Abstract Machines

\[
t_1 \xrightarrow{\beta} t_2 \xrightarrow{\beta} \ldots \xrightarrow{\beta} t_n
\]

\[
t_1 \downarrow t_n
\]

\[
* \xrightarrow{} * \xrightarrow{} * \xrightarrow{} \ldots \xrightarrow{} *
\]
A CEK Machine

inductive

machine :: "lam ⇒ ctx list ⇒ lam ⇒ ctx list ⇒ bool"

(\(\langle _,_\rangle \mapsto \langle _,_\rangle\))

where

\(m_1\): "\langle App e_1 e_2, Es\rangle \mapsto \langle e_1,(CAppL \Box e_2)#Es\rangle" \n\(m_2\): "val v \mapsto \langle v,(CAppL \Box e_2)#Es\rangle \mapsto \langle e_2,(CAppR v \Box)#Es\rangle" \n\(m_3\): "val v \mapsto \langle v,(CAppR (Lam [y].e) \Box)#Es\rangle \mapsto \langle e[y::=v],Es\rangle"

We proved:

- If \(t \Downarrow t'\) then \(\langle t,[]\rangle \mapsto^* \langle t',[]\rangle\).

- If \(\langle t,[]\rangle \mapsto^* \langle v,[]\rangle\) with \(v\) being a value, then \(t \Downarrow v\).

- However, this machine explicitly calculates the substitutions.
nominal datatype val =
  Clos "name" "env" ("Clos [__].__ __")
and env =
  empty |
  cons "name" "val" "env" ("'(_,_)##_")

datatype ctx =
  Hole ("□") |
  CAppL "ctx" "lam" "env" |
  CAppR "val" "ctx"

datatype conf =
  C1 "lam" "env" "ctx list" ("＜__,__,__＞") |
  C2 "ctx list" "val" ("＜__,__＞")
Environments

inductive
emachine :: "conf ⇒ conf ⇒ bool" ("_ ⊸ _")

where
em1: "lookup2 env x v ⊸ Var x, env, Es ⊸ Es, v"
| em2: "Lam [x].t , env, Es ⊸ Es, Clos [x].t env"
| em3: "App t1 t2, env, Es ⊸ t1, env, (CAppL □ t2 env)#Es"
| em4: "((CAppL □ t env)#Es, v) ⊸ t, env, (CAppR v □)#Es"
| em5: "((CAppR (Clos [x].t env) □)#Es, v) ⊸ t, (x,v)##env, Es"

inductive
lookup2 :: "env ⇒ name ⇒ val ⇒ bool"

where
"lookup2 ((x,v)#env) x v"
"[x ≠ y; lookup2 env x v] ⊸ lookup2 ((y,v'#)#env) x v"
lemma type_substitution:
  fixes $\Gamma::\text{(name \times ty) list}$
  assumes a: "$\Gamma \vdash e : T$'
  and b: "$(x,T') \in \text{set } \Gamma$''
  and c: "$\Gamma - [(x,T')] \vdash e' : T''$
  shows "$\Gamma - [(x,T')] \vdash e[x:=e'] : T''$
  using a b c
proof (nominal_induct $\Gamma$ e T avoiding: e' x
            rule: typing.strong_induct)
case (\(t1\) \(\Gamma\) \(y\) \(T\) \(e'\) \(x\))

have a1: "valid \(\Gamma\)" by fact
have a2: "\((y,T) \in \text{set} \ \Gamma\)" by fact
have a3: "\((x,T') \in \text{set} \ \Gamma\)" by fact
have a4: "\(\Gamma - [(x,T')] \vdash e' : T\)" by fact

{ assume eq: "x=y"
  from a1 a2 a3 have "T=T" using eq by (simp add: context_unique)
  with a4 have "\(\Gamma - [(x,T')] \vdash \text{Var}\ y[x::=e'] : T\)" using eq by simp
}

moreover
{ assume ineq: "x\neq y"
  from a1 have "valid (\(\Gamma - [(x,T')]\))" by (simp only: valid_diff)
  moreover
  from a3 have "\((y,T) \in \text{set} (\Gamma - [(x,T')])\)" using ineq by simp
  ultimately
  have "\(\Gamma - [(x,T')] \vdash \text{Var}\ y[x::=e'] : T\)" using ineq by auto
}

ultimately show "\(\Gamma - [(x,T')] \vdash \text{Var}\ y[x::=e'] : T\)" by blast
case (t3 y Γ T₁ ⊢ T₂ e' x)

have vc: "γ#x" "γ#e""γ#Γ" by fact
have a1: "(x, T') ∈ set Γ" by fact
have a2: "Γ - [(x,T')] ⊢ e' : T" by fact
have a3: "(y,T₁)#Γ ⊢ t : T₂" by fact
have ih: "[(x,T') ∈ set ((y,T₁)#Γ); ((y,T₁)#Γ) - [(x,T')] ⊢ e' : T']
            ⇒ ((y,T₁)#Γ) - [(x,T')] ⊢ (t[x::=e']) : T₂" by fact
have "(x,T') ∈ set ((y,T₁)#Γ)" using a1 vc by (simp add: fresh_atm)
moreover
have "valid ((y,T₁)#Γ)" using a3 by (simp add: typing_implies_valid)
then have a5: "valid ((y,T₁)#Γ) - [(x,T')]" by (simp only: valid_diff)
then have "((y,T₁)#Γ) - [(x,T')] ⊢ e' : T" using a2 by (auto intro: weakening)
ultimately have "((y,T₁)#Γ) - [(x,T')] ⊢ (t[x::=e']) : T₂" using ih by blast
then have "((y,T₁)#(Γ - [(x,T')]) ⊢ (t[x::=e']) : T₂" using vc
    by (simp add: fresh_atm)
moreover
have "γ#(Γ - [(x,T')])" using vc by (simp only: fresh_diff)
ultimately have "(Γ - [(x,T')]) ⊢ Lam [y].(t[x::=e']) : T₁ → T₂" by blast
then show "(Γ - [(x,T')]) ⊢ (Lam [y].t)[x::=e'] : T₁ → T₂" using vc by simp
lemma type_substitution:
  assumes a: "(Δ@[x,T'])@Γ ⊢ e : T"
  and b: "Γ ⊢ e' : T"
  shows "(Δ@Γ) ⊢ e[x::=e'] : T"
using a b
proof (nominal_induct Γ''≡"Δ@[x,T']@Γ" e T avoiding: x e' Δ
  rule: typing.strong_induct)

  case (t1 Γ'' y T x e' Δ)
  then have a1: "valid (Δ@[x,T']@Γ)"
      and a2: "(y,T) ∈ set (Δ@[x,T']@Γ)"
      and a3: "Γ ⊢ e' : T" by simp_all
  from a1 have a4: "valid (Δ@Γ)" by (rule valid_insert)
  { assume eq: "x=y"
    from a1 a2 have "T=T" using eq by (auto intro: context_unique)
    with a3 have "Δ@Γ ⊢ Var y[x::=e'] : T" using eq a4
      by (auto intro: weakening) }
  moreover
  { assume ineq: "x≠y"
    from a2 have "(y,T) ∈ set (Δ@Γ)" using ineq by simp
    then have "Δ@Γ ⊢ Var y[x::=e'] : T" using ineq a4 by auto }
  ultimately show "Δ@Γ ⊢ Var y[x::=e'] : T" by blast
qed (force simp add: fresh_list_append fresh_list_cons)
Written Exam

- On 15th July (I think).
- I expect that you can do an induction proof by hand, and define relations such as typing, small-step reduction, big-step evaluation etc.