Nominal Techniques
or, “The Real Thing”

Christian Urban (TU Munich)
http://isabelle.in.tum.de/nominal/

A Formalisation of a CK Machine:

_ ↓ _

CK
Nominal Techniques
or, “The Real Thing”

Christian Urban (TU Munich)
http://isabelle.in.tum.de/nominal/

A Formalisation of a CK Machine:

_ ⊥ _ → CK
Nominal Techniques
or, “The Real Thing”

Christian Urban (TU Munich)
http://isabelle.in.tum.de/nominal/

A Formalisation of a CK Machine:

_ ───┐
    |  CK
    │
_ ───┘

_ ───┐
    | cbv
    }
_ ───┘
Lambda-Terms

We build on the theory Nominal (which in turn builds on HOL). Nominal provides an infrastructure to reason with binders.

```
atom_decl name

nominal_datatype lam =
  Var "name"
| App "lam" "lam"
| Lam "<name>lam" ("Lam [__].__")
```
Lambda-Terms

- We build on the theory Nominal (which in turn builds on HOL). Nominal provides an infrastructure to reason with binders.

```plaintext
atom_decl name

nominal_datatype lam =
  Var "name"
| App "lam" "lam"
| Lam "«name»lam" ("Lam [_._.]"
```

- We allow more than one kind of atoms.
- At the moment we only support single, but nested binders (future: arbitrary binding structures).
```plaintext
datatype ctx =
  Hole ("□")
| CAppL "ctx" "lam"
| CAppR "lam" "ctx"
| CLam "name" "ctx" ("CLam [__].__")

fun
  filling :: "ctx ⇒ lam ⇒ lam" ("_[[\_]]")
where
  "□[t] = t"
| "(CAppL E t')[t] = App (E[t]) t""
| "(CAppR t' E)[t] = App t' (E[t])"
| "(CLam [x].E)[t] = Lam [x].(E[t])"

lemma alpha_test:
  shows "x ≠ y ⇒ (CLam [x].□) ≠ (CLam [y].□)"
and  "(CLam [x].□)[Var x] = (CLam [y].□)[Var y]"
by (simp_all add: ctx.inject lam.inject alpha swap_simps fresh_atm)
```
For our CK machines we actually do not need contexts for lambdas.

```haskell
datatype ctx =
    Hole ("☐")
    | CAppL "ctx" "lam"
    | CAppR "lam" "ctx"

fun
    filling :: "ctx ⇒ lam ⇒ lam" ("_[_]")
where
    "☐[t] = t"
    | "(CAppL E t')[t] = App (E[t]) t"
    | "(CAppR t' E)[t] = App t' (E[t])"
```
fun ctx-compose :: "ctx ⇒ ctx ⇒ ctx" ("_ ○ _")
where
  "□ ○ E' = E''"
| "(CAppL E t') ○ E' = CAppL (E ○ E') t''"
| "(CAppR t' E) ○ E' = CAppR t' (E ○ E')"

lemma ctx-compose:
  shows "(E₁ ○ E₂)[t] = E₁[E₂[t]]"
by (induct E₁ rule: ctx.induct) (simp_all)

types ctxs = "ctx list"

fun ctx-composes :: "ctxs ⇒ ctx" ("_ ↓")
where
  "[] ↓ = □"
| "(E#Es) ↓ = (Es ↓) ○ E"
fun ctx-compose :: "ctx ⇒ ctx ⇒ ctx" ("_ ◦ _")
where
"□ ◦ E' = E''"
| "(CAppL E t') ◦ E' = CAppL (E ◦ E') t''"
| "(CAppR t' E) ◦ E' = CAppR t' (E ◦ E')"

lemma ctx-compose:
shows "(E₁ ◦ E₂)[t] = E₁[E₂[t]]"
by (induct E₁ rule: ctx.induct) (simp_all)

Subgoals

1. □ ◦ E₂[t] = □[E₂[t]]
2. \(\forall \text{ctx lam. } \text{ctx} ◦ E₂[t] = \text{ctx}[E₂[t]] \implies \text{CAppL ctx lam ◦ E₂[t]} = \text{CAppL ctx lam}[E₂[t]]\)
3. \(\forall \text{lam ctx. } \text{ctx} ◦ E₂[t] = \text{ctx}[E₂[t]] \implies \text{CAppR lam ctx ◦ E₂[t]} = \text{CAppR lam ctx}[E₂[t]]\)
Context Composition

fun ctx_compose :: "ctx ⇒ ctx ⇒ ctx" ("_ ◦ _")
where
  "□ ◦ E' = E''"
| "(CAppL E t') ◦ E' = CAppL (E ◦ E') t'"
| "(CAppR t' E) ◦ E' = CAppR t' (E ◦ E')"

lemma ctx_compose:
  shows "(E_1 ◦ E_2)[t] = E_1[E_2[t]]"
by (induct E_1 rule: ctx.induct) (simp_all)

types ctxs = "ctx list"

fun ctx_composes :: "ctxs ⇒ ctx" ("_↓")
where
  "[]↓ = □"
| "(E#Es)↓ = (Es↓) ◦ E"
nominal_datatype $ty =$
  $tVar$ "string"
| $tArr$ "ty" "ty" ("_ $\rightarrow$ _")

$\text{types} \ ty\_\text{ctx} =$ "(name $\times$ ty) list"

abbreviation
"sub_ty\_ctx" :: "ty\_ctx $\Rightarrow$ ty\_ctx $\Rightarrow$ bool" ("_ $\subseteq$ _")

where
"$\Gamma_1 \subseteq \Gamma_2$ $\equiv$ $\forall x. \ x \in$ set $\Gamma_1$ $\longrightarrow$ $x \in$ set $\Gamma_2$"
Definition of Types

nominal_datatype ty =
    tVar "string"
| tArr "ty" "ty" ("_ → _")

types ty_ctx = "(name × ty) list"

abbreviation
    "sub_ty_ctx" :: "ty_ctx ⇒ ty_ctx ⇒ bool" ("_ ⊆ _")
where
    "Γ₁ ⊆ Γ₂ ≡ ∀ x. x ∈ set Γ₁ → x ∈ set Γ₂"

We can overload ⊆, but this might mean we have to give explicit type-annotations so that Isabelle can figure out what is meant.
Typing Judgements

inductive

valid :: "ty_ctx ⇒ bool"

where

v1: "valid []"
| v2: "[valid Γ; x#Γ] ⇒ valid ((x,T)#Γ)"

inductive

typing :: "ty_ctx ⇒ lam ⇒ ty ⇒ bool" ("_ ⊢ _ : _")

where

t_Var: "[valid Γ; (x,T) ∈ set Γ] ⇒ Γ ⊢ Var x : T"
| t_App: "[Γ ⊢ t₁ : T₁ → T₂; Γ ⊢ t₂ : T₁] ⇒ Γ ⊢ App t₁ t₂ : T₂"
| t_Lam: "[x#Γ; (x,T₁)#Γ ⊢ t : T₂] ⇒ Γ ⊢ Lam [x].t : T₁ → T₂"
**Typing Judgements**

\[
\begin{align*}
\text{valid } \Gamma & \quad (x, T) \in \text{set } \Gamma \\
\Gamma \vdash t_1 : T_1 \rightarrow T_2 & \quad \Gamma \vdash t_2 : T_1 \\
\Gamma \vdash \text{Var } x : T & \quad \Gamma \vdash \text{App } t_1 \ t_2 : T_2 \\
x \notin \Gamma & \quad (x, T_1):\Gamma \vdash t : T_2 \\
\Gamma \vdash \text{Lam } [x].t : T_1 \rightarrow T_2 &
\end{align*}
\]

inductive

**valid** :: "\(\text{valid } \Gamma \); \((x, T) \in \text{set } \Gamma\) \[\Rightarrow\] \text{valid } \Gamma"

where

- \(v_1: "\text{valid } \Gamma \); \((x, T) \in \text{set } \Gamma\) \[\Rightarrow\] \text{valid } \Gamma"
- \(v_2: "\text{valid } \Gamma \); \((x, T) \in \text{set } \Gamma\) \[\Rightarrow\] \text{valid } \Gamma"

inductive typing :: "\text{ty}_\text{ctx} \Rightarrow \text{lam} \Rightarrow \text{ty} \Rightarrow \text{bool}" ("\_ \vdash \_ : \_"")

where

- \(t\_\text{Var}: "\text{valid } \Gamma \); \((x, T) \in \text{set } \Gamma\) \[\Rightarrow\] \text{valid } \Gamma \vdash \text{Var } x : T"
- \(t\_\text{App}: "\text{valid } \Gamma \); \((x, T_1) \in \text{set } \Gamma\) \[\Rightarrow\] \text{valid } \Gamma \vdash \text{App } t_1 \ t_2 : T_2"
- \(t\_\text{Lam}: "\text{valid } \Gamma \); \((x, T_1) \in \text{set } \Gamma\) \[\Rightarrow\] \text{valid } \Gamma \vdash \text{Lam } [x].t : T_1 \rightarrow T_2"
Typing Judgements

inductive
valid :: "ty_ctx ⇒ bool"
where
  v₁: "valid []"
| v₂: "[valid Γ; x#Γ] → valid ((x,T)#Γ)"

inductive
typing :: "ty_ctx ⇒ lam ⇒ ty ⇒ bool" ("_ ⊢ _ : _")
where
  t_Var: "[valid Γ; (x,T) ∈ set Γ] → Γ ⊢ Var x : T"
| t_App: "[Γ ⊢ t₁ : T₁ → T₂; Γ ⊢ t₂ : T₁] → Γ ⊢ App t₁ t₂ : T₂"
| t_Lam: "[x#Γ; (x,T₁)#Γ ⊢ t : T₂] → Γ ⊢ Lam [x].t : T₁ → T₂"

declare typing.intros[intro] valid.intros[intro]
Typing Judgements

We want to have the strong induction principle for the typing judgement.

1.) The relation needs to be equivariant.

inductive
valid :: "ty_ctx ⇒ bool"
where
v_1: "valid []"
| v_2: "[valid Γ; x#T] ⇒ valid ((x,T)#Γ)"

inductive
typing :: "ty_ctx ⇒ lam ⇒ ty ⇒ bool" ("_ ⊢ _ : _")
where
t_Var: "[valid Γ; (x,T) ∈ set Γ] ⇒ Γ ⊢ Var x : T"
| t_App: "[Γ ⊢ t_1 : T_1 → T_2; Γ ⊢ t_2 : T_1] ⇒ Γ ⊢ App t_1 t_2 : T_2"
| t_Lam: "[x#Γ; (x,T_1)#Γ ⊢ t : T_2] ⇒ Γ ⊢ Lam [x].t : T_1 → T_2"

declare typing.intros[intro] valid.intros[intro]
Typing Judgements

inductive valid :: "ty_ctx ⇒ bool"
where
  v₁: "valid []"
| v₂: "[valid Γ; x#Γ] ⇒ valid ((x,T)#Γ)"

inductive typing :: "ty_ctx ⇒ lam ⇒ ty ⇒ bool" ("_ ⊢ _ : _")
where
  t_Var: "[valid Γ; (x,T) ∈ set Γ] ⇒ Γ ⊢ Var x : T"
| t_App: "[Γ ⊢ t₁ : T₁ → T₂; Γ ⊢ t₂ : T₁] ⇒ Γ ⊢ App t₁ t₂ : T₂"
| t_Lam: "[x#Γ; (x,T₁)#Γ ⊢ t : T₂] ⇒ Γ ⊢ Lam [x].t : T₁ → T₂"

declare typing.intros[intro] valid.intros[intro]
equivariance valid
equivariance typing
Typing Judgements

Inductive
valid :: "ty_ctx ⇒ |
where
  v₁: "valid []"
| v₂: "[valid Γ; x#Γ] ⇒ valid ((x,T)#Γ)"

Inductive
typing :: "ty_ctx ⇒ lam ⇒ ty ⇒ bool" ("_ ⊢ _ : _")
where
t_Var: "[valid Γ; (x,T) ∈ set Γ] ⇒ Γ ⊢ Var x : T"
| t_App: "[Γ ⊢ t₁ : T₁ → T₂; Γ ⊢ t₂ : T₁] ⇒ Γ ⊢ App t₁ t₂ : T₂"
| t_Lam: "[x#Γ; (x,T₁)#Γ ⊢ t : T₂] ⇒ Γ ⊢ Lam [x].t : T₁ → T₂"

Declare
typing.intros[intro]
valid.intros[intro]
equivariance valid
equivariance typing
Typing Judgements (2)

**inductive**

typing :: "ty_ctx \Rightarrow lam \Rightarrow ty \Rightarrow bool" ("_ \vdash _ : _")

**where**

\[ t_{\text{Var}}: \left[ \begin{array}{l} \text{valid } \Gamma; (x, T) \in \text{set } \Gamma \end{array} \right] \Rightarrow \Gamma \vdash \text{Var } x : T \]

\[ t_{\text{App}}: \left[ \begin{array}{l} \Gamma \vdash t_1 : T_1 \to T_2; \Gamma \vdash t_2 : T_1 \end{array} \right] \Rightarrow \Gamma \vdash \text{App } t_1 \ t_2 : T_2 \]

\[ t_{\text{Lam}}: \left[ \begin{array}{l} x \# \Gamma; (x, T_1) \# \Gamma \vdash t : T_2 \end{array} \right] \Rightarrow \Gamma \vdash \text{Lam } [x].t : T_1 \to T_2 \]

**nominal_inductive** typing
Typing Judgements (2)

inductive
typing :: "ty_ctx ⇒ lam ⇒ ty ⇒ bool" ("_ ⊢ _ : _")

where
t_Var: "\[\text{valid } \Gamma; (x, T) \in \text{set } \Gamma\] \implies \Gamma \vdash \text{Var } x : T"
| t_App: "\[\Gamma \vdash t_1 : T_1 \to T_2; \Gamma \vdash t_2 : T_1\] \implies \Gamma \vdash \text{App } t_1 t_2 : T_2"
| t_Lam: "\[x \# \Gamma; (x, T_1) \# \Gamma \vdash t : T_2\] \implies \Gamma \vdash \text{Lam } [x].t : T_1 \to T_2"

Subgoals

1. \(\forall x \Gamma T_1 \vdash T_2. \ [x \# \Gamma; (x, T_1) : \Gamma \vdash t : T_2] \implies x \# \Gamma\)
2. \(\forall x \Gamma T_1 \vdash T_2. \ [x \# \Gamma; (x, T_1) : \Gamma \vdash t : T_2] \implies x \# \text{Lam } [x].t\)
3. \(\forall x \Gamma T_1 \vdash T_2. \ [x \# \Gamma; (x, T_1) : \Gamma \vdash t : T_2] \implies x \# T_1 \to T_2\)

nominal_inductive typing
**Typing Judgements (2)**

**inductive**

typing :: "ty_ctx ⇒ lam ⇒ ty ⇒ bool" ("_ ⊢ _ : _")

**where**

- t_Var: 
  
  \[ [\text{valid } \Gamma; (x,T) \in \text{set } \Gamma] \implies \Gamma \vdash \text{Var } x : T] \]

- t_App:
  
  \[ [\Gamma \vdash t_1 : T_1 \to T_2; \Gamma \vdash t_2 : T_1] \implies \Gamma \vdash \text{App } t_1 \ t_2 : T_2] \]

- t_Lam:
  
  \[ [x \# \Gamma; (x,T_1) \# \Gamma \vdash t : T_2] \implies \Gamma \vdash \text{Lam } [x].t : T_1 \to T_2] \]

**lemma ty_fresh:**

fixes x::"name"

and T::"ty"

shows "x # T"

by (induct T rule: ty.induct)

(simp_all add: fresh_string)

**nominal_inductive** typing
inductive
typing :: "ty_ctx ⇒ lam ⇒ ty ⇒ bool" ("_ ⊸ _ ⊸ _")

where
  t_Var: "[\[ valid \( Γ \); (x,T) ∈ set \( Γ \) ] ] ⇒ Γ ⊸ Var x : T"
| t_App: "[\( Γ \) ⊸ t_1 : T_1 ⇒ T_2; \( Γ \) ⊸ t_2 : T_1 ] ] ⇒ Γ ⊸ App t_1 t_2 : T_2"
| t_Lam: "[\( x \# Γ \); (x,T_1)\#Γ ⊸ t : T_2 ] ] ⇒ Γ ⊸ Lam [x].t : T_1 ⇒ T_2"

lemma ty_fresh:
  fixes x::"name"
  and T::"ty"
  shows "x#T"
by (induct T rule: ty.induct)
  (simp_all add: fresh_string)

nominal_inductive typing
  by (simp_all add: abs_fresh ty_fresh)
Weakening

lemma weakening:
  fixes $\Gamma_1 \Gamma_2 :: "ty_ctx"
  assumes a: "$\Gamma_1 \vdash t : T"
  and b: "valid $\Gamma_2"
  and c: "$\Gamma_1 \subseteq \Gamma_2"
  shows "$\Gamma_2 \vdash t : T"
using a b c
by (nominal_induct $\Gamma_1 \ t \ T$ avoiding: $\Gamma_2$ rule: typing.strong_induct)
  (auto simp add: atomize_all atomize_imp)
lemma weakening:
  fixes $\Gamma_1 \Gamma_2 :: \"ty\_ctx\"$
  assumes a: "$\Gamma_1 \vdash t : T$" 
  and b: "$\text{valid } \Gamma_2\" 
  and c: "$\Gamma_1 \subseteq \Gamma_2\" 
  shows "$\Gamma_2 \vdash t : T\" 
using a b c 
by (nominal_induct $\Gamma_1 \vdash t : T$ avoiding: $\Gamma_2$, rule: typing.strong_induct) 
  (auto simp add: atomize_all atomize_imp)

- This proof is can be found automatically, but that tells us not much...
Lemmas / Theorems / Corollary are of the form:

```
theorem theorem_name:
  fixes    x::"type"
...
assumes  "assm_1"
and      "assm_2"
...
shows   "statement"
...
```

- Grey parts are optional.
- Assumptions and the (goal)statement must be of type bool.
Lemma / Theorem / Corollary

- Lemmas / Theorems / Corollary are of the form:

```
theorem theorem_name:
  fixes x::"type"
  ...
  assumes "assm_1"
  and "assm_2"
  ...
  shows "statement"
```

- Grey parts are optional.

- Assumptions and the (goal)statement must be of type bool.

- Lemma weakening:

```
lemma weakening:
  fixes Γ₁ Γ₂::"ty_ctx"
  assumes a: "Γ₁ ⊢ t : T"
  and b: "valid Γ₂"
  and c: "Γ₁ ⊆ Γ₂"
  shows "Γ₂ ⊢ t : T"
```
lemma weakening:
fixes $\Gamma_1 \Gamma_2 :: "ty_ctx"
assumes a: "$\Gamma_1 \vdash t : T"
and b: "valid $\Gamma_2"
and c: "$\Gamma_1 \subseteq \Gamma_2"
shows "$\Gamma_2 \vdash t : T"
using a b c

proof (nominal_induct $\Gamma_1 \vdash T$ avoiding: $\Gamma_2$ rule: typing.strong_induct)
  case (t_Var $\Gamma_1 x T$)
  ...
  show "$\Gamma_2 \vdash \text{Var} x : T"
  ...

next
  case (t_App $\Gamma_1 t_1 T_1 T_2 t_2$)
  ...
  show "$\Gamma_2 \vdash \text{App} t_1 t_2 : T_2"
  ...

next
  case (t_Lam $x \times \Gamma_1 T_1 \vdash T_2$)
  ...
  show "$\Gamma_2 \vdash \text{Lam} [x].t : T_1 \rightarrow T_2"
  ...

qed
Each case is of the form:

```
case (Name x ...)  
    have n1: "statement1" by justification  
    have n2: "statement2" by justification  
    ...  
    show "statement" by justification  
```

- Grey parts are optional.
- Justifications can also be: using ... by ...
Cases

Each case is of the form:

case (Name x...)  
have n1: "statement1" by justification  
have n2: "statement2" by justification  
...  
show "statement" by justification

Grey parts are optional.

Justifications can also be: using ... by ...

using ih by ...
using n1 n2 n3 by ...
using lemma_name... by ...
Cases

Each case is of the form:

```
case (Name x...) 
  have n1: "statment1" by justification 
  have n2: "statment2" by justification 
  ... 
  show "statment" by justification 
```

- Grey parts are optional.
- Justifications can also be: using ... by ...
  
  - using ih by ...
  - using n1 n2 n3 by ...
  - using lemma_name...by ...
Justifications

- Omitting proofs
  - sorry

- Assumptions
  - by fact

- Automated proofs
  - by simp: simplification (equations, definitions)
  - by auto: simplification & proof search (many goals)
  - by force: simplification & proof search (first goal)
  - by blast: proof search
  ...

lemma weakening:
                    
  fixes \( \Gamma_1 \) \( \Gamma_2 \)::"ty_ctxt"  
  assumes a: "\( \Gamma_1 \vdash t : T \)"  
  and b: "valid \( \Gamma_2 \)"  
  and c: "\( \Gamma_1 \subseteq \Gamma_2 \)"  
  shows "\( \Gamma_2 \vdash t : T \)"  
  using a b c  

proof(nominal_induct \( \Gamma_1 \vdash T \) avoiding: \( \Gamma_2 \) rule: typing.strong_induct)  
  case (\( t_{\text{Var}} \) \( \Gamma_1 \times T \))  
    have a1: "valid \( \Gamma_2 \)" by fact  
    have a2: "\( \Gamma_1 \subseteq \Gamma_2 \)" by fact  
    have a3: "\( (x,T) \in (\text{set} \ \Gamma_1) \)" by fact  
    have a4: "\( (x,T) \in (\text{set} \ \Gamma_2) \)" using a2 a3 by simp  
    show "\( \Gamma_2 \vdash \text{Var} \ x : T \)" using a1 a4 by auto  
next ...
next

\[ \text{case } (t\_\text{Lam } x \ G_1 \ T_1 \vdash t : T_2) \]

\[ \text{have vc: } "x\#G_2" \text{ by fact} \]

\[ \text{have ih: } "[\text{valid } ((x,T_1)#G_2); (x,T_1)#G_1 \subseteq (x,T_1)#G_2] \]
\[ \implies (x,T_1)#G_2 \vdash t : T_2" \text{ by fact} \]

\[ \text{have a1: } "G_1 \subseteq G_2" \text{ by fact} \]

\[ \text{have a2: } "(x,T_1)#G_1 \subseteq (x,T_1)#G_2" \text{ using a1 by simp} \]

\[ \text{have b1: } "\text{valid } G_2" \text{ by fact} \]

\[ \text{have b2: } "\text{valid } ((x,T_1)#G_2)" \text{ using vc b1 by auto} \]

\[ \text{have b3: } "(x,T_1)#G_2 \vdash t : T_2" \text{ using ih b2 a2 by simp} \]

\[ \text{show } "G_2 \vdash \text{Lam } [x].t : T_1 \rightarrow T_2" \text{ using b3 vc by auto} \]

next . . .
\[
\begin{align*}
\Gamma & \vdash (x, T_1) :: \Gamma \vdash t : T_2 \\
\Gamma & \vdash \text{Lam}[x].t : T_1 \rightarrow T_2
\end{align*}
\]

next

\begin{itemize}
\item case \((t \_ \text{Lam} \times \Gamma_1 \ T_1 \vdash T_2)\)
\item have \(vc: "x \# \Gamma_2" \ \text{by fact}\)
\item have \(ih: "[(\text{valid} ((x, T_1) \# \Gamma_2); (x, T_1) \# \Gamma_1 \subseteq (x, T_1) \# \Gamma_2)] \implies (x, T_1) \# \Gamma_2\vdash t : T_2" \ \text{by fact}\)
\item have \(\Gamma_1 \subseteq \Gamma_2" \ \text{by fact}\)
\item then have \(a2: "(x, T_1) \# \Gamma_1 \subseteq (x, T_1) \# \Gamma_2" \ \text{by simp}\)
\item have \("\text{valid} \ \Gamma_2" \ \text{by fact}\)
\item then have \(b2: "\text{valid} ((x, T_1) \# \Gamma_2)" \ \text{using} \ vc \ \text{by auto}\)
\item have \("(x, T_1) \# \Gamma_2 \vdash t : T_2" \ \text{using} \ ih \ b2 \ a2 \ \text{by simp}\)
\item then show \("\Gamma_2 \vdash \text{Lam}[x].t : T_1 \rightarrow T_2" \ \text{using} \ vc \ \text{by auto}\)
\end{itemize}

next ...
A Sequence of Facts

have n1: “...”
have n2: “...”
...
have nn: “...”
have “...” using n1 n2...nn

have “...”
moreover have “...”
...
moreover have “...”
ultimately have “...”
\[
\frac{x \# \Gamma \quad (x, T_1):\Gamma \vdash t : T_2}{\Gamma \vdash \text{Lam} [x].t : T_1 \rightarrow T_2}
\]

next

\textbf{case} (t_Lam x \Gamma_1 \ T_1 \vdash T_2)

\textbf{have} vc: "x\#\Gamma_2" \textbf{by fact}

\textbf{have} ih: "[valid ((x,T_1)\#\Gamma_2); (x,T_1)\#\Gamma_1 \subseteq (x,T_1)\#\Gamma_2] \implies (x,T_1)\#\Gamma_2 \vdash t: T_2" \textbf{by fact}

\textbf{have} "\Gamma_1 \subseteq \Gamma_2" \textbf{by fact}

\textbf{then have} "(x,T_1)\#\Gamma_1 \subseteq (x,T_1)\#\Gamma_2" \textbf{by simp}

\textbf{moreover}

\textbf{have} "valid \ \Gamma_2" \textbf{by fact}

\textbf{then have} "valid ((x,T_1)\#\Gamma_2)" \textbf{using vc by auto}

\textbf{ultimately have} "(x,T_1)\#\Gamma_2 \vdash t : T_2" \textbf{using ih by simp}

\textbf{then show} "\Gamma_2 \vdash \text{Lam} [x].t : T_1 \rightarrow T_2" \textbf{using vc by auto}

next . . .
next

\[
x \# \Gamma \quad (x, T_1) : : \Gamma \vdash t : T_2 \\
\Gamma \vdash \text{Lam}[x].t : T_1 \rightarrow T_2
\]

case (t_Lam x \Gamma_1 T_1 \vdash T_2)

have vc: "x\#\Gamma_2" by fact

have ih: "[[valid ((x,T_1)\#\Gamma_2); (x,T_1)\#\Gamma_1 \subseteq (x,T_1)\#\Gamma_2]] \\
\implies (x,T_1)\#\Gamma_2 \vdash t:T_2" by fact

have "\Gamma_1 \subseteq \Gamma_2" by fact
then have "(x,T_1)\#\Gamma_1 \subseteq (x,T_1)\#\Gamma_2" by simp

moreover

have "valid \Gamma_2" by fact
then have "valid ((x,T_1)\#\Gamma_2)" using vc by auto
ultimately have "(x,T_1)\#\Gamma_2 \vdash t : T_2" using ih by simp
then show "\Gamma_2 \vdash \text{Lam}[x].t : T_1 \rightarrow T_2" using vc by auto

qed (auto)
We next want to introduce an evaluation relation and a CK machine.

For this we need the notion of capture-avoiding substitution.

consts

\[ \text{subst} :: \text{"lam \to name \to lam \to lam" ("\text{\_\_::=\_}\")} \]

nominal_primrec

\[ \text{\( (\text{Var } x)[y::=s] = (\text{if } x=y \text{ then } s \text{ else } (\text{Var } x)) \)} \]
\[ \text{\( (\text{App } t_1 \ t_2)[y::=s] = \text{App } (t_1[y::=s]) \ (t_2[y::=s]) \)} \]
\[ \text{\( x\#(y,s) \implies (\text{Lam } [x].t)[y::=s] = \text{Lam } [x].(t[y::=s]) \)} \]
We next want to introduce an evaluation relation and a CK machine.

For this we need the notion of capture-avoiding substitution.

\[
\text{consts}
\]

\[
\text{subst} :: "lam \Rightarrow name \Rightarrow lam \Rightarrow lam" ("\_[\_::=\_]")
\]

\[
\text{nominal\_primrec}
\]

"(Var x)[y::=s] = (if x=y then s else (Var x))"

"(App t_1 t_2)[y::=s] = App (t_1[y::=s]) (t_2[y::=s])"

"x\#(y,s) \iff (Lam [x].t)[y::=s] = Lam [x].(t[y::=s])"

Despite its looks, this is a total function!
However there is a problem with the bound names function:

```plaintext
consts
  bnds :: "lam ⇒ name set"

nominal_primrec
  "bnds (Var x) = {}"
  "bnds (App t₁ t₂) = bnds (t₁) ∪ bnds (t₂)"
  "bnds (Lam [x].t) = bnds (t) ∪ {x}"

lemma
  shows "bnds (Lam [x].Var x) = {x}"
  and "bnds (Lam [y].Var y) = {y}"
by (simp_all)
```
However there is a problem with the bound names function:

```
consts
  bnds :: "lam => name set"

nominal_primrec
  "bnds (Var x) = {}"
  "bnds (App t_1 t_2) = bnds (t_1) \cup bnds (t_2)"
  "bnds (Lam [x] . t) = bnds (t) \cup \{x\}"

lemma
  shows "bnds (Lam [x] . Var x) = \{x\}"
  and "bnds (Lam [y] . Var y) = \{y\}"
by (simp_all)
```
However there is a problem with the bound names function:

\[
\text{consts}\nonumber \\
\text{bnds :: "lam name set"}
\]

\[
\text{nominal_pr}
onumber \\
\text{"bnds (Var x) = {}"}
\]

\[
\text{"bnds (App t_1 t_2) = bnds (t_1) \cup bnds (t_2)"}
\]

\[
\text{"bnds (Lam [x].t) = bnds (t) \cup \{x\}"
\]

\[
\text{lemma}
onumber \\
\text{shows "bnds (Lam [x].Var x) = \{x\}"
\]

\[
\text{and "bnds (Lam [y].Var y) = \{y\}"
\]

by (simp_all)
However, there is a problem with the bound names function:

**consts**

\[
\text{consts}
\begin{align*}
\text{bnds :: "lam name set"}
\end{align*}
\]

**nominal_pr**

\[
\text{nominal_pr}
\begin{align*}
\text{"bnds (Var x) = {}"}
\text{"bnds (App } t_1 \text{ } t_2 \text{) = bnds (} t_1 \text{) } \cup \text{ bnds (} t_2 \text{)"}
\text{"bnds (Lam } [x].t \text{) = bnds (} t \text{) } \cup \{x\}"
\end{align*}
\]

**lemma**

\[
\text{lemma}
\begin{align*}
\text{shows "bnds (Lam } [x].\text{Var x) = \{x\}"
\text{and "bnds (Lam } [y].\text{Var y) = \{y\}"}
\text{by (simp_all)}
\end{align*}
\]

**Assume** \( x \neq y \).

\[
\text{Lam } [x].\text{Var x} = \text{Lam } [y].\text{Var y}
\]

\[
\text{bnds (Lam } [x].\text{Var x) = bnds (Lam } [y].\text{Var y)}
\]

However there is a problem with the bound names function:

\[
\text{bnds} :: \text{"lam}\rightarrow \text{name set}
\]

\[
\text{bnds (Var x)} = \{\}
\]

\[
\text{bnds (App t}_1 \text{ t}_2) = \text{bnds (t}_1) \cup \text{bnds (t}_2)
\]

\[
\text{bnds (Lam [x].t)} = \text{bnds (t)} \cup \{x\}
\]

lemma

shows "\text{bnds (Lam [x].Var x)} = \{x\}"

and "\text{bnds (Lam [y].Var y)} = \{y\}"

by (simp_all)
Bound Names Function

However there is a problem with the bound names function:

---

\textbf{consts}

\texttt{bnds :: "lam \Rightarrow name set"}

\textbf{nominal\_primrec}

\texttt{"bnds (Var x) = {}"}
\texttt{"bnds (App t_1 \ t_2) = bnds (t_1) \cup bnds (t_2)"}
\texttt{"bnds (Lam [x].t) = bnds (t) \cup \{x\}"

\textbf{lemma}

\texttt{shows "bnds (Lam [x].Var x) = \{x\}"
\texttt{and "bnds (Lam [y].Var y) = \{y\}"

\texttt{by (simp\_all)
consts

\[ \text{subst} :: \text{"lam} \Rightarrow \text{name} \Rightarrow \text{lam} \Rightarrow \text{lam}" \ ("\_[\_::=_]\") \]

nominal\_primrec

"(Var x)[y::=s] = (if x=y then s else (Var x))"
"(App t_1, t_2)[y::=s] = App (t_1[y::=s]) (t_2[y::=s])"
"x\#(y,s) \rightarrow (Lam [x].t)[y::=s] = Lam [x].(t[y::=s])"
Capture-Avoiding Subst.

consts

\[ \text{subst :: "lam } \Rightarrow \text{name } \Rightarrow \text{lam } \Rightarrow \text{lam" ("\_\_::=_\_\_\")} \]

nominal_primrec

\[ (\text{Var } x)[y::=s] = \text{(if } x=y \text{ then } s \text{ else (Var } x)) \]
\[ (\text{App } t_1 \ t_2)[y::=s] = \text{App } (t_1[y::=s]) (t_2[y::=s]) \]
\[ x\#(y,s) \quad \text{then } \quad (\text{Lam } [x].t)[y::=s] = \text{Lam } [x].(t[y::=s]) \]

Freshness Condition for Binders (FCB)

\[ \forall a \ ts. \ a \ # \ f \ \Rightarrow \ a \ # \ f \ a \ ts \]
Capture-Avoiding Subst.

consts

\[ \text{subst ::= "lam \Rightarrow name \Rightarrow lam \Rightarrow lam" ("_[\_::=_]")} \]

nominal_primrec

"(Var x)[y::=s] = (if x=y then s else (Var x))"
"(App t_1 t_2)[y::=s] = App (t_1[y::=s]) (t_2[y::=s])"
"x#(y,s) \mapsto (Lam [x].t)[y::=s] = Lam [x].(t[y::=s])"

Freshness Condition for Binders (FCB)

\[
\forall a \ ts. \ a \not\# f \Rightarrow a \not\# f \ a \ ts \\
\wedge x_1 \ y_1. \ \ldots \ \ldots \ \Rightarrow x_1 \not\# Lam [x_1].y_1
\]
Capture-Avoiding Subst.

consts

\[ \text{subst} :: \text{"lam} \Rightarrow \text{name} \Rightarrow \text{lam} \Rightarrow \text{lam}" \ ("[_::=_]") \]

nominal_primrec

\[ (\text{Var} \ x)[y::=s] = (\text{if} \ x=y \ \text{then} \ s \ \text{else} \ (\text{Var} \ x))" \]
\[ (\text{App} \ t_1 \ t_2)[y::=s] = \text{App} \ (t_1[y::=s]) \ (t_2[y::=s])" \]
\[ x\#(y,s) \mapsto (\text{Lam} \ [x].t)[y::=s] = \text{Lam} \ [x].(t[y::=s])" \]

apply(finite_guess)+
apply(rule TrueI)+
apply(simp add: abs_fresh)+
apply(fresh_guess)+
done

Freshness Condition for Binders (FCB)

\[ \forall a \ ts. \ a \ # \ f \ \Rightarrow \ a \ # \ f \ a \ ts \]
\[ \land x_1 y_1. \ ... \ ... \ \Rightarrow \ x_1 \ # \ \text{Lam} \ [x_1].y_1 \]
consts

\[ \text{subst :: "lam \name \lam \name \lam (\_\[\_\])} \]

nominal_primrec

"(Var x)[y::=s] = (if x=y then s else (Var x))"
"(App t_1 \ t_2)[y::=s] = App (t_1[y::=s]) (t_2[y::=s])"
"x#(y,s) \implies (Lam [x].t)[y::=s] = Lam [x].(t[y::=s])"

apply(finite_guess)+
apply(rule TrueI)+
apply(simp add: abs_fresh)+
apply(fresh_guess)+
done

Freshness Condition for Binders (FCB)

\[ \forall a \ ts. \ a \# f \implies a \# f \ a \ ts \]
\[ \land x_1 y_1. \ldots \ldots \implies x_1 \# \text{Lam} [x_1].y_1 \]
inductive
  eval :: "lam ⇒ lam ⇒ bool" ("_ ↓ _")
where
  e_Lam: "Lam [x].t ↓ Lam [x].t"
| e_App: "[t_1 ↓ Lam [x].t; t_2 ↓ v'; t[x:=v'] ↓ v] ⇒⇒ App t_1 t_2 ↓ v"

declare eval.intros[intro]
**Evaluation Relation**

**inductive**

\[ \text{eval} :: "\text{lam} \Rightarrow \text{lam} \Rightarrow \text{bool}" \ ("_ \downarrow \_") \]

**where**

\[ \begin{align*} 
\text{e}_\text{Lam}: & \ "\text{Lam} \ [x].t \downarrow \text{Lam} \ [x].t" \\
| \text{e}_\text{App}: & \ "[t_1 \downarrow \text{Lam} \ [x].t; t_2 \downarrow v'; t[x::=v'] \downarrow v] \Longrightarrow \text{App} \ t_1 \ t_2 \downarrow v" 
\end{align*} \]

**declare** eval.intros[intro]

---

\[ \begin{align*} 
\text{Lam} \ [x].t \downarrow \text{Lam} \ [x].t \\
\hline \\
t_1 \downarrow \text{Lam} \ [x].t & t_2 \downarrow v' & t[x::=v'] \downarrow v \\
\hline \\
\text{App} \ t_1 \ t_2 \downarrow v 
\end{align*} \]
Values

inductive
  val :: "lam ⇒ bool"
where
  v_Lam[intro]: "val (Lam [x].e)"

lemma eval_to_val:
  assumes a: "t ⇓ t"
  shows "val t"
using a by (induct) (auto)
Values

inductive
val :: "lam \Rightarrow bool"

where
v_Lam[intro]: "val (Lam [x].e)"

lemma eval_to_val:
  assumes a: "t \downarrow t"
  shows "val t"
  using a by (induct) (auto)

- If our language contained natural numbers, booleans, etc., we would expand on this definition.
A CK machine works on configurations \(\langle \_,\_\rangle\) consisting of a lambda-term and a list of contexts.

**inductive**

machine :: "lam\(\Rightarrow\)ctxs\(\Rightarrow\)lam\(\Rightarrow\)ctxs\(\Rightarrow\)bool" ("\(\langle\_,\_\rangle \leftrightarrow \langle\_,\_\rangle\)"

**where**

\[ m_1: "\langle\text{App}\ e_1\ e_2,Es\rangle \leftrightarrow \langle e_1,(CAppL \square e_2)\#Es\rangle" \]
\[ m_2: "\text{val } v \iff \langle v,(CAppL \square e_2)\#Es\rangle \leftrightarrow \langle e_2,(CAppR v \square)\#Es\rangle" \]
\[ m_3: "\text{val } v \iff \langle v,(CAppR (Lam [x].e) \square)\#Es\rangle \leftrightarrow \langle e[x::=v],Es\rangle" \]
A CK machine works on configurations \( \langle \_, \_ \rangle \) consisting of a lambda-term and a list of contexts.

\[
\text{inductive}
\text{machine} :: "\text{lam} \Rightarrow \text{ctxs} \Rightarrow \text{lam} \Rightarrow \text{ctxs} \Rightarrow \text{bool}" \quad ("\langle \_, \_ \rangle \leftrightarrow \langle \_, \_ \rangle")
\]

where

\[
m_1: \langle \text{App } e_1 \ e_2, \text{Es} \rangle \leftrightarrow \langle e_1, (\text{CAppL } \square \ e_2) \# \text{Es} \rangle
\]
\[
m_2: \text{val } v \quad \leftrightarrow \quad \langle v, (\text{CAppL } \square \ e_2) \# \text{Es} \rangle \leftrightarrow \langle e_2, (\text{CAppR } v \ \square) \# \text{Es} \rangle
\]
\[
m_3: \text{val } v \quad \leftrightarrow \quad \langle v, (\text{CAppR } (\text{Lam } [x].e) \ \square) \# \text{Es} \rangle \leftrightarrow \langle e[x::=v], \text{Es} \rangle
\]

Initial state of the CK machine:
\( \langle \text{t}, [\_] \rangle \)
CK Machine

- A CK machine works on configurations \( \langle \_, \_ \rangle \) consisting of a lambda-term and a list of contexts.

**Inductive**

\[
\text{machine :: "lam} \Rightarrow \text{ctxs} \Rightarrow \text{lam} \Rightarrow \text{ctxs} \Rightarrow \text{bool" ("\langle \_, \_ \rangle \mapsto \langle \_, \_ \rangle")}
\]

where

\[
m_1: \langle \text{App } e_1 \ e_2, \text{Es} \rangle \mapsto \langle e_1, (\text{CAppL } \Box e_2)\#\text{Es} \rangle
\]
\[
m_2: \langle \text{val } v \Rightarrow \langle v, (\text{CAppL } \Box e_2)\#\text{Es} \rangle \mapsto \langle e_2, (\text{CAppR } v \Box)\#\text{Es} \rangle
\]
\[
m_3: \langle \text{val } v \Rightarrow \langle v, (\text{CAppR } (\text{Lam } [x].e) \Box)\#\text{Es} \rangle \mapsto \langle e[x::=v], \text{Es} \rangle
\]

**Inductive**

"machines" :: "lam} \Rightarrow \text{ctxs} \Rightarrow \text{lam} \Rightarrow \text{ctxs} \Rightarrow \text{bool" ("\langle \_, \_ \rangle \mapsto^* \langle \_, \_ \rangle")}

where

\[
\text{ms}_1: \langle e, \text{Es} \rangle \mapsto^* \langle e, \text{Es} \rangle
\]
\[
\text{ms}_2: \langle e_1, \text{Es}_1 \rangle \mapsto \langle e_2, \text{Es}_2 \rangle; \langle e_2, \text{Es}_2 \rangle \mapsto^* \langle e_3, \text{Es}_3 \rangle; \]
\[
\quad \mapsto \langle e_1, \text{Es}_1 \rangle \mapsto^* \langle e_3, \text{Es}_3 \rangle
\]
Our Goal

Our goal is to show that the result the machine calculates corresponds to the value the evaluation relation generates and vice versa. That means:

\[ t \downarrow v \iff \langle t,[] \rangle \mapsto^* \langle v,[] \rangle \]

with \( v \) being a value.
corollary eval_implies_machines:
  assumes a: "t ↓ t"
  shows "⟨t,[]⟩ →* ⟨t',[]⟩"
using a using eval_implies_machines_ctx by simp
lemma ms₃:
  assumes a: "⟨e₁, Es₁⟩ ⟷* ⟨e₂, Es₂⟩" "⟨e₂, Es₂⟩ ⟷* ⟨e₃, Es₃⟩"
  shows "⟨e₁, Es₁⟩ ⟷* ⟨e₃, Es₃⟩"
using a by (induct) (auto)

corollary eval_implies_machines:
  assumes a: "† ↓ †"
  shows "⟨†, []⟩ ⟷* ⟨†', []⟩"
using a using eval_implies_machines_ctx by simp
lemma ms₃:
  assumes a: "⟨e₁,Es₁⟩ ↦* ⟨e₂,Es₂⟩" "⟨e₂,Es₂⟩ ↦* ⟨e₃,Es₃⟩"
  shows "⟨e₁,Es₁⟩ ↦* ⟨e₃,Es₃⟩"
using a by (induct) (auto)

theorem eval_implies_machines_ctx:
  assumes a: "t ↓ t'"
  shows "⟨t,Es⟩ ↦* ⟨t',Es⟩"
using a
by (induct arbitrary: Es)
  (metis eval_to_val machine.intros ms₁ ms₂ ms₃ v_Lam)+

corollary eval_implies_machines:
  assumes a: "t ↓ t'"
  shows "⟨t,[[]]⟩ ↦* ⟨t',[[]]⟩"
using a using eval_implies_machines_ctx by simp
lemma ms3:
  assumes a: "⟨e₁,Es₁⟩ ↦* ⟨e₂,Es₂⟩" "⟨e₂,Es₂⟩ ↦* ⟨e₃,Es₃⟩"
  shows "⟨e₁,Es₁⟩ ↦* ⟨e₃,Es₃⟩"

Sledgehammer:

Can be used at any point in the development.
lemma ms3:
  assumes a: "⟨e₁,Es₁⟩ ⟷* ⟨e₂,Es₂⟩" "⟨e₂,Es₂⟩ ⟷* ⟨e₃,Es₃⟩"
  shows "⟨e₁,Es₁⟩ ⟷* ⟨e₃,Es₃⟩"

Sledgehammer:

Can be used at any point in the development.
lemma ms3:
  assumes a: "\langle e_1, Es_1 \rangle \mapsto^* \langle e_2, Es_2 \rangle" "\langle e_2, Es_2 \rangle \mapsto^* \langle e_3, Es_3 \rangle"
  shows "\langle e_1, Es_1 \rangle \mapsto^* \langle e_3, Es_3 \rangle"

Sledgehammer:

Can be used at any point in the development.
lemma ms₃:
  assumes a: "⟨e₁,Es₁⟩ ⟷* ⟨e₂,Es₂⟩" "⟨e₂,Es₂⟩ ⟷* ⟨e₃,Es₃⟩"
  shows "⟨e₁,Es₁⟩ ⟷* ⟨e₃,Es₃⟩"
using a by (induct) (auto)

theorem eval_implies_machines_ctx:
  assumes a: "t ⧵ t''
  shows "⟨t,Es⟩ ⟷* ⟨t',Es⟩"
using a
by (induct arbitrary: Es)
  (metis eval_to_val_machine.intros ms₁ ms₂ ms₃ v_Lam)+

corollary eval_implies_machines:
  assumes a: "t ⧵ t''
  shows "⟨t,[]⟩ ⟷* ⟨t',[]⟩"
using a using eval_implies_machines_ctx by simp
The statement for the other direction is as follows:

**lemma** machines_implies_eval:

- **assumes** \( a: "\langle \top, [] \rangle \xrightarrow{\star} \langle v, [] \rangle" \)
- **and** \( b: "\text{val } v" \)
- **shows** \( \top \Downarrow v" \)
The statement for the other direction is as follows:

```lean
lemma machines_implies_eval:
  assumes a: "⟨t,[]⟩ \mapsto^* ⟨v,[]⟩"
  and b: "val v"
  shows "t ↓ v"
oops
```
The statement for the other direction is as follows:

**lemma** machines_implies_eval:
- **assumes** a: "⟨t,[]⟩ ⟷* ⟨v,[]⟩"
- **and** b: "val v"
- **shows** "t ↓ v"
- **oops**

We can prove this direction by introducing a small-step reduction relation.
CBV Reduction

inductive
  cbv :: "lam⇒lam⇒bool" ("_ ——> cbv _")

where
  cbv₁: "val v ——> App (Lam [x].t) v ——> cbv t[x::=v]"
| cbv₂: "t ——> cbv t' ——> App t t₂ ——> cbv App t' t₂"
| cbv₃: "t ——> cbv t' ——> App t₂ t ——> cbv App t₂ t'"

Later on we like to use the strong induction principle for this relation.
CBV Reduction

**inductive**

\[
\text{cbv} :: "\text{lam} \Rightarrow \text{lam} \Rightarrow \text{bool}" ("_ \longrightarrow \text{cbv } _")
\]

**where**

\[
\text{cbv}_1: "\text{val } v \longmapsto \text{App } (\text{Lam } [x].t) \; v \longmapsto \text{cbv } t[x::=v]"
\]

\[
| \quad \text{cbv}_2: "t \longmapsto \text{cbv } t' \longmapsto \text{App } t \; t_2 \longmapsto \text{cbv } \text{App } t' \; t_2"
\]

\[
| \quad \text{cbv}_3: "t \longmapsto \text{cbv } t' \longmapsto \text{App } t_2 \; t \longmapsto \text{cbv } \text{App } t_2 \; t'"
\]

- Later on we like to use the strong induction principle for this relation.

**Conditions:**

1. \( \forall v \; x \; t. \; \text{val } v \longmapsto x \; \# \; \text{App } \text{Lam } [x].t \; v \)

2. \( \forall v \; x \; t. \; \text{val } v \longmapsto x \; \# \; t[x::=v] \)
CBV Reduction

inductive

\[ \text{cbv} :: \forall \lambda \alpha \beta \gamma. \alpha \rightarrow \beta \rightarrow \gamma \rightarrow \text{bool} \]  
\( _\_ \rightarrow \text{cbv} \_ \)

where

\[ \text{cbv}_1: \left[ \begin{array}{c} \text{val } \nu; \ x \# \nu \end{array} \right] \rightarrow \text{App} \ (\text{Lam} \ [x] \ . \ t) \ \nu \rightarrow \text{cbv} \ t[x::=\nu] \]

\| \text{cbv}_2[\text{intro}]: \ t \rightarrow \text{cbv} \ t' \rightarrow \text{App} \ t \ \ t_2 \rightarrow \text{cbv} \ \text{App} \ t' \ t_2 \]

\| \text{cbv}_3[\text{intro}]: \ t \rightarrow \text{cbv} \ t' \rightarrow \text{App} \ t_2 \ t \rightarrow \text{cbv} \ \text{App} \ t_2 \ t \]

- The conditions that give us automatically the strong induction principle require us to add the assumption \( x \# \nu \). This makes this rule less useful.
lemma subst_eqvt[eqvt]:
fixes π::"name prm"
shows "π • (t₁[x:=t₂]) = (π • t₁)[(π • x)::=(π • t₂)]"
by (nominal_induct t₁ avoiding: x t₂ rule: lam.strong_induct)
(auto simp add: perm_bij fresh_atm fresh_bij)

lemma fresh_fact:
fixes z::"name"
shows "[z#s; (z=y ∨ z#t)] = z#t[y::=s]"
by (nominal_induct t avoiding: z y s rule: lam.strong_induct)
(auto simp add: abs_fresh fresh_prod fresh_atm)

equivariance val
equivariance cbv
nominal_inductive cbv
by (simp_all add: abs_fresh fresh_fact)
lemma subst_rename:
  assumes a: "y # t"
  shows "t[x::=s] = ([(y,x)]•t)[y::=s]"
using a
by (nominal_induct t avoiding: x y s rule: lam.strong_induct)
  (auto simp add: calc_atm fresh_atm abs_fresh)

lemma better_cbv1[intro]:
  assumes a: "val v"
  shows "App (Lam [x].t) v → cbv t[x::=v]"
proof -
  obtain y::"name" where fs: "y # (x,t,v)"
    by (rule exists_fresh) (auto simp add: fs_name1)
  have "App (Lam [x].t) v = App (Lam [y].([(y,x)]•t)) v" using fs
    by (auto simp add: lam.inject alpha' fresh_prod fresh_atm)
  also have "... → cbv ([(y,x)]•t)[y::=v]" using fs a
    by (auto simp add: cbv1 fresh_prod)
  also have "... = t[x::=v]" using fs
    by (simp add: subst_rename[symmetric] fresh_prod)
finally show "App (Lam [x].t) v → cbv t[x::=v]" by simp
qed
inductive "cbvs" :: "lam ⇒ lam ⇒ bool" ("_ → cbv* _")

where
  cbvs₁[intro]: "e → cbv* e"
| cbvs₂[intro]: "[e₁ → cbv e₂; e₂ → cbv* e₃] ⇒ e₁ → cbv* e₃"

lemma cbvs₃[intro]:
  assumes a: "e₁ → cbv* e₂" "e₂ → cbv* e₃"
  shows "e₁ → cbv* e₃"
using a by (induct) (auto)
CBV Reduction*

inductive
"cbvs" :: "\text{lam} \Rightarrow \text{lam} \Rightarrow \text{bool}" ("\_ \longrightarrow cbv\* \_")

where
\begin{align*}
\text{cbvs}_1[\text{intro}]: & \quad \text{(e \longrightarrow cbv}\* \ e) \\
| \text{cbvs}_2[\text{intro}]: & \quad \left[ (\text{e}_1 \longrightarrow \text{cbv} \ e_2; \ e_2 \longrightarrow \text{cbv}\* \ e_3) \right] \Longrightarrow \text{e}_1 \longrightarrow \text{cbv}\* \ e_3
\end{align*}

lemma \text{cbvs}_3[\text{intro}]:
\begin{align*}
\text{assumes a}: & \quad (\text{e}_1 \longrightarrow \text{cbv}\* \ e_2) \quad (\text{e}_2 \longrightarrow \text{cbv}\* \ e_3) \\
\text{shows}: & \quad \text{e}_1 \longrightarrow \text{cbv}\* \ e_3
\end{align*}

using a by (induct) (auto)

lemma \text{cbv}_{\text{in ctx}}:
\begin{align*}
\text{assumes a}: & \quad \text{t} \longrightarrow \text{cbv} \ t' \\
\text{shows}: & \quad \text{E}[\text{t}] \longrightarrow \text{cbv} \ E[\text{t}]
\end{align*}

using a by (induct E) (auto)
lemma machines_implies_cbvs: 
  assumes a: "⟨e,[]⟩ ↠* ⟨e',[]⟩" 
  shows "e ⟶ cbv* e" 
using a by (auto dest: machines_implies_cbvs_ctx)
lemma machine_implies_cbvs_ctx:
  assumes a: "\langle e,Es \rangle \leftrightarrow \langle e',Es' \rangle"
  shows "\langle Es',[e] \rangle \rightarrow cbv^* \langle Es',[e'] \rangle"
using a by (induct) (auto simp add: ctx-compose intro: cbv_in_ctx)

lemma machines_implies_cbvs:
  assumes a: "\langle e,[] \rangle \leftrightarrow^* \langle e',[] \rangle"
  shows "e \rightarrow cbv^* e"
using a by (auto dest: machine_implies_cbvs_ctx)
lemma machine_implies_cbvs_ctx:
assumes a: "⟨e,Es⟩ ◲ → ⟨e’,Es’⟩"
shows "(Es↓)[e] → cbv* (Es’↓)[e’]"
using a by (induct) (auto simp add: ctx_compose intro: cbv_in_ctx)

If we had not derived the better cbv-rule, then we would have to do an explicit renamining here.

lemma machines_implies_cbvs:
assumes a: "⟨e,[]⟩ ◲* →* ⟨e’,[]⟩"
shows "e → cbv* e"
using a by (auto dest: machines_implies_cbvs_ctxt)
lemma machine_implies_cbvs_ctx:
  assumes a: "\langle e,Es \rangle \longmapsto \langle e',Es' \rangle"
  shows "(Es\downarrow)[e] \longrightarrow cbv^* (Es'\downarrow)[e']"
using a by (induct) (auto simp add: ctx_compose intro: cbv_in_ctx)

lemma machines_implies_cbvs_ctx:
  assumes a: "\langle e,Es \rangle \longmapsto^* \langle e',Es' \rangle"
  shows "(Es\downarrow)[e] \longrightarrow cbv^* (Es'\downarrow)[e']"
using a
by (induct) (auto dest: machine_implies_cbvs_ctx)

lemma machines_implies_cbvs:
  assumes a: "\langle e,[] \rangle \longmapsto^* \langle e',[] \rangle"
  shows "e \longrightarrow cbv^* e"
using a by (auto dest: machines_implies_cbvs_ctx)
CBV* Implies Evaluation

We need the following scaffolding lemmas in order to show that cbv-reduction implies evaluation.

**Lemma eval_val:**
- Assumes `a: "val t"`
- Shows `"t ↓↓ t"`

**Using a by (induct) (auto)**

**Lemma e_App_elim:**
- Assumes `a: "App t₁ t₂ ↓ v"`
- Shows `"∃ x t v'. t₁ ↓ Lam [x].t ∧ t₂ ↓ v' ∧ t[x::=v'] ↓ v"`

**Using a by (cases) (auto simp add: lam.inject)**
lemma cbv_eval:
  assumes a: "t₁ \rightarrow cbv t₂" "t₂ \downarrow t₃"
  shows "t₁ \downarrow t₃"
using a
by (induct arbitrary: t₃)
  (auto intro: eval_val dest!: e_App_elim)

lemma cbvs_eval:
  assumes a: "t₁ \rightarrow cbv* t₂" "t₂ \downarrow t₃"
  shows "t₁ \downarrow t₃"
using a by (induct) (auto simp add: cbv_eval)

lemma cbvs_implies_eval:
  assumes a: "t \rightarrow cbv* v" "val v"
  shows "t \downarrow v"
using a
by (induct)
  (auto simp add: eval_val cbvs_eval dest: cbvs₂)
Right-to-Left Direction

Via the cbv-reduction relation we can finally show that the CK machine implies the evaluation relation.

```isar
theorem machines_implies_eval:
  assumes a: "⟨t_1,[]⟩ ⇠^* ⟨t_2,[]⟩" and b: "val t_2"
  shows "t_1 ⇓ t_2"
proof -
  from a have "t_1 ⇒ CBV^* t_2" by (simp add: machines_implies_cbvs)
  then show "t_1 ⇓ t_2" using b by (simp add: cbvs_implies_eval)
qed
```

Next we like to prove a type preservation and an progress lemma for the cbv-reduction relation.

**Theorem cbv_type Preservation:**

- Assumes: 
  - \( t \rightarrow \text{cbv } t' \)
  - \( \Gamma \vdash t : T \)

- Shows: 
  - \( \Gamma \vdash t' : T \)

**Theorem Progress:**

- Assumes: 
  - \( [] \vdash t : T \)

- Shows: 
  - \( \exists t'. t \rightarrow \text{cbv } t' \) \lor (val \ t) \)
Preservation and Progress

Next we like to prove a type preservation and an progress lemma for the cbv-reduction relation.

**Theorem cbv_type_preservation:**

- **Assumes a:** "\( t \rightarrow \text{cbv} \ t' \)"
- **And b:** "\( \Gamma \vdash t : T \)"
- **Shows:** "\( \Gamma \vdash t' : T \)"

**Theorem progress:**

- **Assumes a:** "\([\] \vdash t : T\)"
- **Shows:** "\((\exists t'. t \rightarrow \text{cbv} t') \lor (\text{val } t)\)"

We need the property of type-substitutivity.
lemma valid_elim:
  assumes a: "valid ((x,T)# Γ)"
  shows "x# Γ ∧ valid Γ"
using a by (cases) (auto)

lemma valid_insert:
  assumes a: "valid (Δ@[(x,T)]@ Γ)"
  shows "valid (Δ@ Γ)"
using a
by (induct Δ)
  (auto simp add: fresh_list_append fresh_list_cons dest!: valid_elim)

lemma fresh_list:
  shows "y#xs = (∀ x ∈ set xs. y#x)"
by (induct xs) (simp_all add: fresh_list_nil fresh_list_cons)

lemma context_unique:
  assumes a1: "valid Γ"
  and a2: "(x,T) ∈ set Γ"
  and a3: "(x,U) ∈ set Γ"
  shows "T = U"
using a1 a2 a3
by (induct) (auto simp add: fresh_list fresh_prod fresh_atm)
corollary type_substitution:

assumes a: "\((x,T') \# \Gamma \vdash e : T\)"
and b: "\(\Gamma \vdash e' : T'\)"

shows "\(\Gamma \vdash e[x::=e'] : T\)"

proof (nominal_induct)

ultimately show "\(\Delta \vdash \text{Var} y[x::=e'] : T\)" by blast

qed (force simp add: fresh_list_append fresh_list_cons)
lemma type_substitution_aux:
  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 (t_Var 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)
lemma type_substitution_aux:
  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 (t_Var 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)
**Type Substitutivity**

**lemma** type_substitution_aux:
  assumes a: "Δ@[((x,T')]@ Γ ⊢ e : T"
  and    b: "Γ ⊢ e' : T"
  shows "Δ@ Γ ⊢ e[x:=e'] : T"

**corollary** type_substitution:
  assumes a: "(x,T')# Γ ⊢ e : T"
  and    b: "Γ ⊢ e' : T"
  shows "Γ ⊢ e[x:=e'] : T"
  using a b type_substitution_aux[where Δ="[]"]
  by (auto)
Inversion Lemmas

lemma t_App_elim:
  assumes a: "Γ ⋬ App t1 t2 : T"
  shows "∃ T'. Γ ⋬ t1 : T' → T ∧ Γ ⋬ t2 : T''
using a by (cases) (auto simp add: lam.inject)

lemma t_Lam_elim:
  assumes ty: "Γ ⋬ Lam [x].t : T"
  and fc: "x#Γ"
  shows "∃ T1 T2. T = T1 → T2 ∧ (x,T1)#Γ ⋬ t : T2"
using ty fc
by (cases rule: typing.strong_cases)
  (auto simp add: alpha lam.inject abs_fresh ty_fresh)
Theorem cbv_type_preservation:

assumes a: "t \rightarrow_{\text{cbv}} t'"
and b: "\Gamma \vdash t : T"

shows "\Gamma \vdash t' : T"

using a b

by (nominal_induct avoiding: \Gamma T rule: cbv.strong_induct)
  (auto dest!: t_Lam_elim t_App_elim
   simp add: type_substitution ty.inject)

Corollary cbvs_type_preservation:

assumes a: "t \rightarrow_{\text{cbv*}} t'"
and b: "\Gamma \vdash t : T"

shows "\Gamma \vdash t' : T"

using a b

by (induct) (auto intro: cbv_type_preservation)
Finally we can establish the progress lemma:

```isar
lemma canonical_tArr:
  assumes a: "[\] \vdash t : T1 \to T2"
  and b: "val t"
  shows "\exists x t'. t = Lam [x].t'"
using b a by (induct) (auto)
```

```isar
theorem progress:
  assumes a: "[\] \vdash t : T"
  shows "(\exists t'. t \longrightarrow cbv t') \lor (val t)"
using a
by (induct \Gamma::"[\]::ty_ctx" t T)
  (auto intro!: cbv.intros dest: canonical_tArr)
```
Finally we can establish the progress lemma:

**Lemma** canonical_tArr:

- assumes a: "\([\_] \vdash t : T1 \rightarrow T2\)"
- and b: "val t"
- shows "\(\exists x . t'. t = \text{Lam } [x].t'\)"

**Theorem** progress:

- assumes a: "\([\_] \vdash t : T\)"
- shows "\((\exists t'. t \rightarrow\rightarrow \text{cbv } t') \lor (\text{val } t)\)"

This lemma is stated with extensions in mind.
Extensions

With only minimal modifications the proofs can be extended to the language given by:

```plaintext
nominal_datatype lam =
  Var "name"
  App "lam" "lam"
  Lam "<name>lam" ("Lam [_.]_.")
  Num "nat"
  Minus "lam" "lam" ("_. -- _.")
  Plus "lam" "lam" ("_. ++ _.")
  TRUE
  FALSE
  IF "lam" "lam" "lam"
  Fix "<name>lam" ("Fix [_.]_.")
  Zet "lam"
  Eqi "lam" "lam"
```
Formalisation of LF
(joint work with Cheney and Berghofer)

\begin{align*}
def & \equiv \quad \text{Proof} \\
& \quad \text{Alg.}
\end{align*}
Formalisation of LF
(joint work with Cheney and Berghofer)
Formalisation of LF
(joint work with Cheney and Berghofer)

1st Solution

\( \text{def} \quad \overset{\equiv}{=} \quad \text{Proof} \quad \overset{\equiv}{=} \quad \text{Alg.} \)

2nd Solution

\( \text{def} + \text{ex} \quad \overset{\equiv}{=} \quad \text{Proof} \quad \overset{\equiv}{=} \quad \text{Alg.} \)

3rd Solution

\( \text{def} \quad \overset{\equiv}{=} \quad \text{Proof} \quad \overset{\equiv}{=} \quad \text{Alg.} \)

(each time one needs to check \(\sim 31\)pp of informal paper proofs)

Formalisation of LF
(joint work with Cheney and Berghofer)

1st Solution

2nd Solution

(each time one needs to check ~31pp of informal paper proofs)
Formalisation of LF
(joint work with Cheney and Berghofer)

1st Solution

2nd Solution

3rd Solution

(each time one needs to check ~31pp of informal paper proofs)
Two Health Warnings ;o)

Theorem provers should come with two health warnings:

- Theorem provers are addictive! (Xavier Leroy: “Building proof scripts is surprisingly addictive, in a videogame kind of way...”)
- Theorem provers cause you to lose faith in your proofs done by hand! (Michael Norrish, Mike Gordon, me, very possibly others)
Theorem provers should come with two health warnings:

- Theorem provers are addictive!
  (Xavier Leroy: “Building [proof] scripts is surprisingly addictive, in a videogame kind of way...”)

Two Health Warnings ;o)

Theorem provers should come with two health warnings:

- Theorem provers are addictive!
  (Xavier Leroy: “Building [proof] scripts is surprisingly addictive, in a videogame kind of way...”)

- Theorem provers cause you to lose faith in your proofs done by hand!
  (Michael Norrish, Mike Gordon, me, very possibly others)
Answers to Exercises

- Given a finite set of atoms. What is the support of this set?
Given a finite set of atoms. What is the support of this set? If $S$ is finite, then $\text{supp}(S) = S$. 
Given a finite set of atoms. What is the support of this set? If $S$ is finite, then \( \text{supp}(S) = S \).

What is the support of the set of all atoms?

Are there any sets of atoms that have infinite support? If both $S$ and $A$ are infinite then \( \text{supp}(S) = A \).
Given a finite set of atoms. What is the support of this set? If $S$ is finite, then $\text{supp}(S) = S$.

What is the support of the set of all atoms? Let $A = \{a_0, a_1 \ldots\}$, then $\text{supp}(A) = \emptyset$. 
Answers to Exercises

Given a finite set of atoms. What is the support of this set? If $S$ is finite, then $\text{supp}(S) = S$.

What is the support of the set of all atoms? Let $A = \{a_0, a_1 \ldots\}$, then $\text{supp}(A) = \emptyset$.

From the set of all atoms take one atom out. What is the support of the resulting set?
Given a finite set of atoms. What is the support of this set? If $S$ is finite, then $\text{supp}(S) = S$.

What is the support of the set of all atoms? Let $A = \{a_0, a_1 \ldots\}$, then $\text{supp}(A) = \emptyset$.

From the set of all atoms take one atom out. What is the support of the resulting set? $\text{supp}(A - \{a\}) = \{a\}$.
Answers to Exercises

- Given a finite set of atoms. What is the support of this set? If $S$ is finite, then $\text{supp}(S) = S$.

- What is the support of the set of all atoms? Let $A = \{a_0, a_1 \ldots\}$, then $\text{supp}(A) = \emptyset$.

- From the set of all atoms take one atom out. What is the support of the resulting set? $\text{supp}(A - \{a\}) = \{a\}$.

- Are there any sets of atoms that have infinite support?
Given a finite set of atoms. What is the support of this set? If \( S \) is finite, then \( \text{supp}(S) = S \).

What is the support of the set of all atoms? Let \( A = \{a_0, a_1 \ldots\} \), then \( \text{supp}(A) = \emptyset \).

From the set of all atoms take one atom out. What is the support of the resulting set? \( \text{supp}(A - \{a\}) = \{a\} \).

Are there any sets of atoms that have infinite support? If both \( S \) and \( A - S \) are infinite then \( \text{supp}(S) = A \).
Thank you very much!