Welcome!

- Files and Programme at:

  `http://isabelle.in.tum.de/nominal/ijcar-09.html`

- Have you already installed Nominal Isabelle?

- Can you step through Minimal.thy without getting an error message?

If yes, then very good.
If not, then please ask us **now!**

Nominal Isabelle

Stefan Berghofer and Christian Urban
TU Munich

Quick overview: a formalisation of a CK machine:

$$\begin{tikzcd}
_ \Downarrow _ \arrow[rr] & & CK \arrow[dl] \\
& _ \longrightarrow_{cbv} _ \arrow[ul] &
\end{tikzcd}$$

A Quick and Dirty Overview of Nominal Isabelle

- Nominal Isabelle is a definitional extension of Isabelle/HOL (i.e. no additional axioms, only HOL),

A Quick and Dirty Overview of Nominal Isabelle

- Nominal Isabelle is a definitional extension of Isabelle/HOL (i.e. no additional axioms, only HOL),
- provides an infrastructure for reasoning about named binders,

A Quick and Dirty Overview of Nominal Isabelle

- Nominal Isabelle is a definitional extension of Isabelle/HOL (i.e. no additional axioms, only HOL),
- provides an infrastructure for reasoning about named binders,
- for example lets you define

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

- which give you named α-equivalence classes:

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

A Quick and Dirty Overview of Nominal Isabelle

- Nomi
  Isab
  HOL)
- provi
  name
- for e

  nor

  |
  | Lam

That means Nominal Isabelle is aimed at helping you with formalising results from:

- programming language theory
- term-rewriting
- logic
- ...

- which give you named α-equivalence classes:

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

A Quick and Dirty Overview of Nominal Isabelle

- Nomi
  Isabe
  HOL)
- provi
  name
- for e

  nom

  |
  | Lam "«name»lam" ("Lam [_]._")

That means Nominal Isabelle is aimed at helping you with formalising results from:

- programming language theory
- term-rewriting
- logic
- ...

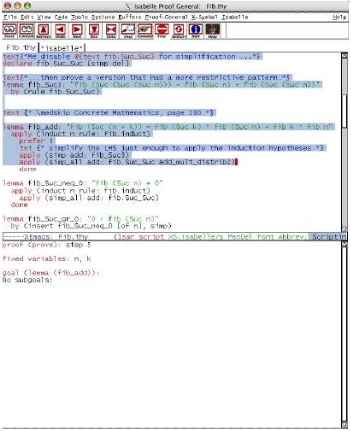
...not just the lambda-calculus!

- which give you named α-equivalence classes:

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

A Six-Slides Crash-Course on How to Use Isabelle

Proof General



Important buttons:

- $\boxed{\textbf{Next}}$ and $\boxed{\textbf{Undo}}$ advance / retract the processed part
- $\boxed{\textbf{Goto}}$ jumps to the current cursor position, same as **ctrl-c/ctrl-return**

Feedback:

- warning messages are given in yellow
- error messages in red

X-Symbols

- ... provide a nice way to input non-ascii characters; for example:

$$\forall\,,\ \exists\,,\ \Downarrow,\ \#,\ \bigwedge,\ \boldsymbol{\Gamma},\ \times\,,\ \neq,\ \in,\ \ldots$$

- they need to be input via the combination
  \<name-of-x-symbol>

X-Symbols

- …provide a nice way to input non-ascii characters; for example:

$$\forall\,,\ \exists\,,\Downarrow, \#, \bigwedge, \boldsymbol{\Gamma}, \times\,, \neq, \in, \ldots$$

- they need to be input via the combination
  \<name-of-x-symbol>

- short-cuts for often used symbols

$$\begin{array}{ccccccccccc} [| & \ldots & [\![& & \texttt{==>} & \ldots & \Longrightarrow & & /\backslash & \ldots & \wedge \\ |] & \ldots &]\!] & & \texttt{=>} & \ldots & \Rightarrow & & \backslash/ & \ldots & \vee \end{array}$$

Isabelle Proof-Scripts

- Every proof-script (theory) is of the form

```
theory Name
  imports T1...Tn
begin
...
end
```

Isabelle Proof-Scripts

- Every proof-script (theory) is of the form

```
theory Name
  imports T1...Tn
begin
...
end
```

- For Nominal Isabelle proof-scripts, T_1 will normally be the theory **Nominal**.
- We use here the theory Lambda.thy, which contains the definition for lambda-terms and for capture-avoiding substitution.

Types

- Isabelle is typed, has polymorphism and overloading.
    - Base types: nat, bool, string, lam, ...
    - Type-formers: 'a list, 'a $\times$ 'b, 'c set, ...
    - Type-variables: 'a, 'b, 'c, ...

Types

- Isabelle is typed, has polymorphism and overloading.
  - Base types: nat, bool, string, lam, ...
  - Type-formers: 'a list, 'a $\times$ 'b, 'c set, ...
  - Type-variables: 'a, 'b, 'c, ...

- Types can be queried in Isabelle using:

  typ nat
  typ bool
  typ lam
  typ "('a $\times$ 'b)"
  typ "'c set"
  typ "nat $\Rightarrow$ bool"

Terms

- The well-formedness of terms can be queried using:

```
term c
term "1::nat"
term 1
term "{1, 2, 3::nat}"
term "[1, 2, 3]"
term "Lam [x].(Var x)"
term "App t₁ t₂"
```

Terms

- The well-formedness of terms can be queried using:

  term c
  term "1::nat"
  term 1
  term "{1, 2, 3::nat}"
  term "[1, 2, 3]"
  term "Lam [x].(Var x)"
  term "App t_1 t_2"

- Isabelle provides some useful colour feedback

term "True"	gives	"True" :: "bool"
term "true"	gives	"true" :: "'a"
term "∀ x. P x"	gives	"∀ x. P x" :: "bool"

Formulae

- Every formula in Isabelle needs to be of type bool

```
term "True"
term "True ∧ False"
term "{1,2,3} = {3,2,1}"
term "∀ x. P x"
term "A ⟶ B"
```

Formulae

- Every formula in Isabelle needs to be of type bool

  term "True"
  term "True $\wedge$ False"
  term "{1,2,3} = {3,2,1}"
  term "$\forall$ x. P x"
  term "$A \longrightarrow B$"

- When working with Isabelle, you are confronted with an objet logic (HOL) and a meta-logic (Pure)

  term "$A \longrightarrow B$" '=' **term** "$A \Longrightarrow B$"
  term "$\forall$ x. P x" '=' **term** "$\bigwedge$x. P x"

Formulae

- Every formula in Isabelle needs to be of type bool

term "True"
term "True $\wedge$ False"
term "{1,2,3} = {3,2,1}"
term "$\forall$ x. P x"
term "$A \longrightarrow B$"

- When working with Isabelle, you are confronted with an objet logic (HOL) and a meta-logic (Pure)

term "$A \longrightarrow B$" '=' term "$A \Longrightarrow B$"
term "$\forall$ x. P x" '=' term "$\bigwedge$x. P x"

term "$A \Longrightarrow B \Longrightarrow C$" = term "$[\![A; B]\!] \Longrightarrow C$"

Definition for the Evaluation Relation, Contexts and the CK Machine on Six Slides

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"
```

Evaluation Relation

a name

inductive

  eval :: "lam ⇒ lam ⇒ bool"   ("_ ⇓ _")

where

  e_Lam: "Lam [x].t ⇓ Lam [x].t"

| e_App: "$\llbracket t_1 \Downarrow$ Lam [x].t; $t_2 \Downarrow v'$; t[x::=v'] ⇓ v$\rrbracket \Longrightarrow$ App t_1 t_2 ⇓ v"

Evaluation Relation

a type

inductive
  eval :: "lam ⇒ lam ⇒ bool"   ("_ ⇓ _")
where
  e_Lam: "Lam [x].t ⇓ Lam [x].t"
| e_App: "$\llbracket t_1 \Downarrow$ Lam [x].t; $t_2 \Downarrow v'$; t[x::=v'] $\Downarrow v \rrbracket \Longrightarrow$ App t_1 $t_2 \Downarrow v$"

Evaluation Relation

optionally
pretty syntax

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"

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"
```

a clause

another clause

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"
```

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

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

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"
```

optionally
a name

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"
```

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

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"
```

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

- The attribute [intro] adds the corresponding clause to the hint theorem base (later more).

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]

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

- The attribute [intro] adds the corresponding clause to the hint theorem base (later more).

Theorems

- Isabelle's theorem database can be querried using

  thm e_Lam
  thm e_App
  thm conjI
  thm conjunct1

Theorems

- Isabelle's theorem database can be querried using

  thm e_Lam
  thm e_App
  thm conjI
  thm conjunct1

e_Lam: $\mathsf{Lam}\ [?x].?t \Downarrow \mathsf{Lam}\ [?x].?t$

e_App: $[\![?t_1 \Downarrow \mathsf{Lam}\ [?x].?t;\ ?t_2 \Downarrow ?v';\ ?t[?x{::=}?v'] \Downarrow ?v]\!] \Longrightarrow \mathsf{App}\ ?t_1\ ?t_2 \Downarrow ?v$

conjI: $[\![?P;\ ?Q]\!] \Longrightarrow ?P \wedge ?Q$

conjunct1: $?P \wedge ?Q \Longrightarrow ?P$

Theorems

- Isabelle's theorem database can be querried using

  thm e_Lam
  thm e_App
  thm conjI
  thm conjunct1

schematic variables

e_Lam:	$\mathsf{Lam}\ [?x].?t \Downarrow \mathsf{Lam}\ [?x].?t$
e_App:	$[\![?t_1 \Downarrow \mathsf{Lam}\ [?x].?t;\ ?t_2 \Downarrow ?v';\ ?t[?x::=?v'] \Downarrow ?v]\!]$
	$\Longrightarrow \mathsf{App}\ ?t_1\ ?t_2 \Downarrow ?v$
conjI:	$[\![?P;\ ?Q]\!] \Longrightarrow ?P \wedge ?Q$
conjunct1:	$?P \wedge ?Q \Longrightarrow ?P$

Theorems

- Isabelle's theorem database can be querried using

  thm e_Lam[no_vars]
  thm e_App[no_vars]
  thm conjI[no_vars]
  thm conjunct1[no_vars] ← attributes

e_Lam:	$\mathrm{Lam}\ [x].t \Downarrow \mathrm{Lam}\ [x].t$
e_App:	$[\![t_1 \Downarrow \mathrm{Lam}\ [x].t;\ t_2 \Downarrow v';\ t[x::=v'] \Downarrow v]\!] \Longrightarrow$
	$\mathrm{App}\ t_1\ t_2 \Downarrow v$
conjI:	$[\![P;\ Q]\!] \Longrightarrow P \wedge Q$
conjunct1:	$P \wedge Q \Longrightarrow P$

Generated Theorems

- Most definitions result in automatically generated theorems; for example

```
thm eval.intros[no_vars]
thm eval.induct[no_vars]
```

Generated Theorems

- Most definitions result in automatically generated theorems; for example

```
thm eval.intros[no_vars]
thm eval.induct[no_vars]
```

intr's: $\mathrm{Lam}\ [x].t \Downarrow \mathrm{Lam}\ [x].t$

$[\![t_1 \Downarrow \mathrm{Lam}\ [x].t;\ t_2 \Downarrow v';\ t[x::=v'] \Downarrow v]\!] \Longrightarrow \mathrm{App}\ t_1\ t_2 \Downarrow v$

ind'ct: $[\![x_1 \Downarrow x_2;$

$\bigwedge x\ t.\ P\ \mathrm{Lam}\ [x].t\ \mathrm{Lam}\ [x].t;$

$\bigwedge t_1\ x\ t\ t_2\ v'\ v.\ [\![t_1 \Downarrow \mathrm{Lam}\ [x].t;\ P\ t_1\ \mathrm{Lam}\ [x].t;\ t_2 \Downarrow v';\ P\ t_2\ v';\ t[x::=v'] \Downarrow v;\ P\ t[x::=v']\ v]\!] \Longrightarrow P\ (\mathrm{App}\ t_1\ t_2)\ v;]\!]$

$\Longrightarrow P\ x_1\ x_2$

Theorem / Lemma / Corollary

- ... they 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. Assumptions can have labels.

Theorem / Lemma / Corollary

- ...they are of the form:

```
lemma alpha_equ:
  shows "Lam [x].Var x = Lam [y].Var y"
...

lemma Lam_freshness:
  assumes a: "x ≠ y"
  shows "y # Lam [x].t ⟹ y # t"
...

lemma neutral_element:
  fixes x::"nat"
  shows "x + 0 = x"
...
```

- Grey parts
- Assumptio ... of type bool.

Datatypes

- We define contexts with a single hole as the datatype:

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

Datatypes

- We define contexts with a single hole as the datatype:

a name

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

Datatypes

- We define contexts with a single hole as the datatype:

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

constr's

constr's

constr's

Datatypes

- We define contexts with a single hole as the datatype:

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

arg type   arg type

Datatypes

- We define contexts with a single hole as the datatype:

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

pretty syntax

Datatypes

- We define contexts with a single hole as the datatype:

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

- Isabelle now knows about:

```
typ ctx
term "□"
term "CAppL"
term "CAppL □ (Var x)"
```

Datatypes

- We define contexts with a single hole as the datatype:

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

- Isabelle now knows about:

```
typ ctx
term "□"
term "CAppL"
term "CAppL □ (Var x)"

types ctxs = "ctx list"            (a type abbreviation)
```

CK Machine

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

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 $\square$ e_2)#Es$\rangle$"
| m_2: "val v $\Longrightarrow$ $\langle$v,(CAppL $\square$ e_2)#Es$\rangle \mapsto \langle e_2$,(CAppR v $\square$)#Es$\rangle$"
| m_3: "val v $\Longrightarrow$ $\langle$v,(CAppR (Lam [x].e) $\square$)#Es$\rangle \mapsto \langle$e[x::=v],Es$\rangle$"

CK Machine

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

inductive
machine :: "lam⇒ctxs⇒lam⇒ctxs⇒bool" ("⟨_,_⟩ ↦ ⟨_,_⟩")
where
m_1: "⟨App e_1 e_2,Es⟩ ↦ ⟨e_1,(CAppL □ e_2)#Es⟩"
| m_2: "val v ⟹ ⟨v,(CAppL □ e_2)#Es⟩ ↦ ⟨e_2,(CAppR v □)#Es⟩"
| m_3: "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 framestack.

inductive
  machine :: "lam⇒ctxs⇒lam⇒ctxs⇒bool"   ("⟨_,_⟩ ↦ ⟨_,_⟩")
where
  m_1: "⟨App e_1 e_2,Es⟩ ↦ ⟨e_1,(CAppL □ e_2)#Es⟩"
| m_2: "val v ⟹ ⟨v,(CAppL □ e_2)#Es⟩ ↦ ⟨e_2,(CAppR v □)#Es⟩"
| m_3: "val v ⟹ ⟨v,(CAppR (Lam [x].e) □)#Es⟩ ↦ ⟨e[x::=v],Es⟩"

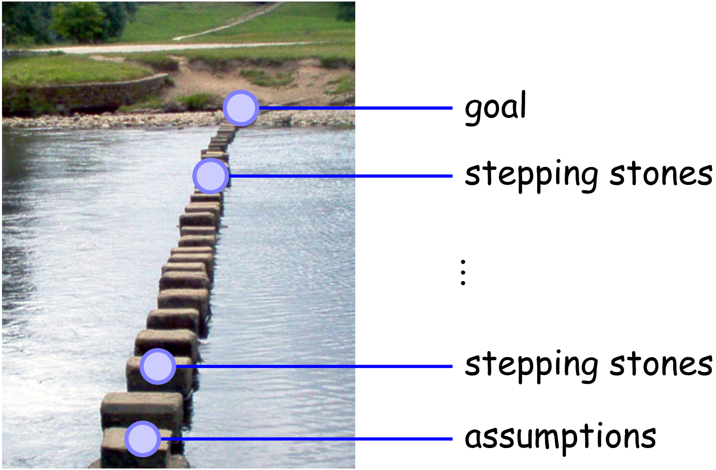
inductive
  machines :: "lam⇒ctxs⇒lam⇒ctxs⇒bool" ("⟨_,_⟩ ↦* ⟨_,_⟩")
where
  ms_1: "⟨e,Es⟩ ↦* ⟨e,Es⟩"
| ms_2: "⟦⟨e_1,Es_1⟩ ↦ ⟨e_2,Es_2⟩; ⟨e_2,Es_2⟩ ↦* ⟨e_3,Es_3⟩⟧
        ⟹ ⟨e_1,Es_1⟩ ↦* ⟨e_3,Es_3⟩"

An Isar Proof for Evaluation implying the CK Machine

An Isar Proof . . .

- The Isar proof language has been conceived by Markus Wenzel, the main developer behind Isabelle.

An Isar Proof . . .



- The Isar proof language has been conceived by Markus Wenzel, the main developer behind Isabelle.

An Isar Proof . . .

- A Rough Schema of an Isar Proof:

```
have     "assumption"
have     "assumption"
...
have     "statement"
have     "statement"
...
show "statement"
qed
```

An Isar Proof . . .

- A Rough Schema of an Isar Proof:

```
have n1: "assumption"
have n2: "assumption"
...
have  n:  "statement"
have  m:  "statement"
...
show "statement"
qed
```

- each have-statement can be given a label

An Isar Proof . . .

- A Rough Schema of an Isar Proof:

```
have n1: "assumption" by justification
have n2: "assumption" by justification
...
have  n:  "statement" by justification
have  m:  "statement" by justification
...
show "statement" by justification
qed
```

- each have-statement can be given a label
- obviously, everything needs to have a justifiation

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

  ...

Justifications

- Omitting proofs

  sorry

- Assumptions

  by fact

- Automated proofs

  by simp
  by auto

  by force

  by blast

  ...

Automatic justifications can also be:

using ... **by** ...

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

First Exercise

- Lets try to prove a simple lemma. Remember we defined

Transitive Closure of the CK Machine:

$$\frac{}{\langle e,Es\rangle \mapsto^* \langle e,Es\rangle}\,ms_1$$

$$\frac{\langle e_1,Es_1\rangle \mapsto \langle e_2,Es_2\rangle \qquad \langle e_2,Es_2\rangle \mapsto^* \langle e_3,Es_3\rangle}{\langle e_1,Es_1\rangle \mapsto^* \langle e_3,Es_3\rangle}\,ms_2$$

lemma
  assumes a: "$\langle e_1,Es_1\rangle \mapsto^* \langle e_2,Es_2\rangle$"
  and b: "$\langle e_2,Es_2\rangle \mapsto^* \langle e_3,Es_3\rangle$"
  shows "$\langle e_1,Es_1\rangle \mapsto^* \langle e_3,Es_3\rangle$"

First Exercise

- Lets try to prove a simple lemma. Remember we defined

Transitive Closure of the CK Machine:

$$\frac{}{\langle e,Es\rangle \mapsto^* \langle e,Es\rangle}\,ms_1$$

$$\frac{\langle e_1,Es_1\rangle \mapsto \langle e_2,Es_2\rangle \quad \langle e_2,Es_2\rangle \mapsto^* \langle e_3,Es_3\rangle}{\langle e_1,Es_1\rangle \mapsto^* \langle e_3,Es_3\rangle}\,ms_2$$

lemma
  assumes a: "$\langle e_1,Es_1\rangle \mapsto^* \langle e_2,Es_2\rangle$"
  and b: "$\langle e_2,Es_2\rangle \mapsto^* \langle e_3,Es_3\rangle$"
  shows "$\langle e_1,Es_1\rangle \mapsto^* \langle e_3,Es_3\rangle$"
using a b
proof (induct)

Proofs by Induction

- Proofs by induction involve cases, which are of the form:

```
proof (induct)
 case (Case-Name x...)
  have "assumption" by justification
  ...
  have "statment" by justification
  ...
  show "statment" by justification
next
 case (Another-Case-Name y...)
  ...
```

Your Turn

$$\frac{}{\langle e, Es\rangle \mapsto^* \langle e, Es\rangle}\,ms_1$$

$$\frac{\langle e_1, Es_1\rangle \mapsto \langle e_2, Es_2\rangle \qquad \langle e_2, Es_2\rangle \mapsto^* \langle e_3, Es_3\rangle}{\langle e_1, Es_1\rangle \mapsto^* \langle e_3, Es_3\rangle}\,ms_2$$

```
lemma
  assumes a: "⟨e1,Es1⟩ ↦* ⟨e2,Es2⟩"
  and     b: "⟨e2,Es2⟩ ↦* ⟨e3,Es3⟩"
  shows "⟨e1,Es1⟩ ↦* ⟨e3,Es3⟩"
using a b
proof (induct)
  case (ms1 e1 Es1)
  have c: "⟨e1,Es1⟩ ↦* ⟨e3,Es3⟩" by fact
  show "⟨e1,Es1⟩ ↦* ⟨e3,Es3⟩" sorry
next
  case (ms2 e1 Es1 e2 Es2 e2' Es2')
  have ih: "⟨e2',Es2'⟩ ↦* ⟨e3,Es3⟩ ⟹ ⟨e2,Es2⟩ ↦* ⟨e3,Es3⟩" by fact
  have d1: "⟨e2',Es2'⟩ ↦* ⟨e3,Es3⟩" by fact
  have d2: "⟨e1,Es1⟩ ↦ ⟨e2,Es2⟩" by fact

  show "⟨e1,Es1⟩ ↦* ⟨e3,Es3⟩" sorry
qed
```

Your Turn

$$\frac{}{\langle e, Es\rangle \mapsto^* \langle e, Es\rangle}\ ms_1$$

$$\frac{\langle e_1, Es_1\rangle \mapsto \langle e_2, Es_2\rangle \quad \langle e_2, Es_2\rangle \mapsto^* \langle e_3, Es_3\rangle}{\langle e_1, Es_1\rangle \mapsto^* \langle e_3, Es_3\rangle}\ ms_2$$

lemma
  assumes a: "$\langle e_1, Es_1\rangle \mapsto^* \langle e_2, Es_2\rangle$"
  and b: "$\langle e_2, Es_2\rangle \mapsto^* \langle e_3, Es_3\rangle$"
  shows "$\langle e_1, Es_1\rangle \mapsto^* \langle e_3, Es_3\rangle$"
using a b
proof (induct)
  case (ms_1 e_1 Es_1)
  have c: "$\langle e_1, Es_1\rangle \mapsto^* \langle e_3, Es_3\rangle$" **by** fact
  show "$\langle e_1, Es_1\rangle \mapsto^* \langle e_3, Es_3\rangle$" **sorry**
next
  case (ms_2 e_1 Es_1 e_2 Es_2 e_2' Es_2')
  have ih: "$\langle e_2', Es_2'\rangle \mapsto^* \langle e_3, Es_3\rangle \Longrightarrow \langle e_2, Es_2\rangle \mapsto^* \langle e_3, Es_3\rangle$" **by** fact
  have d1: "$\langle e_2', Es_2'\rangle \mapsto^* \langle e_3, Es_3\rangle$" **by** fact
  have d2: "$\langle e_1, Es_1\rangle \mapsto \langle e_2, Es_2\rangle$" **by** fact

  show "$\langle e_1, Es_1\rangle \mapsto^* \langle e_3, Es_3\rangle$" **sorry**
qed

Your Turn

$$\frac{}{\langle e,Es\rangle \mapsto^* \langle e,Es\rangle}\,ms_1$$

$$\frac{\langle e_1,Es_1\rangle \mapsto \langle e_2,Es_2\rangle \quad \langle e_2,Es_2\rangle \mapsto^* \langle e_3,Es_3\rangle}{\langle e_1,Es_1\rangle \mapsto^* \langle e_3,Es_3\rangle}\,ms_2$$

```
lemma
  assumes a: "⟨e1,Es1⟩ ↦* ⟨e2,Es2⟩"
  and     b: "⟨e2,Es2⟩ ↦* ⟨e3,Es3⟩"
  shows "⟨e1,Es1⟩ ↦* ⟨e3,Es3⟩"
using a b
proof (induct)
  case (ms1 e1 Es1)
  have c: "⟨e1,Es1⟩ ↦* ⟨e3,Es3⟩" by fact
  show "⟨e1,Es1⟩ ↦* ⟨e3,Es3⟩" using c by simp
next
  case (ms2 e1 Es1 e2 Es2 e2' Es2')
  have ih: "⟨e2',Es2'⟩ ↦* ⟨e3,Es3⟩ ⟹ ⟨e2,Es2⟩ ↦* ⟨e3,Es3⟩" by fact
  have d1: "⟨e2',Es2'⟩ ↦* ⟨e3,Es3⟩" by fact
  have d2: "⟨e1,Es1⟩ ↦ ⟨e2,Es2⟩" by fact

  show "⟨e1,Es1⟩ ↦* ⟨e3,Es3⟩" sorry
qed
```

Your Turn

```
lemma
  assumes a: "⟨e₁,Es₁⟩ ↦* ⟨e₂,Es₂⟩"
  and     b: "⟨e₂,Es₂⟩ ↦* ⟨e₃,Es₃⟩"
  shows "⟨e₁,Es₁⟩ ↦* ⟨e₃,Es₃⟩"
using a b
proof (induct)
  case (ms₁ e₁ Es₁)
  have c: "⟨e₁,Es₁⟩ ↦* ⟨e₃,Es₃⟩" by fact
  show "⟨e₁,Es₁⟩ ↦* ⟨e₃,Es₃⟩" using c by simp
next
  case (ms₂ e₁ Es₁ e₂ Es₂ e₂' Es₂')
  have ih: "⟨e₂',Es₂'⟩ ↦* ⟨e₃,Es₃⟩ ⟹ ⟨e₂,Es₂⟩ ↦* ⟨e₃,Es₃⟩" by fact
  have d1: "⟨e₂',Es₂'⟩ ↦* ⟨e₃,Es₃⟩" by fact
  have d2: "⟨e₁,Es₁⟩ ↦ ⟨e₂,Es₂⟩" by fact
  have d3: "⟨e₂,Es₂⟩ ↦* ⟨e₃,Es₃⟩" using ih d1 by auto
  show "⟨e₁,Es₁⟩ ↦* ⟨e₃,Es₃⟩" sorry
qed
```

$$\frac{}{\langle e,Es\rangle \mapsto^* \langle e,Es\rangle}\,ms_1$$

$$\frac{\langle e_1,Es_1\rangle \mapsto \langle e_2,Es_2\rangle \quad \langle e_2,Es_2\rangle \mapsto^* \langle e_3,Es_3\rangle}{\langle e_1,Es_1\rangle \mapsto^* \langle e_3,Es_3\rangle}\,ms_2$$

Your Turn

```
lemma
  assumes a: "⟨e₁,Es₁⟩ ↦* ⟨e₂,Es₂⟩"
  and     b: "⟨e₂,Es₂⟩ ↦* ⟨e₃,Es₃⟩"
  shows "⟨e₁,Es₁⟩ ↦* ⟨e₃,Es₃⟩"
using a b
proof (induct)
  case (ms₁ e₁ Es₁)
  have c: "⟨e₁,Es₁⟩ ↦* ⟨e₃,Es₃⟩" by fact
  show "⟨e₁,Es₁⟩ ↦* ⟨e₃,Es₃⟩" using c by simp
next
  case (ms₂ e₁ Es₁ e₂ Es₂ e₂' Es₂')
  have ih: "⟨e₂',Es₂'⟩ ↦* ⟨e₃,Es₃⟩ ⟹ ⟨e₂,Es₂⟩ ↦* ⟨e₃,Es₃⟩" by fact
  have d1: "⟨e₂',Es₂'⟩ ↦* ⟨e₃,Es₃⟩" by fact
  have d2: "⟨e₁,Es₁⟩ ↦ ⟨e₂,Es₂⟩" by fact
  have d3: "⟨e₂,Es₂⟩ ↦* ⟨e₃,Es₃⟩" using ih d1 by auto
  show "⟨e₁,Es₁⟩ ↦* ⟨e₃,Es₃⟩"  using d2 d3 by auto
qed
```

$$\frac{}{\langle e,Es\rangle \mapsto^* \langle e,Es\rangle}\,ms_1$$

$$\frac{\langle e_1,Es_1\rangle \mapsto \langle e_2,Es_2\rangle \quad \langle e_2,Es_2\rangle \mapsto^* \langle e_3,Es_3\rangle}{\langle e_1,Es_1\rangle \mapsto^* \langle e_3,Es_3\rangle}\,ms_2$$

Your Turn

```
lemma
  assumes a: "⟨e₁,Es₁⟩ ↦* ⟨e₂,Es₂⟩"
  and       b: "⟨e₂,Es₂⟩ ↦* ⟨e₃,Es₃⟩"
  shows "⟨e₁,Es₁⟩ ↦* ⟨e₃,Es₃⟩"
using a b
proof (induct)
  case (ms₁ e₁ Es₁)
  have c: "⟨e₁,Es₁⟩ ↦* ⟨e₃,Es₃⟩" by fact
  show "⟨e₁,Es₁⟩ ↦* ⟨e₃,Es₃⟩" using c by simp
next
  case (ms₂ e₁ Es₁ e₂ Es₂ e₂' Es₂')
  have ih: "⟨e₂',Es₂'⟩ ↦* ⟨e₃,Es₃⟩ ⟹ ⟨e₂,Es₂⟩ ↦* ⟨e₃,Es₃⟩" by fact
  have d1: "⟨e₂',Es₂'⟩ ↦* ⟨e₃,Es₃⟩" by fact
  have d2: "⟨e₁,Es₁⟩ ↦ ⟨e₂,Es₂⟩" by fact
  have d3: "⟨e₂,Es₂⟩ ↦* ⟨e₃,Es₃⟩" using ih d1 by auto
  show "⟨e₁,Es₁⟩ ↦* ⟨e₃,Es₃⟩"  using d2 d3 by auto
qed
```

Your Turn

```
lemma
  assumes a: "⟨e₁,Es₁⟩ ↦* ⟨e₂,Es₂⟩"
  and     b: "⟨e₂,Es₂⟩ ↦* ⟨e₃,Es₃⟩"
  shows "⟨e₁,Es₁⟩ ↦* ⟨e₃,Es₃⟩"
using a b
proof (induct)
  case (ms₁ e₁ Es₁)
  show "⟨e₁,Es₁⟩ ↦* ⟨e₃,Es₃⟩" by fact
next
  case (ms₂ e₁ Es₁ e₂ Es₂ e₂' Es₂')
  have ih: "⟨e₂',Es₂'⟩ ↦* ⟨e₃,Es₃⟩ ⟹ ⟨e₂,Es₂⟩ ↦* ⟨e₃,Es₃⟩" by fact
  have d1: "⟨e₂',Es₂'⟩ ↦* ⟨e₃,Es₃⟩" by fact
  have d2: "⟨e₁,Es₁⟩ ↦ ⟨e₂,Es₂⟩" by fact
  have d3: "⟨e₂,Es₂⟩ ↦* ⟨e₃,Es₃⟩" using ih d1 by auto
  show "⟨e₁,Es₁⟩ ↦* ⟨e₃,Es₃⟩" using d2 d3 by auto
qed
```

Your Turn

```
lemma
  assumes a: "⟨e1,Es1⟩ ↦* ⟨e2,Es2⟩"
  and     b: "⟨e2,Es2⟩ ↦* ⟨e3,Es3⟩"
  shows "⟨e1,Es1⟩ ↦* ⟨e3,Es3⟩"
using a b
proof (induct)
  case (ms1 e1 Es1)
  show "⟨e1,Es1⟩ ↦* ⟨e3,Es3⟩" by fact
next
  case (ms2 e1 Es1 e2 Es2 e2' Es2')
  have ih: "⟨e2',Es2'⟩ ↦* ⟨e3,Es3⟩ ⟹ ⟨e2,Es2⟩ ↦* ⟨e3,Es3⟩" by fact
  have d2: "⟨e1,Es1⟩ ↦ ⟨e2,Es2⟩" by fact
  have d1: "⟨e2',Es2'⟩ ↦* ⟨e3,Es3⟩" by fact
  have d3: "⟨e2,Es2⟩ ↦* ⟨e3,Es3⟩" using ih d1 by auto
  show "⟨e1,Es1⟩ ↦* ⟨e3,Es3⟩" using d2 d3 by auto
qed
```

Your Turn

```
lemma
  assumes a: "⟨e₁,Es₁⟩ ↦* ⟨e₂,Es₂⟩"
  and      b: "⟨e₂,Es₂⟩ ↦* ⟨e₃,Es₃⟩"
  shows "⟨e₁,Es₁⟩ ↦* ⟨e₃,Es₃⟩"
using a b
proof (induct)
  case (ms₁ e₁ Es₁)
  show "⟨e₁,Es₁⟩ ↦* ⟨e₃,Es₃⟩" by fact
next
  case (ms₂ e₁ Es₁ e₂ Es₂ e₂' Es₂')
  have ih: "⟨e₂',Es₂'⟩ ↦* ⟨e₃,Es₃⟩ ⟹ ⟨e₂,Es₂⟩ ↦* ⟨e₃,Es₃⟩" by fact
  have d2: "⟨e₁,Es₁⟩ ↦ ⟨e₂,Es₂⟩" by fact
  have "⟨e₂',Es₂'⟩ ↦* ⟨e₃,Es₃⟩" by fact
  then have d3: "⟨e₂,Es₂⟩ ↦* ⟨e₃,Es₃⟩" using ih by auto
  show "⟨e₁,Es₁⟩ ↦* ⟨e₃,Es₃⟩" using d2 d3 by auto
qed
```

A Chain of Facts

- Isar allows you to build a chain of facts as follows:

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

...

have ni: "..."
have "..." **using** n1 n2 ... ni

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

...

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

- also works for **show**

Your Turn

```
lemma
  assumes a: "⟨e₁,Es₁⟩ ↦* ⟨e₂,Es₂⟩"
  and     b: "⟨e₂,Es₂⟩ ↦* ⟨e₃,Es₃⟩"
  shows "⟨e₁,Es₁⟩ ↦* ⟨e₃,Es₃⟩"
using a b
proof (induct)
  case (ms₁ e₁ Es₁)
  show "⟨e₁,Es₁⟩ ↦* ⟨e₃,Es₃⟩" by fact
next
  case (ms₂ e₁ Es₁ e₂ Es₂ e₂' Es₂')
  have ih: "⟨e₂',Es₂'⟩ ↦* ⟨e₃,Es₃⟩ ⟹ ⟨e₂,Es₂⟩ ↦* ⟨e₃,Es₃⟩" by fact
  have "⟨e₁,Es₁⟩ ↦ ⟨e₂,Es₂⟩" by fact
  moreover
  have "⟨e₂',Es₂'⟩ ↦* ⟨e₃,Es₃⟩" by fact
  then have "⟨e₂,Es₂⟩ ↦* ⟨e₃,Es₃⟩" using ih by auto
  ultimately show "⟨e₁,Es₁⟩ ↦* ⟨e₃,Es₃⟩" by auto
qed
```

Automatic Proofs

- Do not expect Isabelle to be able to solve automatically **show** "P=NP", but...

lemma
  assumes a: "$\langle e_1, Es_1\rangle \mapsto^* \langle e_2, Es_2\rangle$"
  and b: "$\langle e_2, Es_2\rangle \mapsto^* \langle e_3, Es_3\rangle$"
  shows "$\langle e_1, Es_1\rangle \mapsto^* \langle e_3, Es_3\rangle$"
using a b
by (induct) (auto)

Eval Implies CK

```
theorem
  assumes a: "t ⇓ t'"
  shows "⟨t,[]⟩ ↦* ⟨t',[]⟩"
using a
proof (induct)
  case (e_Lam x t)                                    (no assumption avail.)
  show "⟨Lam [x].t,[]⟩ ↦* ⟨Lam [x].t,[]⟩" sorry
next
  case (e_App t₁ x t t₂ v' v)
  have a1: "t₁ ⇓ Lam [x].t" by fact                  (all assumptions)
  have ih1: "⟨t₁,[]⟩ ↦* ⟨Lam [x].t,[]⟩" by fact
  have a2: "t₂ ⇓ v'" by fact
  have ih2: "⟨t₂,[]⟩ ↦* ⟨v',[]⟩" by fact
  have a3: "t[x::=v'] ⇓ v" by fact
  have ih3: "⟨t[x::=v'],[]⟩ ↦* ⟨v,[]⟩" by fact

  show "⟨App t₁ t₂,[]⟩ ↦* ⟨v,[]⟩" sorry
qed
```

Eval Implies CK

```
theorem
  assumes a: "t ⇓ t'"
  shows "⟨t,[]⟩ ↦* ⟨t',[]⟩"
using a
proof (induct)
  case (e_Lam x t)                                   (no assumption avail.)
  show "⟨Lam [x].t,[]⟩ ↦* ⟨Lam [x].t,[]⟩" sorry
next
  case (e_App t1 x t t2 v' v)
  have a1: "t1 ⇓ Lam [x].t" by fact                   (all assumptions)
  have ih1: "⟨t1,[]⟩ ↦* ⟨Lam [x].t,[]⟩" by fact
  have a2: "t2 ⇓ v'" by fact
  have ih2: "⟨t2,[]⟩ ↦* ⟨v',[]⟩" by fact
  have a3: "t[x::=v'] ⇓ v" by fact
  have ih3: "⟨t[x::=v'],[]⟩ ↦* ⟨v,[]⟩" by fact

  show "⟨App t1 t2,[]⟩ ↦* ⟨v,[]⟩" sorry
qed
```

Eval Implies CK

```
theorem
  assumes a: "t ⇓ t'"
  shows "⟨t,[]⟩ ↦* ⟨t',[]⟩"
using a
proof (induct)
  case (e_Lam x t)                                   (no assumption avail.)
  show "⟨Lam [x].t,[]⟩ ↦* ⟨Lam [x].t,[]⟩" sorry
next
  case (e_App t₁ x t t₂ v' v)
  have a1: "t₁ ⇓ Lam [x].t" by fact                  (all assumptions)
  have ih1: "⟨t₁,[]⟩ ↦* ⟨Lam [x].t,[]⟩" by fact
  have a2: "t₂ ⇓ v'" by fact
  have ih2: "⟨t₂,[]⟩ ↦* ⟨v',[]⟩" by fact
  have a3: "t[x::=v'] ⇓ v" by fact
  have ih3: "⟨t[x::=v'],[]⟩ ↦* ⟨v,[]⟩" by fact

  show "⟨App t₁ t₂,[]⟩ ↦* ⟨v,[]⟩" sorry
qed
```

```
thm machine.intros
thm machines.intros
thm eval_to_val
```

Eval Implies CK

Proof Idea:

$\langle \text{App } t_1\ t_2, [] \rangle$
$\mapsto^* \langle t_1, [\text{CAppL } \Box\ t_2] \rangle$
$\mapsto^* \langle \text{Lam } [x].t, [\text{CAppL } \Box\ t_2] \rangle$
$\mapsto^* \langle t_2, [\text{CAppR } (\text{Lam } [x].t)\ \Box] \rangle$
$\mapsto^* \langle v', [\text{CAppR } (\text{Lam } [x].t)\ \Box] \rangle$
$\mapsto^* \langle t[x::=v'], [] \rangle$
$\mapsto^* \langle v, [] \rangle$

thm machine.intros
thm machines.intros
thm eval_to_val

(no assumption avail.)

m $[x].t, []\rangle$" **sorry**

next
  case (e_App t_1 x t t_2 v' v) (all assumptions)
  have a1: "$t_1 \Downarrow \text{Lam } [x].t$" **by** fact
  have ih1: "$\langle t_1, [] \rangle \mapsto^* \langle \text{Lam } [x].t, [] \rangle$" **by** fact
  have a2: "$t_2 \Downarrow v'$" **by** fact
  have ih2: "$\langle t_2, [] \rangle \mapsto^* \langle v', [] \rangle$" **by** fact
  have a3: "$t[x::=v'] \Downarrow v$" **by** fact
  have ih3: "$\langle t[x::=v'], [] \rangle \mapsto^* \langle v, [] \rangle$" **by** fact

  show "$\langle \text{App } t_1\ t_2, [] \rangle \mapsto^* \langle v, [] \rangle$" **sorry**
qed

Eval Implies CK

```
theorem
  assumes a: "t ⇓ t'"
  shows "⟨t,[]⟩ ↦* ⟨t',[]⟩"
using a
proof (induct)
  case (e_Lam x t)                                  (no assumption avail.)
  show "⟨Lam [x].t,[]⟩ ↦* ⟨Lam [x].t,[]⟩" sorry
next
  case (e_App t₁ x t t₂ v' v)
  have a1: "t₁ ⇓ Lam [x].t" by fact                 (all assumptions)
  have ih1: "⟨t₁,[]⟩ ↦* ⟨Lam [x].t,[]⟩" by fact
  have a2: "t₂ ⇓ v'" by fact
  have ih2: "⟨t₂,[]⟩ ↦* ⟨v',[]⟩" by fact
  have a3: "t[x::=v'] ⇓ v" by fact
  have ih3: "⟨t[x::=v'],[]⟩ ↦* ⟨v,[]⟩" by fact

  show "⟨App t₁ t₂,[]⟩ ↦* ⟨v,[]⟩" sorry
qed
```

```
thm machine.intros
thm machines.intros
thm eval_to_val
```

Eval Implies CK

```
theorem
  assumes a: "t ⇓ t'"
  shows "⟨t,[]⟩ ↦* ⟨t',[]⟩"
using a
proof (induct)
  case (e_Lam x t)                                   (no assumption avail.)
  show "⟨Lam [x].t,[]⟩ ↦* ⟨Lam [x].t,[]⟩" sorry
next
  case (e_App t₁ x t t₂ v' v)
  have a1: "t₁ ⇓ Lam [x].t" by fact                  (all assumptions)
  have ih1: "⟨t₁,[]⟩ ↦* ⟨Lam [x].t,[]⟩" by fact
  have a2: "t₂ ⇓ v'" by fact
  have ih2: "⟨t₂,[]⟩ ↦* ⟨v',[]⟩" by fact
  have a3: "t[x::=v'] ⇓ v" by fact
  have ih3: "⟨t[x::=v'],[]⟩ ↦* ⟨v,[]⟩" by fact

  show "⟨App t₁ t₂,[]⟩ ↦* ⟨v,[]⟩" sorry
qed
```

```
thm machine.intros
thm machines.intros
thm eval_to_val
```

Eval Implies CK

```
theorem
  assumes a: "t ⇓ t'"
  shows "⟨t,Es⟩ ↦* ⟨t',Es⟩"
using a
proof (induct arbitrary: Es)
  case (e_Lam x t)                                  (no assumption avail.)
  show "⟨Lam [x].t,Es⟩ ↦* ⟨Lam [x].t,Es⟩" sorry
next
  case (e_App t₁ x t t₂ v' v)
  have a1: "t₁ ⇓ Lam [x].t" by fact                 (all assumptions)
  have ih1: "⋀Es. ⟨t₁,Es⟩ ↦* ⟨Lam [x].t,Es⟩" by fact
  have a2: "t₂ ⇓ v'" by fact
  have ih2: "⋀Es. ⟨t₂,Es⟩ ↦* ⟨v',Es⟩" by fact
  have a3: "t[x::=v'] ⇓ v" by fact
  have ih3: "⋀Es. ⟨t[x::=v'],Es⟩ ↦* ⟨v,Es⟩" by fact

  show "⟨App t₁ t₂,Es⟩ ↦* ⟨v,Es⟩" sorry
qed
```

```
thm machine.intros
thm machines.intros
thm eval_to_val
```

Finally: Eval Implies CK

```
theorem eval_implies_machines_ctx:
  assumes a: "t ⇓ t'"
  shows "⟨t,Es⟩ ↦* ⟨t',Es⟩"
using a
proof (induct arbitrary: Es)
...

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

Finally: Eval Implies CK

```
theorem eval_implies_machines_ctx:
  assumes a: "t ⇓ t'"
  shows "⟨t,Es⟩ ↦* ⟨t',Es⟩"
using a
proof (induct arbitrary: Es)
...

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

```
thm eval_implies_machines_ctx
```

gives

$?t \Downarrow ?t' \Longrightarrow \langle ?t, ?Es \rangle \mapsto^* \langle ?t', ?Es \rangle$

Weakening Lemma (trivial / routine)

Definition of Types

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

Definition of Types

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

$$\frac{(x\!:\!T) \in \Gamma \quad \mathsf{valid}\ \Gamma}{\Gamma \vdash x : T}
\qquad
\frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}$$

$$\frac{x \# \Gamma \quad (x\!:\!T_1)::\Gamma \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \to T_2}$$

$$\frac{}{\mathsf{valid}\ []}
\qquad
\frac{x \# \Gamma \quad \mathsf{valid}\ \Gamma}{\mathsf{valid}\ (x\!:\!T)::\Gamma}$$

Typing Judgements

```
types ty_ctx = "(name×ty) list"

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

#: list cons
#: freshness (\<sharp>)

types ty_ctx = "(name $\times$ ty) list"

inductive
  valid :: "ty_ctx $\Rightarrow$ bool"
where
  v_1: "valid []"
| v_2: "$[\![$valid Γ; $\boxed{x\#\Gamma}$$]\!] \Longrightarrow$ valid $\boxed{((x,T)\#\Gamma)}$"

inductive
  typing :: "ty_ctx $\Rightarrow$ lam $\Rightarrow$ ty $\Rightarrow$ bool" ("_ $\vdash$ _ : _")
where
  t_Var: "$[\![$valid Γ; $(x,T) \in$ set $\Gamma]\!] \Longrightarrow \Gamma \vdash$ Var x : T"
| t_App: "$[\![\Gamma \vdash t_1 : T_1 \rightarrow T_2; \Gamma \vdash t_2 : T_1]\!] \Longrightarrow \Gamma \vdash$ App t_1 t_2 : T_2"
| t_Lam: "$[\![\boxed{x\#\Gamma}; \boxed{(x,T_1)\#\Gamma} \vdash t : T_2]\!] \Longrightarrow \Gamma \vdash$ Lam [x].t : $T_1 \rightarrow T_2$"

Freshness

- Freshness is a concept automatically defined in Nominal Isabelle; it corresponds roughly to the notion of "not-free-in".

```
lemma
  fixes x::"name"
  shows "x#Lam [x].t"
  and   "x#t₁ ∧ x#t₂ ⟹ x#App t₁ t₂"
  and   "x#(Var y) ⟹ x#y"
  and   "⟦x#t₁; x#t₂⟧ ⟹ x#(t₁,t₂)"
  and   "⟦x#l₁; x#l₂⟧ ⟹ x#(l₁@l₂)"
  and   "x#y ⟹ x≠y"
by (simp_all add: abs_fresh fresh_list_append fresh_atm)
```

Freshness

- Freshness is a concept automatically defined in Nominal Isabelle; it corresponds roughly to the notion of "not-free-in".

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

Freshness

- Freshness is a concept automatically defined in Nominal Isabelle; it corresponds roughly to the notion of “not-free-in”.

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

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

The Weakening Lemma

- We can overload ⊆ for typing contexts, but this means we have to give explicit type-annotations.

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

lemma weakening:
  fixes Γ₁ Γ₂::"(name×ty) list"
  assumes a: "Γ₁ ⊢ t : T"
  and     b: "valid Γ₂"
  and     c: "Γ₁ ⊆ Γ₂"
  shows "Γ₂ ⊢ t : T"
using a b c
proof (induct arbitrary: Γ₂)
```

Your Turn: Variable Case

```
lemma
  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 (induct arbitrary: Γ₂)
  case (t_Var Γ₁ x T)
  have a1: "valid Γ₁" by fact
  have a2: "(x,T) ∈ set Γ₁" by fact
  have a3: "valid Γ₂" by fact
  have a4: "Γ₁ ⊆ Γ₂" by fact
    . . .
  show "Γ₂ ⊢ Var x : T" sorry
```

My Proof for the Variable Case

```
lemma
  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 (induct arbitrary: Γ₂)
  case (t_Var Γ₁ x T)
  have "Γ₁ ⊆ Γ₂" by fact
  moreover
  have "valid Γ₂" by fact
  moreover
  have "(x,T)∈ set Γ₁" by fact
  ultimately show "Γ₂ ⊢ Var x : T" by auto
```

Induction Principle for Typing

- The induction principle that comes with the typing definition is as follows:

$$\forall \Gamma\, x\, T.\ (x\!:\!T) \in \Gamma \wedge \mathsf{valid}\,\Gamma \Rightarrow P\,\Gamma\,(x)\,T$$

$$\begin{array}{l}\forall \Gamma\, t_1\, t_2\, T_1\, T_2. \\ P\,\Gamma\,t_1\,(T_1 \to T_2) \wedge P\,\Gamma\,t_2\,T_1 \Rightarrow P\,\Gamma\,(t_1\,t_2)\,T_2\end{array}$$

$$\begin{array}{l}\forall \Gamma\, x\, t\, T_1\, T_2. \\ x \# \Gamma \wedge P\,((x\!:\!T_1)::\Gamma)\,t\,T_2 \Rightarrow P\,\Gamma(\lambda x.t)\,(T_1 \to T_2)\end{array}$$

$$\overline{\quad \Gamma \vdash t : T \Rightarrow P\,\Gamma\,t\,T \quad}$$

Note the quantifiers!

Proof Idea for the Lambda Cs.

$$\frac{x \# \Gamma \quad (x:T_1)::\Gamma \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \rightarrow T_2}$$

- If $\Gamma_1 \vdash t : T_1$ then $\forall \Gamma_2.$ valid $\Gamma_2 \wedge \Gamma_1 \subseteq \Gamma_2 \Rightarrow \Gamma_2 \vdash t : T_2$

Proof Idea for the Lambda Cs.

$$\frac{x \# \Gamma \quad (x{:}T_1)::\Gamma \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \to T_2}$$

- If $\Gamma_1 \vdash t : T_1$ then $\forall \Gamma_2.\ \text{valid}\ \Gamma_2 \wedge \Gamma_1 \subseteq \Gamma_2 \Rightarrow \Gamma_2 \vdash t : T_2$

<u>For all Γ_1, x, t, T_1 and T_2:</u>

- We know:
  $\forall \Gamma_3.\ \text{valid}\ \Gamma_3 \wedge (x{:}T_1)::\Gamma_1 \subseteq \Gamma_3 \Rightarrow \Gamma_3 \vdash t : T_1$
  $x \# \Gamma_1$
  $\text{valid}\ \Gamma_2$
  $\Gamma_1 \subseteq \Gamma_2$

- We have to show:
  $\Gamma_2 \vdash \lambda x.t : T_1 \to T_2$

Proof Idea for the Lambda Cs.

$$\frac{x \# \Gamma \quad (x{:}T_1)::\Gamma \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \rightarrow T_2}$$

- If $\Gamma_1 \vdash t : T_1$ then $\forall \Gamma_2.\,\text{valid}\,\Gamma_2 \wedge \Gamma_1 \subseteq \Gamma_2 \Rightarrow \Gamma_2 \vdash t : T_2$

For all Γ_1, x, t, T_1 and T_2:

- We know:
  $\forall \Gamma_3.\,\text{valid}\,\Gamma_3 \wedge (x{:}T_1)::\Gamma_1 \subseteq \Gamma_3 \Rightarrow \Gamma_3 \vdash t : T_1$
  $x \# \Gamma_1$
  $\text{valid}\,\Gamma_2$
  $\Gamma_1 \subseteq \Gamma_2$

- We have to show:
  $\Gamma_2 \vdash \lambda x.t : T_1 \rightarrow T_2$

Proof Idea for the Lambda Cs.

$$\frac{x \# \Gamma \quad (x:T_1)::\Gamma \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \rightarrow T_2}$$

- If $\Gamma_1 \vdash t : T_1$ then $\forall \Gamma_2.\ \text{valid}\ \Gamma_2 \wedge \Gamma_1 \subseteq \Gamma_2 \Rightarrow \Gamma_2 \vdash t : T_2$

For all Γ_1, x, t, T_1 and T_2:

$\boxed{\Gamma_3 \mapsto (x:T_1)::\Gamma_2}$

- We know:
  $\forall \Gamma_3.\ \text{valid}\ \Gamma_3 \wedge (x:T_1)::\Gamma_1 \subseteq \Gamma_3 \Rightarrow \Gamma_3 \vdash t : T_1$
  $x \# \Gamma_1$
  valid Γ_2
  $\Gamma_1 \subseteq \Gamma_2$

- We have to show:
  $\Gamma_2 \vdash \lambda x.t : T_1 \rightarrow T_2$

Your Turn: Lambda Case

```
lemma
  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 (induct arbitrary: Γ2)
  case (t_Lam x Γ1 T1 t T2)
  have ih: "⋀Γ3. ⟦valid Γ3; (x,T1)#Γ1 ⊆ Γ3⟧ ⟹ Γ3 ⊢ t : T2" by fact
  have a0: "x#Γ1" by fact
  have a1: "valid Γ2" by fact
  have a2: "Γ1 ⊆ Γ2" by fact
    ...
  show "Γ2 ⊢ Lam [x].t : T1 → T2" sorry
```

Strong Induction Principle

$$\forall \Gamma\, x\, T\,.\ (x\!:\!T) \in \Gamma \wedge \mathsf{valid}\,\Gamma \Rightarrow P\ \Gamma\,(x)\,T$$

$$\begin{array}{l} \forall \Gamma\, t_1\, t_2\, T_1\, T_2\,. \\ P\,\Gamma\, t_1\,(T_1 \to T_2) \wedge P\,\Gamma\, t_2\, T_1 \\ \qquad\qquad\qquad\qquad \Rightarrow P\,\Gamma\,(t_1\, t_2)\, T_2 \end{array}$$

$$\begin{array}{l} \forall \Gamma\, x\, t\, T_1\, T_2\,. \\ x \,\#\, \Gamma\ \ \wedge \\ P\,((x\!:\!T_1)\!::\!\Gamma)\, t\, T_2 \Rightarrow P\,\Gamma(\lambda x.t)\,(T_1 \to T_2) \end{array}$$

$$\overline{\qquad\qquad \Gamma \vdash t : T \Rightarrow P\,\Gamma\, t\, T \qquad\qquad}$$

- Instead we are going to use the strong induction principle and set up the induction so that the binder "avoids" Γ_2.

2nd Attempt

```
lemma
  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 (induct arbitrary: Γ₂)
  case (t_Lam x Γ₁ T₁ t T₂)
  have ih: "⋀Γ₃. ⟦valid Γ₃; (x,T₁)#Γ₁ ⊆ Γ₃⟧ ⟹ Γ₃ ⊢ t : T₂" by fact
  have a0: "x#Γ₁" by fact
  have a1: "valid Γ₂" by fact
  have a2: "Γ₁ ⊆ Γ₂" by fact
    ...
  show "Γ₂ ⊢ Lam [x].t : T₁ → T₂" sorry
```

2nd Attempt

```
lemma
  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 avoiding: Γ₂ rule: typing.strong_induct)
  case (t_Lam x Γ₁ T₁ t T₂)
  have vc: "x#Γ₂" by fact
  have ih: "⋀Γ₃. ⟦valid Γ₃; (x,T₁)#Γ₁ ⊆ Γ₃⟧ ⟹ Γ₃ ⊢ t : T₂" by fact
  have a0: "x#Γ₁" by fact
  have a1: "valid Γ₂" by fact
  have a2: "Γ₁ ⊆ Γ₂" by fact
   . . .
  show "Γ₂ ⊢ Lam [x].t : T₁ → T₂" sorry
```

```
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 avoiding: Γ₂ rule: typing.strong_induct)
  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)
```

```
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 avoiding: Γ₂ rule: typing.strong_induct)
   (auto)
```

```
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 avoiding: Γ2 rule: typing.strong_induct)
   (auto)
```

- Perhaps the weakening lemma is after all trivial / routine / obvious ;o)
- We shall late see that the work we put into the stronger induction principle needs a bit of thinking. For you, of course, it is provided automatially.

Function Definitions and the Simplifier

Function Definitions

- Later on we will need a few functions about contexts:

```
fun
  filling :: "ctx ⇒ lam ⇒ lam" ("_⟦_⟧")
where
  "□⟦t⟧ = t"
| "(CAppL E t')⟦t⟧ = App (E⟦t⟧) t'"
| "(CAppR t' E)⟦t⟧ = App t' (E⟦t⟧)"
```

Function Definitions

- Later on we will need a few functions about contexts:

a name

```
fun
  filling :: "ctx ⇒ lam ⇒ lam" ("_⟦_⟧")
where
  "□⟦t⟧ = t"
| "(CAppL E t')⟦t⟧ = App (E⟦t⟧) t'"
| "(CAppR t' E)⟦t⟧ = App t' (E⟦t⟧)"
```

Function Definitions

- Later on we will need a few functions about contexts:

a type

```
fun
  filling :: "ctx ⇒ lam ⇒ lam" ("_⟦_⟧")
where
  "□⟦t⟧ = t"
| "(CAppL E t')⟦t⟧ = App (E⟦t⟧) t'"
| "(CAppR t' E)⟦t⟧ = App t' (E⟦t⟧)"
```

Function Definitions

- Later on we will need a few functions about contexts:

pretty syntax

```
fun
  filling :: "ctx ⇒ lam ⇒ lam" ("_⟦_⟧")
where
  "□⟦t⟧ = t"
| "(CAppL E t')⟦t⟧ = App (E⟦t⟧) t'"
| "(CAppR t' E)⟦t⟧ = App t' (E⟦t⟧)"
```

Function Definitions

- Later on we will need a few functions about contexts:

```
fun
  filling :: "ctx ⇒ lam ⇒ lam" ("_⟦_⟧")
where
  "□⟦t⟧ = t"
| "(CAppL E t')⟦t⟧ = App (E⟦t⟧) t'"
| "(CAppR t' E)⟦t⟧ = App t' (E⟦t⟧)"
```

char. eqs

Function Definitions

- Later on we will need a few functions about contexts:

```
fun
  filling :: "ctx ⇒ lam ⇒ lam" ("_⟦_⟧")
where
  "□⟦t⟧ = t"
| "(CAppL E t')⟦t⟧ = App (E⟦t⟧) t'"
| "(CAppR t' E)⟦t⟧ = App t' (E⟦t⟧)"
```

- Once a function is defined, the simplifier will be able to solve equations like

```
lemma
  shows "(CAppL □ (Var x))⟦Var y⟧ = App (Var y) (Var x)"
  by simp
```

Context Composition

```
fun
 ctx_compose :: "ctx ⇒ ctx ⇒ ctx" ("_ ∘ _" [101,100] 100)
where
  "□ ∘ E' = E'"
| "(CAppL E t') ∘ E' = CAppL (E ∘ E') t'"
| "(CAppR t' E) ∘ E' = CAppR t' (E ∘ E')"

fun
  ctx_composes :: "ctxs ⇒ ctx" ("_↓" [110] 110)
where
    "[]↓ = □"
  | "(E#Es)↓ = (Es↓) ∘ E"
```

Context Composition

```
fun
 ctx_compose :: "ctx ⇒ ctx ⇒ ctx" ("_ ∘ _" [101,100] 100)
where
  "□ ∘ E' = E'"
| "(CAppL E t') ∘ E' = CAppL (E ∘ E') t'"
| "(CAppR t' E) ∘ E' = CAppR t' (E ∘ E')"
```

precedence

```
fun
  ctx_composes :: "ctxs ⇒ ctx" ("_↓" [110] 110)
where
   "[]↓ = □"
 | "(E#Es)↓ = (Es↓) ∘ E"
```

precedence

- Explicit preedences are given in order to enforce the notation:

$$(E_1 \circ E_2) \circ E_3 \qquad (E_1 \circ E_2)\downarrow$$

Context Composition

```
fun
 ctx_compose :: "ctx ⇒ ctx ⇒ ctx" ("_ ∘ _" [101,100] 100)
where
  "□ ∘ E' = E'"
| "(CAppL E t') ∘ E' = CAppL (E ∘ E') t'"
| "(CAppR t' E) ∘ E' = CAppR t' (E ∘ E')"
```

precedence

```
fun
  ctx_composes :: "ctxs ⇒ ctx" ("_↓" [110] 110)
where
   "[]↓ = □"
 | "(E#Es)↓ = (Es↓) ∘ E"
```

precedence

- Explicit preedences are given in order to enforce the notation:

$$(E_1 \circ E_2) \circ E_3 \qquad (E_1 \circ E_2)\!\downarrow$$

Your Turn

```
datatype ctx =
  Hole
| CAppL "ctx" "lam"
| CAppR "lam" "ctx"
```

```
lemma ctx_compose:
  shows "(E₁ ∘ E₂)⟦t⟧ = E₁⟦E₂⟦t⟧⟧"
proof (induct E₁)
  case Hole
  show "□ ∘ E₂⟦t⟧ = □⟦E₂⟦t⟧⟧" sorry
next
  case (CAppL E₁ t')
  have ih: "(E₁ ∘ E₂)⟦t⟧ = E₁⟦E₂⟦t⟧⟧" by fact
  show "((CAppL E₁ t') ∘ E₂)⟦t⟧ = (CAppL E₁ t')⟦E₂⟦t⟧⟧" sorry
next
  case (CAppR t' E₁)
  have ih: "(E₁ ∘ E₂)⟦t⟧ = E₁⟦E₂⟦t⟧⟧" by fact
  show "((CAppR t' E₁) ∘ E₂)⟦t⟧ = (CAppR t' E₁)⟦E₂⟦t⟧⟧" sorry
qed
```

Your Turn

```
datatype ctx =
  Hole
| CAppL "ctx" "lam"
| CAppR "lam" "ctx"
```

```
lemma ctx_compose:
  shows "(E₁ ∘ E₂)⟦t⟧ = E₁⟦E₂⟦t⟧⟧"
proof (induct E₁)
  case Hole
  show "□ ∘ E₂⟦t⟧ = □⟦E₂⟦t⟧⟧" sorry
next
  case (CAppL E₁ t')
  have ih: "(E₁ ∘ E₂)⟦t⟧ = E₁⟦E₂⟦t⟧⟧" by fact
  show "((CAppL E₁ t') ∘ E₂)⟦t⟧ = (CAppL E₁ t')⟦E₂⟦t⟧⟧" sorry
next
  case (CAppR t' E₁)
  have ih: "(E₁ ∘ E₂)⟦t⟧ = E₁⟦E₂⟦t⟧⟧" by fact
  show "((CAppR t' E₁) ∘ E₂)⟦t⟧ = (CAppR t' E₁)⟦E₂⟦t⟧⟧" sorry
qed
```

```
thm filling.simps[no_vars]
thm ctx_compose.simps[no_vars]
```

Your Turn Again

- Assuming:

```
lemma neut_hole: shows "E ∘ □ = E"
lemma circ_assoc:  shows "(E₁ ∘ E₂) ∘ E₃ = E₁ ∘ (E₂ ∘ E₃)"
```

- Prove

```
lemma shows "(Es₁ @ Es₂)↓ = (Es₂↓) ∘ (Es₁↓)"
proof (induct Es₁)
  case Nil
  show "([] @ Es₂)↓ = Es₂↓ ∘ []↓" sorry
next
  case (Cons E Es₁)
  have ih: "(Es₁ @ Es₂)↓ = Es₂↓ ∘ Es₁↓" by fact

  show "((E#Es₁) @ Es₂)↓ = Es₂↓ ∘ (E#Es₁)↓" sorry
qed
```

Your Turn Again

- Assuming:

```
lemma neut_hole: shows "E ∘ □ = E"
lemma circ_assoc:  shows "(E₁ ∘ E₂) ∘ E₃ = E₁ ∘ (E₂ ∘ E₃)"
```

- Prove

```
lemma shows "(Es₁ @ Es₂)↓ = (Es₂↓) ∘ (Es₁↓)"
proof (induct Es₁)
  case Nil
  show "([] @ Es₂)↓ = Es₂↓ ∘ []↓" sorry
next
  case (Cons E Es₁)
  have ih: "(Es₁ @ Es₂)↓ = Es₂↓ ∘ Es₁↓" by fact

  show "((E#Es₁) @ Es₂)↓ = Es₂↓ ∘ (E#Es₁)↓" sorry
qed
```

My Solution

```
lemma
  shows "(Es₁ @ Es₂)↓ = (Es₂↓) ∘ (Es₁↓)"
proof (induct Es₁)
  case Nil
  show "([]@Es₂)↓ = Es₂↓ ∘ []↓" using neut_hole by simp
next
  case (Cons E Es₁)
  have ih: "(Es₁ @ Es₂)↓ = Es₂↓ ∘ Es₁↓" by fact
  have lhs: "((E#Es₁) @ Es₂)↓ = (Es₁ @ Es₂)↓ ∘ E" by simp
  have lhs': "(Es₁ @ Es₂)↓ ∘ E = (Es₂↓ ∘ Es₁↓) ∘ E" using ih by simp
  have rhs: "Es₂↓ ∘ (E#Es₁)↓ = Es₂↓ ∘ (Es₁↓ ∘ E)" by simp
  show "((E#Es₁) @ Es₂)↓ = Es₂↓ ∘ (E#Es₁)↓"
    using lhs lhs' rhs circ_assoc by simp
qed
```

Equational Reasoning in Isar

- One frequently wants to prove an equation $t_1 = t_n$ by means of a chain of equations, like

$$t_1 = t_2 = t_3 = t_4 = \ldots = t_n$$

Equational Reasoning in Isar

- One frequently wants to prove an equation $t_1 = t_n$ by means of a chain of equations, like

$$t_1 = t_2 = t_3 = t_4 = \ldots = t_n$$

- This kind of reasoning is supported in Isar as:

```
have "t1 = t2" by just.
also have "... = t3" by just.
also have "... = t4" by just.
...
also have "... = tn" by just.
finally have "t1 = tn" by simp
```

A Readable Solution

```
lemma
  shows "(Es₁ @ Es₂)↓ = (Es₂↓) ∘ (Es₁↓)"
proof (induct Es₁)
  case Nil
  show "([]@Es₂)↓ = Es₂↓ ∘ []↓" using neut_hole by simp
next
  case (Cons E Es₁)
  have ih: "(Es₁ @ Es₂)↓ = Es₂↓ ∘ Es₁↓" by fact
  have "((E#Es₁) @ Es₂)↓ = (Es₁ @ Es₂)↓ ∘ E" by simp
  also have "... = (Es₂↓ ∘ Es₁↓) ∘ E" using ih by simp
  also have "... = Es₂↓ ∘ (Es₁↓ ∘ E)" using circ_assoc by simp
  also have "... = Es₂↓ ∘ (E#Es₁)↓" by simp
  finally show "((E#Es₁) @ Es₂)↓ = Es₂↓ ∘ (E#Es₁)↓" by simp
qed
```

Capture-Avoiding Substitution and the Substitution Lemma

Capture-Avoiding Subst.

- Lambda.thy contains a definition of capture-avoiding substitution with the characteristic equations:

"(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) $\Longrightarrow$ (Lam [x].t)[y::=s] = Lam [x].(t[y::=s])"

Capture-Avoiding Subst.

- Lambda.thy contains a definition of capture-avoiding substitution with the characteristic equations:

"(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) $\Longrightarrow$ (Lam [x].t)[y::=s] = Lam [x].(t[y::=s])"

- Despite its looks, this is a total function!

Substitution Lemma: If $x \not\equiv y$ and $x \notin \mathsf{fv}(L)$, then
$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]]$$

Proof: By induction on the structure of M.

- **Case 1:** M is a variable.

  Case 1.1. $M \equiv x$. Then both sides equal $N[y := L]$ since $x \not\equiv y$.

  Case 1.2. $M \equiv y$. Then both sides equal L, for $x \notin \mathsf{fv}(L)$ implies $L[x := \ldots] \equiv L$.

  Case 1.3. $M \equiv z \not\equiv x, y$. Then both sides equal z.

- **Case 2:** $M \equiv \lambda z.M_1$. By the variable convention we may assume that $z \not\equiv x, y$ and z is not free in N, L.

$$\begin{aligned}(\lambda z.M_1)[x := N][y := L] &\equiv \lambda z.(M_1[x := N][y := L]) \\ &\equiv \lambda z.(M_1[y := L][x := N[y := L]]) \\ &\equiv (\lambda z.M_1)[y := L][x := N[y := L]].\end{aligned}$$

- **Case 3:** $M \equiv M_1 M_2$. The statement follows again from the induction hypothesis. $\square$

Substitution Lemma: If $x \not\equiv y$ and $x \notin \mathsf{fv}(L)$, then
$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]]$$

Proof: By induction on the structure of M.

- **Case 1:** M is a variable.

  Case 1.1. $M \equiv x$. Then both sides equal $N[y := L]$ since $x \not\equiv y$.

  Case 1.2. $M \equiv y$. Then both sides equal L, for $x \notin \mathsf{fv}(L)$ implies $L[x := \ldots] \equiv L$.

  Case 1.3. $M \equiv z \not\equiv x, y$. Then both sides equal z.

- **Case 2:** $M \equiv \lambda z.M_1$. By the variable convention we may assume that $z \not\equiv x, y$ and z is not free in N, L.

$$\begin{aligned}(\lambda z.M_1)[x := N][y := L] &\equiv \lambda z.(M_1[x := N][y := L]) \\ &\equiv \lambda z.(M_1[y := L][x := N[y := L]]) \\ &\equiv (\lambda z.M_1)[y := L][x := N[y := L]].\end{aligned}$$

- **Case 3:** $M \equiv M_1 M_2$. The statement follows again from the induction hypothesis. $\square$

Substitution Lemma: If $x \not\equiv y$ and $x \notin \mathsf{fv}(L)$, then
$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]]$$

Proof: By induction on the structure of M.

- **Case 1:** M is a variable.

  Case 1.1. $M \equiv x$. Then both sides equal $N[y := L]$ since $x \not\equiv y$.

  Case 1.2. $M \equiv y$. Then both sides equal L, for $x \notin \mathsf{fv}(L)$ implies $L[x := \ldots] \equiv L$.

  Case 1.3. $M \equiv z \not\equiv x, y$. Then both sides equal z.

- **Case 2:** $M \equiv \lambda z.M_1$. By the variable convention we may assume that $z \not\equiv x, y$ and z is not free in N, L.

$$\begin{aligned}(\lambda z.M_1)[x := N][y := L] &\equiv \lambda z.(M_1[x := N][y := L])\\ &\equiv \lambda z.(M_1[y := L][x := N[y := L]])\\ &\equiv (\lambda z.M_1)[y := L][x := N[y := L]].\end{aligned}$$

- **Case 3:** $M \equiv M_1 M_2$. The statement follows again from the induction hypothesis. $\square$

Substitution Lemma: If $x \not\equiv y$ and $x \notin \mathsf{fv}(L)$, then
$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]]$$

Proof: By induction on the structure of M.

- **Case 1:** M is a variable.

  Case 1.1. $M \equiv x$. Then both sides equal $N[y := L]$ since $x \not\equiv y$.

  Case 1.2. $M \equiv y$. Then both sides equal L, for $x \notin \mathsf{fv}(L)$ implies $L[x := \ldots] \equiv L$.

  Case 1.3. $M \equiv z \not\equiv x, y$. Then both sides equal z.

- **Case 2:** $M \equiv \lambda z.M_1$. By the variable convention we may assume that $z \not\equiv x, y$ and z is not free in N, L.

$$\begin{aligned}(\lambda z.M_1)[x := N][y := L] &\equiv \lambda z.(M_1[x := N][y := L]) \\ &\equiv \lambda z.(M_1[y := L][x := N[y := L]]) \\ &\equiv (\lambda z.M_1)[y := L][x := N[y := L]].\end{aligned}$$

- **Case 3:** $M \equiv M_1 M_2$. The statement follows again from the induction hypothesis. □

Substitution Lemma: If $x \not\equiv y$ and $x \notin \mathrm{fv}(L)$, then
$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]]$$

Proof: By induction on the structure of M.

- **Case 1:** M
  Case 1.1. M
  x
  Case 1.2. M
  in
  Case 1.3. M
- **Case 2:** M
  assume th
  $(\lambda z.M_1)[x$

Remember only if $y \neq x$ and $x \notin \mathrm{fv}(N)$ then
$$(\lambda y.M)[x := N] = \lambda y.(M[x := N])$$

$$\begin{array}{lll} & (\lambda z.M_1)[x := N][y := L] & \\ \equiv & (\lambda z.(M_1[x := N]))[y := L] & \xleftarrow{1} \\ \equiv & \lambda z.(M_1[x := N][y := L]) & \xleftarrow{2} \\ \equiv & \lambda z.(M_1[y := L][x := N[y := L]]) & \text{IH} \\ \equiv & (\lambda z.(M_1[y := L]))[x := N[y := L]] & \xrightarrow{2} \ ! \\ \equiv & (\lambda z.M_1)[y := L][x := N[y := L]]. & \xrightarrow{1} \end{array}$$

- **Case 3:** $M \equiv M_1 M_2$. The statement follows again from the induction hypothesis. $\square$

Substitution Lemma: If $x \not\equiv y$ and $x \notin \mathsf{fv}(L)$, then
$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]]$$

Proof: By induction on the structure of M.

- **Case 1:** M is a variable.

  Case 1.1. $M \equiv x$. Then both sides equal $N[y := L]$ since $x \not\equiv y$.

  Case 1.2. $M \equiv y$. Then both sides equal L, for $x \notin \mathsf{fv}(L)$ implies $L[x := \ldots] \equiv L$.

  Case 1.3. $M \equiv z \not\equiv x, y$. Then both sides equal z.

- **Case 2:** $M \equiv \lambda z.M_1$. By the variable convention we may assume that $z \not\equiv x, y$ and z is not free in N, L.

$$\begin{aligned}(\lambda z.M_1)[x := N][y := L] &\equiv \lambda z.(M_1[x := N][y := L])\\ &\equiv \lambda z.(M_1[y := L][x := N[y := L]])\\ &\equiv (\lambda z.M_1)[y := L][x := N[y := L]].\end{aligned}$$

- **Case 3:** $M \equiv M_1 M_2$. The statement follows again from the induction hypothesis. $\square$

Case Distintions

- Assuming $P_1 \lor P_2 \lor P_3$ is true then:

```
{ assume "P₁"
  ...
  have "something" ...}
moreover
{ assume "P₂"
  ...
  have "something" ...}
moreover
{ assume "P₃"
  ...
  have "something" ...}
ultimately have "something" by blast
```

Case Distintions

- Assuming $P_1 \vee P_2 \vee P_3$ is true then:

$P_1 \mapsto (z{=}x)$
$P_2 \mapsto (z{=}y) \wedge (z{\neq}x)$
$P_3 \mapsto (z{\neq}y) \wedge (z{\neq}x)$

```
{ assume "P1"
  ...
  have "something" ...}
moreover
{ assume "P2"
  ...
  have "something" ...}
moreover
{ assume "P3"
  ...
  have "something" ...}
ultimately have "something" by blast
```

Case Distintions

- Assuming $P_1 \vee P_2 \vee P_3$ is true then:

```
{ assume "P1"
  ...
  have "something" ...}
moreover
{ assume "P2"
  ...
  have "something" ...}
moreover
{ assume "P3"
  ...
  have "something" ...}
ultimately have "something" by blast
```

$$\frac{P_1 \Longrightarrow smth \quad P_2 \Longrightarrow smth \quad P_3 \Longrightarrow smth}{smth}$$

```
lemma substitution_lemma:
  assumes a: "x≠y" "x # L"
  shows "M[x::=N][y::=L] = M[y::=L][x::=N[y::=L]]"
using a proof (nominal_induct M avoiding: x y N L rule: lam.strong_induct)
case (Var z)
  have a1: "x≠y" by fact
  have a2: "x#L" by fact
  show "Var z[x::=N][y::=L] = Var z[y::=L][x::=N[y::=L]]" (is "?LHS = ?RHS")
  proof -
   { assume c1: "z=x"
     have "(1)": "?LHS = N[y::=L]" using c1 by simp
     have "(2)": "?RHS = N[y::=L]" using c1 a1 by simp
      have "?LHS = ?RHS" using "(1)" "(2)" by simp }
   moreover
   { assume c2: "z=y" "z≠x"

     have "?LHS = ?RHS" sorry }
   moreover
   { assume c3: "z≠x" "z≠y"

     have "?LHS = ?RHS" sorry }
   ultimately show "?LHS = ?RHS" by blast
  qed
```

```
lemma substitution_lemma:
  assumes a: "x≠y" "x # L"
  shows "M[x::=N][y::=L] = M[y::=L][x::=N[y::=L]]"
using a proof (nominal_induct M avoiding: x y N L rule: lam.strong_induct)
case (Var z)
  have a1: "x≠y" by fact
  have a2: "x#L" by fact
  show "Var z[x::=N][y::=L] = Var z[y::=L][x::=N[y::=L]]" (is "?LHS = ?RHS")
  proof -
   { assume c1: "z=x"
     have "(1)": "?LHS = N[y::=L]" using c1 by simp
     have "(2)": "?RHS = N[y::=L]" using c1 a1 by simp
      have "?LHS = ?RHS" using "(1)" "(2)" by simp }
   moreover
   { assume c2: "z=y" "z≠x"

     have "?LHS = ?RHS" sorry }
   moreover
   { assume c3: "z≠x" "z≠y"

     have "?LHS = ?RHS" sorry }
   ultimately show "?LHS = ?RHS" by blast
  qed
```

```
lemma substitution_lemma:
  assumes a: "x≠y" "x # L"
  shows "M[x::=N][y::=L] = M[y::=L][x::=N[y::=L]]"
using a proof (nominal_induct M avoiding: x y N L rule: lam.strong_induct)
case (Var z)
  have a1: "x≠y" by fact
  have a2: "x#L" by fact
  show "Var z[x::=N][y::=L] = Var z[y::=L][x::=N[y::=L]]" (is "?LHS = ?RHS")
  proof -
   { assume c1: "z=x"
     have "(1)": "?LHS = N[y::=L]" using c1 by simp
     have "(2)": "?RHS = N[y::=L]" using c1 a1 by simp
      have "?LHS = ?RHS" using "(1)" "(2)" by simp }
   moreover
   { assume c2: "z=y" "z≠x"

     have "?LHS = ?RHS" sorry }
   moreover
   { assume c3: "z≠x" "z≠y"

     have "?LHS = ?RHS" sorry }
   ultimately show "?LHS = ?RHS" by blast
  qed
```

```
lemma substitution_lemma:
  assumes a: "x≠y" "x # L"
  shows "M[x::=N][y::=L] = M[y::=L][x::=N[y::=L]]"
using a proof (nominal_induct M avoiding: x y N L rule: lam.strong_induct)
case (Var z)
  have a1: "x≠y" by fact
  have a2: "x#L" by fact
  show "Var z[x::=N][y::=L] = Var z[y::=L][x::=N[y::=L]]" (is "?LHS = ?RHS")
  proof -
   { assume c1: "z=x"
     have "(1)": "?LHS = N[y::=L]" using c1 by simp
     have "(2)": "?RHS = N[y::=L]" using c1 a1 by simp
      have "?LHS = ?RHS" using "(1)" "(2)" by simp }
   moreover
   { assume c2: "z=y" "z≠x"

     have "?LHS = ?RHS" sorry }
   moreover
   { assume c3: "z≠x" "z≠y"

     have "?LHS = ?RHS" sorry }
   ultimately show "?LHS = ?RHS" by blast
  qed
```

```
lemma substitution_lemma:
  assumes a: "x≠y" "x # L"
  shows "M[x::=N][y::=L] = M[y::=L][x::=N[y::=L]]"
using a proof (nominal_induct M avoiding: x y N L rule: lam.strong_induct)
case (Var z)
  have a1: "x≠y" by fact
  have a2: "x#L" by fact
  show "Var z[x::=N][y::=L] = Var z[y::=L][x::=N[y::=L]]" (is "?LHS = ?RHS")
  proof -
   { assume c1: "z=x"
     have "(1)": "?LHS = N[y::=L]" using c1 by simp
     have "(2)": "?RHS = N[y::=L]" using c1 a1 by simp
      have "?LHS = ?RHS" using "(1)" "(2)" by simp }
   moreover
   { assume c2: "z=y" "z≠x"

     have "?LHS = ?RHS" sorry }
   moreover
   { assume c3: "z≠x" "z≠y"

     have "?LHS = ?RHS" sorry }
   ultimately show "?LHS = ?RHS" by blast
  qed
```

```
lemma substitution_lemma:
  assumes a: "x≠y" "x # L"
  shows "M[x::=N][y::=L] = M[y::=L][x::=N[y::=L]]"
using a proof (nominal_induct M avoiding: x y N L rule: lam.strong_induct)
case (Var z)
  have a1: "x≠y" by fact
  have a2: "x#L" by fact
  show "Var z[x::=N][y::=L] = Var z[y::=L][x::=N[y::=L]]" (is "?LHS = ?RHS")
  proof -
   { assume c1: "z=x"
     have "(1)": "?LHS = N[y::=L]" using c1 by simp
     have "(2)": "?RHS = N[y::=L]" using c1 a1 by simp
      have "?LHS = ?RHS" using "(1)" "(2)" by simp }
   moreover
   { assume c2: "z=y" "z≠x"

     have "?LHS = ?RHS" sorry }
   moreover
   { assume c3: "z≠x" "z≠y"

     have "?LHS = ?RHS" sorry }
   ultimately show "?LHS = ?RHS" by blast
  qed
```

```
lemma substitution_lemma:
  assumes a: "x≠y" "x # L"
  shows "M[x::=N][y::=L] = M[y::=L][x::=N[y::=L]]"
using a proof (nominal_induct M avoiding: x y N L rule: lam.strong_induct)
case (Var z)
  have a1: "x≠y" by fact
  have a2: "x#L" by fact
  show "Var z[x::=N][y::=L] = Var z[y::=L][x::=N[y::=L]]" (is "?LHS = ?RHS")
  proof -
   { assume c1: "z=x"
     have "(1)": "?LHS = N[y::=L]" using c1 by simp
     have "(2)": "?RHS = N[y::=L]" using c1 a1 by simp
      have "?LHS = ?RHS" using "(1)" "(2)" by simp }
   moreover
   { assume c2: "z=y" "z≠x"

     have "?LHS = ?RHS" sorry }
   moreover
   { assume c3: "z≠x" "z≠y"

     have "?LHS = ?RHS" sorry }
   ultimately show "?LHS = ?RHS" by blast
  qed
```

```
thm forget:
x # L ⟹ L[x::=P] = L
```

```
lemma substitution_lemma:
  assumes a: "x≠y" "x # L"
  shows "M[x::=N][y::=L] = M[y::=L][x::=N[y::=L]]"
using a proof (nominal_induct M avoiding: x y N L rule: lam.strong_induct)
case (Var z)
  have a1: "x≠y" by fact
  have a2: "x#L" by fact
  show "Var z[x::=N][y::=L] = Var z[y::=L][x::=N[y::=L]]" (is "?LHS = ?RHS")
  proof -
   { assume c1: "z=x"
     have "(1)": "?LHS = N[y::=L]" using c1 by simp
     have "(2)": "?RHS = N[y::=L]" using c1 a1 by simp
      have "?LHS = ?RHS" using "(1)" "(2)" by simp }
   moreover
   { assume c2: "z=y" "z≠x"

     have "?LHS = ?RHS" sorry }
   moreover
   { assume c3: "z≠x" "z≠y"

     have "?LHS = ?RHS" sorry }
   ultimately show "?LHS = ?RHS" by blast
  qed
```

```
lemma substitution_lemma:
  assumes a: "x≠y" "x # L"
  shows "M[x::=N][y::=L] = M[y::=L][x::=N[y::=L]]"
using a proof (nominal_induct M avoiding: x y N L rule: lam.strong_induct)
 case (Lam z M₁)
 have ih: "⟦x≠y; x#L⟧ ⟹ M₁[x::=N][y::=L] = M₁[y::=L][x::=N[y::=L]]" by fact
 have "x≠y" by fact
 have "x#L" by fact
 have vc: "z#x" "z#y" "z#N" "z#L" by fact+
 then have "z#N[y::=L]" by (simp add: fresh_fact)
 show "(Lam [z].M₁)[x::=N][y::=L]=(Lam [z].M₁)[y::=L][x::=N[y::=L]]" (is "?LHS=?RHS")
 proof -
   have "?LHS = ..." sorry

   also have "... = ?RHS" sorry
   finally show "?LHS = ?RHS" by simp
 qed
next
```

```
lemma substitution_lemma:
  assumes a: "x≠y" "x # L"
  shows "M[x::=N][y::=L] = M[y::=L][x::=N[y::=L]]"
using a proof (nominal_induct M avoiding: x y N L rule: lam.strong_induct)
 case (Lam z M₁)
 have ih: "⟦x≠y; x#L⟧ ⟹ M₁[x::=N][y::=L] = M₁[y::=L][x::=N[y::=L]]" by fact
 have "x≠y" by fact
 have "x#L" by fact
 have vc: "z#x" "z#y" "z#N" "z#L" by fact+
 then have "z#N[y::=L]" by (simp add: fresh_fact)
 show "(Lam [z].M₁)[x::=N][y::=L]=(Lam [z].M₁)[y::=L][x::=N[y::=L]]" (is "?LHS=?RHS")
 proof -
   have "?LHS = ..." sorry

   also have "... = ?RHS" sorry
   finally show "?LHS = ?RHS" by simp
 qed
next
```

```
lemma substitution_lemma:
  assumes a: "x≠y" "x # L"
  shows "M[x::=N][y::=L] = M[y::=L][x::=N[y::=L]]"
using a proof (nominal_induct M avoiding: x y N L rule: lam.strong_induct)
 case (Lam z M1)
 have ih: "⟦x≠y; x#L⟧ ⟹ M1[x::=N][y::=L] = M1[y::=L][x::=N[y::=L]]" by fact
 have "x≠y" by fact
 have "x#L" by fact
 have vc: "z#x" "z#y" "z#N" "z#L" by fact+
 then have "z#N[y::=L]" by (simp add: fresh_fact)
 show "(Lam [z].M1)[x::=N][y::=L]=(Lam [z].M1)[y::=L][x::=N[y::=L]]" (is "?LHS=?RHS")
 proof -
   have "?LHS = ..." sorry

   also have "... = ?RHS" sorry
   finally show "?LHS = ?RHS" by simp
 qed
next
```

```
lemma substitution_lemma:
  assumes a: "x≠y" "x # L"
  shows "M[x::=N][y::=L] = M[y::=L][x::=N[y::=L]]"
using a proof (nominal_induct M avoiding: x y N L rule: lam.strong_induct)
 case (Lam z M1)
 have ih: "⟦x≠y; x#L⟧ ⟹ M1[x::=N][y::=L] = M1[y::=L][x::=N[y::=L]]" by fact
 have "x≠y" by fact
 have "x#L" by fact
 have vc: "z#x" "z#y" "z#N" "z#L" by fact+
 then have "z#N[y::=L]" by (simp add: fresh_fact)
 show "(Lam [z].M1)[x::=N][y::=L]=(Lam [z].M1)[y::=L][x::=N[y::=L]]" (is "?LHS=?RHS")
 proof -
  have "?LHS = ..." sorry

  also have "... = ?RHS" sorry
  finally show "?LHS = ?RHS" by simp
 qed
next
```

```
lemma substitution_lemma:
  assumes a: "x≠y" "x # L"
  shows "M[x::=N][y::=L] = M[y::=L][x::=N[y::=L]]"
using a proof (nominal_induct M avoiding: x y N L rule: lam.strong_induct)
 case (Lam z M1)
 have ih: "⟦x≠y; x#L⟧ ⟹ M1[x::=N][y::=L] = M1[y::=L][x::=N[y::=L]]" by fact
 have "x≠y" by fact
 have "x#L" by fact
 have vc: "z#x" "z#y" "z#N" "z#L" by fact+
 then have "z#N[y::=L]" by (simp add: fresh_fact)
 show "(Lam [z].M1)[x::=N][y::=L]=(Lam [z].M1)[y::=L][x::=N[y::=L]]" (is "?LHS=?RHS")
 proof -
   have "?LHS = ..." sorry

   also have "... = ?RHS" sorry
   finally show "?LHS = ?RHS" by simp
 qed
next
```

Substitution Lemma: If $x \not\equiv y$ and $x \notin \mathsf{fv}(L)$, then
$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]]$$

Proof: By induction on the structure of M.

- **Case 1:** M is a variable.

  Case 1.1. $M \equiv x$. Then both sides equal $N[y := L]$ since $x \not\equiv y$.

  Case 1.2. $M \equiv y$. Then both sides equal L, for $x \notin \mathsf{fv}(L)$ implies $L[x := \ldots] \equiv L$.

  Case 1.3. $M \equiv z \not\equiv x, y$. Then both sides equal z.

- **Case 2:** $M \equiv \lambda z.M_1$. By the variable convention we may assume that $z \not\equiv x, y$ and z is not free in N, L.

$$\begin{aligned}(\lambda z.M_1)[x := N][y := L] &\equiv \lambda z.(M_1[x := N][y := L]) \\ &\equiv \lambda z.(M_1[y := L][x := N[y := L]]) \\ &\equiv (\lambda z.M_1)[y := L][x := N[y := L]].\end{aligned}$$

- **Case 3:** $M \equiv M_1 M_2$. The statement follows again from the induction hypothesis. $\square$

Substitution Lemma

- The strong structural induction principle for lambda-terms allowed us to follow Barendregt's proof quite closely. It also enables Isabelle to find this proof automatically:

```
lemma substitution_lemma:
  assumes asm: "x≠y" "x#L"
  shows "M[x::=N][y::=L] = M[y::=L][x::=N[y::=L]]"
  using asm
by (nominal_induct M avoiding: x y N L rule: lam.strong_induct)
   (auto simp add: fresh_fact forget)
```

How To Prove False Using the Variable Convention (on Paper)

So Far So Good

- A Faulty Lemma with the Variable Convention?

> **Variable Convention:**
>
> If $M_1, \ldots, M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.
>
> Barendregt in "The Lambda-Calculus: Its Syntax and Semantics"

Inductive Definitions:

$$\frac{\mathsf{prem}_1 \ldots \mathsf{prem}_n \;\; \mathsf{scs}}{\mathsf{concl}}$$

Rule Inductions:

1.) Assume the property for the premises. Assume the side-conditions.

2.) Show the property for the conclusion.

Faulty Reasoning

- Consider the two-place relation foo:

$$\frac{}{x \mapsto x} \qquad \frac{}{t_1\ t_2 \mapsto t_1\ t_2} \qquad \frac{t \mapsto t'}{\lambda x.t \mapsto t'}$$

Faulty Reasoning

- Consider the two-place relation foo:

$$\frac{}{x \mapsto x} \qquad \frac{}{t_1\ t_2 \mapsto t_1\ t_2} \qquad \frac{t \mapsto t'}{\lambda x.t \mapsto t'}$$

- The lemma we going to prove:

  Let $t \mapsto t'$. If $y \# t$ then $y \# t'$.

Faulty Reasoning

- Consider the two-place relation foo:

$$\frac{}{x \mapsto x} \qquad \frac{}{t_1\ t_2 \mapsto t_1\ t_2} \qquad \frac{t \mapsto t'}{\lambda x.t \mapsto t'}$$

- The lemma we going to prove:

  Let $t \mapsto t'$. If $y \# t$ then $y \# t'$.

- Cases 1 and 2 are trivial:
  - If $y \# x$ then $y \# x$.
  - If $y \# t_1\, t_2$ then $y \# t_1\, t_2$.

Faulty Reasoning

- Consider the two-place relation foo:

$$\frac{}{x \mapsto x} \qquad \frac{}{t_1\ t_2 \mapsto t_1\ t_2} \qquad \frac{t \mapsto t'}{\lambda x.t \mapsto t'}$$

- The lemma we going to prove:

  Let $t \mapsto t'$. If $y \;\#\; t$ then $y \;\#\; t'$.

- Case 3:
  - We know $y \;\#\; \lambda x.t$. We have to show $y \;\#\; t'$.
  - The IH says: if $y \;\#\; t$ then $y \;\#\; t'$.

Variable Convention:

If $M_1, \ldots, M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.

In our case:

The free variables are y and t'; the bound one is x.

By the variable convention we conclude that $x \neq y$.

Let $t \mapsto t'$. If $y \# t$ then $y \# t'$.

- Case 3:
  - We know $y \# \lambda x.t$. We have to show $y \# t'$.
  - The IH says: if $y \# t$ then $y \# t'$.

Variable Convention:

If $M_1, \ldots, M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.

In our case:

The free variables are y and t'; the bound one is x.

By the variable convention we conclude that $x \neq y$.

Let $t \mapsto t'$. If $y \# t$ then $y \# t'$

$$y \not\in \mathsf{fv}(\lambda x.t) \iff y \not\in \mathsf{fv}(t) - \{x\} \overset{x \neq y}{\iff} y \not\in \mathsf{fv}(t)$$

- Case 3:
  - We know $y \# \lambda x.t$. We have to show $y \# t'$.
  - The IH says: if $y \# t$ then $y \# t'$.

Variable Convention:

If $M_1, \ldots, M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.

In our case:

The free variables are y and t'; the bound one is x.

By the variable convention we conclude that $x \neq y$.

Let $t \mapsto t'$. If $y \# t$ then $y \# t'$.

$$y \notin \mathsf{fv}(\lambda x.t) \iff y \notin \mathsf{fv}(t) - \{x\} \overset{x \neq y}{\iff} y \notin \mathsf{fv}(t)$$

- Case 3:
  - We know $y \# \lambda x.t$. We have to show $y \# t'$.
  - The IH says: if $y \# t$ then $y \# t'$.
  - So we have $y \# t$. Hence $y \# t'$ by IH. Done!

Faulty Reasoning

- Consider the two-place relation foo:

$$\frac{}{x \mapsto x} \qquad \frac{}{t_1\ t_2 \mapsto t_1\ t_2} \qquad \frac{t \mapsto t'}{\lambda x.t \mapsto t'}$$

- The lemma we going to prove:

  Let $t \mapsto t'$. If $y \# t$ then $y \# t'$.

- Case 3:
  - We know $y \# \lambda x.t$. We have to show $y \# t'$.
  - The IH says: if $y \# t$ then $y \# t'$.
  - So we have $y \# t$. Hence $y \# t'$ by IH. Done!

VC-Compatibility

- We introduced two conditions that make the VC safe to use in rule inductions:
  - the relation needs to be **equivariant**, and
  - the binder is not allowed to occur in the **support** of the conclusion (not free in the conclusion)

VC-Compatibility

- We introduced two conditions that make the VC safe to use in rule inductions:
  - the relation needs to be **equivariant**, and
  - the binder is not allowed to occur in the

A relation R is **equivariant** iff

$$\forall \pi\, t_1 \ldots t_n$$
$$R\, t_1 \ldots t_n \Rightarrow R(\pi \bullet t_1) \ldots (\pi \bullet t_n)$$

This means the relation has to be invariant under permutative renaming of variables.

VC-Compatibility

- We introduced two conditions that make the VC safe to use in rule inductions:
  - the relation needs to be **equivariant**, and
  - the binder is not allowed to occur in the **support** of the conclusion (not free in the conclusion)

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"

equivariance typing
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"

equivariance typing
nominal_inductive typing
```

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

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"

equivariance typing
nominal_inductive typing
  by (simp_all add: abs_fresh ty_fresh)
```

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. $\bigwedge x\ \Gamma\ T_1\ t\ T_2.\ [\![x \# \Gamma; (x, T_1)::\Gamma \vdash t : T_2]\!] \Longrightarrow x \# T_1 \to T_2$

CK Machine Implies the Evaluation Relation (Via A Small-Step Reduction)

A Direct Attempt

- The statement for the other direction is as follows:

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

A Direct Attempt

- 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
```

A Direct Attempt

- 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
  cbv$_1$: "⟦val v; x#v⟧ ⟹ App (Lam [x].t) v ⟶cbv t[x::=v]"
| cbv$_2$[intro]: "t ⟶cbv t' ⟹ App t t$_2$ ⟶cbv App t' t$_2$"
| cbv$_3$[intro]: "t ⟶cbv t' ⟹ App t$_2$ t ⟶cbv App t$_2$ 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.

Better Introduction Rule

```
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
```

Better Introduction Rule

```
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
```

Better Introduction Rule

```
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
```

Better Introduction Rule

```
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
```

Better Introduction Rule

```
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
```

Better Introduction Rule

```
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
```

Better Introduction Rule

```
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
cbvs$_1$[intro]: "e ⟶cbv* e"
| cbvs$_2$[intro]: "⟦e_1 ⟶cbv e_2; e_2 ⟶cbv* e_3⟧ ⟹ e_1 ⟶cbv* e_3"

lemma cbvs$_3$[intro]:
assumes a: "e_1 ⟶cbv* e_2" "e_2 ⟶cbv* e_3"
shows "e_1 ⟶cbv* e_3"
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)
```

Is another such exercise needed?

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)
```

Your Turn

```
lemma machine_implies_cbvs_ctx:
  assumes a: "⟨e,Es⟩ ↦ ⟨e',Es'⟩"
  shows "(Es↓)⟦e⟧ ⟶cbv* (Es'↓)⟦e'⟧"
using a proof (induct)
  case (m1 t1 t2 Es)

  show "Es↓⟦App t1 t2⟧ ⟶cbv* (CAppL □ t2#Es)↓⟦t1⟧"  sorry
next
  case (m2 v t2 Es)
  have "val v" by fact

  show "(CAppL □ t2#Es)↓⟦v⟧ ⟶cbv* (CAppR v □#Es)↓⟦t2⟧" sorry
next
  case (m3 v x t Es)
  have "val v" by fact

  show "(CAppR Lam [x].t □#Es)↓⟦v⟧ ⟶cbv* (Es↓)⟦t[x::=v]⟧" sorry
qed
```

CBV* Implies Evaluation

- We need the following auxiliary 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 proof(induct arbitrary: t3)
  case (cbv1 v x t t3)
  have a1: "val v" by fact
  have a2: "t[x::=v] ⇓ t3" by fact
  show "App Lam [x].t v ⇓ t3" sorry
next
  case (cbv2 t t' t2 t3)
  have ih: "⋀t3. t' ⇓ t3 ⟹ t ⇓ t3" by fact
  have "App t' t2 ⇓ t3" by fact
  then obtain x t'' v'
    where a1: "t' ⇓ Lam [x].t''"
      and a2: "t2 ⇓ v'"
      and a3: "t''[x::=v'] ⇓ t3" using e_App_elim by blast
  have "t ⇓  Lam [x].t''" using ih a1 by auto
  then show "App t t2 ⇓ t3" using a2 a3 by auto
qed (auto dest!: e_App_elim)
```

```
lemma cbv_eval:
  assumes a: "t₁ ⟶cbv t₂" "t₂ ⇓ t₃"
  shows "t₁ ⇓ t₃"
using a proof(induct arbitrary: t₃)
  case (cbv₁ v x t t₃)
  have a1: "val v" by fact
  have a2: "t[x::=v] ⇓ t₃" by fact
  show "App Lam [x].t v ⇓ t₃" using eval_val a1 a2 by auto
next
  case (cbv₂ t t' t₂ t₃)
  have ih: "⋀t₃. t' ⇓ t₃ ⟹ t ⇓ t₃" by fact
  have "App t' t₂ ⇓ t₃" by fact
  then obtain x t'' v'
    where a1: "t' ⇓ Lam [x].t''"
      and a2: "t₂ ⇓ v'"
      and a3: "t''[x::=v'] ⇓ t₃" using e_App_elim by blast
  have "t ⇓  Lam [x].t''" using ih a1 by auto
  then show "App t t₂ ⇓ t₃" using a2 a3 by auto
qed (auto dest!: e_App_elim)
```

```
lemma cbv_eval:
  assumes a: "t₁ ⟶cbv t₂" "t₂ ⇓ t₃"
  shows "t₁ ⇓ t₃"
using a proof(induct arbitrary: t₃)
  case (cbv₁ v x t t₃)
  have a1: "val v" by fact
  have a2: "t[x::=v] ⇓ t₃" by fact
  show "App Lam [x].t v ⇓ t₃" using eval_val a1 a2 by auto
next
  case (cbv₂ t t' t₂ t₃)
  have ih: "⋀t₃. t' ⇓ t₃ ⟹ t ⇓ t₃" by fact
  have "App t' t₂ ⇓ t₃" by fact
  then obtain x t'' v'
    where a1: "t' ⇓ Lam [x].t''"
      and a2: "t₂ ⇓ v'"
      and a3: "t''[x::=v'] ⇓ t₃" using e_App_elim by blast
  have "t ⇓  Lam [x].t''" using ih a1 by auto
  then show "App t t₂ ⇓ t₃" using a2 a3 by auto
qed (auto dest!: e_App_elim)
```

```
lemma cbv_eval:
  assumes a: "t1 ⟶cbv t2" "t2 ⇓ t3"
  shows "t1 ⇓ t3"
using a proof(induct
  case (cbv1 v x t t3)
  have a1: "val v" by fact
  have a2: "t[x::=v] ⇓ t3" by fact
  show "App Lam [x].t v ⇓ t3" using eval_val a1 a2 by auto
next
  case (cbv2 t t' t2 t3)
  have ih: "⋀t3. t' ⇓ t3 ⟹ t ⇓ t3" by fact
  have "App t' t2 ⇓ t3" by fact
  then obtain x t'' v'
    where a1: "t' ⇓ Lam [x].t''"
      and a2: "t2 ⇓ v'"
      and a3: "t''[x::=v'] ⇓ t3" using e_App_elim by blast
  have "t ⇓  Lam [x].t''" using ih a1 by auto
  then show "App t t2 ⇓ t3" using a2 a3 by auto
qed (auto dest!: e_App_elim)
```

```
lemma e_App_elim:
assumes a: "App t1 t2 ⇓ v"
shows "∃ x t v'. t1 ⇓ Lam [x].t ∧ t2 ⇓ v' ∧ t[x::=v'] ⇓ v"
```

```
lemma cbv_eval:
  assumes a: "t1 ⟶cbv t2" "t2 ⇓ t3"
  shows "t1 ⇓ t3"
using a proof(induct
  case (cbv1 v x t t3)
  have a1: "val v" by fact
  have a2: "t[x::=v] ⇓ t3" by fact
  show "App Lam [x].t v ⇓ t3" using eval_val a1 a2 by auto
next
  case (cbv2 t t' t2 t3)
  have ih: "⋀t3. t' ⇓ t3 ⟹ t ⇓ t3" by fact
  have "App t' t2 ⇓ t3" by fact
  then obtain x t'' v'
    where a1: "t' ⇓ Lam [x].t''"
      and a2: "t2 ⇓ v'"
      and a3: "t''[x::=v'] ⇓ t3" using e_App_elim by blast
  have "t ⇓  Lam [x].t''" using ih a1 by auto
  then show "App t t2 ⇓ t3" using a2 a3 by auto
qed (auto dest!: e_App_elim)
```

```
lemma e_App_elim:
assumes a: "App t1 t2 ⇓ v"
shows "∃ x t v'. t1 ⇓ Lam [x].t ∧ t2 ⇓ v' ∧ t[x::=v'] ⇓ v"
```

```
lemma cbv_eval:
  assumes a: "t₁ ⟶cbv t₂" "t₂ ⇓ t₃"
  shows "t₁ ⇓ t₃"
using a proof(induct arbitrary: t₃)
  case (cbv₁ v x t t₃)
  have a1: "val v" by fact
  have a2: "t[x::=v] ⇓ t₃" by fact
  show "App Lam [x].t v ⇓ t₃" using eval_val a1 a2 by auto
next
  case (cbv₂ t t' t₂ t₃)
  have ih: "⋀t₃. t' ⇓ t₃ ⟹ t ⇓ t₃" by fact
  have "App t' t₂ ⇓ t₃" by fact
  then obtain x t'' v'
    where a1: "t' ⇓ Lam [x].t''"
        and a2: "t₂ ⇓ v'"
        and a3: "t''[x::=v'] ⇓ t₃" using e_App_elim by blast
  have "t ⇓  Lam [x].t''" using ih a1 by auto
  then show "App t t₂ ⇓ t₃" using a2 a3 by auto
qed (auto dest!: e_App_elim)
```

```
lemma cbv_eval:
  assumes a: "t₁ ⟶cbv t₂" "t₂ ⇓ t₃"
  shows "t₁ ⇓ t₃"
using a proof(induct arbitrary: t₃)
  case (cbv₁ v x t t₃)
  have a1: "val v" by fact
  have a2: "t[x::=v] ⇓ t₃" by fact
  show "App Lam [x].t v ⇓ t₃" using eval_val a1 a2 by auto
next
  case (cbv₂ t t' t₂ t₃)
  have ih: "⋀t₃. t' ⇓ t₃ ⟹ t ⇓ t₃" by fact
  have "App t' t₂ ⇓ t₃" by fact
  then obtain x t'' v'
    where a1: "t' ⇓ Lam [x].t''"
      and a2: "t₂ ⇓ v'"
      and a3: "t''[x::=v'] ⇓ t₃" using e_App_elim by blast
  have "t ⇓  Lam [x].t''" using ih a1 by auto
  then show "App t t₂ ⇓ t₃" using a2 a3 by auto
qed (auto dest!: e_App_elim)
```

```
lemma cbv_eval:
  assumes a: "t₁ ⟶cbv t₂" "t₂ ⇓ t₃"
  shows "t₁ ⇓ t₃"
using a proof(induct arbitrary: t₃)
  case (cbv₁ v x t t₃)
  have a1: "val v" by fact
  have a2: "t[x::=v] ⇓ t₃" by fact
  show "App Lam [x].t v ⇓ t₃" using eval_val a1 a2 by auto
next
  case (cbv₂ t t' t₂ t₃)
  have ih: "⋀t₃. t' ⇓ t₃ ⟹ t ⇓ t₃" by fact
  have "App t' t₂ ⇓ t₃" by fact
  then obtain x t'' v'
    where a1: "t' ⇓ Lam [x].t''"
      and a2: "t₂ ⇓ v'"
      and a3: "t''[x::=v'] ⇓ t₃" using e_App_elim by blast
  have "t ⇓  Lam [x].t''" using ih a1 by auto
  then show "App t t₂ ⇓ t₃" using a2 a3 by auto
qed (auto dest!: e_App_elim)
```

Nothing Interesting

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

lemma cbvs_implies_eval:
  assumes a: "t ⟶cbv* v" "val v"
  shows "t ⇓ v"
using a by (induct) (auto intro: eval_val cbvs_eval)

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

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"
```

Honest Toil, No Theft!

- The sacred principle of HOL:

> "The method of 'postulating' what we want has many advantages; they are the same as the advantages of theft over honest toil."
>
> B. Russell, Introduction of Mathematical Philosophy

- I will show next that the weak structural induction principle implies the strong structural induction principle.

  (I am only going to show the lambda-case.)

Permutations

A permutation **acts** on variable names as follows:

$$
\begin{aligned}
[]\bullet a &\stackrel{\text{def}}{=} a \\
((a_1\,a_2)::\pi)\bullet a &\stackrel{\text{def}}{=} \begin{cases} a_1 & \text{if } \pi\bullet a = a_2 \\ a_2 & \text{if } \pi\bullet a = a_1 \\ \pi\bullet a & \text{otherwise} \end{cases}
\end{aligned}
$$

- $[]$ stands for the empty list (the identity permutation), and
- $(a_1\,a_2)::\pi$ stands for the permutation π followed by the swapping $(a_1\,a_2)$.

Permutations on Lambda-Terms

- Permutations act on lambda-terms as follows:

$$
\begin{array}{rcl}
\pi \bullet x & \stackrel{\text{def}}{=} & \text{“action on variables”} \\
\pi \bullet (t_1\ t_2) & \stackrel{\text{def}}{=} & (\pi \bullet t_1)\ (\pi \bullet t_2) \\
\pi \bullet (\lambda x.t) & \stackrel{\text{def}}{=} & \lambda(\pi \bullet x).(\pi \bullet t)
\end{array}
$$

- Alpha-equivalence can be defined as:

$$\frac{t_1 = t_2}{\lambda x.t_1 = \lambda x.t_2}$$

$$\frac{x \neq y \quad t_1 = (x\,y) \bullet t_2 \quad x \# t_2}{\lambda x.t_1 = \lambda y.t_2}$$

Permutations on Lambda-Terms

- Permutations act on lambda-terms as follows:

$$\begin{aligned}\pi \bullet x &\stackrel{\text{def}}{=} \text{"action on variables"}\\ \pi \bullet (t_1\ t_2) &\stackrel{\text{def}}{=} (\pi \bullet t_1)\ (\pi \bullet t_2)\\ \pi \bullet (\lambda x.t) &\stackrel{\text{def}}{=} \lambda(\pi \bullet x).(\pi \bullet t)\end{aligned}$$

- Alpha-equivalence can be defined as:

$$\frac{t_1 = t_2}{\lambda x.t_1 = \lambda x.t_2}$$

$$\frac{x \neq y \quad t_1 = (x\ y) \bullet t_2 \quad x \,\#\, t_2}{\lambda x.t_1 = \lambda y.t_2}$$

Notice, I wrote equality here!

My Claim

$$\frac{\forall x.\, P\ x \qquad \forall t_1\, t_2.\, P\ t_1 \wedge P\ t_2 \Rightarrow P\ (t_1\ t_2) \qquad \forall x\, t.\, P\ t \Rightarrow P\ (\lambda x.t)}{P\ t}$$

implies

$$\frac{\forall x\, c.\, P\, c\, x \qquad \forall t_1\, t_2\, c.\, (\forall d.\, P\, d\, t_1) \wedge (\forall d.\, P\, d\, t_2) \Rightarrow P\, c\, (t_1\, t_2) \qquad \forall x\, t\, c.\, x \# c \wedge (\forall d.\, P\, d\, t) \Rightarrow P\, c\, (\lambda x.t)}{P\, c\, t}$$

Proof for the Strong Induction Principle

- We prove $P\,c\,t$ by induction on t.

Proof for the Strong Induction Principle

- We prove $\forall \pi\, c.\ P c\, (\pi \bullet t)$ by induction on t.

Proof for the Strong Induction Principle

- We prove $\forall \pi\ c.\ P c\,(\pi \bullet t)$ by induction on t.
- I.e., we have to show $P c\,(\pi \bullet (\lambda x.t))$.

Proof for the Strong Induction Principle

- We prove $\forall \pi\, c.\ P c\, (\pi \bullet t)$ by induction on t.
- I.e., we have to show $P c\, \lambda(\pi \bullet x).(\pi \bullet t)$.

Proof for the Strong Induction Principle

- We prove $\forall \pi\, c.\ P c\, (\pi \bullet t)$ by induction on t.
- I.e., we have to show $P c\, \lambda(\pi \bullet x).(\pi \bullet t)$.
- We have $\forall \pi\, c.\ P c\, (\pi \bullet t)$ by induction.

Proof for the Strong Induction Principle

- We prove $\forall \pi\, c.\ P c\, (\pi \bullet t)$ by induction on t.
- I.e., we have to show $P c\, \lambda(\pi \bullet x).(\pi \bullet t)$.
- We have $\forall \pi\, c.\ P c\, (\pi \bullet t)$ by induction.
- Our weaker precondition says that:

$$\forall x\, t\, c.\, x \,\#\, c \wedge (\forall c.\, P c\, t) \Rightarrow P c\, (\lambda x.t)$$

Proof for the Strong Induction Principle

- We prove $\forall \pi\, c.\ P c\, (\pi \bullet t)$ by induction on t.
- I.e., we have to show $P c\, \lambda(\pi \bullet x).(\pi \bullet t)$.
- We have $\forall \pi\, c.\ P c\, (\pi \bullet t)$ by induction.
- Our weaker precondition says that:

$$\forall x\, t\, c.\, x \;\#\; c \wedge (\forall c.\, P c\, t) \Rightarrow P c\, (\lambda x.t)$$

- We choose a fresh y such that $y \;\#\; (\pi \bullet x, \pi \bullet t, c)$.

Proof for the Strong Induction Principle

- We prove $\forall \pi\, c.\ P\, c\, (\pi \bullet t)$ by induction on t.
- I.e., we have to show $P\, c\, \lambda(\pi \bullet x).(\pi \bullet t)$.
- We have $\forall \pi\, c.\ P\, c\, (\pi \bullet t)$ by induction.
- Our weaker precondition says that:

$$\forall x\, t\, c.\, x \;\#\; c \wedge (\forall c.\, P\, c\, t) \Rightarrow P\, c\, (\lambda x.t)$$

- We choose a fresh y such that $y \;\#\; (\pi \bullet x, \pi \bullet t, c)$.
- Now we can use $\forall c.\ P\, c\, (((y\ \ \pi \bullet x) :: \pi) \bullet t)$

Proof for the Strong Induction Principle

- We prove $\forall \pi\, c.\ P c\, (\pi \bullet t)$ by induction on t.
- I.e., we have to show $P c\, \lambda(\pi \bullet x).(\pi \bullet t)$.
- We have $\forall \pi\, c.\ P c\, (\pi \bullet t)$ by induction.
- Our weaker precondition says that:

$$\forall x\, t\, c.\, x \# c \wedge (\forall c.\, P c\, t) \Rightarrow P c\, (\lambda x.t)$$

- We choose a fresh y such that $y \# (\pi \bullet x, \pi \bullet t, c)$.
- Now we can use $\forall c.\ P c\, ((y\ \ \pi \bullet x) \bullet \pi \bullet t)$

Proof for the Strong Induction Principle

- We prove $\forall \pi\, c.\ P c\, (\pi \bullet t)$ by induction on t.
- I.e., we have to show $P c\, \lambda(\pi \bullet x).(\pi \bullet t)$.
- We have $\forall \pi\, c.\ P c\, (\pi \bullet t)$ by induction.
- Our weaker precondition says that:

$$\forall x\, t\, c.\, x \ \# \ c \wedge (\forall c.\, P c\, t) \Rightarrow P c\, (\lambda x.t)$$

- We choose a fresh y such that $y \ \# \ (\pi \bullet x, \pi \bullet t, c)$.
- Now we can use $\forall c.\ P c\, ((y\ \ \pi \bullet x) \bullet \pi \bullet t)$ to infer

$$P\, c\, \lambda y.((y\ \ \pi \bullet x) \bullet \pi \bullet t)$$

Proof for the Strong Induction Principle

- We prove $\forall \pi\ c.\ P c\,(\pi \bullet t)$ by induction on t.
- I.e., we have to show $P c\, \lambda(\pi \bullet x).(\pi \bullet t)$.
- We have $\forall \pi\ c.\ P c\,(\pi \bullet t)$ by induction.
- Our weak

$$\frac{x \neq y \quad t_1 = (x\ y)\bullet t_2 \quad y \,\#\, t_2}{\lambda y.t_1 = \lambda x.t_2}$$

$\forall x\ t$ ... $t)$

- We choose a fresh y such that $y \,\#\, (\pi \bullet x, \pi \bullet t, c)$.
- Now we can use $\forall c.\ P c\,((y\ \ \pi \bullet x)\bullet \pi \bullet t)$ to infer

$$P\,c\,\lambda y.((y\ \ \pi \bullet x)\bullet \pi \bullet t)$$

- However

$$\lambda y.((y\ \ \pi \bullet x)\bullet \pi \bullet t) = \lambda(\pi \bullet x).(\pi \bullet t)$$

Proof for the Strong Induction Principle

- We prove $\forall \pi\, c.\; P c\, (\pi \bullet t)$ by induction on t.
- I.e., we have to show $P c\, \lambda(\pi \bullet x).(\pi \bullet t)$.
- We have $\forall \pi\, c.\; P c\, (\pi \bullet t)$ by induction.
- Our weaker precondition says that:

$$\forall x\, t\, c.\, x \;\#\; c \wedge (\forall c.\, P c\, t) \Rightarrow P c\, (\lambda x.t)$$

- We choose a fresh y such that $y \;\#\; (\pi \bullet x, \pi \bullet t, c)$.
- Now we can use $\forall c.\; P c\, ((y\;\; \pi \bullet x) \bullet \pi \bullet t)$ to infer

$$P\, c\, \lambda y.((y\;\; \pi \bullet x) \bullet \pi \bullet t)$$

- However

$$\lambda y.((y\;\; \pi \bullet x) \bullet \pi \bullet t) = \lambda(\pi \bullet x).(\pi \bullet t)$$

- Therefore $P\, c\, \lambda(\pi \bullet x).(\pi \bullet t)$ and we are done.

This Proof in Isabelle

```
lemma lam_strong_induct:
  fixes c::"'a::fs_name"
  assumes h₁: "⋀x c. P c (Var x)"
  and     h₂: "⋀t₁ t₂ c. ⟦∀d. P d t₁; ∀d. P d t₂⟧ ⟹ P c (App t₁ t₂)"
  and     h₃: "⋀x t c. ⟦x#c; ∀d. P d t⟧ ⟹ P c (Lam [x].t)"
  shows "P c t"
proof -
  have "∀(π::name prm) c. P c (π•t)"   . . .
  then have "P c (([]::name prm)•t)" by blast
  then show "P c t" by simp
qed
```

interesting bit

Interesting Bit

```
...
have "∀ (π::name prm) c. P c (π • t)"
proof (induct t rule: lam.induct)
  case (Lam x t)
  have ih: "∀ (π::name prm) c. P c (π • t)" by fact
  { fix π::"name prm" and c::"'a::fs_name"
    obtain y::"name" where fc: "y#(π • x,π • t,c)"
      by (rule exists_fresh) (auto simp add: fs_name1)
    from ih have "∀ c. P c (([(y,π • x)]@π) • t)" by simp
    then have "∀ c. P c ([(y,π • x)] • (π • t))" by (auto simp only: pt_name2)
    with h3 have "P c (Lam [y].[(y,π • x)] • (π • t))" using fc by (simp add: fresh_prod)
    moreover
    have "Lam [y].[(y,π • x)] • (π • t) = Lam [(π • x)].(π • t)"
      using fc by (simp add: lam.inject alpha fresh_atm fresh_prod)
    ultimately have "P c (Lam [(π • x)].(π • t))" by simp
  }
  then have "∀ (π::name prm) c. P c (Lam [(π • x)].(π • t))" by simp
  then show "∀ (π::name prm) c. P c (π • (Lam [x].t))" by simp
qed (auto intro: h1 h2)
...
```

Interesting Bit

```
...
have "∀ (π::name prm) c. P c (π • t)"
proof (induct t rule: lam.induct)
  case (Lam x t)
  have ih: "∀ (π::name prm) c. P c (π • t)" by fact
  { fix π::"name prm" and c::"'a::fs_name"
    obtain y::"name" where fc: "y#(π • x,π • t,c)"
      by (rule exists_fresh) (auto simp add: fs_name1)
    from ih have "∀ c. P c (([(y,π • x)]@π) • t)" by simp
    then have "∀ c. P c ([(y,π • x)] • (π • t))" by (auto simp only: pt_name2)
    with h3 have "P c (Lam [y].[(y,π • x)] • (π • t))" using fc by (simp add: fresh_prod)
    moreover
    have "Lam [y].[(y,π • x)] • (π • t) = Lam [(π • x)].(π • t)"
      using fc by (simp add: lam.inject alpha fresh_atm fresh_prod)
    ultimately have "P c (Lam [(π • x)].(π • t))" by simp
  }
  then have "∀ (π::name prm) c. P c (Lam [(π • x)].(π • t))" by simp
  then show "∀ (π::name prm) c. P c (π • (Lam [x].t))" by simp
qed (auto intro: h1 h2)
...
```

Interesting Bit

```
...
have "∀ (π::name prm) c. P c (π•t)"
proof (induct t rule: lam.induct)
  case (Lam x t)
  have ih: "∀ (π::name prm) c. P c (π•t)" by fact
  { fix π::"name prm" and c::"'a::fs_name"
    obtain y::"name" where fc: "y#(π•x,π•t,c)"
      by (rule exists_fresh) (auto simp add: fs_name1)
    from ih have "∀ c. P c (([(y,π•x)]@π)•t)" by simp
    then have "∀ c. P c ([(y,π•x)]•(π•t))" by (auto simp only: pt_name2)
    with h3 have "P c (Lam [y].[(y,π•x)]•(π•t))" using fc by (simp add: fresh_prod)
    moreover
    have "Lam [y].[(y,π•x)]•(π•t) = Lam [(π•x)].(π•t)"
      using fc by (simp add: lam.inject alpha fresh_atm fresh_prod)
    ultimately have "P c (Lam [(π•x)].(π•t))" by simp
  }
  then have "∀ (π::name prm) c. P c (Lam [(π•x)].(π•t))" by simp
  then show "∀ (π::name prm) c. P c (π•(Lam [x].t))" by simp
qed (auto intro: h1 h2)
...
```

Interesting Bit

```
...
have "∀(π::name prm) c. P c (π•t)"
proof (induct t rule: lam.induct)
  case (Lam x t)
  have ih: "∀(π::name prm) c. P c (π•t)" by fact
  { fix π::"name prm" and c::"'a::fs_name"
    obtain y::"name" where fc: "y#(π•x,π•t,c)"
      by (rule exists_fresh) (auto simp add: fs_name1)
    from ih have "∀c. P c (([(y,π•x)]@π)•t)" by simp
    then have "∀c. P c ([(y,π•x)]•(π•t))" by (auto simp only: pt_name2)
    with h₃ have "P c (Lam [y].[(y,π•x)]•(π•t))" using fc by (simp add: fresh_prod)
    moreover
    have "Lam [y].[(y,π•x)]•(π•t) = Lam [(π•x)].(π•t)"
      using fc by (simp add: lam.inject alpha fresh_atm fresh_prod)
    ultimately have "P c (Lam [(π•x)].(π•t))" by simp
  }
  then have "∀(π::name prm) c. P c (Lam [(π•x)].(π•t))" by simp
  then show "∀(π::name prm) c. P c (π•(Lam [x].t))" by simp
qed (auto intro: h₁ h₂)
...
```

Interesting Bit

```
...
have "∀(π::name prm) c. P c (π•t)"
proof (induct t rule: lam.induct)
  case (Lam x t)
  have ih: "∀(π::name prm) c. P c (π•t)" by fact
  { fix π::"name prm" and c::"'a::fs_name"
    obtain y::"name" where fc: "y#(π•x,π•t,c)"
      by (rule exists_fresh) (auto simp add: fs_name1)
    from ih have "∀c. P c (([(y,π•x)]@π)•t)" by simp
    then have "∀c. P c ([(y,π•x)]•(π•t))" by (auto simp only: pt_name2)
    with h3 have "P c (Lam [y].[(y,π•x)]•(π•t))" using fc by (simp add: fresh_prod)
    moreover
    have "Lam [y].[(y,π•x)]•(π•t) = Lam [(π•x)].(π•t)"
      using fc by (simp add: lam.inject alpha fresh_atm fresh_prod)
    ultimately have "P c (Lam [(π•x)].(π•t))" by simp
  }
  then have "∀(π::name prm) c. P c (Lam [(π•x)].(π•t))" by simp
  then show "∀(π::name prm) c. P c (π•(Lam [x].t))" by simp
qed (auto intro: h1 h2)
...
```

Interesting Bit

```
...
have "∀(π::name prm) c. P c (π•t)"
proof (induct t rule: lam.induct)
  case (Lam x t)
  have ih: "∀(π::name prm) c. P c (π•t)" by fact
  { fix π::"name prm" and c::"'a::fs_name"
    obtain y::"name" where fc: "y#(π•x,π•t,c)"
      by (rule exists_fresh) (auto simp add: fs_name1)
    from ih have "∀c. P c (([(y,π•x)]@π)•t)" by simp
    then have "∀c. P c ([(y,π•x)]•(π•t))" by (auto simp only: pt_name2)
    with h3 have "P c (Lam [y].[(y,π•x)]•(π•t))" using fc by (simp add: fresh_prod)
    moreover
    have "Lam [y].[(y,π•x)]•(π•t) = Lam [(π•x)].(π•t)"
      using fc by (simp add: lam.inject alpha fresh_atm fresh_prod)
    ultimately have "P c (Lam [(π•x)].(π•t))" by simp
  }
  then have "∀(π::name prm) c. P c (Lam [(π•x)].(π•t))" by simp
  then show "∀(π::name prm) c. P c (π•(Lam [x].t))" by simp
qed (auto intro: h1 h2)
...
```

Interesting Bit

```
...
have "∀(π::name prm) c. P c (π•t)"
proof (induct t rule: lam.induct)
  case (Lam x t)
  have ih: "∀(π::name prm) c. P c (π•t)" by fact
  { fix π::"name prm" and c::"'a::fs_name"
    obtain y::"name" where fc: "y#(π•x,π•t,c)"
      by (rule exists_fresh) (auto simp add: fs_name1)
    from ih have "∀c. P c (([(y,π•x)]@π)•t)" by simp
    then have "∀c. P c ([(y,π•x)]•(π•t))" by (auto simp only: pt_name2)
    with h₃ have "P c (Lam [y].[(y,π•x)]•(π•t))" using fc by (simp add: fresh_prod)
    moreover
    have "Lam [y].[(y,π•x)]•(π•t) = Lam [(π•x)].(π•t)"
      using fc by (simp add: lam.inject alpha fresh_atm fresh_prod)
    ultimately have "P c (Lam [(π•x)].(π•t))" by simp
  }
  then have "∀(π::name prm) c. P c (Lam [(π•x)].(π•t))" by simp
  then show "∀(π::name prm) c. P c (π•(Lam [x].t))" by simp
qed (auto intro: h₁ h₂)
...
```

h_3: "$\bigwedge$x t c. ⟦x # c; ∀d. P d t⟧ $\Longrightarrow$ P c Lam [x].t"

Interesting Bit

```
...
have "∀(π::name prm) c. P c (π•t)"
proof (induct t rule: lam.induct)
  case (Lam x t)
  have ih: "∀(π::name prm) c. P c (π•t)" by fact
  { fix π::"name prm" and c::"'a::fs_name"
    obtain y::"name" where fc: "y#(π•x,π•t,c)"
      by (rule exists_fresh) (auto simp add: fs_name1)
    from ih have "∀c. P c (([(y,π•x)]@π)•t)" by simp
    then have "∀c. P c ([(y,π•x)]•(π•t))" by (auto simp only: pt_name2)
    with h₃ have "P c (Lam [y].[(y,π•x)]•(π•t))" using fc by (simp add: fresh_prod)
    moreover
    have "Lam [y].[(y,π•x)]•(π•t) = Lam [(π•x)].(π•t)"
      using fc by (simp add: lam.inject alpha fresh_atm fresh_prod)
    ultimately have "P c (Lam [(π•x)].(π•t))" by simp
  }
  then have "∀(π::name prm) c. P c (Lam [(π•x)].(π•t))" by simp
  then show "∀(π::name prm) c. P c (π•(Lam [x].t))" by simp
qed (auto intro: h₁ h₂)
...
```

Interesting Bit

```
...
have "∀(π::name prm) c. P c (π•t)"
proof (induct t rule: lam.induct)
  case (Lam x t)
  have ih: "∀(π::name prm) c. P c (π•t)" by fact
  { fix π::"name prm" and c::"'a::fs_name"
    obtain y::"name" where fc: "y#(π•x,π•t,c)"
      by (rule exists_fresh) (auto simp add: fs_name1)
    from ih have "∀c. P c (([(y,π•x)]@π)•t)" by simp
    then have "∀c. P c ([(y,π•x)]•(π•t))" by (auto simp only: pt_name2)
    with h₃ have "P c (Lam [y].[(y,π•x)]•(π•t))" using fc by (simp add: fresh_prod)
    moreover
    have "Lam [y].[(y,π•x)]•(π•t) = Lam [(π•x)].(π•t)"
      using fc by (simp add: lam.inject alpha fresh_atm fresh_prod)
    ultimately have "P c (Lam [(π•x)].(π•t))" by simp
  }
  then have "∀(π::name prm) c. P c (Lam [(π•x)].(π•t))" by simp
  then show "∀(π::name prm) c. P c (π•(Lam [x].t))" by simp
qed (auto intro: h₁ h₂)
...
```

Interesting Bit

```
...
have "∀ (π::name prm) c. P c (π • t)"
proof (induct t rule: lam.induct)
  case (Lam x t)
  have ih: "∀ (π::name prm) c. P c (π • t)" by fact
  { fix π::"name prm" and c::"'a::fs_name"
    obtain y::"name" where fc: "y#(π • x,π • t,c)"
      by (rule exists_fresh) (auto simp add: fs_name1)
    from ih have "∀ c. P c (([(y,π • x)]@π) • t)" by simp
    then have "∀ c. P c ([(y,π • x)] • (π • t))" by (auto simp only: pt_name2)
    with h₃ have "P c (Lam [y].[(y,π • x)] • (π • t))" using fc by (simp add: fresh_prod)
    moreover
    have "Lam [y].[(y,π • x)] • (π • t) = Lam [(π • x)].(π • t)"
      using fc by (simp add: lam.inject alpha fresh_atm fresh_prod)
    ultimately have "P c (Lam [(π • x)].(π • t))" by simp
  }
  then have "∀ (π::name prm) c. P c (Lam [(π • x)].(π • t))" by simp
  then show "∀ (π::name prm) c. P c (π • (Lam [x].t))" by simp
qed (auto intro: h₁ h₂)
...
```

Some Examples

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

Some Examples

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

$$\frac{t \mapsto t'}{\mathrm{Lam}\,[x].t \mapsto t'}$$

Some Examples

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

$$\frac{t \mapsto t'}{\mathrm{Lam}\ [x].t \mapsto t'}$$

Some Examples

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

$$\frac{t \mapsto t'}{\mathsf{Lam}\,[x].t \mapsto t'}$$

$$\frac{\Gamma \vdash_\Sigma A_1 : \mathsf{Type} \quad (x, A_1)::\Gamma \vdash_\Sigma M_2 : A_2 \quad x \# (\Gamma, A_1)}{\Gamma \vdash_\Sigma \mathsf{Lam}\,[x{:}A_1].M_2 : \Pi[x{:}A_1].A_2}$$

Some Examples

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

$$\frac{t \mapsto t'}{\ldots[x].t \mapsto t'}$$

free

$$\frac{\Gamma \vdash_{\Sigma} A_1 : \mathsf{Type} \qquad (x, A_1)::\Gamma \vdash_{\Sigma} M_2 : A_2 \qquad x \# (\Gamma, A_1)}{\Gamma \vdash_{\Sigma} \mathsf{Lam}\ [x{:}A_1].M_2 : \Pi[x{:}A_1].A_2}$$

bound

bound

Some Examples

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

$$\frac{t \mapsto t'}{\mathsf{Lam}\,[x].t \mapsto t'}$$

$$\frac{\Gamma \vdash_\Sigma A_1 : \mathsf{Type} \qquad (x, A_1)::\Gamma \vdash_\Sigma M_2 : A_2 \qquad x \;\#\; (\Gamma, A_1)}{\Gamma \vdash_\Sigma \mathsf{Lam}\,[x \ldots \Pi[x{:}A_1].\ldots}$$

free free free

$$\frac{(x, \tau_1)::\Delta \vdash_\Sigma \mathsf{App}\,M\,(\mathsf{Var}\,x) \Leftrightarrow \mathsf{App}\,N\,(\mathsf{Var}\,x) : \tau_2 \qquad x \;\#\; (\Delta, M, N)}{\Delta \vdash_\Sigma M \Leftrightarrow N : \tau_1 \to \tau_2}$$

Some Examples

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

$$\frac{t \mapsto t'}{\mathsf{Lam}\,[x].t \mapsto t'}$$

$$\frac{\Gamma \vdash_\Sigma A_1 : \mathsf{Type} \quad (x, A_1)::\Gamma \vdash_\Sigma M_2 : A_2 \quad x \# (\Gamma, A_1)}{\Gamma \vdash_\Sigma \mathsf{Lam}\,[x{:}A_1].M_2 : \Pi[x{:}A_1].A_2}$$

$$\frac{\begin{array}{c}(x, \tau_1)::\Delta \vdash_\Sigma \mathsf{App}\,M\,(\mathsf{Var}\,x) \Leftrightarrow \mathsf{App}\,N\,(\mathsf{Var}\,x) : \tau_2 \\ x \# (\Delta, M, N)\end{array}}{\Delta \vdash_\Sigma M \Leftrightarrow N : \tau_1 \to \tau_2}$$

Formalisation of LF

```
nominal_datatype
    kind =   Type
           | KPi "ty" "«name»kind"
and ty =   TConst "id"
           | TApp "ty" "trm"
           | TPi "ty" "«name»ty"
and trm = Const "id"
           | Var "name"
           | App "trm" "trm"
           | Lam "ty" "«name»trm"

abbreviation KPi_syn :: "name ⇒ ty ⇒ kind ⇒ kind" ("Π[_:_]._")
where "Π[x:A].K ≡ KPi A x K"

abbreviation TPi_syn :: "name ⇒ ty ⇒ ty ⇒ ty" ("Π[_:_]._")
where "Π[x:A1].A2 ≡ TPi A1 x A2"

abbreviation Lam_syn :: "name ⇒ ty ⇒ trm ⇒ trm" ("Lam [_:_]._")
where  "Lam [x:A].M ≡ Lam A x M"
```

Formalisation of LF

(joint work with Cheney and Berghofer)

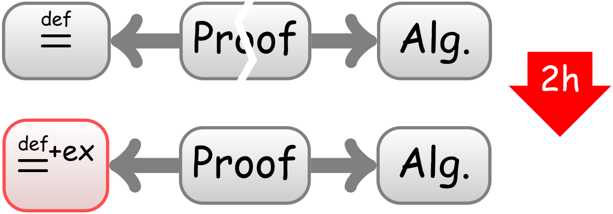
Formalisation of LF

(joint work with Cheney and Berghofer)

Formalisation of LF

(joint work with Cheney and Berghofer)

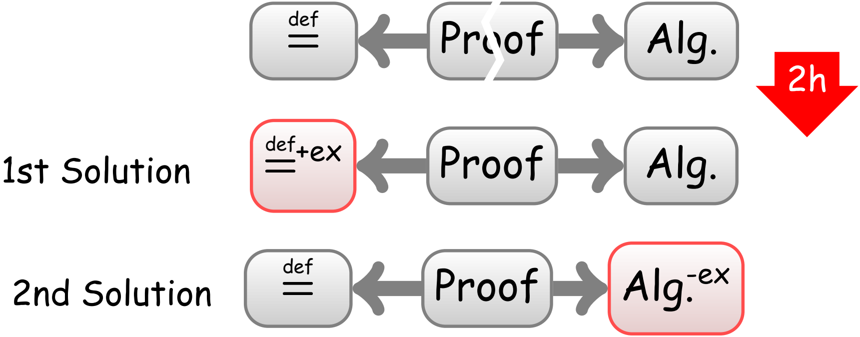
1st Solution



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

Formalisation of LF

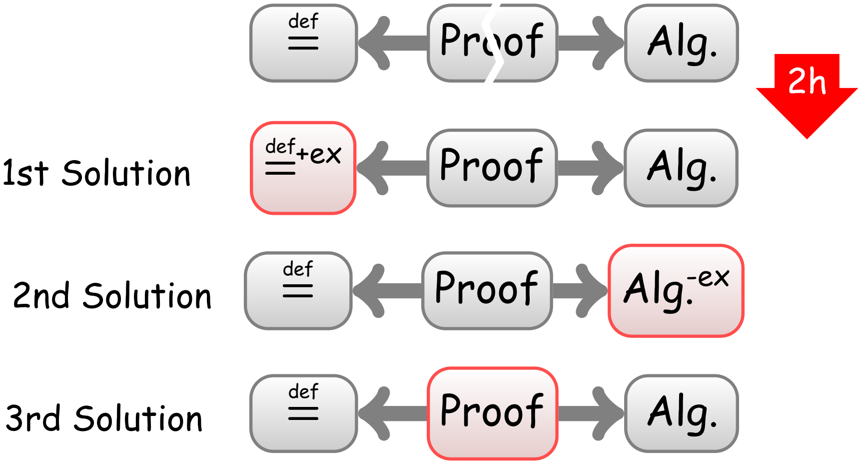
(joint work with Cheney and Berghofer)



(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)

In My PhD

```
nominal_datatype trm =
    Ax   "name" "coname"
  | Cut  "«coname»trm" "«name»trm"                  ("Cut ⟨_⟩._ (_)._")
  | NotR "«name»trm" "coname"                       ("NotR (_)._ _")
  | NotL "«coname»trm" "name"                       ("NotL ⟨_⟩._ _")
  | AndR "«coname»trm" "«coname»trm" "coname"       ("AndR ⟨_⟩._ ⟨_⟩._ _")
  | AndL1 "«name»trm" "name"                        ("AndL1 (_)._ _")
  | AndL2 "«name»trm" "name"                        ("AndL2 (_)._ _")
  | OrR1 "«coname»trm" "coname"                     ("OrR1 ⟨_⟩._ _")
  | OrR2 "«coname»trm" "coname"                     ("OrR2 ⟨_⟩._ _")
  | OrL "«name»trm" "«name»trm" "name"              ("OrL (_)._ (_)._ _")
  | ImpR "«name»(«coname»trm)" "coname"             ("ImpR (_).⟨_⟩._ _")
  | ImpL "«coname»trm" "«name»trm" "name"           ("ImpL ⟨_⟩._ (_)._ _")
```

- A SN-result for cut-elimination in CL: reviewed by Henk Barendregt and Andy Pitts, and reviewers of conference and journal paper. Still, I found errors in central lemmas; fortunately the main claim was correct :o)

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)

Conclusions

- The Nominal Isabelle automatically derives the strong structural induction principle for **all** nominal datatypes (not just the lambda-calculus);
- also for rule inductions (though they have to satisfy a vc-condition).
- They are easy to use: you just have to think carefully what the variable convention should be.
- We can explore the dark corners of the variable convention: when and where it can actually be used.

Conclusions

- The Nominal Isabelle automatically derives the strong structural induction principle for <u>**all**</u> nominal datatypes (not just the lambda-calculus);
- also for rule inductions (though they have to satisfy a vc-condition).
- They are easy to use: you just have to think carefully what the variable convention should be.
- We can explore the dark corners of the variable convention: when and where it can actually be used.
- **Main Point:** Actually these proofs using the variable convention are all trivial / obvious / routine... **provided** you use Nominal Isabelle. ;o)

Thank you very much!