σ* '^⋄=_σ' *σ′*" (at level 30).

Inductive *translate_st_l* : *state_i* → *state_l* → *Prop* :=
| *T_st_l* : ∀ *n t u ℐ ℐ*$_\mu$ *ℰ 𝒟 𝒟*$_\mu$ *ℰ*$_\mu$ *ℰ*$'_\mu$ *𝒮 𝒮′*,

$$\frac{[t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu]^\diamond =_c [u, \mathcal{D}, \mathcal{D}_\mu, \mathcal{E}'_\mu] \qquad \mathcal{S}^\diamond =_s \mathcal{S}'}{\langle t, n, \mathcal{I}, \mathcal{I}_\mu, \mathcal{E}, \mathcal{E}_\mu, \mathcal{S}\rangle^\diamond =_\sigma \langle u, \mathcal{D}, \mathcal{D}_\mu, \mathcal{E}'_\mu, \mathcal{S}'\rangle}$$

where "*σ* '^⋄=_σ' *σ′*" := (*translate_st_l σ σ′*).

Scheme *flatten_mut* := **Induction for** *flatten* **Sort** *Prop*
  with *map_flatten_mut* := **Induction for** *map_flatten* **Sort** *Prop*
  with *translate_clos_l_mut* := **Induction for** *translate_clos_l* **Sort** *Prop*
  with *translate_stack_l_mut* := **Induction for** *translate_stack_l* **Sort** *Prop*
  with *translate_k_env_l_mut* := **Induction for** *translate_k_env_l* **Sort** *Prop*.

Lemma *flatten_unique* : ∀ *n ℰ ℐ 𝒟 𝒟′*, *flatten n ℰ ℐ 𝒟* → *flatten n ℰ ℐ 𝒟′* → *𝒟* = *𝒟′*
  with *map_flatten_unique* : ∀ *n ℰ ℐ*$_\mu$ *𝒟*$_\mu$ *𝒟*$'_\mu$,
    map_flatten n ℰ ℐ$_\mu$ *𝒟*$_\mu$ → *map_flatten n ℰ ℐ*$_\mu$ *𝒟*$'_\mu$ → *𝒟*$_\mu$ = *𝒟*$'_\mu$
  with *translate_clos_l_unique* : ∀ *c c*$_2$ *c*$'_2$, *c*$^\diamond$ =$_c$ *c*$_2$ → *c*$^\diamond$ =$_c$ *c*$'_2$ → *c*$_2$ = *c*$'_2$
  with *translate_stack_l_unique* : ∀ *s s*$_2$ *s*$'_2$, *s*$^\diamond$ =$_s$ *s*$_2$ → *s*$^\diamond$ =$_s$ *s*$'_2$ → *s*$_2$ = *s*$'_2$
  with *translate_k_env_l_unique* : ∀ *k k*$_2$ *k*$'_2$, *k*$^\diamond$ =$_k$ *k*$_2$ → *k*$^\diamond$ =$_k$ *k*$'_2$ → *k*$_2$ = *k*$'_2$.
Proof.
 Tactic Notation "*translate_l_unique_tac*" tactic(L) :=
  repeat clear_last;
  introv; revert_last;
  intros y1 H H0; revert H0; revert H; revert y1;
  apply (L
    (**fun** *n ℰ ℐ 𝒟 h* => ∀ *𝒟′*, *flatten n ℰ ℐ 𝒟′* → *𝒟* = *𝒟′*)
    (**fun** *n ℰ ℐ*$_\mu$ *𝒟*$_\mu$ *h* => ∀ *𝒟*$'_\mu$, *map_flatten n ℰ ℐ*$_\mu$ *𝒟*$'_\mu$ → *𝒟*$_\mu$ = *𝒟*$'_\mu$)
    (**fun** *c c*$_2$ *h* => ∀ *c*$'_2$, *c*$^\diamond$ =$_c$ *c*$'_2$ → *c*$_2$ = *c*$'_2$)
    (**fun** *s s*$_2$ *h* => ∀ *s*$'_2$, *s*$^\diamond$ =$_s$ *s*$'_2$ → *s*$_2$ = *s*$'_2$)
    (**fun** *k k*$_2$ *h* => ∀ *k*$'_2$, *k*$^\diamond$ =$_k$ *k*$'_2$ → *k*$_2$ = *k*$'_2$));
  (intros; revert_last; intro D2; inversion D2;

```
    repeat progress ( eauto || econstructor || gen_subst || crush )).
  - translate_l_unique_tac flatten_mut.
  - translate_l_unique_tac map_flatten_mut.
  - translate_l_unique_tac translate_clos_l_mut.
  - translate_l_unique_tac translate_stack_l_mut.
  - translate_l_unique_tac translate_k_env_l_mut.
Qed.

Hint Resolve flatten_unique : gen_subst_db.
Hint Resolve map_flatten_unique : gen_subst_db.
Hint Resolve translate_clos_l_unique : gen_subst_db.
Hint Resolve translate_stack_l_unique : gen_subst_db.
Hint Resolve translate_k_env_l_unique : gen_subst_db.

Lemma translate_st_l_unique : ∀ σ σ2 σ2', σ° =σ σ2 → σ° =σ σ2' → σ2 = σ2'.
Proof.
  introv. revert_last.
  intros y1 H H0. revert H0. revert y1.
  induction H;
    (intros y2 D2; inversion D2; repeat progress ( eauto || econstructor || gen_subst || crush )).
Qed.

Hint Resolve translate_st_l_unique : gen_subst_db.
```

4.3.1 Soundness

```
Lemma minus_next : ∀ n k g, n − k = g → Sn − k = Sg.
Proof.
  intros n k g 𝒟1.
  induction 𝒟1.
  - apply Minus_1.
  - apply Minus_2.
    assumption.
Qed.

Lemma weaken_flatten : ∀ {n ℰ ℐ 𝒮}, ∀ c', flatten n ℰ ℐ 𝒮 → flatten (Sn) (c' :: ℰ) ℐ 𝒮.
Proof.
  intros n ℰ ℐ 𝒮 c' 𝒟_1.
  induction 𝒟_1.
  - apply T_flatten_1.
  - inversion_clear H.
    assert (𝒟_4 := minus_next _ _ _ H1).
    {
      eapply T_flatten_2.
      - {
          eapply Compute_l_1.
          - exact 𝒟_4.
          - apply Fetch_2.
            exact H2.
        }
      - assumption.
      - assumption.
    }
Qed.

Lemma weaken_map_flatten : ∀ {n ℰ ℐμ 𝒮μ}, ∀ c',
```

map_flatten n $\mathscr{E}$ $\mathscr{I}_\mu$ $\mathscr{L}_\mu$ → map_flatten (Sn) (c'::$\mathscr{E}$) $\mathscr{I}_\mu$ $\mathscr{L}_\mu$.

Proof.
intros *n $\mathscr{E}$ $\mathscr{I}_\mu$ $\mathscr{L}_\mu$ c' $\mathscr{D}$_1*.
induction $\mathscr{D}$_1.
- apply T_map_flatten_1.
- eapply weaken_flatten in H.
  {
   apply T_map_flatten_2.
   - exact H.
   - assumption.
  }

Qed.

Lemma *map_flatten_sound* : ∀ *n $\mathscr{E}$ $\mathscr{I}'$ $\mathscr{I}_\mu$ α*,
$\mathscr{I}_\mu$(α)=$\mathscr{I}'$ → ∀ $\mathscr{L}_\mu$, map_flatten n $\mathscr{E}$ $\mathscr{I}_\mu$ $\mathscr{L}_\mu$ → ∃ $\mathscr{L}'$, flatten n $\mathscr{E}$ $\mathscr{I}'$ $\mathscr{L}'$ ∧ $\mathscr{L}_\mu$(α)=$\mathscr{L}'$.

Proof.
intros n $\mathscr{E}$ $\mathscr{I}$' $\mathscr{I}$_μ α $\mathscr{D}$_2.
induction $\mathscr{D}$_2.
- intros $\mathscr{L}$_μ $\mathscr{D}$_1.
  inversion $\mathscr{D}$_1; subst.
  exists $\mathscr{L}$.
  split.
  exact H3.
  apply Fetch_1.
- intros $\mathscr{L}$_μ $\mathscr{D}$_1.
  inversion $\mathscr{D}$_1; subst.
  assert (D34 := IH$\mathscr{D}$_2 _ H5).
  destruct D34.
  exists x.
  destruct H.
  split.
  exact H.
  apply Fetch_2.
  assumption.

Qed.

Lemma *fetch_sound_l* : ∀ *n l k c $\mathscr{I}$*,
$\mathscr{I}(l)=k \to \forall \mathscr{E}, \mathscr{E}(n-k)=c \to \forall \mathscr{L}$, flatten n $\mathscr{E}$ $\mathscr{I}$ $\mathscr{L}$ → ∃c', c°=$_c$ c' ∧ $\mathscr{L}(l)=c'$.

Proof.
intros *n l k c $\mathscr{I}$* D1.
induction D1.
- intros $\mathscr{E}$' D2 $\mathscr{L}$ D3.
  inversion D3.
  inversion D2.
  inversion H1.
  subst.
  exists c'.
  assert (H666 := minus_unique _ _ _ _ H7 H13).
  subst.
  assert (H555 := fetch_unique _ _ _ _ _ H8 H14).
  subst.
  {
   split.
   - assumption.
   - apply Fetch_1.
  }

```
 - intros 𝓔' D2 𝓛 D3.
   inversion D3.
   subst.
   assert (H888 := IHD1 _ D2 _ H6).
   destruct H888.
   destruct H.
   exists x.
   {
    split.
    - assumption.
    - apply Fetch_2.
      assumption.
   }
Qed.

Lemma fetch_mu_sound_l : ∀ n 𝓢 𝓔_μ, 𝓔_μ(n) = 𝓢 → ∀𝓔'_μ, 𝓔^◇_μ =_k 𝓔'_μ → ∃𝓢', 𝓢^◇ =_s 𝓢' ∧ 𝓔'_μ(n) = 𝓢'.
Proof.
 intros n S 𝓔_μ D2.
 induction D2.
 - intros 𝓔'_μ D1.
   inversion D1.
   exists 𝓢'.
   {
    split.
    - assumption.
    - apply Fetch_1.
   }
 - intros 𝓔'_μ D1.
   inversion D1.
   apply IHD2 in H3.
   destruct H3.
   destruct H3.
   exists x.
   {
    split.
    - exact H3.
    - apply Fetch_2.
      exact H4.
   }
Qed.

Theorem soundness_l : ∀ σ_1 σ_2 σ'_1, σ_1 ~>^i σ_2 → σ^◇_1 =_σ σ'_1 → ∃σ'_2, σ'_1 ~>^l σ'_2 ∧ σ_2^◇ =_σ σ'_2.
Proof.
 intros σ_1 σ_2 σ_1' D1.
 induction D1.
 - intro D2.
   inversion D2.
   inversion H9.
   inversion H.
   subst.
   assert (H111 := Compute_l_1 _ _ _ _ _ H25 H0).
   assert (H222 := fetch_sound_l _ _ _ _ _ H24 _ H111 _ H14).
   destruct H222.
   destruct H1.
   destruct x.
   exists (⟨t0, 𝓛0, 𝓛_μ0, 𝓔_μ0, 𝓢'⟩).
```

```
 {
  split.
  - apply L_var.
    assumption.
  - subst.
    apply T_st_l.
    destruct H0.
    assumption.
    assumption.
    assumption.
 }
- intro D2.
 inversion D2.
 inversion H7.
 subst.
 exists (⟨t, 𝒮, 𝒮_μ, ℰ_μ', [u, 𝒮, 𝒮_μ, ℰ_μ']:: 𝒮'⟩).
 {
  split.
  - apply L_app.
  - {
      apply T_st_l.
      - apply T_cl_l.
        assumption.
        assumption.
        assumption.
      - apply T_stack_l_2.
        apply T_cl_l.
        assumption.
        assumption.
        assumption.
    assumption.
  }
 }
- intro D2.
 inversion D2.
 inversion H7.
 inversion H8.
 subst.
 exists (⟨t, (c':: 𝒮), 𝒮_μ, ℰ_μ', 𝒮'0⟩).
 {
  split.
  - apply L_abs.
  - {
      apply T_st_l.
      - {
          apply T_cl_l.
          - {
              eapply T_flatten_2.
              - {
                  eapply Compute_l_1.
                  - apply minus_id.
                  - apply Fetch_1.
                }
              - assumption.
              - apply weaken_flatten.
                assumption.
```

```
          }
        - apply weaken_map_flatten.
          assumption.
        - assumption.
      }
    - assumption.
  }
}
- intro D2.
  inversion D2.
  inversion H7.
  subst.
  exists (⟨t, 𝒮, (𝒮 :: ℒ_μ), (𝒮' :: ℰ_μ'), 𝒮'⟩).
  {
    split.
    - apply L_catch.
    - {
        apply T_st_l.
        - {
            apply T_cl_l.
            - assumption.
            - {
                apply T_map_flatten_2.
                - assumption.
                - assumption.
              }
            - {
                apply T_k_env_l_2.
                - assumption.
                - assumption.
              }
          }
        - assumption.
      }
  }
- intro D2.
  inversion D2.
  inversion H9.
  subst.
  assert (H333 := (fetch_mu_sound_l _ _ _ H0 _ H23)).
  destruct H333 as [ 𝒮2 H444 ].
  destruct H444.
  assert (H555 := (map_flatten_sound _ _ _ _ _ H _ H22)).
  destruct H555.
  destruct H3.
  exists (⟨t, x, 𝒮 ℒ_μ, ℰ_μ', 𝒮2⟩).
  {
    split.
    - {
        apply L_throw.
        - assumption.
        - assumption.
      }
    - {
        apply T_st_l.
        - {
```

```
          apply T_cl_l.
          - assumption.
          - assumption.
          - assumption.
        }
      - assumption.
    }
  }
Qed.
```

4.3.2 Completeness

Lemma *map_flatten_complete* : $\forall$ {$n\ \mathscr{E}\ \mathscr{L}'\ \mathscr{L}_\mu\ \alpha$},
$\mathscr{L}_\mu(\alpha)=\mathscr{L}' \rightarrow \forall\ \mathscr{I}_\mu$, *map_flatten* $n\ \mathscr{E}\ \mathscr{I}_\mu\ \mathscr{L}_\mu \rightarrow \exists\ \mathscr{I}'$, *flatten* $n\ \mathscr{E}\ \mathscr{I}'\ \mathscr{L}' \wedge \mathscr{I}_\mu(\alpha)=\mathscr{I}'$.
Proof.

```
 intros n ℰ ℒ' ℒ_μ α 𝒟_2.
 induction 𝒟_2.
 - intros ℐ_μ 𝒟_1.
   inverts 𝒟_1 as D11 D12.
   exists ℐ0.
   {
    split.
    - exact D11.
    - apply Fetch_1.
   }
 - intros ℐ_μ 𝒟_1.
   inverts 𝒟_1 as D11 D12.
   assert (D34 := IH𝒟_2 _ D12).
   destruct D34 as [ℐ' D34].
   exists ℐ'.
   destruct D34 as [ D3 D4 ].
   {
    split.
    - exact D3.
    - apply Fetch_2.
      exact D4.
   }
Qed.
```

Lemma *fetch_complete_l* : $\forall$ {$n\ l\ c'\ \mathscr{I}\ \mathscr{E}\ \mathscr{L}$},
flatten $n\ \mathscr{E}\ \mathscr{I}\ \mathscr{L} \rightarrow \mathscr{L}(l)=c' \rightarrow \exists\ k\ c,\ \mathscr{E}(n-k)=c\ \wedge\ c^{\diamond}=_c c' \wedge \mathscr{I}(l)=k$.
Proof.

```
 intros n l c' ℐ ℰ ℒ D1 D2.
 revert D1.
 revert ℐ.
 induction D2.
 - intros ℐ0 D1.
   inverts D1 as D31 D32 D33.
   exists k.
   exists c.
   {
    split.
    - exact D31.
    - {
       split.
```

```
        - exact D32.
        - apply Fetch_1.
      }
    }
  - intros 𝒥0 D1.
    inverts D1 as D31 D32 D33.
    assert (D2411 := IHD2 _ D33).
    destruct D2411 as [k0 D2411].
    destruct D2411 as [c0 D2411].
    exists k0 c0.
    destruct D2411 as [D20 D411].
    destruct D411 as [D4 D11].
    {
     split.
     - exact D20.
     - {
        split.
        - exact D4.
        - apply Fetch_2.
          exact D11.
      }
    }
Qed.

Lemma fetch_mu_complete_l : ∀ {n 𝒮′ ℰ′μ},
    ℰ′μ(n) = 𝒮′ → ∀ℰμ, ℰ°μ =k ℰ′μ → ∃𝒮, 𝒮° =s 𝒮′ ∧ ℰμ(n) = 𝒮.
Proof.
  intros n S ℰ′μ D2.
  induction D2.
  - intros ℰμ D1.
    inversion D1.
    exists 𝒮.
    {
     split.
     - assumption.
     - apply Fetch_1.
    }
  - intros ℰμ D1.
    inversion D1.
    apply IHD2 in H3.
    destruct H3.
    destruct H3.
    exists x.
    {
     split.
     - exact H3.
     - apply Fetch_2.
       exact H4.
    }
Qed.

Theorem completeness_l : ∀ σ1′ σ2′ σ1, σ1′ ⇝l σ2′ → σ1° =σ σ1′ → ∃σ2, σ1 ⇝i σ2 ∧ σ2° =σ σ2′.
Proof.
  intros σ1′ σ2′ σ1 D1.
  induction D1.
  - intro D2.
```

```
  inverts keep D2 as D21 D22.
  inverts keep D21 as D211 D212 D213.
  assert (H222 := fetch_complete_l D211 H).
  destruct H222 as [ k0 H222 ].
  destruct H222 as [ c H222 ].
  destruct H222 as [ D112_12 D31_111 ].
  inverts D112_12 as D112 D12.
  destruct D31_111 as [ D31 D111 ].
  destruct c as [ t0 n0 ℱ0 ℐ_μ0 ℰ0 ℰ_μ1 ].
  exists (⟨t0, n0, ℱ0, ℐ_μ0, ℰ0, ℰ_μ1, 𝒮0⟩).
  {
   split.
  - {
     eapply I_var.
     - {
        eapply Compute_1.
        - exact D111.
        - exact D112.
       }
     - exact D12.
    }
  - {
     apply T_st_l.
     - exact D31.
     - exact D22.
    }
  }
- intro D2.
  inverts D2 as D21 D22.
  inverts D21 as D211 D212 D213.
  exists (⟨t, n, ℱ, ℐ_μ, ℰ, ℰ_μ0, [u, n, ℱ, ℐ_μ, ℰ, ℰ_μ0]:: 𝒮0⟩).
  {
   split.
   - apply I_app.
   - {
      apply T_st_l.
      - {
         apply T_cl_l.
         - exact D211.
         - exact D212.
         - exact D213.
        }
      - {
         apply T_stack_l_2.
         {
          apply T_cl_l.
          - exact D211.
          - exact D212.
          - exact D213.
         }
         - exact D22.
        }
     }
  }
- intro D2.
  inverts D2 as D21 D22.
```

```
  inverts D21 as D211 D212 D213.
  inverts D22 as D221 D222.
  exists (⟨t, (Sn), (Sn)::ℐ, ℐ_μ, (c0::ℰ), ℰ_μ0, 𝒮0⟩).
  {
   split.
   - apply I_abs.
   - assert (D1 := minus_id (S n)).
     assert (D2 := weaken_flatten c0 D211).
     assert (D3 := weaken_map_flatten c0 D212).
     {
      apply T_st_l.
      - {
         apply T_cl_l.
         - {
            eapply T_flatten_2.
            - {
               eapply Compute_l_1.
               - exact D1.
               - apply Fetch_1.
              }
            - exact D221.
            - exact D2.
           }
         - exact D3.
         - exact D213.
        }
      - exact D222.
     }
  }
- intro D2.
  inverts D2 as D21 D22.
  inverts D21 as D211 D212 D213.
  exists (⟨t, n, ℐ, (ℐ::ℐ_μ), ℰ, (𝒮0::ℰ_μ0), 𝒮0⟩).
  {
   split.
   - apply I_catch.
   - {
      apply T_st_l.
      - {
         apply T_cl_l.
         - exact D211.
         - {
            eapply T_map_flatten_2.
            -  exact D211.
            -  exact D212.
           }
         - {
            apply T_k_env_l_2.
            - exact D22.
            - exact D213.
           }
        }
      - exact D22.
     }
  }
- intro D2.
```

```
  inverts D2 as D21 D22.
  inverts D21 as D211 D212 D213.
  assert (D32_12 := fetch_mu_complete_l H0 _ D213).
  assert (D5_11 := map_flatten_complete H _ D212).
  destruct D32_12 as [ 𝒮1 D32_12 ].
  destruct D32_12 as [ D32 D12 ].
  destruct D5_11 as [ ℱ1 D5_11 ].
  destruct D5_11 as [ D5 D11 ].
  exists (⟨t, n, ℱ1, ℱ_μ, ℰ, ℰ_μ0, 𝒮1⟩).
  {
   split.
   - {
       apply I_throw.
       - exact D11.
       - exact D12.
     }
   - {
       apply T_st_l.
       - {
           apply T_cl_l.
           - exact D5.
           - exact D212.
           - exact D213.
         }
       - exact D32.
     }
  }
Qed.
```

5 Conclusion and future work