Welcome!

- Files and Programme at: `http://goo.gl/Aslc9`
- Have you already installed Nominal Isabelle?

  If yes, good.

  ```
  isabelle jedit -l HOL-Nominal2 Tutorial1.thy
  ```

  If no, install it now.

Nominal Isabelle

Cezary Kaliszyk and Christian Urban

Quick overview:

$$
\begin{tikzcd}
\_ \Downarrow \_ \arrow[rr] & & \text{machine} \arrow[dl] \\
& \_ \longrightarrow\text{cbv*} \_ \arrow[ul] &
\end{tikzcd}
$$

+ type preservation and progress

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

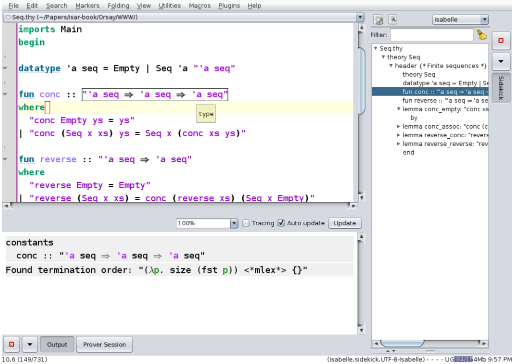
```
nominal_datatype lam =
  Var "name"
| App "lam" "lam"
| Lam x::"name" l::"lam" bind x in l ("Lam [_]. _")
```

- which give you named α-equivalence classes:

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

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

Isabelle/jEdit



Important points:

- the complete buffer is checked
- checking also as you type

Symbols

- ... jEdit provides 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-symbol or \\<name-of-symbol>

Symbols

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

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

- they need to be input via the combination

name-of-symbol or \<name-of-symbol>

- short-cuts for often used symbols

$$\begin{array}{cccccccc} \texttt{==>} & \ldots & \Longrightarrow & & \texttt{/\textbackslash} & \ldots & \bigwedge \\ \texttt{=>} & \ldots & \Rightarrow & & \texttt{\textbackslash/} & \ldots & \bigvee \end{array}$$

Isabelle Theories

- Every theory is of the form

theory Name
imports $T_1 \ldots T_n$
begin
...
end

Isabelle Theories

- Every theory is of the form

  theory Name
  imports $\mathsf{T}_1 \ldots \mathsf{T}_n$
  begin
  ...
  end

- Normally, one T will be the theory **Main**.

Types

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

- Types can be queried in Isabelle using:

```
typ nat
typ bool
typ lam
typ "('a × 'b)"
typ "'c set"
typ "'a list"
typ "lam ⇒ nat"
```

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 "(True, ''c'')"
term "Suc 0"
term "Lam [x].Var x"
term "App t1 t2"
term "atom (x::name)"
```

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 "(True, ''c'')"
term "Suc 0"
term "Lam [x].Var x"
term "App t1 t2"
term "atom (x::name)"
```

- 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"
term "atom x # t"
```

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"
  term "atom x # t"

- When working with Isabelle, one deals with an object logic (that is HOL) and Isabelle's rule framework (called Pure).

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

Inductive Predicates and Theorems

```
inductive
  eval :: "lam ⇒ lam ⇒ bool" ("_ ⇓ _" [60, 60] 60)
where
  e_Lam[intro]:  "Lam [x].t ⇓ Lam [x].t"
| e_App[intro]:
   "⟦t1 ⇓ Lam [x].t; t2 ⇓ v'; t[x::=v'] ⇓ v⟧ ⟹ App t1 t2 ⇓ v"
```

```
inductive
  eval :: "lam ⇒ lam ⇒ bool" ("_ ⇓ _" [60, 60] 60)
where
  e_Lam[intro]:  "Lam [x].t ⇓ Lam [x].t"
| e_App[intro]:
   "⟦t1 ⇓ Lam [x].t; t2 ⇓ v'; t[x::=v'] ⇓ v⟧ ⟹ App t1 t2 ⇓ v"
```

- The type of the predicate is always something to bool.
- The attribute [intro] adds the corresponding clause to the hint-theorem base.
- The clauses correspond to the rules

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

$$\frac{t1 \Downarrow \text{Lam }[x].\ t \qquad t2 \Downarrow v' \qquad t\ [x ::= v'] \Downarrow v}{\text{App } t1\ t2 \Downarrow v}$$

Theorems

- Isabelle's theorem database can be queried using

```
thm e_Lam
thm e_App
thm conjI
thm conjunct1
```

Theorems

- Isabelle's theorem database can be queried using

  thm e_Lam
  thm e_App
  thm conjI
  thm conjunct1

e_Lam:	Lam [?x]. ?t $\Downarrow$ Lam [?x]. ?t
e_App:	$[\![$?t1.0 $\Downarrow$ Lam [?x]. ?t; ?t2.0 $\Downarrow$?v'; ?t [?x ::= ?v'] $\Downarrow$?v$]\!]$
	$\Longrightarrow$ App ?t1.0 ?t2.0 $\Downarrow$?v
conjI:	?P $\Longrightarrow$?Q $\Longrightarrow$?P $\wedge$?Q
conjunct1:	?P $\wedge$?Q $\Longrightarrow$?P

Theorems

- Isabelle's theorem database can be queried using

  thm e_Lam
  thm e_App
  thm conjI
  thm conjunct1

schematic variables

e_Lam:	$\mathrm{Lam}\ [?x].\ ?t \Downarrow \mathrm{Lam}\ [?x].\ ?t$
e_App:	$[\![?t1.0 \Downarrow \mathrm{Lam}\ [?x].\ ?t;\ ?t2.0 \Downarrow ?v';\ ?t\ [?x ::= ?v'] \Downarrow ?v]\!]$
	$\Longrightarrow \mathrm{App}\ ?t1.0\ ?t2.0 \Downarrow ?v$
conjI:	$?P \Longrightarrow ?Q \Longrightarrow ?P \wedge ?Q$
conjunct1:	$?P \wedge ?Q \Longrightarrow ?P$

Generated Theorems

- Most definitions result in automatically generated theorems; for example

```
thm eval.intros
thm eval.induct
```

Theorem / Lemma / Corollary

- ... they are of the form:

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

- Grey parts are optional.

Theorem / Lemma / Corollary

- . . . they are

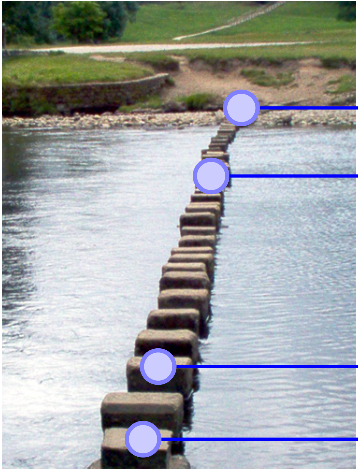
```
lemma alpha_equ:
  shows "Lam [x].Var x = Lam [y].Var y"

lemma Lam_freshness:
  assumes a: "atom y # Lam [x].t"
  shows "(y = x) ∨ (y ≠ x ∧ atom y # t)"

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

- Grey parts are optional.

Isar Proofs

An Isar Proof . . .

An Isar Proof . . .



goal

stepping stones

⋮

stepping stones

assumptions

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 / name

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 / name
- obviously, everything needs to have a justifiation

Justifications

- Omitting proofs

  sorry

- Available facts

  by fact

- Automated proofs

by simp	simplification (equations, ...)
by auto	simplification & proof search
by blast	proof search

  ...

Justifications

- Omitting proofs

  sorry

- Available facts

  by fact

- Automated proofs

  by simp
  by auto
  by blast
  ...

justifications can also be of the form:

using ... by ...

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

Proofs by Induction

- Proofs by induction involve cases, which can be stated as:

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

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

Eval Implies Machine

```
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 Machine

```
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 Machine

```
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 Machine

Proof Idea:

$$
\begin{aligned}
& \langle \mathrm{App}\ t_1\ t_2, [\,] \rangle \\
\mapsto^* \ & \langle t_1, [\mathrm{CAppL}\ \square\ t_2] \rangle \\
\mapsto^* \ & \langle \mathrm{Lam}\ [x].t, [\mathrm{CAppL}\ \square\ t_2] \rangle \\
\mapsto^* \ & \langle t_2, [\mathrm{CAppR}\ (\mathrm{Lam}\ [x].t)\ \square] \rangle \\
\mapsto^* \ & \langle v', [\mathrm{CAppR}\ (\mathrm{Lam}\ [x].t)\ \square] \rangle \\
\mapsto^* \ & \langle t[x::=v'], [\,] \rangle \\
\mapsto^* \ & \langle v, [\,] \rangle
\end{aligned}
$$

thm machine.intros
thm machines.intros
thm eval_to_val

(no assumption avail.)

m [x].t,[]⟩" **sorry**

next

case (e_App t_1 x t t_2 v' v)

(all assumptions)

have a1: "$t_1 \Downarrow \mathrm{Lam}\ [x].t$" **by** fact

have ih1: "$\langle t_1, [\,] \rangle \mapsto^* \langle \mathrm{Lam}\ [x].t, [\,] \rangle$" **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 "$\langle \mathrm{App}\ t_1\ t_2, [\,] \rangle \mapsto^* \langle v, [\,] \rangle$" **sorry**

qed

Eval Implies Machine

```
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 Machine

```
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 Machine

```
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
```

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" .
```

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" ("_ → _")
```

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

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}$$

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

$$\frac{}{\text{valid } []}$$

$$\frac{\text{atom } x \ \# \ \Gamma \quad \text{valid } \Gamma}{\text{valid } (x{:}T) :: \Gamma}$$

Typing Judgements

```
types ty_ctx = "(name×ty) list"

inductive
  valid :: "ty_ctx ⇒ bool"
where
  v1: "valid []"
| v2: "⟦valid Γ; atom 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: "⟦atom x#Γ; (x,T1)#Γ ⊢ t : T2⟧ ⟹ Γ ⊢ Lam [x].t : T1 → T
```

Typing Judgements

```
types ty_ctx = "(name×ty) list"

inductive
  valid :: "ty_ctx ⇒ bool"
where
  v1: "valid []"
| v2: "⟦valid Γ; atom 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: "⟦atom x#Γ; (x,T1)#Γ ⊢ t : T2⟧ ⟹ Γ ⊢ Lam [x].t : T1 → T
```

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

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 "atom x # Lam [x].t"
  and   "atom x # (t1, t2) ⟹ atom x # App t1 t2"
  and   "atom x # Var y ⟹ atom x # y"
  and   "⟦atom x # t1; atom x # t2⟧ ⟹ atom x # (t1, t2)"
  and   "⟦atom x # l1; atom x # l2⟧ ⟹ atom x # (l1 @ l2)"
  and   "atom x # y ⟹ x ≠ y"
  by (simp_all add: lam.fresh fresh_append fresh_at_base)
```

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 "atom x # T"
by (induct T rule: ty.induct)
   (simp_all add: ty.fresh pure_fresh)
```

The Weakening Lemma

```
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
```

Our 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:

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

$$\forall \Gamma\, t_1\, t_2\, T_1\, T_2.$$
$$P\,\Gamma\,t_1\,(T_1 \to T_2) \wedge P\,\Gamma\,t_2\,T_1 \Rightarrow P\,\Gamma\,(t_1\,t_2)\,T_2$$

$$\forall \Gamma\, x\, t\, T_1\, T_2.$$
$$x \# \Gamma \wedge P\,((x : T_1) :: \Gamma)\,t\,T_2 \Rightarrow P\,\Gamma(\lambda x.t)\,(T_1 \to T_2)$$

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

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 $\Gamma_2 \wedge \Gamma_1 \sqsubseteq \Gamma_2 \Rightarrow \Gamma_2 \vdash t : T_2$

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 $\forall \Gamma_2.\, \text{valid}\ \Gamma_2 \wedge \Gamma_1 \sqsubseteq \Gamma_2 \Rightarrow \Gamma_2 \vdash t:T_2$

For all Γ_1, x, t, T_1 and T_2:

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

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

Proof Idea for the Lambda Cs.

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

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

For all Γ_1, x, t, T_1 and T_2:

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

- 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 $\forall \Gamma_2.\ \text{valid}\ \Gamma_2 \wedge \Gamma_1 \sqsubseteq \Gamma_2 \Rightarrow \Gamma_2 \vdash t{:}T_2$

For all Γ_1, x, t, T_1 and T_2:

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

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

- 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: "atom 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

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

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

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

$$\overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}$$

$$\Gamma \vdash t : T \Rightarrow P\,\Gamma\, t\, T$$

- 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: "atom 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: "atom x#Γ₂" by fact
  have ih: "⋀Γ₃. ⟦valid Γ₃; (x,T₁)#Γ₁ ⊑ Γ₃⟧ ⟹ Γ₃ ⊢ t : T₂" by fact
  have a0: "atom 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: "atom 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)
```

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:

$$\frac{\text{prem}_1 \ldots \text{prem}_n \ \text{scs}}{\text{concl}}$$

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 $x \neq y$.

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

- 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 $y \# t'$.

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

- Case 3:
  - We know $y \# \lambda x.t$. We have to show $y \# t'$.
  - The IH says: if $y \# t$ then $y \# t'$.
  - 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 **support** of the conclusion (not free 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: "⟦atom x#Γ; (x,T₁)#Γ ⊢ t : T₂⟧ ⟹ Γ ⊢ Lam [x].t : T₁ → T

equivariance typing
nominal_inductive typing
  avoids t_Lam: "x"
```

Subgoals

1. $\bigwedge x\ \Gamma\ T_1\ t\ T_2.$
$[\![\text{atom } x \# \Gamma; (x, T_1)\cdot\Gamma \vdash t : T_2]\!] \Longrightarrow \{\text{atom } x\} \#^* (\Gamma, \text{Lam } [x]. t, T_1 \to T_2)$
2. $\bigwedge x\ \Gamma\ T_1\ t\ T_2.\ [\![\text{atom } x \# \Gamma; (x, T_1)\cdot\Gamma \vdash t : T_2]\!] \Longrightarrow \text{finite } \{\text{atom } x\}$

in

w

|

| t_Lam: "$[\![\text{atom } x\#\Gamma; (x,T_1)\#\Gamma \vdash t : T_2]\!] \Longrightarrow \Gamma \vdash \text{Lam } [x].t : T_1 \to T$

equivariance typing
nominal_inductive typing
 avoids t_Lam: "x"

Subgoals

1. $\bigwedge x\ \Gamma\ T_1\ t\ T_2.$
$[\![\text{atom } x \ \# \ \Gamma;\ (x, T_1)\cdot\Gamma \vdash t : T_2]\!] \Longrightarrow \{\text{atom } x\} \ \#^* \ (\Gamma, \text{Lam } [x].\ t,\ T_1 \rightarrow T_2)$
2. $\bigwedge x\ \Gamma\ T_1\ t\ T_2.\ [\![\text{atom } x \ \# \ \Gamma;\ (x, T_1)\cdot\Gamma \vdash t : T_2]\!] \Longrightarrow \text{finite } \{\text{atom } x\}$

in

w

| t_Lam: "$[\![\text{atom } x\#\Gamma;\ (x,T_1)\#\Gamma \vdash t : T_2]\!] \Longrightarrow \Gamma \vdash \text{Lam } [x].t : T_1 \rightarrow T$

equivariance typing
nominal_inductive typing
 avoids t_Lam: "x"
 unfolding fresh_star_def
 by (simp_all add: fresh_Pair lam.fresh ty_fresh)

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])"

  "atom 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])"

  "atom 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. $\square$

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
  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 $x \notin \mathsf{fv}(N)$ then

$$(\lambda y.M)[x := N] = \lambda y.(M[x := N])$$

$$
\begin{array}{lll}
 & (\lambda z.M_1)[x := N][y := L] & \\
\equiv & (\lambda z.(M_1[x := N]))[y := L] & \stackrel{1}{\leftarrow} \\
\equiv & \lambda z.(M_1[x := N][y := L]) & \stackrel{2}{\leftarrow} \\
\equiv & \lambda z.(M_1[y := L][x := N[y := L]]) & \mathrm{IH} \\
\equiv & (\lambda z.(M_1[y := L]))[x := N[y := L]] & \stackrel{2}{\rightarrow} \; ! \\
\equiv & (\lambda z.M_1)[y := L][x := N[y := L]]. & \stackrel{1}{\rightarrow}
\end{array}
$$

- **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. $\square$

```
lemma substitution_lemma:
  assumes a: "x ≠ y" "atom 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: "atom 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" "atom 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: "atom 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" "atom 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: "atom 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" "atom 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: "atom 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" "atom 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: "atom 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" "atom 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: "atom 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" "atom 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: "atom 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:
atom x # t ⟹ t [x ::= s] = t
```

```
lemma substitution_lemma:
  assumes a: "x ≠ y" "atom 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: "atom 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" "atom 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; atom x#L⟧ ⟹ M₁[x::=N][y::=L] = M₁[y::=L][x::=N[y::=L]]" by fact
 have "x≠y" by fact
 have "atom x#L" by fact
 have vc: "atom z#x" "atom z#y" "atom z#N" "atom z#L" by fact+
 then have "atom 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" "atom 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; atom x#L⟧ ⟹ M1[x::=N][y::=L] = M1[y::=L][x::=N[y::=L]]" by fact
 have "x≠y" by fact
 have "atom x#L" by fact
 have vc: "atom z#x" "atom z#y" "atom z#N" "atom z#L" by fact+
 then have "atom 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" "atom 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; atom x#L⟧ ⟹ M₁[x::=N][y::=L] = M₁[y::=L][x::=N[y::=L]]" by fact
 have "x≠y" by fact
 have "atom x#L" by fact
 have vc: "atom z#x" "atom z#y" "atom z#N" "atom z#L" by fact+
 then have "atom 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" "atom 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; atom x#L⟧ ⟹ M1[x::=N][y::=L] = M1[y::=L][x::=N[y::=L]]" by fact
 have "x≠y" by fact
 have "atom x#L" by fact
 have vc: "atom z#x" "atom z#y" "atom z#N" "atom z#L" by fact+
 then have "atom 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" "atom 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; atom x#L⟧ ⟹ M₁[x::=N][y::=L] = M₁[y::=L][x::=N[y::=L]]" by fact
 have "x≠y" by fact
 have "atom x#L" by fact
 have vc: "atom z#x" "atom z#y" "atom z#N" "atom z#L" by fact+
 then have "atom 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" "atom 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)
```