Nominal Techniques
or, “The Real Thing”

Christian Urban (TU Munich)

http://isabelle.in.tum.de/nominal/

A Formalisation of a CK Machine:

$\_ \Downarrow \_$ CK

Nominal Techniques

or, “The Real Thing”

Christian Urban (TU Munich)

http://isabelle.in.tum.de/nominal/

A Formalisation of a CK Machine:

$$\_ \Downarrow \_ \longrightarrow \text{CK}$$

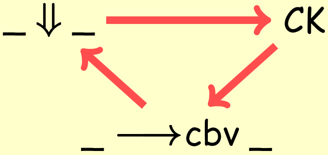
Nominal Techniques

or, “The Real Thing”

Christian Urban (TU Munich)

http://isabelle.in.tum.de/nominal/

A Formalisation of a CK Machine:



Lambda-Terms

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

```
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).

Contexts

```
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)
```

Backtrack One Step

- For our CK machines we actually do not need contexts for lambdas.

```
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⟧)"
```

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 "(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"
```

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 "(E₁ ∘ E₂)⟦t⟧ = E₁⟦E₂⟦t⟧⟧"
by (induct E₁ rule: ctx.induct) (simp_all)
```

Subgoals

1. $\Box \circ E_2[\![t]\!] = \Box[\![E_2[\![t]\!]]\!]$
2. $\bigwedge ctx\ lam.\ ctx \circ E_2[\![t]\!] = ctx[\![E_2[\![t]\!]]\!] \Longrightarrow CAppL\ ctx\ lam \circ E_2[\![t]\!] = CAppL\ ctx\ lam[\![E_2[\![t]\!]]\!]$
3. $\bigwedge lam\ ctx.\ ctx \circ E_2[\![t]\!] = ctx[\![E_2[\![t]\!]]\!] \Longrightarrow CAppR\ lam\ ctx \circ E_2[\![t]\!] = CAppR\ lam\ ctx[\![E_2[\![t]\!]]\!]$

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 "(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"
```

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"
```

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 $\subseteq$, 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

$$\frac{\text{valid}\ \Gamma \qquad (x, T) \in \text{set}\ \Gamma}{\Gamma \vdash \text{Var}\ x : T} \qquad \frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash \text{App}\ t_1\ t_2 : T_2}$$

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

induct
valid
where
v_1: "v
| v_2: "⟦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 ⊢ 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"

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

```
inductive
  valid :: "ty_ctx ⇒
where
  v1: "valid []"
| v2: "⟦valid Γ; x#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]
```

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

1.) The relation needs to be equivariant.

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)"
```

This proves for us:

$$\text{valid } \Gamma \Longrightarrow \text{valid } (\pi \bullet \Gamma)$$

$$\Gamma \vdash t : T \Longrightarrow \pi \bullet \Gamma \vdash \pi \bullet t : \pi \bullet T$$

```
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 ⇒ 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"
```

```
nominal_inductive typing
```

Typing Judgements (2)

```
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"
```

Subgoals

1. $\bigwedge x\ \Gamma\ T_1\ t\ T_2.\ [\![x \# \Gamma; (x, T_1)::\Gamma \vdash t : T_2]\!] \Longrightarrow x \# \Gamma$
2. $\bigwedge x\ \Gamma\ T_1\ t\ T_2.\ [\![x \# \Gamma; (x, T_1)::\Gamma \vdash t : T_2]\!] \Longrightarrow x \# \mathrm{Lam}\ [x].t$
3. $\bigwedge x\ \Gamma\ T_1\ t\ T_2.\ [\![x \# \Gamma; (x, T_1)::\Gamma \vdash t : T_2]\!] \Longrightarrow x \# T_1 \to T_2$

```
nominal_inductive typing
```

Typing Judgements (2)

```
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"

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
```

Typing Judgements (2)

```
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"

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 Γ1 Γ2::"ty_ctx"
  assumes a: "Γ1 ⊢ t : T"
  and     b: "valid Γ2"
  and     c: "Γ1 ⊆ Γ2"
  shows "Γ2 ⊢ t : T"
using a b c
by (nominal_induct Γ1 t T avoiding: Γ2 rule: typing.strong_induct)
   (auto simp add: atomize_all atomize_imp)
```

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"
using a b c
by (nominal_induct Γ1 t T avoiding: Γ2 rule: typing.strong_induct)
  (auto simp add: atomize_all atomize_imp)
```

- This proof is can be found automatically, but that tells us not much. . .

Lemma / Theorem / Corollary

- Lemmas / Theorems / Corollary are of the form:

```
theorem theorem_name:
 fixes      x::"type"
 ...
 assumes    "assm1"
 and        "assm2"
 ...
 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  "assm1"
  and      "assm2"
  ...
  shows   "s
  ...
```

- Grey parts are optional.
- Assumptions and the (g… type bool.

lemma weakening:
  fixes Γ_1 Γ_2::"ty_ctx"
  assumes a: "$\Gamma_1 \vdash t : T$"
  and b: "valid Γ_2"
  and c: "$\Gamma_1 \subseteq \Gamma_2$"
  shows "$\Gamma_2 \vdash t : T$"

Struct. of an Ind. Proof

```
lemma weakening:
  fixes Γ₁ Γ₂::"ty_ctx"
  assumes a: "Γ₁ ⊢ t : T"
  and     b: "valid Γ₂"
  and     c: "Γ₁ ⊆ Γ₂"
  shows "Γ₂ ⊢ t : T"
using a b c
proof(nominal_induct Γ₁ t T avoiding: Γ₂ rule: typing.strong_induct)
  case (t_Var Γ₁ x T)
  ...
  show "Γ₂ ⊢ Var x : T"
  ...
next
  case (t_App Γ₁ t₁ T₁ T₂ t₂)
  ...
  show "Γ₂ ⊢ App t₁ t₂ : T₂"
  ...
next
  case (t_Lam x Γ₁ T₁ t T₂)
  ...
  show "Γ₂ ⊢ Lam [x].t : T₁ → T₂"
  ...
qed
```

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** ...

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 . . .
```

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

  . . .

$$\frac{\text{valid } \Gamma \qquad (x, T) \in \text{set } \Gamma}{\Gamma \vdash \text{Var } x : T}$$

```
lemma weakening:
  fixes Γ₁ Γ₂::"ty_ctx"
  assumes a: "Γ₁ ⊢ t : T"
  and     b: "valid Γ₂"
  and     c: "Γ₁ ⊆ Γ₂"
  shows "Γ₂ ⊢ t : T"
using a b c
proof(nominal_induct Γ₁ t T avoiding: Γ₂ rule: typing.strong_induct)
  case (t_Var Γ₁ x T)
  have a1: "valid Γ₂" by fact
  have a2: "Γ₁ ⊆ Γ₂" by fact
  have a3: "(x,T) ∈ (set Γ₁)" by fact
  have a4: "(x,T) ∈ (set Γ₂)" using a2 a3 by simp
  show "Γ₂ ⊢ Var x : T" using a1 a4 by auto
next ...
```

$$\frac{x \# \Gamma \qquad (x, T_1)::\Gamma \vdash t : T_2}{\Gamma \vdash \text{Lam } [x].t : T_1 \to T_2}$$

```
next
  case (t_Lam x Γ1 T1 t T2)
  have vc: "x#Γ2" by fact
  have ih: "⟦valid ((x,T1)#Γ2); (x,T1)#Γ1 ⊆ (x,T1)#Γ2⟧
                                   ⟹ (x,T1)#Γ2 ⊢ t:T2" by fact
  have a1: "Γ1 ⊆ Γ2" by fact
  have a2: "(x,T1)#Γ1 ⊆ (x,T1)#Γ2" using a1 by simp
  have b1: "valid Γ2" by fact
  have b2: "valid ((x,T1)#Γ2)" using vc b1 by auto
  have b3: "(x,T1)#Γ2 ⊢ t : T2" using ih b2 a2 by simp
  show "Γ2 ⊢ Lam [x].t : T1→T2" using b3 vc by auto
next ...
```

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

```
next
  case (t_Lam x Γ₁ T₁ t T₂)
  have vc: "x#Γ₂" by fact
  have ih: "⟦valid ((x,T₁)#Γ₂); (x,T₁)#Γ₁ ⊆ (x,T₁)#Γ₂⟧
                                    ⟹ (x,T₁)#Γ₂ ⊢ t:T₂" by fact
  have "Γ₁ ⊆ Γ₂" by fact
  then have a2: "(x,T₁)#Γ₁ ⊆ (x,T₁)#Γ₂" by simp
  have "valid Γ₂" by fact
  then have b2: "valid ((x,T₁)#Γ₂)" using vc by auto
  have "(x,T₁)#Γ₂ ⊢ t : T₂" using ih b2 a2 by simp
  then show "Γ₂ ⊢ Lam [x].t : T₁→T₂" using vc by auto
next ...
```

A Sequence of Facts

```
have n1: "..."
have n2: "..."

...

have nn: "..."
have "..." using n1 n2...nn
```

```
have "..."
moreover have "..."

...

moreover have "..."
ultimately have "..."
```

$$\frac{x \# \Gamma \qquad (x, T_1)::\Gamma \vdash t : T_2}{\Gamma \vdash \mathrm{Lam}\ [x].t : T_1 \rightarrow T_2}$$

```
next
  case (t_Lam x Γ₁ T₁ t T₂)
  have vc: "x#Γ₂" by fact
  have ih: "⟦valid ((x,T₁)#Γ₂); (x,T₁)#Γ₁ ⊆ (x,T₁)#Γ₂⟧
                                   ⟹ (x,T₁)#Γ₂ ⊢ t:T₂" by fact
  have "Γ₁ ⊆ Γ₂" by fact
  then have "(x,T₁)#Γ₁ ⊆ (x,T₁)#Γ₂" by simp
  moreover
  have "valid Γ₂" by fact
  then have "valid ((x,T₁)#Γ₂)" using vc by auto
  ultimately have "(x,T₁)#Γ₂ ⊢ t : T₂" using ih by simp
  then show "Γ₂ ⊢ Lam [x].t : T₁→T₂" using vc by auto
next ...
```

$$\frac{x \# \Gamma \qquad (x, T_1)::\Gamma \vdash t : T_2}{\Gamma \vdash \mathrm{Lam}\ [x].t : T_1 \rightarrow T_2}$$

```
next
  case (t_Lam x Γ1 T1 t T2)
  have vc: "x#Γ2" by fact
  have ih: "⟦valid ((x,T1)#Γ2); (x,T1)#Γ1 ⊆ (x,T1)#Γ2⟧
                                  ⟹ (x,T1)#Γ2 ⊢ t:T2" by fact
  have "Γ1 ⊆ Γ2" by fact
  then have "(x,T1)#Γ1 ⊆ (x,T1)#Γ2" by simp
  moreover
  have "valid Γ2" by fact
  then have "valid ((x,T1)#Γ2)" using vc by auto
  ultimately have "(x,T1)#Γ2 ⊢ t : T2" using ih by simp
  then show "Γ2 ⊢ Lam [x].t : T1→T2" using vc by auto
qed (auto)
```

Capture-Avoiding Subst.

- We next want to introduce an evaluation relation and a CK machine.
- For this we need the notion of capture-avoiding substitution.

```
consts
 subst :: "lam ⇒ name ⇒ lam ⇒ lam"  ("_[_::=_]")

nominal_primrec
 "(Var x)[y::=s] = (if x=y then s else (Var x))"
 "(App t₁ t₂)[y::=s] = App (t₁[y::=s]) (t₂[y::=s])"
 "x#(y,s) ⟹ (Lam [x].t)[y::=s] = Lam [x].(t[y::=s])"
```

Capture-Avoiding Subst.

- We next want to introduce an evaluation relation and a CK machine.
- For this we need the notion of capture-avoiding substitution.

```
consts
 subst :: "lam ⇒ name ⇒ lam ⇒ lam"  ("_[_::=_]")

nominal_primrec
 "(Var x)[y::=s] = (if x=y then s else (Var x))"
 "(App t₁ t₂)[y::=s] = App (t₁[y::=s]) (t₂[y::=s])"
 "x#(y,s) ⟹ (Lam [x].t)[y::=s] = Lam [x].(t[y::=s])"
```

- Despite its looks, this is a total function!

Bound Names Function

- However there is a problem with the bound names function:

```
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)
```

Bound Names Function

Assume $x \neq y$.

- Howe
  funct

consts
  bnds :: "lo

nominal_pr
  "bnds (Var x) = {}"
  "bnds (App t_1 t_2) = bnds (t_1) ∪ bnds (t_2)"
  "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)

Bound Names Function

- Howe
  funct

consts
bnds :: "lo

nominal_pr
"bnds (Var x) = {}"
"bnds (App t_1 t_2) = bnds (t_1) ∪ bnds (t_2)"
"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)

Assume $x \neq y$.

Lam [x].Var x = Lam [y].Var y

Bound Names Function

- Howe
  funct

consts
bnds :: "lc

nominal_pr
"bnds (Var x) = {}"
"bnds (App t_1 t_2) = bnds (t_1) ∪ bnds (t_2)"
"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)

Assume $x \neq y$.

Lam [x].Var x = Lam [y].Var y

bnds (Lam [x].Var x) = bnds (Lam [y].Var y)

Bound Names Function

- Howe
  funct

consts
  bnds :: "lc

nominal_pr
  "bnds (Var x) = {}"
  "bnds (App t_1 t_2) = bnds (t_1) ∪ bnds (t_2)"
  "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)

Assume $x \neq y$.

Lam [x].Var x = Lam [y].Var y

bnds (Lam [x].Var x) = bnds (Lam [y].Var y)

{x} = {y}

Bound Names Function

- However there is a problem with the bound names function:

```
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)
```

Capture-Avoiding Subst.

consts

subst :: "lam ⇒ name ⇒ lam ⇒ 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) ⟹ (Lam [x].t)[y::=s] = Lam [x].(t[y::=s])"

Capture-Avoiding Subst.

consts

subst :: "lam ⇒ name ⇒ lam ⇒ 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) ⟹ (Lam [x].t)[y::=s] = Lam [x].(t[y::=s])"

Freshness Condition for Binders (FCB)

$$\forall a\, ts.\ a \# f \Rightarrow a \# f\, a\, ts$$

Capture-Avoiding Subst.

```
consts
  subst :: "lam ⇒ name ⇒ lam ⇒ lam"  ("_[_::=_]")

nominal_primrec
  "(Var x)[y::=s] = (if x=y then s else (Var x))"
  "(App t₁ t₂)[y::=s] = App (t₁[y::=s]) (t₂[y::=s])"
  "x#(y,s) ⟹ (Lam [x].t)[y::=s] = Lam [x].(t[y::=s])"
```

Freshness Condition for Binders (FCB)

$$\forall a\, ts.\ a \# f \Rightarrow a \# f\ a\ ts$$

$$\bigwedge x1\ y1.\ \ldots\ \ldots \Longrightarrow x1 \# \mathrm{Lam}\ [x1].y1$$

Capture-Avoiding Subst.

```
consts
  subst :: "lam ⇒ name ⇒ lam ⇒ lam"  ("_[_::=_]")

nominal_primrec
  "(Var x)[y::=s] = (if x=y then s else (Var x))"
  "(App t₁ t₂)[y::=s] = App (t₁[y::=s]) (t₂[y::=s])"
  "x#(y,s) ⟹ (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)

$$\forall a\, ts.\ a \# f \Rightarrow a \# f\ a\ ts$$

⋀x1 y1. ... ... ⟹ x1 # Lam [x1].y1

Capture-Avoiding Subst.

FCB for Bound Variable Function:

$\bigwedge$x1 y1. ... ... ⟹ x1 # (y1 ∪ {x1})

consts
subst :: "lam ⇒ name ⇒ lam ⇒ 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) ⟹ (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)

$$\forall a\, ts.\ a \# f \Rightarrow a \# f\ a\ ts$$

$\bigwedge$x1 y1. ... ... ⟹ x1 # Lam [x1].y1

Evaluation Relation

```
inductive
  eval :: "lam ⇒ lam ⇒ bool" ("_ ⇓ _")
where
  e_Lam: "Lam [x].t ⇓ Lam [x].t"
| e_App: "⟦t₁⇓ Lam [x].t; t₂⇓ v'; t[x::=v']⇓ v⟧ ⟹ App t₁ t₂ ⇓ v"

declare eval.intros[intro]
```

Evaluation Relation

```
inductive
  eval :: "lam ⇒ lam ⇒ bool" ("_ ⇓ _")
where
  e_Lam: "Lam [x].t ⇓ Lam [x].t"
| e_App: "⟦t₁⇓ Lam [x].t; t₂⇓ v'; t[x::=v']⇓ v⟧ ⟹ App t₁ t₂ ⇓ v"

declare eval.intros[intro]
```

$$\frac{}{\mathsf{Lam}\ [x].t \Downarrow \mathsf{Lam}\ [x].t}$$

$$\frac{t_1 \Downarrow \mathsf{Lam}\ [x].t \qquad t_2 \Downarrow v' \qquad t[x::=v'] \Downarrow v}{\mathsf{App}\ t_1\ t_2 \Downarrow v}$$

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 ⇒ 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)
```

- If our language contained natural numbers, booleans, etc., we would expand on this definition.

CK Machine

- A CK machine works on configurations ⟨_,_⟩ consisting of a lambda-term and a list of contexts.

```
inductive
  machine :: "lam⇒ctxs⇒lam⇒ctxs⇒bool"   ("⟨_,_⟩ ↦ ⟨_,_⟩")
where
  m1: "⟨App e1 e2,Es⟩ ↦ ⟨e1,(CAppL □ e2)#Es⟩"
| m2: "val v ⟹ ⟨v,(CAppL □ e2)#Es⟩ ↦ ⟨e2,(CAppR v □)#Es⟩"
| m3: "val v ⟹ ⟨v,(CAppR (Lam [x].e) □)#Es⟩ ↦ ⟨e[x::=v],Es⟩"
```

CK Machine

- A CK machine works on configurations ⟨_,_⟩ consisting of a lambda-term and a list of contexts.

```
inductive
  machine :: "lam⇒ctxs⇒lam⇒ctxs⇒bool"   ("⟨_,_⟩ ↦ ⟨_,_⟩")
where
  m1: "⟨App e1 e2,Es⟩ ↦ ⟨e1,(CAppL □ e2)#Es⟩"
| m2: "val v ⟹ ⟨v,(CAppL □ e2)#Es⟩ ↦ ⟨e2,(CAppR v □)#Es⟩"
| m3: "val v ⟹ ⟨v,(CAppR (Lam [x].e) □)#Es⟩ ↦ ⟨e[x::=v],Es⟩"
```

Initial state of the CK machine:

⟨t,[]⟩

CK Machine

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

inductive
  machine :: "lam⇒ctxs⇒lam⇒ctxs⇒bool"   ("$\langle\_,\_\rangle \mapsto \langle\_,\_\rangle$")
where
  m_1: "$\langle$App e_1 e_2,Es$\rangle \mapsto \langle e_1$,(CAppL □ e_2)#Es$\rangle$"
| m_2: "val v $\Longrightarrow$ $\langle$v,(CAppL □ e_2)#Es$\rangle \mapsto \langle e_2$,(CAppR v □)#Es$\rangle$"
| m_3: "val v $\Longrightarrow$ $\langle$v,(CAppR (Lam [x].e) □)#Es$\rangle \mapsto \langle$e[x::=v],Es$\rangle$"

inductive
  "machines" :: "lam⇒ctxs⇒lam⇒ctxs⇒bool" ("$\langle\_,\_\rangle \mapsto^* \langle\_,\_\rangle$")
where
  ms_1: "$\langle$e,Es$\rangle \mapsto^* \langle$e,Es$\rangle$"
| ms_2: "$[\![\langle e_1,Es_1\rangle \mapsto \langle e_2,Es_2\rangle; \langle e_2,Es_2\rangle \mapsto^* \langle e_3,Es_3\rangle]\!]$
    $\Longrightarrow \langle e_1,Es_1\rangle \mapsto^* \langle e_3,Es_3\rangle$"

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:

$$t \Downarrow v \Longleftrightarrow \langle t, [] \rangle \mapsto^* \langle v, [] \rangle$$

with v being a value.

Left-to-Right Direction

```
corollary eval_implies_machines:
  assumes a: "t ⇓ t'"
  shows "⟨t,[]⟩ ↦* ⟨t',[]⟩"
using a using eval_implies_machines_ctx by simp
```

Left-to-Right Direction

```
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: "t ⇓ t'"
  shows "⟨t,[]⟩ ↦* ⟨t',[]⟩"
using a using eval_implies_machines_ctx by simp
```

Left-to-Right Direction

```
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
```

Left-to-Right Direction

lemma ms_3:
  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.

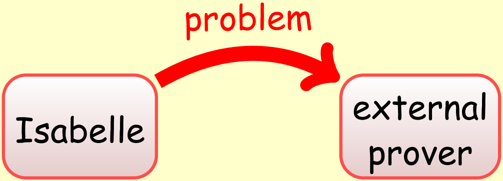
Isabelle

Left-to-Right Direction

lemma ms$_3$:
  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.

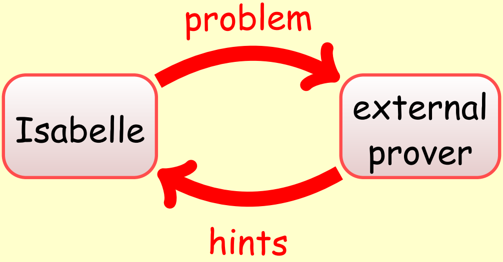


Left-to-Right Direction

lemma ms_3:
  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.



Left-to-Right Direction

```
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
```

Right-to-Left Direction

- The statement for the other direction is as follows:

```
lemma machines_implies_eval:
  assumes a: "⟨t,[]⟩ ↦* ⟨v,[]⟩"
  and     b: "val v"
  shows "t ⇓ v"
```

Right-to-Left Direction

- 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
```

Right-to-Left Direction

- 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
  cbv_1: "val v ⟹ App (Lam [x].t) v ⟶cbv t[x::=v]"
| cbv_2: "t ⟶cbv t' ⟹ App t t_2 ⟶cbv App t' t_2"
| cbv_3: "t ⟶cbv t' ⟹ App t_2 t ⟶cbv App t_2 t'"

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

CBV Reduction

inductive
cbv :: "lam⇒lam⇒bool" ("_ ⟶cbv _")
where
cbv_1: "val v ⟹ App (Lam [x].t) v ⟶cbv t[x::=v]"
| cbv_2: "t ⟶cbv t' ⟹ App t t_2 ⟶cbv App t' t_2"
| cbv_3: "t ⟶cbv t' ⟹ App t_2 t ⟶cbv App t_2 t'"

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

Conditions:

1. ⋀v x t. val v ⟹ x # App Lam [x].t v
2. ⋀v x t. val v ⟹ x # t[x::=v]

CBV Reduction

```
inductive
  cbv :: "lam⇒lam⇒bool" ("_ ⟶cbv _")
where
  cbv₁: "⟦val v; x#v⟧ ⟹ App (Lam [x].t) v ⟶cbv t[x::=v]"
| cbv₂[intro]: "t ⟶cbv t' ⟹ App t t₂ ⟶cbv App t' t₂"
| cbv₃[intro]: "t ⟶cbv t' ⟹ App t₂ t ⟶cbv App t₂ t'"
```

- The conditions that give us automatically the strong induction principle require us to add the assumption $x \# v$. This makes this rule less useful.

Strong Induction Principle

```
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
```

CBV Reduction*

```
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" :: "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)

lemma cbv_in_ctx:
  assumes a: "t ⟶cbv t'"
  shows "E⟦t⟧ ⟶cbv E⟦t'⟧"
using a by (induct E) (auto)
```

CK Machine Implies CBV*

```
lemma machines_implies_cbvs:
  assumes a: "⟨e,[]⟩ ↦* ⟨e',[]⟩"
  shows "e ⟶cbv* e'"
using a by (auto dest: machines_implies_cbvs_ctx)
```

CK Machine Implies CBV*

```
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)
```

```
lemma machines_implies_cbvs:
  assumes a: "⟨e,[]⟩ ↦* ⟨e',[]⟩"
  shows "e ⟶cbv* e'"
using a by (auto dest: machines_implies_cbvs_ctx)
```

CK Machine Implies CBV*

```
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 renaming here.

```
lemma machines_implies_cbvs:
  assumes a: "⟨e,[]⟩ ↦* ⟨e',[]⟩"
  shows "e ⟶cbv* e'"
using a by (auto dest: machines_implies_cbvs_ctx)
```

CK Machine Implies CBV*

```
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)

lemma machines_implies_cbvs_ctx:
  assumes a: "⟨e,Es⟩ ↦* ⟨e',Es'⟩"
  shows "(Es↓)⟦e⟧ ⟶cbv* (Es'↓)⟦e'⟧"
using a
by (induct) (auto dest: machine_implies_cbvs_ctx)

lemma machines_implies_cbvs:
  assumes a: "⟨e,[]⟩ ↦* ⟨e',[]⟩"
  shows "e ⟶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 ⟶cbv t₂" "t₂ ⇓ t₃"
  shows "t₁ ⇓ t₃"
using a
by (induct arbitrary: t₃)
   (auto intro: eval_val dest!: e_App_elim)

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

lemma cbvs_implies_eval:
  assumes a: "t ⟶cbv* v" "val v"
  shows "t ⇓ v"
using a
by (induct)
   (auto simp add: eval_val cbvs_eval dest: cbvs₂)
```

Right-to-Left Direction

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

```
theorem machines_implies_eval:
  assumes a: "⟨t₁,[]⟩ ↦* ⟨t₂,[]⟩"
  and      b: "val t₂"
  shows "t₁ ⇓ t₂"
proof -
  from a have "t₁ ⟶cbv* t₂" by (simp add: machines_implies_cbvs)
  then show "t₁ ⇓ t₂" using b by (simp add: cbvs_implies_eval)
qed
```

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 ⟶cbv t'"
  and       b: "Γ ⊢ t : T"
  shows "Γ ⊢ t' : T"
```

```
theorem progress:
  assumes a: "[] ⊢ t : T"
  shows "(∃ t'. t ⟶cbv t') ∨ (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 ⟶cbv t'"
  and     b: "Γ ⊢ t : T"
  shows "Γ ⊢ t' : T"
```

```
theorem progress:
  assumes a: "[] ⊢ t : T"
  shows "(∃ t'. t ⟶cbv t') ∨ (val t)"
```

- We need the property of type-substitutivity.

Some Side-Lemmas

```
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)
```

```
lemma type_substitution_aux:
  assum
  and
  shows
using a
proof (r
                                    avoiding: x e' Δ rule: typing.strong_induct)
```

```
corollary type_substitution:
  assumes a: "(x,T')#Γ ⊢ e : T"
  and     b: "Γ ⊢ e' : T'"
  shows "Γ ⊢ e[x::=e'] : T"
```

```
  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)+
```

$$\frac{\text{valid } \Gamma \quad (x, T) \in \text{set } \Gamma}{\Gamma \vdash \text{Var } x : T}$$

```
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)
```

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash \mathsf{App}\ t_1\ t_2 : T_2} \qquad \frac{x \# \Gamma \quad (x, T_1)::\Gamma \vdash t : T_2}{\Gamma \vdash \mathsf{Lam}\ [x].t : T_1 \to T_2}$$

Type Preservation

```
theorem cbv_type_preservation:
  assumes a: "t ⟶cbv t'"
  and       b: "Γ ⊢ t : T"
  shows "Γ ⊢ t' : T"
using a b
by (nominal_induct avoiding: Γ 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 ⟶cbv* t'"
  and     b: "Γ ⊢ t : T"
  shows "Γ ⊢ t' : T"
using a b
by (induct) (auto intro: cbv_type_preservation)
```

Progress Lemma

- Finally we can establish the progress lemma:

```
lemma canonical_tArr:
  assumes a: "[] ⊢ t : T1 → T2"
  and       b: "val t"
  shows "∃ x t'. t = Lam [x].t'"
using b a by (induct) (auto)



theorem progress:
  assumes a: "[] ⊢ t : T"
  shows "(∃ t'. t ⟶cbv t') ∨ (val t)"
using a
by (induct Γ≡"[]::ty_ctx" t T)
    (auto intro!: cbv.intros dest: canonical_tArr)
```

Progress Lemma

- Finally we can establish the progress lemma:

```
lemma canonical_tArr:
  assumes a: "[] ⊢ t : T1 → T2"
  and      b: "val t"
  shows "∃ x t'. t = Lam [x].t'"
using b a by (induct) (auto)
```

- This lemma is stated with extensions in mind.

```
theorem progress:
  assumes a: "[] ⊢ t : T"
  shows "(∃ t'. t ⟶cbv t') ∨ (val t)"
using a
by (induct Γ≡"[]::ty_ctx" t T)
   (auto intro!: cbv.intros dest: canonical_tArr)
```

Extensions

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

```
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

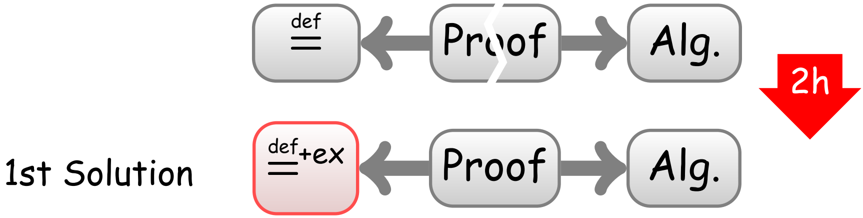
(joint work with Cheney and Berghofer)

Formalisation of LF

(joint work with Cheney and Berghofer)

Formalisation of LF

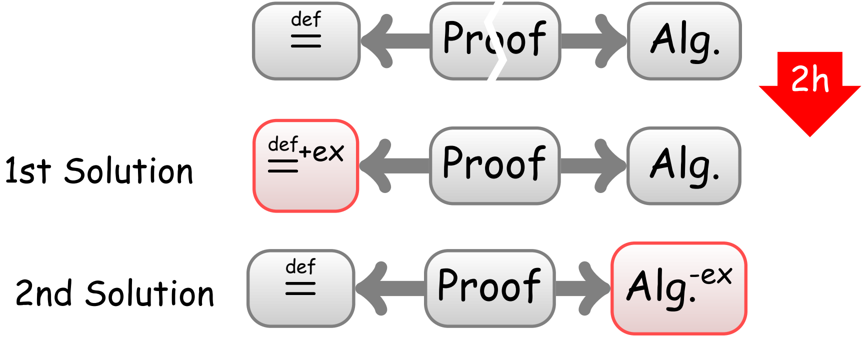
(joint work with Cheney and Berghofer)



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

Formalisation of LF

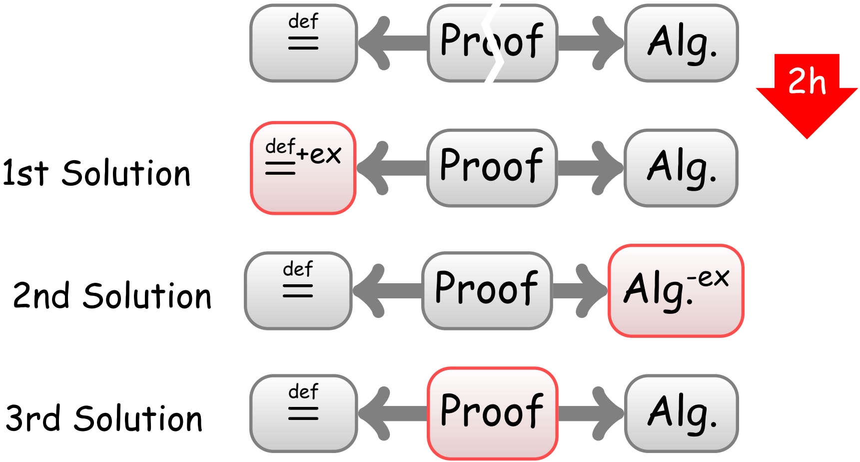
(joint work with Cheney and Berghofer)



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

Formalisation of LF

(joint work with Cheney and Berghofer)



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

Two Health Warnings ;o)

Theorem provers should come with two health warnings:

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...”)

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?

Answers to Exercises

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

Answers to Exercises

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

Answers to Exercises

- Given a finite set of atoms. What is the support of this set? If S is finite, then $\mathsf{supp}(S) = S$.
- What is the support of the set of all atoms? Let $A = \{a_0, a_1 \ldots\}$, then $\mathsf{supp}(A) = \varnothing$.

Answers to Exercises

- Given a finite set of atoms. What is the support of this set? If S is finite, then $\mathrm{supp}(S) = S$.
- What is the support of the set of all atoms? Let $A = \{a_0, a_1 \ldots\}$, then $\mathrm{supp}(A) = \varnothing$.
- From the set of all atoms take one atom out. What is the support of the resulting set?

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 $\text{supp}(A) = \varnothing$.
- 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 $\mathrm{supp}(S) = S$.
- What is the support of the set of all atoms? Let $A = \{a_0, a_1 \ldots\}$, then $\mathrm{supp}(A) = \varnothing$.
- From the set of all atoms take one atom out. What is the support of the resulting set? $\mathrm{supp}(A - \{a\}) = \{a\}$.
- Are there any sets of atoms that have infinite support?

Answers to Exercises

- Given a finite set of atoms. What is the support of this set? If S is finite, then $\mathrm{supp}(S) = S$.
- What is the support of the set of all atoms? Let $A = \{a_0, a_1 \ldots\}$, then $\mathrm{supp}(A) = \varnothing$.
- From the set of all atoms take one atom out. What is the support of the resulting set? $\mathrm{supp}(A - \{a\}) = \{a\}$.
- Are there any sets of atoms that have infinite support? If both S and $A - S$ are infinite then $\mathrm{supp}(S) = A$.

Thank you very much!