Nominal Isabelle[1]

Christian Urban   Stefan Berghofer

with contributions from Julien Narboux

December 22, 2008

[1]This manual is released while being written. The hope is that it is still useful to others.

Contents

Chapter 1

Introduction

Dealing with binders, renaming of bound variables, capture-avoiding substitution, etc., is very often a major problem in formal proofs, especially in proofs by structural and rule induction. *Nominal Isabelle* is designed to make such proofs easy to formalise: it provides an infrastructure for declaring nominal datatypes (that is alpha-equivalence classes) and for defining functions over them by structural recursion. It also provides induction principles that have Barendregt's variable convention already built in. At present Nominal Isabelle is still under heavy development, and also theoretical and implementation issues still remain unsolved. Nevertheless it can and *has* already been used to formalise a number of results about languages involving binders. This includes the standard strong normalisation proof for the simply-typed lambda-calculus, the Tait/Martin-Löf proof and Takahshi/Pollack proof for the Church-Rosser property of beta-reduction, typical proofs from structural operational semantics, a chapter by Crary on logical relations, a paper by Harper and Pfenning on LF and parts of the PoplMark Challenge. Urban also used it to verify the results in his PhD-thesis on strong normalisation of cut-elimination in classical logic.

There is a mailing list for Nominal Isabelle to which users are very welcome to subsribe. Details can be found under "Mailing List" at

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

A great source of information about Isabelle and its old-style proof-scripts is the tutorial by Nipkow et al. [2002]. It can be downloaded in the "Documentation" section at

http://isabelle.in.tum.de/documentation.html

In the same place is a short tutorial about the Isar proof-language developed by Wenzel. This manual assumes that the reader is familiar with Isabelle (more precisely Isabelle/HOL) or learns it by using these notes as a source of examples and by consulting the material referred above for further information.

1.1 Installation

To run Nominal Isabelle you need four packages: PolyML, Isabelle, an emacs and Proof-General. Nominal Isabelle is part of the Isabelle distribution, which you can find at

http://isabelle.in.tum.de

We make occasionally "nominal" releases, which can be downloaded from the "Download" section at

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

To unpack and install Nominal Isabelle and the additional packages (such as Proof-General and PolyML) follow the installation notes of Isabelle. You can find them in the file

```
[ISABELLE_HOME]/INSTALL
```

where `[ISABELLE_HOME]` refers to the directory in which you have unpacked the Isabelle sources. The build process for Nominal Isabelle needs to be started inside this directory with the command:

```
./build -m HOL-Nominal
```

There are no pre-compiled heap files for Nominal Isabelle. After the build completes (this can take a few minutes), the files need to be installed with the command

`./bin/isabelle install -p` *name-of-a-directory*

where the directory is a place in which the system can find executable files. Under Linux this is typically `/usr/local/bin`.

Isabelle together with the Proof-General interface can be started with `isabelle emacs` *name-of-a-theory-file* `&`

You might have to explicitly specify where Isabelle can find your emacs by providing an option `-p` *path-to-emacs*.

This way of starting Isabelle means that the nominal theory will be re-checked when you start your theory. This results in a delay of approximately 30 seconds at the beginning of your interactive session. The delay can be avoided by enabling the option "HOL-Nominal" in the menu "Isabelle → Logics". Unfortunately, one cannot save the state of this option. So if you always want to have HOL-Nominal turned on, then you can either start Isabelle with the command

`isabelle emacs -l HOL-Nominal` *name-of-a-theory-file* `&`

or set the environment variable

```
ISABELLE_LOGIC="HOL-Nominal"
```

in your seetings file for Isabelle (usually under `~/.isabelle/etc`).

Also when Isabelle is first started, check that the X-Symbols are enabled (see "Proof-General → Options") and save any changes (see "Proof-General → Options → Save Options"). The package X-Symbol is not essential for working with Isabelle, but can *greatly* enhance the readability of proof scripts.

Nominal Isabelle contains numerous examples of formalisations for properties about terms with binders. See Chapter 6.

Chapter 2

A Quick Example

This chapter provides a quick overview over Nominal Isabelle by giving the complete proof of the Church-Rosser theorem. The proof we show is due to Takahashi [1995] with some improvements by Pollack [1995].

Informal reasoning about lambda-terms often starts by stating that there is a countably infinite supply of variables. The formal equivalent in Nominal Isabelle for this statement is the declaration

> **atom_decl** name

This declaration introduces a type name whose elements can stand for binders. They are needed when we want to define lambda-terms using the command **nominal_datatype**.

> **nominal_datatype** lam =
>   Var "name"
> | App "lam" "lam"
> | Lam "≪name≫lam"   ("Lam [_]._")

The ≪name≫lam indicates that a name is bound in the term that follows. By convention we introduce the syntax annotation [_]._ for binders. In this way we can write Lam [x].t, instead of Lam x t. Other conventions are possible. Unfortunately, however, the usual syntax λx. t is already used by Isabelle's HOL-logic.

The main point of using Nominal Isabelle is that the definition above results in *alpha-equated* lambda-terms. We can, for example, show the following *equality*:

> **lemma**
> **shows** "Lam [x].Var x = Lam [y].Var y"
> **by** (auto simp add: lam.inject alpha swap_simps fresh_atm)

This means the HOL-logic allows us without any further ado to replace in *any* context a lambda-term by an alpha-equivalent lambda-term. A lot of convenience in Nominal Isabelle arises from this fact.

The most important operation we need for lambda-terms is capture-avoiding substitution. This operation can be defined in Nominal Isabelle as total function using the command **nominal_primrec**:

nominal_primrec
  subst :: "lam ⇒ name ⇒ lam ⇒ lam"   ("_[_::=_]")
where
  "(Var x)[y::=s] = (if x=y then s else (Var x))"
| "(App t_1 t_2)[y::=s] = App (t_1[y::=s]) (t_2[y::=s])"
| "x ♯ (y,s) ⟹ (Lam [x].t)[y::=s] = Lam [x].(t[y::=s])"

By convention we use the notation t[y::=s] for writing substitutions (the more obvious convention t[y:=s] is unfortunately already used in Isabelle for updating maps).

The crucial line in the definition above is the third one, which includes the precondition x ♯ (y, s) standing for the variable x being fresh for the variable y and the lambda-term s. Clearly, only in this case we should be able to move a substitution under the binder, otherwise we defined a function that does not respect alpha-equivalence. It is actually easy to write down functions over lambda-terms that do not respect alpha-equivalence and thus it is easy to prove faulty statements. Nominal Isabelle prevents this by requiring us to prove several properties about the definition of substitution. The corresponding subgoals can be discharged by the following five lines

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

which are typical for many function definitions over lambda-terms.

The notion _ ♯ _ about freshness is central to Nominal Isabelle and automatically defined for nominal datatypes and common datatypes, such as products, lists, natural numbers and so on. A variable being fresh is in most cases equivalent to the variable not being free.

An important lemma we need to prove about substitution is the substitution lemma. To establish it, we need two facts about freshness and substitution. The first states that substitutions can be "forgotten" provided the variable that is being substituted is not free (i.e. is fresh).

lemma forget:
  assumes a: "x ♯ t"
  shows "t[x::=s] = t"
using a
by (nominal_induct t avoiding: x s rule: lam.strong_induct)
    (auto simp add: abs_fresh fresh_atm)

The second is about conditions when a variable is fresh for a substitution: more precisely it states that the variable z will be fresh for z ♯ t[y::=s] provided that z is fresh for s and either z = y or z also fresh for t. This can be proved as follows:

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

With these two facts at our disposal, the substitution lemma can then be easily established.

lemma substitution_lemma:
  assumes a: "x≠y" "x ♯ u"
  shows "t[x::=s][y::=u] = t[y::=u][x::=s[y::=u]]"
using a
by (nominal_induct t avoiding: x y s u rule: lam.strong_induct)
    (auto simp add: fresh_fact forget)

It is a fun exercise to unfold the automation in this proof: one ends up with a formal proof that looks uncannily like the informal proof in Barendregt [1981]. If you do not like to do this exercise yourself, have a look in the file `CR.thy`.

While the previous three lemmas are quite standard and can be found in many informal Church-Rosser proofs, we need two more facts about substitution. First we establish that substitution is *equivariant*. As we will see later on, much of the reasoning in Nominal Isabelle depends on this property. Equivariance for substitution states that a permutation (a bijective renaming of variables) applied on the "outside" of a substitution can be pushed inside to the arguments. Permutations and a permutation applied to a term are concepts provided by Nominal Isabelle. Applying a permutation is written infix as _ • _.

```
lemma subst_eqvt[eqvt]:
  fixes π::"name prm"
  shows "π•(t[x::=s]) = (π•t)[(π•x)::=(π•s)]"
by (nominal_induct t avoiding: x s rule: lam.strong_induct)
   (auto simp add: perm_bij fresh_atm fresh_bij)
```

Since this property is needed frequently "behind the scenes", we make Nominal Isabelle aware of it by giving this lemma the attribute [eqvt].

In two of the later lemmas it will also be necessary to rename a variable being substituted with a fresh variable. This is formalised in the following lemma:

```
lemma subst_rename:
  assumes a: "y ♯ t"
  shows "t[x::=s] = ([(y,x)]•t)[y::=s]"
using a
by (nominal_induct t avoiding: x y s rule: lam.strong_induct)
   (auto simp add: swap_simps fresh_atm abs_fresh)
```

where [(y, x)] • t stands for the *swapping* of y and x in t. A swapping is a permutation consisting of a single pair of variables. You can think of it as the simultaneous renaming of y for x and x for y.

Next we need the definition for beta-reduction. It can be defined as inductive predicate using the Isabelle command **inductive**.

```
inductive
  "Beta" :: "lam ⇒ lam ⇒ bool"   (" _ ⟶β _")
where
  b1[intro]: "t1 ⟶β t2 ⟹ App t1 s ⟶β App t2 s"
| b2[intro]: "s1 ⟶β s2 ⟹ App t s1 ⟶β App t s2"
| b3[intro]: "t1 ⟶β t2 ⟹ Lam [x].t1 ⟶β Lam [x].t2"
| b4[intro]: "App (Lam [x].t) s ⟶β t[x::=s]"
```

We introduce the readable notation $\_ \longrightarrow_\beta \_$ for beta-reduction and we also tag each rule with the attribute [intro] so that the automatic proof-search tools of Isabelle are aware of these rules. The first three rules in this definition are the usual concruence rules and the fourth is the beta-contraction. In Nominal Isabelle it is also possible to define beta-reduction via the notion of a context with a hole (see the examples in `Context.thy` and `CK_Machine.thy`).

The transitive closure of beta-reduction is again an inductive definition.

inductive
  "Beta_star" :: "lam ⇒ lam ⇒ bool" (" _ $\longrightarrow_\beta{}^*$ _")
where
  bs1[intro]: "t $\longrightarrow_\beta{}^*$ t"
| bs2[intro]: "t $\longrightarrow_\beta$ s $\Longrightarrow$ t $\longrightarrow_\beta{}^*$ s"
| bs3[intro]: "$[\![t_1 \longrightarrow_\beta{}^* t_2;\ t_2 \longrightarrow_\beta{}^* t_3]\!] \Longrightarrow t_1 \longrightarrow_\beta{}^* t_3$"

There is an equivalent definition for the transitive closure of $\longrightarrow_\beta$ involving only two rules, but the version above will turn out to be more convenient in some of the subsequent proofs.

The point of the Church-Rosser proof is to show that every two $\longrightarrow_\beta{}^*$-reductions can be joined. No proof in the literature, however, establish this property directly for this reduction relation. Instead an auxiliary reduction relation, written $\longrightarrow_1$, is introduced for which it is easier to establish Church-Rosser. It can also be shown that its transitive closure is equivalent to $\longrightarrow_\beta{}^*$. The main reasoning given below will, therefore, be over $\longrightarrow_1$ defined next.

inductive
  One :: "lam ⇒ lam ⇒ bool" (" _ $\longrightarrow_1$ _")
where
  o1[intro]: "Var x$\longrightarrow_1$ Var x"
| o2[intro]: "$[\![t_1 \longrightarrow_1 t_2;\ s_1 \longrightarrow_1 s_2]\!] \Longrightarrow$ App t_1 s_1 $\longrightarrow_1$ App t_2 s_2"
| o3[intro]: "$t_1 \longrightarrow_1 t_2 \Longrightarrow$ Lam [x].t_1 $\longrightarrow_1$ Lam [x].t_2"
| o4[intro]: "$[\![x \sharp (s_1,s_2);\ t_1 \longrightarrow_1 t_2;\ s_1 \longrightarrow_1 s_2]\!]$
    $\Longrightarrow$ App (Lam [x].t_1) s_1 $\longrightarrow_1$ t_2[x::=s_2]"

This definition is similar to what one can find in the literature, except for the last rule where the freshness condition $x \sharp (s_1, s_2)$ is included as a premise. This premise is needed in Nominal Isabelle in order to derive a stronger induction principle for this definition. This stronger induction principle has the usual variable convention already built in and so will make the reasoning about this definition much more convenient. In order to derive the stronger induction principle, Nominal Isabelle first has know that this definition is equivariant. Since it knows that substitution is equivariant (remember the [eqvt] attribute on the corresponding lemma) and also that freshness is equivariant, equivariance for $\longrightarrow_1$ can be derived automatically by stating:

equivariance One

This gives us the property

If $t \longrightarrow_1 s$ then $\pi \bullet t \longrightarrow_1 \pi \bullet s$.

named One.eqvt. We need it in one lemma below and as mentioned above also need it in the next **nominal_inductive** statement:

```
nominal_inductive One
  by (simp_all add: abs_fresh fresh_fact)
```

This statement derives for us the stronger induction principle for $\longrightarrow_1$. To do so, it checks that the binder in the rules o3 and o4 are not free in the conclusions of those rules. This fact is clear for o3 with the conclusion

$$\text{Lam }[x].t_1 \longrightarrow_1 \text{Lam }[x].t_2$$

but for the conclusion of o4

$$\text{App (Lam }[x].t_1)\ s_1 \longrightarrow_1 t_2[x::=s_2]$$

it only holds if we have the premise $x \sharp (s_1, s_2)$.

If we do not do this test, then deriving the stronger induction principle for an inductive definition can potentially lead to faulty proofs. See the examples in `VC_Condition.thy` where an inconsistency is derived using the variable convention. The unfortunate point with needing this additional premise in rule o4 is that it makes this rule harder to apply; it has no influence on what reductions can be derived, because we can show that this premise is actually superfluous:

```
1   lemma better_o4_intro:
2     assumes a: "t1 --->1 t2" "s1 --->1 s2"
3     shows "App (Lam [x].t1) s1 --->1 t2[x::=s2]"
4   proof -
5     obtain y::"name" where fs: "y ♯ (x,t1,s1,t2,s2)"
6       by (rule exists_fresh, rule fin_supp, blast)
7     have "App (Lam [x].t1) s1 = App (Lam [y].([(y,x)]•t1)) s1" using fs
8       by (auto simp add: lam.inject alpha' fresh_prod fresh_atm)
9     also have "... --->1 ([(y,x)]•t2)[y::=s2]" using fs a
10      by (auto simp add: One.eqvt)
11    also have "... = t2[x::=s2]" using fs by (simp add: subst_rename[symmetric])
12    finally show "App (Lam [x].t1) s1 --->1 t2[x::=s2]" by simp
13  qed
```

The proof follows a quite standard pattern: a fresh name y is chosen in lines 5 and 6. Line 7 alpha-renames the term before the reduction. In line 9 we apply the introduction rule of $\longrightarrow_1$ for the alpha-renamed term (this requires the property One.eqvt for the premise $t_1 \longrightarrow_1 t_2$). Then we rename the substitution back to the original term in line 11.

To make progress in our Church-Rosser proof, we need to establish two properties for the reduction $\longrightarrow_1$: reflexivity and substitutivity.

```
lemma One_refl:
  shows "t ⟶₁ t"
by (nominal_induct t rule: lam.strong_induct) (auto)

lemma One_subst:
  assumes a: "t₁ ⟶₁ t₂" "s₁ ⟶₁ s₂"
  shows "t₁[x::=s₁] ⟶₁ t₂[x::=s₂]"
using a
by (nominal_induct t₁ t₂ avoiding: s₁ s₂ x rule: One.strong_induct)
    (auto simp add: substitution_lemma fresh_atm fresh_fact)
```

In the proof of the substitutivity property it is essential to have the stronger induction principle for $\longrightarrow_1$ at our disposal, because otherwise there is no convenient way to move the substitutions under binders.

We also need later on the following four inversion principles for $\longrightarrow_1$. We derive them as elimination rules using **obtains**.

```
lemma One_Var:
  assumes a: "Var x ⟶₁ t"
  obtains "t = Var x"
using a by (cases rule: One.cases) (simp_all)

lemma One_Lam:
  assumes a: "Lam [x].t ⟶₁ s" "x ♯ s"
  obtains t' where "s = Lam [x].t'" "t ⟶₁ t'"
using a
by (cases rule: One.strong_cases)
    (auto simp add: lam.inject abs_fresh alpha)

lemma One_App:
  assumes a: "App t s ⟶₁ r"
  obtains t' s' where "r = App t' s'" "t ⟶₁ t'" "s ⟶₁ s'"
     | x p p' s' where "r = p'[x::=s']" "t = Lam [x].p" "p ⟶₁ p'"
                        "s ⟶₁ s'" "x ♯ (s,s')"
using a by (cases rule: One.cases)
           (auto simp add: lam.inject)

lemma One_Redex:
  assumes a: "App (Lam [x].t) s ⟶₁ r" "x ♯ (s,r)"
  obtains t' s' where "r = App (Lam [x].t') s'" "t ⟶₁ t'" "s ⟶₁ s'"
       |  t' s' where "r = t'[x::=s']" "t ⟶₁ t'" "s ⟶₁ s'"
using a
by (cases rule: One.strong_cases)
    (auto elim!: One_Lam simp add: lam.inject abs_fresh alpha fresh_prod)
```

The transitive closure of $\longrightarrow_1$ is defined next.

```
inductive
  "One_star" :: "lam ⇒ lam ⇒ bool" (" _ ⟶₁* _")
where
  os1[intro]: "t ⟶₁* t"
| os2[intro]: "t ⟶₁ s ⟹ t ⟶₁* s"
| os3[intro]: "⟦t₁⟶₁* t₂; t₂ ⟶₁* t₃⟧ ⟹ t₁ ⟶₁* t₃"
```

The beautiful trick in the Church-Rosser proof by Takahashi is to introduce a third reduction relation for which a triangle property can be easily established.

```
inductive
  Dev :: "lam ⇒ lam ⇒ bool" (" _ ⟶d _")
where
  d1[intro]: "Var x ⟶d Var x"
| d2[intro]: "t ⟶d s ⟹ Lam [x].t ⟶d Lam[x].s"
| d3[intro]: "⟦¬(∃y t'. t₁ = Lam [y].t'); t₁ ⟶d t₂; s₁ ⟶d s₂⟧
                ⟹ App t₁ s₁ ⟶d App t₂ s₂"
| d4[intro]: "⟦x ♯ (s₁,s₂); t₁ ⟶d t₂; s₁ ⟶d s₂⟧
                ⟹ App (Lam [x].t₁) s₁ ⟶d t₂[x::=s₂]"
```

Note that this reduction relation is defined so that no rule overlaps with another. It will be crucial again to use a stronger induction principle with this definition, so we have to show equivariance and "nominal inductiveness".

```
equivariance Dev

nominal_inductive Dev
  by (simp_all add: abs_fresh fresh_fact)
```

Like with $\longrightarrow_1$ the proof in **nominal_inductive** will only go through if we have the premise $x \sharp (s_1, s_2)$ in rule d4. However, it is otherwise unnecessary and therefore we derive a better introduction rule.

```
lemma better_d4_intro:
  assumes a: "t₁ ⟶d t₂" "s₁ ⟶d s₂"
  shows "App (Lam [x].t₁) s₁ ⟶d t₂[x::=s₂]"
proof -
  obtain y::"name" where fs: "y ♯ (x,t₁,s₁,t₂,s₂)"
    by (rule exists_fresh, rule fin_supp,blast)
  have "App (Lam [x].t₁) s₁ = App (Lam [y].([(y,x)]•t₁)) s₁" using fs
    by (auto simp add: lam.inject alpha' fresh_prod fresh_atm)
  also have "... ⟶d ([(y,x)]•t₂)[y::=s₂]" using fs a
    by (auto simp add: Dev.eqvt)
  also have "... = t₂[x::=s₂]" using fs by (simp add: subst_rename[symmetric])
  finally show "App (Lam [x].t₁) s₁ ⟶d t₂[x::=s₂]" by simp
qed
```

We will need an inversion principle for the reduction $\longrightarrow_d$ and for this it will be advantageous to know that freshness is preserved by the $\longrightarrow_d$ reduction.

```
lemma Dev_preserves_fresh:
  fixes x::"name"
  assumes a: "t⟶d s"
  shows "x ♯ t ⟹ x ♯ s"
using a by (induct) (auto simp add: abs_fresh fresh_fact)
```

The next lemma characterises the term to which a lambda-abstraction d-reduces. Since we are going to invert a rule involving a binder, the inversion only works if the binder is fresh with respect to the judgement to be inverted. If we do not impose this restriction, we obtain only a much weaker statement that is inconvenient to use.

To do the inversion, we know in the proof below that x ♯ Lam [x].t holds (Line 5) and so by the preceding lemma also x ♯ s (Line 6). With these two fact at our disposal, we can infer the form of the reduct.

```
1   lemma Dev_Lam:
2     assumes a: "Lam [x].t ⟶d s"
3     shows "∃s'. s = Lam [x].s' ∧ t ⟶d s'"
4   proof -
5     from a have "x ♯ Lam [x].t" by (simp add: abs_fresh)
6     with a have "x ♯ s" by (simp add: Dev_preserves_fresh)
7     with a show "∃s'. s = Lam [x].s' ∧ t ⟶d s'"
8       by (cases rule: Dev.strong_cases)
9          (auto simp add: lam.inject abs_fresh alpha)
10  qed
```

We can now show that for every lambda-term there exists a d-reduct.

```
lemma Development_existence:
  shows "∃t'. t ⟶d t'"
by (nominal_induct t rule: lam.strong_induct)
   (auto dest!: Dev_Lam intro: better_d4_intro)
```

The reason for introducing the reduction $\longrightarrow_d$ is that we can show a "little" Church-Rosser lemma, namely that a d-reduction and a 1-reduction can always be joined.

```
lemma Triangle:
  assumes a: "t ⟶d t1" "t ⟶1 t2"
  shows "t2 ⟶1 t1"
  using a
```

We prove this lemma by induction on $t \longrightarrow_d t_1$. Any binder that is used in the proof should be fresh for t_2. This requires us to set up the induction as follows:

proof (nominal_induct avoiding: t_2 rule: Dev.strong_induct)

The only interesting case is d4 where we have to be careful to apply the One_Redex inversion instead of One_App. It might be possible to make this proof completely automatic on the expense of folding these two inversion principles into one complicated case analysis. However below we are not doing this and instead analyse the case explicitly:

case (d4 x s_1 s_2 t_1 t_1' t_2)
have fc: "x $\sharp$ t_2" "x $\sharp$ s_1" **by** fact+
have "App (Lam [x].t_1) $s_1 \longrightarrow_1 t_2$" **by** fact

The fc assumptions are needed in order to invert the assumption about how App (Lam [x].t_1) s_1 reduces. The inversion gives us the following two possibilities: either some term inside the redex reduces, or the redex itself.

then obtain t' s' **where** reds:
"(t_2 = App (Lam [x].t') s' $\wedge$ $t_1 \longrightarrow_1$ t' $\wedge$ $s_1 \longrightarrow_1$ s') $\vee$
(t_2 = t'[x::=s'] $\wedge$ $t_1 \longrightarrow_1$ t' $\wedge$ $s_1 \longrightarrow_1$ s')"
using fc **by** (auto elim!: One_Redex)

Having obtained the terms t' and s', we can instantiate the two induction hypotheses in this case.

have ih1: "$t_1 \longrightarrow_1$ t' $\Longrightarrow$ t' $\longrightarrow_1$ t_1'" **by** fact
have ih2: "$s_1 \longrightarrow_1$ s' $\Longrightarrow$ s' $\longrightarrow_1$ s_2" **by** fact

Now we only have to analyse the two cases and in each of them establish that t_2 reduces to t_1'[x::=s_2].

{ **assume** "$t_1 \longrightarrow_1$ t'" "$s_1 \longrightarrow_1$ s'"
then have "App (Lam [x].t') s' $\longrightarrow_1$ t_1'[x::=s_2]"
using ih1 ih2 **by** (auto intro: better_o4_intro)
}
moreover
{ **assume** "$t_1 \longrightarrow_1$ t'" "$s_1 \longrightarrow_1$ s'"
then have "t'[x::=s'] $\longrightarrow_1$ t_1'[x::=s_2]"
using ih1 ih2 **by** (auto intro: One_subst)
}

This allows us to conclude this case.

ultimately show ”$t_2 \longrightarrow_1 t_1$‘[x::=$s_2$]” **using** reds **by** auto
qed (auto elim!: One_App One_Var One_Lam)

The other three cases can be discharged by the automatic proof-search tools.

Using the triangle property and the existence of a development, it is now straightforward to establish the diamond property for $\longrightarrow_1$.

lemma Diamond_for_One:
  assumes a: ”$t \longrightarrow_1 t_1$” ”$t \longrightarrow_1 t_2$”
  shows ”$\exists t_3.\ t_2 \longrightarrow_1 t_3 \wedge t_1 \longrightarrow_1 t_3$”
using a **by** (metis Development_existence Triangle)

We only need to string together the lemma about the existence of a d-reduct and then use the triangle lemma to prove that the reductions are joinable. And from this the following rectangle property can be easily deduced.

lemma Rectangle_for_One:
  assumes a: ”$t \longrightarrow_1{}^* t_1$” ”$t \longrightarrow_1 t_2$”
  shows ”$\exists t_3.\ t_1 \longrightarrow_1 t_3 \wedge t_2 \longrightarrow_1{}^* t_3$”
using a Diamond_for_One **by** (induct arbitrary: t_2) (blast)+

Finally we can show Church-Rosser for $\longrightarrow_1{}^*$.

lemma CR_for_One_star:
  assumes a: ”$t \longrightarrow_1{}^* t_1$” ”$t \longrightarrow_1{}^* t_2$”
    shows ”$\exists t_3.\ t_2 \longrightarrow_1{}^* t_3 \wedge t_1 \longrightarrow_1{}^* t_3$”
using a Rectangle_for_One **by** (induct arbitrary: t_2) (blast)+

It remains to show that the reductions $\longrightarrow_1{}^*$ and $\longrightarrow_\beta{}^*$ are equivalent. This involves tedious proofs that unfortunately cannot be automated much. First we show the following three congruence rules for $\longrightarrow_\beta{}^*$.

lemma Beta_Lam_cong:
  assumes a: ”$t_1 \longrightarrow_\beta{}^* t_2$”
  shows ”Lam [x].$t_1 \longrightarrow_\beta{}^*$ Lam [x].t_2”
using a **by** (induct) (blast)+

lemma Beta_App_cong_aux:
  assumes a: ”$t_1 \longrightarrow_\beta{}^* t_2$”
  shows ”App t_1 s$\longrightarrow_\beta{}^*$ App t_2 s”
    and ”App s $t_1 \longrightarrow_\beta{}^*$ App s t_2”
using a **by** (induct) (blast)+

lemma Beta_App_cong:
  assumes a: ”$t_1 \longrightarrow_\beta{}^* t_2$” ”$s_1 \longrightarrow_\beta{}^* s_2$”
  shows ”App t_1 $s_1 \longrightarrow_\beta{}^*$ App t_2 s_2”
using a **by** (blast intro: Beta_App_cong_aux)

lemmas Beta_congs = Beta_Lam_cong Beta_App_cong

These congruence rules allow us to show that a single 1-reduction can be matched by zero or more beta-reductions.

lemma One_implies_Beta_star:
 assumes a: "$t \longrightarrow_1 s$"
 shows "$t \longrightarrow_\beta{}^* s$"
using a **by** (induct) (auto intro!: Beta_congs)

For the other direction we need to show the following congruence lemma for the reduction $\longrightarrow_1{}^*$.

lemma One_congs:
 assumes a: "$t_1 \longrightarrow_1{}^* t_2$"
 shows "Lam [x].$t_1 \longrightarrow_1{}^*$ Lam [x].t_2"
 and "App t_1 s $\longrightarrow_1{}^*$ App t_2 s"
 and "App s $t_1 \longrightarrow_1{}^*$ App s t_2"
using a **by** (induct) (auto intro: One_refl)

A consequence is now that a single beta-reduction can be matched by zero or more 1-reductions.

lemma Beta_implies_One_star:
 assumes a: "$t_1 \longrightarrow_\beta t_2$"
 shows "$t_1 \longrightarrow_1{}^* t_2$"
using a **by** (induct) (auto intro: One_refl One_congs better_o4_intro)

Stringing both directions together gives us the lemma that the reduction $\longrightarrow_1{}^*$ and $\longrightarrow_\beta{}^*$ are in fact equal.

lemma Beta_star_equals_One_star:
 shows "$t_1 \longrightarrow_1{}^* t_2 = t_1 \longrightarrow_\beta{}^* t_2$"
proof
 assume "$t_1 \longrightarrow_1{}^* t_2$"
 then show "$t_1 \longrightarrow_\beta{}^* t_2$" **by** (induct) (auto intro: One_implies_Beta_star)
next
 assume "$t_1 \longrightarrow_\beta{}^* t_2$"
 then show "$t_1 \longrightarrow_1{}^* t_2$" **by** (induct) (auto intro: Beta_implies_One_star)
qed

Since we have already established that $\longrightarrow_1{}^*$ is Church-Rosser, the reduction $\longrightarrow_\beta{}^*$ must be too.

theorem CR_for_Beta_star:
assumes a: "$t \longrightarrow_\beta{}^* t_1$" "$t \longrightarrow_\beta{}^* t_2$"
shows "$\exists t_3.\ t_1 \longrightarrow_\beta{}^* t_3 \wedge t_2 \longrightarrow_\beta{}^* t_3$"
proof -
from a **have** "$t \longrightarrow_1{}^* t_1$" **and** "$t \longrightarrow_1{}^* t_2$" **by** (simp_all add: Beta_star_equals_One_star)
then have "$\exists t_3.\ t_1 \longrightarrow_1{}^* t_3 \wedge t_2 \longrightarrow_1{}^* t_3$" **by** (simp add: CR_for_One_star)
then show "$\exists t_3.\ t_1 \longrightarrow_\beta{}^* t_3 \wedge t_2 \longrightarrow_\beta{}^* t_3$" **by** (simp add: Beta_star_equals_One_star)
qed

This theorem completes the Church-Rosser proof.

Chapter 3

Nominal Reasoning Infrastructure

3.1 Preliminaries

Nominal Isabelle can be included in a theory using the usual Isabelle convention:

> **theory** Foo
> **imports** Nominal
> **begin**
> . . . *declarations, definitions and proofs* . . .
> **end**

The most important constants introduced in Nominal Isabelle are:

	X-Symbol	ASCII
permutation operation	$\_ \bullet \_$	`\< bullet >`
freshness	$\_ \sharp \_$	`\< sharp >`
support	supp $\_$	
abstractions	$[\_].\_$	

Note that the symbol for freshness is different from Isabelle's symbol # used for list-cons.

These constants are all polymorphic and therefore often need explicit type-annotations to be meaningful (see Appendix A.3). Lack of explicit type-annotations is usually what causes rules to not being applicable or to other non-intuitive behaviour. The meaning of these constants is described in more detail in the remaining sections of this chapter.

3.2 Atom Declarations

Before any nominal datatype can be defined, Nominal Isabelle needs to know which atom types will be used. Atom types stand for different kinds of (object) variables that might be bound in nominal datatypes.

Atom types can be declared using the command **atom_decl** with the syntax

> **atom_decl** $\text{type}_1 \ldots \text{type}_n$

where $\text{type}_1 \ldots \text{type}_n$ stand for some identifiers. A concrete example is

> **atom_decl** var tyvar

which might be used to formalise variables in lambda-terms and type-variables in simple types. After this declaration one can write

> **lemma** foo:
> **fixes** x::"var"
> **and** T::"tyvar"
> . . .

to fix the types for the (Isabelle) variables x and T. Note that the atom declaration is a "global" declaration, in the sense that it can be issued only once. Therefore it needs to contain *all* atom types that are ever going to be used in a formalisation.

Each atom type contains countable infinitely many elements. They can be constructed by using the atom-type name as constructor and a natural number as argument. For example one can show:

> **lemma** concrete_atoms:
> **shows** "var 0 $\neq$ var 1"
> **by** simp

However concrete atoms, such as var 0, are only rarely used in Nominal Isabelle. One example where they are used is to define the sets of "even" and "odd" atoms in the example file Support.thy.

When the atom declaration is issued, the Nominal Isabelle generates a number of lemmas that have been specialised according to the given atom types. One example is the lemma

> **lemma** exists_fresh′:
> **assumes** "finite ((supp x)::var set)"
> **shows** "$\exists$ a::var. a $\sharp$ x"
> . . .

which states that for every finitely supported x there exists an atom a of type var which is fresh for x (support and freshness will be explained in Section 3.4). If more than one atom type is declared, then the lemma exists_fresh′ will contain such a statement for every atom type. For example, in case we declare "**atom_decl** var", then exists_fresh′ will be the lemma

finite ((supp x)::var set) $\Longrightarrow$ $\exists$ a::var. a $\sharp$ x

while in case of "**atom_decl** var tyvar", exists_fresh′ will be the collection of lemmas

finite ((supp x)::var set) $\Longrightarrow$ $\exists$ a::var. a $\sharp$ x
finite ((supp x)::tyvar set) $\Longrightarrow$ $\exists$ a::tyvar. a $\sharp$ x

This kind of specialisation to concrete atom-types is provided automatically for lemmas that are frequently needed. For less frequently used lemmas only general versions are provided in the file Nominal.thy. Such general lemmas need to be specialised manually. For example the lemma

at_exists_fresh′[OF at_var_inst]

is equivalent to

finite ((supp x)::var set) $\Longrightarrow$ $\exists$ a::var. a $\sharp$ x

See also Section 3.6 for more details on the instantiation of generic lemmas.

3.3 Permutations

Permutations are bijective mappings from atoms to atoms. In Nominal Isabelle such permutations are represented as lists of atom-pairs. For this the type-abbreviation

‘x prm

standing for (‘x $\times$ ‘x) list has been introduced. The type variable ‘x can be instantiated with any declared atom type. For example one can write:

lemma another_foo:
 fixes π_1::"var prm"
 and π_2::"tyvar prm"
 ...

Since permutations are lists, they can be written using the standard notation for lists in Isabelle/HOL. Three examples permutations are

$$[(a, b)] \qquad [(a, a), (b, c)] \qquad []$$

where the first stands for the permutation that swaps the atoms a and b; the second swaps first b and c, and then a and a; the last stands for the identity permutation.

Two permutations can be composed by using list-append, written π_1 @ π_2. The inverse of a permutation is given by reversing the list. For this the standard reversal function, written (rev π), of Isabelle is used. Note however that (rev [(a,b)]) is *not* [(b,a)], but is equal to [(a,b)]:

lemma swap_rev:
 fixes a b::"var"
 shows "rev [(a,b)] = [(a,b)]"
 by simp

The infix operation _•_ that applies a permutation to an object has the polymorphic type

$$\text{`x prm} \Rightarrow \text{`a} \Rightarrow \text{`a}.$$

One can write for example

lemma swap_id:
 fixes a::"tyvar"
 shows "[(a,a)] • x = x"
 ...

where the permutation [(a, a)] is applied to the object x, or

lemma perm_bij:
 fixes π::"var prm"
 shows "(rev π) • (π • x) = x"
 ...

where π is applied to x, followed by an application of the inverse of π.

For several standard types, Nominal Isabelle defines automatically permutation operations. Some examples are:

pairs:	$\pi \bullet (x, y) = (\pi \bullet x, \pi \bullet y)$
sets:	$\pi \bullet X = \{\pi \bullet x \mid x.\ x \in X\}$
lists:	$\pi \bullet [] = []$
	$\pi \bullet (x \# xs) = \pi \bullet x \# \pi \bullet xs$
functions:	$\pi \bullet f = (\lambda x.\ \pi \bullet f\ (\text{rev}\ \pi \bullet x))$
natural numbers:	$\pi \bullet n = n$
booleans:	$\pi \bullet b = b$

Permutation operations on atoms are defined such that

$$
\begin{array}{rcl}
[\,] \bullet c & = & c \\
[(a,b)]\#\pi \bullet c & = & \begin{cases} b & \text{if } \pi \bullet c = a \\ a & \text{if } \pi \bullet c = b \\ \pi \bullet c & \text{otherwise} \end{cases}
\end{array}
$$

where the permutations has the same type as the atom c. In case the permutation is of different type than the atom they are acting on, then the permutation operation is defined as $\pi \bullet a = a$. For example one can show that

lemma perm_ineq:
 fixes a::"var"
 shows "$\forall\,\pi$::tyvar prm. $\pi \bullet a = a$"
 by (simp add: calc_atm)

where the lemma calc_atm contains all lemmas that are needed to analyse a permutation applied to an atom.

While the representation of permutations as lists is very convenient for composing two permutations and for building the inverse of a permutation, it is not a unique representation. Nominal Isabelle therefore defines the relation $\triangleq$ that states when two permutations (necessarily having the same type) are equal, namely:

constdefs
 prm_eq :: "'x prm $\Rightarrow$ 'x prm $\Rightarrow$ bool" (" _ $\triangleq$ _ " [80,80] 80)
 "$\pi_1 \triangleq \pi_2 \equiv \forall\,$a::'x. $\pi_1 \bullet a = \pi_2 \bullet a$"

The tactic perm_simp can be used to analyse many instances[1] of permutation operations applied to an arbitrary object. Its syntax is similar to the standard simp-tactic. For example one can add lemmas to, or delete from, the simplifier by issuing

(perm_simp add: ... del: ...)

Three examples of equations which perm_simp solves are:

lemma bar:
 fixes a b::"var"
 and l::"var list"
 and π::"var prm"
 shows "[(a,b)]•a = b" **and** "[(a,b)]•[(a,b)]•l = l" **and** "π•(rev π)•l = l"
 by perm_simp+

[1]Unfortunately it is not known whether a suitable equational theory involving permutations applied to arbitrary objects is decidable.

3.4 Support

The most interesting feature of the nominal work is the notion of *support* (see Pitts [2003]). This notion corresponds roughly to the usual notion of what the set of free variables of an object is, however it is more general and this generality is crucial in a number of places in Nominal Isabelle. The definition of support is

constdefs
  supp :: "'a ⇒ ('x set)"
  "supp x ≡ {a . (infinite {b . [(a,b)]•x ≠ x})}"

where the right-hand side stands for a set of atoms (in this case of type 'x). Note that this definition is different from some other definitions one can find in the nominal literature.

For finitary structures, such as pairs, lists, lambda-terms, etc, the notion of support coincides with the usual notion of free variables. For example, assuming that the atom types var and tyvar are declared, then the support of a var-atom is

lemma supp_atm:
  fixes a::"var"
  shows "(supp a) = ({a}::var set)"
  and "(supp a) = ({}::tyvar set)"
  ...

Note that the in the first statement the support of "a" is calculated with respect to the atom type var, while in the second with respect to tyvar. Because of the typing constraints for the constant supp, it is impossible in Nominal Isabelle to express in a single set what the support of an object is with respect to *all* atom types (the reason is that the set ({a}::var set) ∪ ({}::tyvar set) is ill-typed).

For pairs one can easily show that their support is equal to the support of their components (see Figure 3.1 for the calculation in Isar). The support of an empty list and list-cons is

supp [] = {}
supp (x # xs) = supp x ∪ supp xs

Natural numbers and booleans have empty support:

supp n = {}
supp b = {}

```
11 lemma supp_prod:
12   shows "supp (x1,x2) = (supp x1) ∪ (supp x2)"
13 proof -
14   have "supp (x1,x2) = {a. infinite {b. [(a,b)]•(x1,x2) ≠ (x1,x2)}}" by (simp only: supp_def)
15   also have "... = {a. infinite {b. ([(a,b)]•x1,[(a,b)]•x2) ≠ (x1,x2)}}" by simp
16   also have "... = {a. infinite {b. [(a,b)]•x1 ≠ x1 ∨ [(a,b)]•x2 ≠ x2}}" by simp
17   also have "... = {a. infinite ({b. [(a,b)]•x1 ≠ x1} ∪ {b. [(a,b)]•x2 ≠ x2})}"
18     by (simp only: Collect_disj_eq)
19   also have "... = {a. infinite {b. [(a,b)]•x1 ≠ x1} ∨ infinite {b. [(a,b)]•x2 ≠ x2}}" by simp
20   also have "... = {a. infinite {b. [(a,b)]•x1 ≠ x1}} ∪ {a. infinite {b. [(a,b)]•x2 ≠ x2}}"
21     by (simp only: Collect_disj_eq)
22   also have "... = (supp x1) ∪ (supp x2)" by (simp only: supp_def)
23   finally show "supp (x1,x2) = (supp x1) ∪ (supp x2)" .
24 qed
```

Figure 3.1: A proof for calculating the support of a pair: Line 6 uses the property when a pair is not equal, Line 9 uses the property when a union of two sets is infinite; Lines 7–8 and 10–11 transform a disjunction inside a collection into a union of two sets.

The support of an abstraction, a construct that will be described in more detail in Sec. 3.7, is

```
supp ([a].x) = supp x - supp a
```

However this equation has the proviso that x must have finite support. The support of a finite set, say X, is the union of the support of every element in X, that is

```
lemma supp_fin_set:
  assumes "finite X"
  shows   "(supp X) = (⋃x∈X. (supp x))"
  ...
```

More complicated is the situation for infinitary structures, such as infinite sets and functions. For example it can be easily shown that the set of all atoms of type var has empty support (the set of all atoms of type var is in Isabelle represented by UNIV::var set). We first show that swapping two atoms in this set leaves the set unchanged:

```
lemma swap_UNIV_var:
  fixes a b::"var"
  shows "[(a,b)]•(UNIV::var set) = UNIV"
  by (auto simp add: perm_set_eq calc_atm)
```

Now it is straightforward to conclude that UNIV::var set has empty support.

```
lemma supp_UNIV_var:
  shows "supp (UNIV::var set) = ({}::var set)"
  by (simp add: supp_def swap_UNIV_var)
```

However, there are also objects in the HOL-logic that have infinite support. In the example file Support.thy it is shown that the set of "even", respectively "odd", atoms have infinite support. Pitts [2006] gives an example for a function that has infinite support (see Example 3.4 in Pitts [2006]).

3.5 Freshness

Derived from the concept of support is the notion of *freshness* (again for more background information see Pitts [2003]). Freshness is defined as

```
constdefs
  fresh :: "'x ⇒ 'a ⇒ bool" ("_ ♯ _" [80,80] 80)
  "a ♯ x ≡ a ∉ supp x"
```

Recall that there are countable infinitely many elements in an atom type. Consequently if a given object x has finite support, then there must exist an atom outside the support of x; that is there must be an atom that is fresh for x. This fact is proved in the lemma exists_fresh′

```
lemma exists_fresh':
  assumes "finite ((supp x)::var set)"
  shows   "∃c::var. c ♯ x"
  ...
```

Note that this lemma contains such facts for all declared atom types.

For two atoms of the same type the notion of freshness coincides with inequality:

```
lemma fresh_atm:
  fixes a b::"var"
  shows "(a ♯ b) = (a ≠ b)"
  ...
```

Unwinding the definition of support for other types gives the following facts:

pairs:	$a \sharp (x, y) = (a \sharp x \wedge a \sharp y)$
sets:	$a \sharp \{\}$
	$[\![\text{pt TYPE('a) TYPE('b); at TYPE('b)}]\!] \Longrightarrow a \sharp \{x\} = a \sharp x$
lists:	$a \sharp []$
	$a \sharp (x \# xs) = (a \sharp x \wedge a \sharp xs)$
natural numbers:	$a \sharp n$
booleans:	$a \sharp b$

The crucial property of freshness is that if two atoms are fresh for an object, then swapping those atoms leaves x unchanged, that is

lemma perm_fresh_fresh:
fixes a b::"var"
assumes "a $\sharp$ x" **and** "b $\sharp$ x"
shows "[(a,b)]•x = x"
...

In order to keep formulae in goals manageable, one often wants to switch between a tupled version of freshness and an equivalent un-tupled series of freshness-conjunctions. For this the built-in simplifier has been set up so that it can solve automatically goals such as:

lemma fresh_tuples:
fixes a::"var"
shows "a $\sharp$ $(t_1,t_2) \Longrightarrow$ a $\sharp$ $t_1 \wedge$ a $\sharp$ t_2"
and "$[\![$a $\sharp$ t_1; a $\sharp$ $t_2]\!] \Longrightarrow$ a $\sharp$ (t_1,t_2)"
and "a $\sharp$ $(t_1,t_2) \Longrightarrow$ a $\sharp$ t_2"
and "a $\sharp$ $(t_1,t_2) \Longrightarrow$ a $\sharp$ (t_2,t_1,t_1,t_2)"
by simp_all

Although the reasoning infrastructure in Nominal Isabelle is designed to minimise the need of generating fresh atoms, sometimes there is no way around generating one. To facilitate this, the tactic generate_fresh has been introduced. For example a fresh atom with type var is generated by applying the tactic

apply(generate_fresh "var")

This tactic will scan for all variables in the current goal and then introduce a new assumption with an atom being fresh for all the variables for which the tactic can determine that they have finite support.

3.6 Permutation Types and Finitely Supported Types

The file Nominal.thy is a library containing general facts about support and freshness. These facts depend on the permutation operation to behave properly on every type it acts on. In order to make precise what behaving properly means, the following predicate is defined in Nominal.thy:

$$\begin{aligned}\text{pt TYPE('a) TYPE('x)} \equiv\ & \forall x.\ [\,] \bullet x = x \\ & \wedge \forall \pi_1\ \pi_2\ x.\ (\pi_1\ @\ \pi_2) \bullet x = \pi_1 \bullet \pi_2 \bullet x \\ & \wedge \forall \pi_1\ \pi_2\ x.\ \ \pi_1 \triangleq \pi_2 \ \longrightarrow \pi_1 \bullet x = \pi_2 \bullet x\end{aligned} \tag{3.1}$$

This predicate takes two *types* as arguments (indicated by the keyword TYPE[2]): the first type is an arbitrary type over which the permutaion acts and the second is an atom-type over which permutations are built. These arguments impose the following type-constraints on the right-hand side: in the first clause x has type ‘a and the empty list is of type ‘x prm; in the second and third clause x has type ‘a, and π_1 and π_2 have type ‘x prm. The type argement of pt are necessary because HOL does not allow any free type-variables on the right-hand sides of definitions.

The idea of the predicate pt is that a type ‘a is a permutation type with respect to the atom-type ‘x provided the identity permutation (empty list) leaves all elements of ‘x unchanged, that a composition of two permutations can be “un-composed” and that the application of two equal permutations has to produce the same result (see Page 23 for the definition of permutation equality). These three properties are sufficient to infer most facts about support and freshness.

The predicate pt is used, for example, in the proof of the fact about all swappings leaving an object unchanged implies that also all permutations leave this object unchanged.

> **lemma** pt_swap_eq_aux:
> **fixes** y :: ”‘a”
> **assumes** ”pt TYPE(‘a) TYPE(‘x)”
> **and** ”$\forall$ (a::‘x) (b::‘x). [(a,b)]•y = y”
> **shows** ”$\forall\,\pi$::‘x prm. π•y = y”
> ...

Nominal.thy proves a number of lemmas which show that pt holds “hereditarily” through the type-structure. For example

> **lemma**
> **shows** pt_unit_inst: ”pt TYPE(unit) TYPE(‘x)”
> **and** pt_bool_inst: ”pt TYPE(bool) TYPE(‘x)”
> **and** pt_list_inst: ”pt TYPE(‘a) TYPE(‘x) $\Longrightarrow$ pt TYPE(‘a list) TYPE(‘x)”
> **and** pt_prod_inst: ”$[\![$pt TYPE(‘a) TYPE(‘x); pt TYPE(‘b) TYPE(‘x)$]\!]$
> $\Longrightarrow$ pt TYPE(‘a $\times$ ‘b) TYPE(‘x)”
> ...

Such lemmas allow one to lift the above lemma pt_swap_eq_aux to apply to ‘a lists, for example. Resolving pt_swap_eq_aux with pt_list_inst by writing

> pt_swap_eq_aux[OF pt_list_inst]

gives the lemma

[2] See Section 2.3.2 in Wenzel [2007] for more deatils about this construction.

lemma pt_swap_eq_aux_for_lists:
fixes y :: "'a list"
assumes "pt TYPE('a) TYPE('x)"
and "$\forall$ (a::'x) (b::'x). [(a,b)]•y = y"
shows "$\forall$ π::'x prm. π•y = y"
...

where the difference is that y has type 'a list. Such lifting of lemmas is automated in Nominal Isabelle employing axiomatic type classes (which are described in Wenzel [2004]). This mechanism is explained next.

The atom declaration generates for each atom type two axiomatic type-classes, which state the conditions for permutation types and finitely supported types. These type-classes are named pt_type$_i$ and fs_type$_i$ where type$_i$ ranges over the atom types introduced by **atom_decl**. Nominal Isabelle derives a number of instance proofs so that Isabelle's type-system can in most circumstances automatically infer when a type is a permutation type and/or a finitely supported type. Nominal datatypes are always permutation types and their elements are always finitely supported, see Chapter 4. Sets and functions are permutation types, but *not* finitely supported types (there are some infinite sets and some functions that do not have finite support).

3.7 Abstractions and Alpha-Equivalence

As we shall see in Chapter 4 nominal datatypes represent alpha-equivalence classes.

Chapter 4

Nominal Datatypes

4.1 Declaration

In contrast to standard datatypes of Isabelle/HOL, nominal datatypes represent alpha-equivalence classes. They can be declared using the **nominal_datatype** keyword. Its syntax is similar to the standard datatype declaration, except it allows one to specify where binders occur using ≪_≫_. For example alpha-equated lambda-terms can be specified as:

> **nominal_datatype** lam =
> Var "var"
> | App "lam" "lam"
> | Lam "≪var≫lam"

assuming that var has been declared as an atom type. The constructor Lam is then treated as a constructor with two arguments: a var and a lam. This means one can write Lam a t. However, it is often more convenient to introduce special syntax annotations for constructors with binders. One example of such an annotation is

> **nominal_datatype** lam =
> Var "var"
> | App "lam" "lam"
> | Lam "≪var≫lam" ("Lam [_]._" [100,100] 100)

which allows one to write, instead of Lam a t, the more memorable Lam [a].t for lambda-abstractions. (Note that λ is already in use by Isabelle and cannot be used in special syntax.) However this kind of direct syntax annotation does not work in situations when one wants to change the order of arguments. For example the lambda-calculus with let can be specified as

```
nominal_datatype lam =
  Var "var"
| App "lam" "lam"
| Lam "«var»lam" ("Lam [_]._" [100,100] 100)
| Let "«var»lam" "lam"
```

To obtain the usual notation for Lets, one can use the following abbreviation:

```
abbreviation
  LetBe :: "var⇒lam⇒lam⇒lam" ("Let _ be _ in _" [100,100,100] 100)
where
 "Let x be t1 in t2 ≡ lam.Let x t2 t1"
```

Nested binders can be specified as shown in the following nominal datatype where the first argument in ImpR abstracts over a name and a coname.

```
nominal_datatype trm =
  Ax   "name" "coname"
| Cut  "«coname»trm" "«name»trm"
| ImpR "«name»«coname»trm" "coname"
| ImpL "«coname»trm" "«name»trm" "name"
```

Currently the following restrictions are in place for nominal datatypes:

- inside «_» only one atom type is allowed
- no type variables (that is 'a and so on) are allowed in nominal datatype definitions
- no nested types are allowed; they need to be explicitly defined as mutual recursive datatype, and
- no function types can be used (the reason is that it is generally too difficult to ensure automatically that functions have finite support).

The first and third restriction seem in practice the most inconvenient ones and the hope is that they can be lifted in future versions of Nominal Isabelle. This is desirable, because at the moment type-schemes from Hindley-Milner typing algorithm, for example, need to be specified as

```
nominal_datatype ty =
  TVar "var"
| Fun "ty" "ty" ("_ → _" [100,100] 100)
nominal_datatype tyS =
  Ty "ty"
| All "«var»tyS" ("∀ [_]._" [100,100] 100)
```

instead of abstracting a list or set of names. Also the third restriction is annoying: while with standard datatypes one can declare

datatype trm =
  Var ”var”
| Fun ”trm list”

this is not yet possible with nominal datatypes. Instead, the trm-lists need to be explicitly unfolded like

nominal_datatype trm =
  Var ”var”
| Fun ”trm_list”
and trm_list =
  MyNil
| MyCons ”trm” ”trm_list”

This is quite awkward and can at the moment only be circumvented in cases the nominal datatype has no binders. In such cases one can declare the datatype using the normal **datatype**-mechanism and manually provide a permutation operation for this datatype (see Section ?? for further details).

For nominal datatypes a number of lemmas are automatically generated. The most important are named

nominal-datatype-name.perm

nominal-datatype-name.supp

nominal-datatype-name.fresh

nominal-datatype-name.inject

In case of the lambda-calculus they are defined as follows:

lemma lam.perm:
 fixes π::”var prm”
 shows ”$\pi \bullet$ Var x = Var ($\pi \bullet$ x)”
 and ”$\pi \bullet$ App t_1 t_2 = App ($\pi \bullet t_1$) ($\pi \bullet t_2$)”
 and ”$\pi \bullet$ Lam [x].t = Lam [($\pi \bullet$ x)].($\pi \bullet$ t)”

lemma lam.supp:
 shows ”supp (Var x) = (supp x)”
 and ”supp (App t_1 t_2) = (supp t_1) $\cup$ (supp t_2)”
 and ”supp (Lam [x].t) = (supp ([x].t))”

lemma lam.fresh:
shows "a ♯ (Var x) = a ♯ x"
and "a ♯ (App t_1 t_2) = (a ♯ t_1 ∧ a ♯ t_2)"
and "a ♯ (Lam [x].t) = (a ♯ [x].t)"

lemma lam.inject:
shows "(Var x = Var y) = (x = y)"
and "(App t_1 t_2 = App s_1 s_2) = (t_1 = s_1 ∧ t_2 = s_2)"
and "(Lam [x].t_1 = Lam [y].t_2) = ([x].t_1 = [y].t_2)"

While the first three lemmas are added to the simplifier, please note that for nominal datatypes the lemma *nominal-datatype-name*.inject is *not* automatically added to the simplifier (unlike for standard datatypes).

4.2 Strong Structural Inductions

Nominal Isabelle provides for every nominal datatype a "weak" induction principle, whose name is *nominal-datatype-name*.induct. In case of the lambda-calculus it is as follows:

$$\frac{\begin{array}{l}\bigwedge x.\ P\ (\mathrm{Var}\ x)\\ \bigwedge t_1\ t_2.\ [\![P\ t_1; P\ t_2]\!] \Longrightarrow P\ (\mathrm{App}\ t_1\ t_2)\\ \bigwedge x\ t.\ P\ t \Longrightarrow P\ (\mathrm{Lam}\ [x].t)\end{array}}{P\ t} \tag{4.1}$$

This induction principle can be used to show, for example, that swapping the atom a with itself behaves like an identity on lambda terms, see Figure 4.1 for a detailed Isar-proof of this fact.

This induction principle is "weak", because one has to prove the lambda-case for *all* binders x (note the meta-quantification $\bigwedge x$ in (4.1)). While this is no problem in the proof shown in Figure 4.1, in more complicated circumstances it means that one has to find a suitable renaming to get the Lam-case through. In order to reduce the amount of effort in such circumstances, strong induction principles are automatically generated for nominal datatypes. In case of the lambda-calculus the strong induction principle

```
lemma swap_id:
 fixes a::"var" and t::"lam"
 shows "[(a,a)] • t = t"
proof (induct t rule: lam.induct)
 case (Var x)
 show "[(a,a)] • Var x = Var x" by (simp add: calc_atm)
next
 case (App t1 t2)
 have ih1: "[(a,a)] • t1 = t1" and  ih2: "[(a,a)] • t2 = t2" by fact+
 then show "[(a,a)] • App t1 t2 = App t1 t2" by simp
next
 case (Lam x t)
 have ih: "[(a,a)] • t = t" by fact
 then show "[(a,a)] • Lam [x].t = Lam [x].t" by (simp add: calc_atm)
qed
```

Figure 4.1: A simple proof illustrating the use of the "weak" induction principle lam.induct: the proof shows that swapping an atom a with itself leaves lambda-terms unchanged.

(called lam.induct) is as follows:

$$\frac{\begin{array}{l}\bigwedge x\ c.\ P\ c\ (\mathrm{Var}\ x)\\ \bigwedge t_1\ t_2\ c.\ [\![\bigwedge c'.\ P\ c'\ t_1;\ \bigwedge c'.\ P\ c'\ t_2]\!] \Longrightarrow P\ c\ (\mathrm{App}\ t_1\ t_2)\\ \bigwedge x\ t\ c.\ [\![x \sharp c;\ \bigwedge c'.\ P\ c'\ t]\!] \Longrightarrow P\ c\ (\mathrm{Lam}\ [x].t)\end{array}}{P\ c\ t} \tag{4.2}$$

where c stands for the *freshness context* of the induction. The precondition $x \sharp c$ in the third case implies that one has to prove the lambda-case for only those binders x that are fresh with respect to this context. This usually makes the binder cases as simple as the informal reasoning where the variable convention is employed.

Before we give an example, we shall describe how strong structural inductions can be used (the usual method induct does not work with them). To apply a strong structural induction principle one has to use the method nominal_induct, which applies a strong induction rule and helps with instantiating the freshness context. Its syntax is

```
(nominal_induct x1...xn
           arbitrary: y1...ym
            avoiding: z1...zk
                rule: name-of-the-strong-induction-rule)
```

where arbitrary and avoiding are optional. The meaning of the variables are:

- the x_i stand for the variables over which the induction is done (in structural inductions this is only a single variable),
- the y_j stand for the variables that are generalised in the induction, and
- the z_l stand for the variables in the freshness context of the nominal induction—this context describes what is usually meant be the variable convention.

For example when proving the usual substitution lemma[1] in the lambda-calculus

```
lemma substitution_lemma:
  assumes a: ”x ≠ y” and b: ”x ♯ L”
  shows ”M[x::=N][y::=L] = M[y::=L][x::=N[y::=L]]”
```

the induction proceeds over the structure of M (no generalisation is needed for the induction to go through). If one follows the proof of this lemma given in Barendregt [1981], then in the lambda-case it is assumed that the binder, say z, has to satisfy the following constraints: z ≠ x, z ≠ y, z ♯ N and z ♯ L. These assumptions are usually justified by the *variable convention*. To formalise Barendregt's proof, one can set up the induction using:

```
(nominal_induct M avoiding: x y N L rule: lam.induct)
```

In the lambda-case one can then assume the constraints from the variable convention. This means we can formalise the proof of the substitution lemma roughly as follows:

```
lemma substitution_lemma:
  assumes a: ”x ≠ y” and b: ”x ♯ L”
  shows ”M[x::=N][y::=L] = M[y::=L][x::=N[y::=L]]”
using a b
proof (nominal_induct M avoiding: x y N L rule: lam.strong_induct)
  case (Var z) . . .
next
  case (App M1 M2) . . .
next
  case (Lam z M1)
  have vc: ”z ♯ x” ”z ♯ y” ”z ♯ N” ”z ♯ L” by fact
   . . .
qed
```

[1]The substitution function for the nominal datatype lam will be defined in Section 4.4.

where Line 11 contains the freshness assumptions about the binder z. With those assumptions the proof of the substitution lemma is quite simple and does not need any renaming of binders—just like the proof of Barendregt.

4.3 Equivariance Lemmas

An important concept in the nominal logic work is *equivariance* [Pitts, 2003]. In Nominal Isabelle, definition of functions and inductive definition (described in the next sections) rely on this concept. Therefore a special attribute, called [eqvt], has been introduced for indicating when a lemma establishes the equivariance of a function or a relation. A function f is equivariant provided

$$\pi \bullet (\mathrm{f}\ \mathrm{x}_1 \ldots \mathrm{x}_n) = \mathrm{f}\ (\pi \bullet \mathrm{x}_1) \ldots (\pi \bullet \mathrm{x}_n)$$

holds where the x_i stand for all the arguments of f. A relation R is equivariant provided

$$\text{if } \mathrm{R}\ \mathrm{x}_1 \ldots \mathrm{x}_n \text{ then also } \mathrm{R}\ (\pi \bullet \mathrm{x}_1) \ldots (\pi \bullet \mathrm{x}_n)$$

holds. Again the x_i stand for all the arguments of R. In order to make such lemmas known to the system, they can be tagged with the attribute [eqvt]. Two examples are

```
lemma substitution_is_equivariant[eqvt]:
  fixes π::"var prm"
  shows "π•(t₁[x::=t₂]) = (π•t₁)[(π•x)::=(π•t₂)]"
  ...
lemma equality_over_lam_is_equivariant[eqvt]:
  fixes π::"var prm"
  and   s t::"lam"
  assumes "s = t"
  shows "(π•s) = (π•t)"
  ...
```

When the [eqvt]-attribute is attached to a lemma, the system checks that certain conditions are satisfied. In case of functions, where the equivariance lemma must be of the form $lhs = rhs$, the conditions state that the left-hand side must be of the form $\pi \bullet expr$ and the right-hand side must be equal to $expr$ except that every free variable, say x, in it is replaced by $\pi \bullet x$ (see lemma substitution_is_equivariant above). In case of relations the conclusion must be equal to the premise, except again that the permutation π acts on every free variable in the conclusion. When the conditions are not met then the following error message is displayed at the *end* of the proof:

```
*** Error in attribute "Nominal.eqvt":
*** "offending lemma" does not comply with the
    form of an equivariance lemma ...
```

A number of lemmas that are generated by Nominal Isabelle get automatically tagged with the [eqvt]-attribute. Some of them are listed in Appendix B.1. Also all constructors of a nominal datatype have equivariance lemmas. For example for lambda-terms we have:

lemma lam.perm[eqvt]:
 fixes π::"var prm"
 shows "$\pi \bullet$ Var x = Var ($\pi \bullet$ x)"
 and "$\pi \bullet$ App t_1 t_2 = App ($\pi \bullet t_1$) ($\pi \bullet t_2$)"
 and "$\pi \bullet$ Lam [x].t = Lam [($\pi \bullet$ x)].($\pi \bullet$ t)"

The equivariance lemmas are collected under the theorem called eqvts. Therefore one can use them in proofs using the standard proving tools. For example

- (simp add: eqvts)
- (rule eqvts)

The crucial point of equivariance is that an equivariant function (similar relations) have empty support. This fact will play a crucial role in the next section.

4.4 Function Definitions

To be done.

nominal_primrec
 subst :: "lam⇒var⇒lam⇒lam" ("_[_::=_]" [100,100,100] 100)
where
 "(Var x)[y::=t‘] = (if x=y then t‘ else (Var x))"
| "(App t_1 t_2)[y::=t‘] = App (t_1[y::=t‘]) (t_2[y::=t‘])"
| "x $\sharp$ (y,t‘) $\Longrightarrow$ (Lam [x].t)[y::=t‘] = Lam [x].(t[y::=t‘])"
apply(finite_guess)+
apply(rule TrueI)+
apply(simp add: abs_fresh)
apply(fresh_guess)+
done

4.5 Inductive Definitions and Strong Rule Induction Principles

Inductive definition can be declared in Isabelle using the **inductive** mechanism. If such a definition contains nominal datatypes, then one often needs a stronger induction principle that has the variable convention built in. The strengthening can be achieved using the command **nominal_inductive**. While the strengthening is completely automatic in case of structural inductions (see Sec. 4.2), there is in some work needed from the user in case of strong inductions for inductive definitions. This is because the strengthening relies on three facts

- the inductive definition has to be equivariant and
- every rule of an inductive definition has to be formulated so that no variable standing for a binder occurs in the support of this rule, and
- if there are more than one binder in a rule, then these binders also need to be distinct.

While the first property can often be automatically derived using the command **equivariance**, the latter two require some proof obligations to be discharged.

As an example consider the inductive definition for the typing relation in the simply-typed lambda-calculus. The nominal datatypes for terms and types are defined as:

nominal_datatype lam =
 Var ”var”
| App ”lam” ”lam”
| Lam ”≪var≫lam” (”Lam [_]._” [100,100] 100)

nominal_datatype ty =
 TVar ”string”
| TArr ”ty” ”ty” (”_→_” [100,100] 100)

Typing contexts will be represented as lists of (var,ty)-pairs. We need to ensure that a var in a context has only one associated type. This can be achieved with a validity predicate defined inductively as:

inductive
 valid :: ”(var×ty) list ⇒ bool”
where
 [intro]: ”valid []”
| [intro]: ”⟦valid Γ; x ♯ Γ⟧⟹ valid ((x,T)#Γ)”

The typing judgement can then be defined by the rules:

inductive
typing :: "(var×ty) list⇒lam⇒ty⇒bool" (" _ ⊢ _ : _ " [60,60,60] 60)
where
ty_var[intro]: "⟦valid Γ; (x,T) ∈ set Γ⟧ ⟹ Γ ⊢ (Var x) : T"
| ty_app[intro]: "⟦Γ ⊢ t_1 : T_1→T_2; Γ ⊢ t_2 : T_1⟧ ⟹ Γ ⊢ App t_1 t_2 : T_2"
| ty_lam[intro]: "⟦x ♯ Γ; (x,T_1)#Γ ⊢ t : T_2⟧ ⟹ Γ ⊢ Lam [x].t : T_1→T_2"

With this inductive definition comes automatically an induction principle, called typing.induct. However, as said before, this induction principle is too weak for establishing any non-trivial fact about typing. In order to strengthen it, we need to make sure that the typing relation is equivariant, that means the property

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

holds. For this, we have to make sure that every side-condition in the definition of typing is equivariant. That means, for example, for the rule ty_var that valid, pairing, ∈, set and Var must be equivariant. Nominal Isabelle knows that the latter four are equivariant, but not that the user-defined predicate valid is. Therefore attempting to prove equivariance of typing using the command **equivariance** will fail: the statement

equivariance typing

results in the error message

```
*** Could not prove equivariance for introduction rule
*** ⟦valid Γ; (x, T) ∈ set Γ⟧ ⟹  Γ ⊢ Var x : T
      ...
```

where the second line indicates the rule, which causes **equivariance** to fail (in this case ty_var). To solve the problem equivariance of valid needs to be proved first. This can done by stating

equivariance valid

This succeeds because [], ♯, # and pairing, which are used in the definition of valid, are all equivariant. Having equivariance of valid at the disposal, the statement

equivariance typing

succeeds.

Now we can strengthen the induction principle for the typing relation using **nominal_inductive**. Issuing

nominal_inductive typing

will open a proof requiring to show the following three goals:

$$\llbracket x \sharp \Gamma;\ \ (x, T_1)\ \#\ \Gamma \vdash t : T_2\ \rrbracket \Longrightarrow x \sharp \Gamma$$
$$\llbracket x \sharp \Gamma;\ \ (x, T_1)\ \#\ \Gamma \vdash t : T_2\ \rrbracket \Longrightarrow x \sharp \text{Lam } [x].t$$
$$\llbracket x \sharp \Gamma;\ \ (x, T_1)\ \#\ \Gamma \vdash t : T_2\ \rrbracket \Longrightarrow x \sharp T_1 \rightarrow T_2$$

These goals correspond to the rule ty_lam of the typing-relation (the only rule where a binder occurs). They ensure that the binder x does not appear freely in the conclusion of ty_lam, i.e. $\Gamma \vdash \text{Lam } [x].t : T_1 \rightarrow T_2$. The first one is implied by the side-condition $x \sharp \Gamma$ in ty_lam; the second holds because x is bound in Lam [x].t; the last one holds since vars cannot appear in types. However we need a little side-lemma which establishes this last fact:

lemma fresh_ty:
 fixes x::"var"
 and T::"ty"
 shows "x ♯ T"
by (nominal_induct T rule: ty.strong_induct)
 (simp_all add: fresh_string)

Armed with this side-lemma, we can complete the proof opened by **nominal_inductive** as follows:

nominal_inductive typing
 by (simp_all add: fresh_ty abs_fresh)

This results in the following strong induction principle, called typing.strong_induct, being proved:

$$\frac{\begin{array}{l}\Gamma \vdash t : T \\ \llbracket \text{valid } \Gamma;\ (x, T) \in \text{set } \Gamma \rrbracket \Longrightarrow P\ c\ \Gamma\ (\text{Var } x)\ T \\ \llbracket \Gamma \vdash t_1 : T_1 \rightarrow T_2;\ \bigwedge c'.\ P\ c'\ \Gamma\ t_1\ (T_1 \rightarrow T_2);\ \Gamma \vdash t_2 : T_1;\ \bigwedge c'.\ P\ c'\ \Gamma\ t_2\ T_1 \rrbracket \Longrightarrow P\ c\ \Gamma\ (\text{App } t_1\ t_2)\ T_2 \\ \llbracket x \sharp c;\ x \sharp \Gamma;\ (x,T_1)\#\Gamma \vdash t : T_2;\ \bigwedge c'.\ P\ c'\ ((x,T_1)\#\Gamma)\ t\ T_2 \rrbracket \Longrightarrow P\ c\ \Gamma\ (\text{Lam } [x].t)\ (T_1 \rightarrow T_2)\end{array}}{P\ c\ \Gamma\ t\ T}$$

We will show next that this stronger version of the induction principle is quite useful for formalising proofs that use the variable convention. Like in the case of the strong structural induction principles for nominal datatypes, we can instantiate the freshness context c of the induction so that the binder x in the lambda-case is fresh with respect to this context. Consider Figure 4.2 where the weakening lemma is proved. In the lambda-case of this proof the fact that the binder x is fresh for the typing-context Γ_2 is needed—otherwise we would have to do explicit renamings as the rule ty_lam requires that $x \sharp \Gamma_2$ and we also cannot establish that valid $((x, T_1) \# \Gamma_2)$ holds. However we can establish these facts by stating in nominal_induct that Γ_2 should be avoided.

A disadvantage of **nominal_inductive** is that the automatic proof often requires that one states the rules for the inductively defined predicate slightly different than one is used to from "pencil-and-paper" reasoning. Recall that one requirement for the automatic strengthening is that no binder should not be in the upport of the conclusion of a rule. Now consider the following definition of beta-reduction:

inductive
beta :: "lam⇒lam⇒bool" ("_ $\longrightarrow_\beta$ _" [80,80] 80)
where
AppL: "$s_1 \longrightarrow_\beta s_2 \Longrightarrow$ App s_1 t $\longrightarrow_\beta$ App s_2 t"
| AppR: "$s_1 \longrightarrow_\beta s_2 \Longrightarrow$ App t $s_1 \longrightarrow_\beta$ App t s_2"
| Lam: "$s_1 \longrightarrow_\beta s_2 \Longrightarrow$ Lam [x].$s_1 \longrightarrow_\beta$ Lam [x].s_2"
| Beta: "App (Lam [x].s_1) $s_2 \longrightarrow_\beta s_1$[x::=$s_2$]"

The rules lam and beta contain the binder x and in order to strengthen the induction principle in both cases the x needs to be fresh for the rules' conclusion. When stating

nominal_inductive beta

we are required to discharge the following four proof-obligations

$s_1 \longrightarrow_\beta s_2 \Longrightarrow x \sharp$ Lam [x].s_2
$s_1 \longrightarrow_\beta s_2 \Longrightarrow x \sharp$ Lam [x].s_2
$x \sharp$ App (Lam [x].s_1) s_2
$x \sharp s_1$[x::=s_2]

where we can easily discharge the first two (so rule lam is ok), but the last two are unprovable: x is fresh for Lam [x].s_1, but it may occur in s_2. As for the term s_1[x::=s_2], we know that any occurrences of x in s_1 will be masked by the substitution, but x may again be free in s_2. To obtain automatically a strong induction principle for beta-reduction, it must be defined as:

inductive
beta :: "lam⇒lam⇒bool" ("_ $\longrightarrow_\beta$ _" [80,80] 80)
where

```
1  abbreviation
2    sub_ctxt :: "(var×ty) list⇒(var×ty) list⇒bool" ("_ ⊆ _" [60,60] 60)
3  where
4    "Γ1 ⊆ Γ2 ≡ ∀x T. (x,T) ∈ set Γ1 ⟶ (x,T) ∈ set Γ2"
5
6  lemma weakening:
7    fixes Γ1::"(var×ty) list"
8    and   t ::"lam"
9    and   T ::"ty"
10   assumes asms: "Γ1 ⊢ t : T"  "valid Γ2"  "Γ1 ⊆ Γ2"
11   shows "Γ2 ⊢ t : T"
12 using asms
13 proof (nominal_induct avoiding: Γ2 rule: typing.strong_induct)
14   case (ty_var Γ1 x T)                                        (variable case)
15   have "Γ1 ⊆ Γ2" by fact
16   moreover
17   have "valid Γ2" by fact
18   moreover
19   have "(x,T) ∈ set Γ1" by fact
20   ultimately show "Γ2 ⊢ Var x : T" by auto
21 next
22   case (ty_lam x Γ1 T1 t T2)                                  (lambda case)
23   have vc: "x ♯ Γ2" by fact
24   have ih: "⋀Γ3. ⟦valid Γ3; (x,T1)#Γ1 ⊆ Γ3⟧ ⟹ Γ3 ⊢ t : T2" by fact
25   have "Γ1 ⊆ Γ2" by fact
26   then have "(x,T1)#Γ1 ⊆ (x,T1)#Γ2" by simp
27   moreover
28   have "valid Γ2" by fact
29   then have "valid ((x,T1)#Γ2)" using vc by auto
30   ultimately have "(x,T1)#Γ2 ⊢ t : T2" using ih by simp
31   with vc show "Γ2 ⊢ Lam [x].t : T1→T2" by auto
32 qed (auto)                                  (application case is automatic)
```

Figure 4.2: The Isar-proof for the weakening lemma. In Lines 1-4 the notion of a sub-context is defined for lists. The interesting case in the weakening lemma (Lines 22-31) crucially depends on the assumption that $x \sharp \Gamma_2$ holds (Line 23). This fact is used in Lines 29 and 31. It is obtained by using the strong induction principle for the typing relation and setting up the induction so that it avoids Γ_2 (Line 13).

```
  AppL:  "s1 ⟶β s2 ⟹ App s1 t ⟶β App s2 t"
| AppR:  "s1 ⟶β s2 ⟹ App t s1 ⟶β App t s2"
| Lam:   "s1 ⟶β s2 ⟹ Lam [x].s1 ⟶β Lam [x].s2"
| Beta:  "x ♯ s2 ⟹ App (Lam [x].s1) s2 ⟶β s1[x::=s2]"
```

where in the last clause the freshness constraint $x \sharp s_2$ is added. With this additional constraint, we are able to use **nominal_inductive** to generate the strong induction principle beta.strong_induct.

```
nominal_inductive beta
  by (simp_all add: abs_fresh fresh_subst)
```

Adding the constraint, however, is annoying because both versions of beta can be shown to define the same reduction relation. While it is prossible to use the simpler version of beta and then derive manually a strong induction principle, the possibility of having the strong induction principle automatically derived by **nominal_inductive** often outweighs the benefits of being word-for-word faithful to the "pencil-and-paper" definitions. If one really prefers the simpler version for the beta-rule, then one can prove the following lemma which alpha-converts the terms before applying the more restricted version of the beta-rule:

```
lemma better_beta_intro:
  shows "App (Lam [x].s1) s2 ⟶β s1[x::=s2]"
proof -
  obtain x'::"var" where fs: "x' ♯ (x,s1,s2)" by (rule exists_fresh, rule fin_supp, blast)
  have "App (Lam [x].s1) s2 = App (Lam [x'].([(x',x)]•s1)) s2" using fs
    by (rule_tac sym, perm_simp add: lam.inject alpha fresh_atm fresh_prod)
  also have "... ⟶β ([(x',x)]•s1)[x'::=s2]" using fs by (simp add: beta.Beta)
  also have "... = s1[x::=s2]" using fs by (simp add: subst_rename)
  finally show "App (Lam [x].s1) s2 ⟶β s1[x::=s2]" by simp
qed
```

where we need the side-lemma

```
lemma subst_rename:
  assumes a: "x' ♯ s1"
  shows "([(x',x)]•s1)[x'::=s2] = s1[x::=s2]"
using a
by (nominal_induct s1 avoiding: x x' s2 rule: lam.strong_induct)
   (auto simp add: calc_atm fresh_atm abs_fresh)
```

Nominal_inductive has one advanced feature that has not yet shown up in the given examples: in some cases one wants to strengthen over a variable which is *not* a binder. For example Crary [2005] defines the following types for lambda-terms:

```
nominal_datatype ty =
  TBase
 | TUnit
 | Arrow "ty" "ty" ("_→_" [100,100] 100)
```

and then defines the following two mutually recursive relations

```
inductive
 alg_equiv:: "(var×ty) list⇒lam⇒lam⇒ty⇒bool" ("_ ⊢ _ ⇔ _ : _")
and
 alg_path_equiv:: "(var×ty) list⇒lam⇒lam⇒ty⇒bool" ("_ ⊢_ ↔ _ : _")
where
 QAT_Base:   "⟦s ⇓ p; t ⇓ q; Γ ⊢ p ↔ q : TBase ⟧ ⟹ Γ ⊢ s ⇔ t : TBase"
| QAT_Arrow: "⟦x ♯ (Γ,s,t); (x,T₁)#Γ ⊢ App s (Var x) ⇔ App t (Var x) : T₂⟧
               ⟹ Γ ⊢ s ⇔ t : T₁ → T₂"
| QAT_One:   "valid Γ ⟹ Γ ⊢ s ⇔ t : TUnit"
| QAP_Var:   "⟦valid Γ; (x,T) ∈ set Γ⟧ ⟹ Γ ⊢ Var x ↔ Var x : T"
| QAP_App:   "⟦Γ ⊢ p ↔ q : T₁ → T₂; Γ ⊢ s ⇔ t : T₁⟧ ⟹ Γ ⊢ App p s ↔ App q t : T₂"
```

The interesting rule is QAT_Arrow. Although no binder occurs in the definition at all, one wants in proofs by induction over this definition to be able to assume that the variable x is sufficiently fresh (not just fresh for Γ, s and t). Although not obvious, this is permitted because the x does not appear in the conclusion of that rule. The strengthening of the induction principle over this variable can be achieved by giving to **nominal_inductive** the information that the strengthening should happen over x. For this one uses **avoids**. For example strengthening the induction principle for the relations above can be achieved by

```
equivariance alg_equiv

nominal_inductive alg_equiv
 avoids QAT_Arrow: x
 by (simp_all add: fresh_ty fresh_prod)
```

Note that in mutual inductive definitions only one of the predicates needs to be mentioned in **equivariance** and **nominal_inductive**. The strong induction principle called

alg_equiv_alg_path_equiv.strong_inducts

(note the "s" at the end used in mutually inductive definitions) can now be used to do inductions over both relations.

Three technical points concerning **nominal_inductive** should be noted: the strengthening only works with inductive definition that have been declared using **inductive** (does not work); in case the strengthening does not require any subgoals to be discharged (this happens in the cases where the inductive definition does not contain any binders), then the opened proof needs to be closed by a dot, for example:

> **nominal_inductive** *foo* .

The general syntax for **nominal_inductive** is

> **nominal_inductive** *name-of-inductive-relation*
>   **avoids** *rule-name*$_1$: $x_1\ x_2 \ldots$
>   **and** *rule-name*$_2$: $y_1\ y_2 \ldots$
> $\ldots$

At the moment there is also one limitation placed upon **nominal_inductive**: so far it cannot deal correctly with any logical connective in rules.

(Equivariance for different atom-types)

Chapter 5

Advanced Topics

5.1 Functions That Need to Generate Fresh Names

Sometimes functions explicitly depend on choosing fresh names. To abstract away from any particular choice of such names, Pitts introduced the function fresh_fun (he called this function just fresh). This function takes as argument a function that is of type atom to 'a. The main property of fresh_fun is

if a ♯ (h, ha) then fresh_fun h = h a

where the function h needs to be finitely supported. With this, one can effectively control when the choice of a fresh name is made in a proof. An example where fresh_fun needs to be used is the CPS-translation of Plotkin:

```
nominal_primrec
  CPS :: "lam ⇒ lam"
where
  "CPS (Var x) = fresh_fun (λk. Lam [k].(App (Var k) (Var x)))"
| "CPS (Lam [x].M) =
        fresh_fun (λk. Lam [k].(App (Var k) (Lam [x].(CPS M))))"
| "CPS (App M N) = fresh_fun (λk. Lam [k].(App (CPS M)
            (fresh_fun (λm. Lam [m].(App (CPS N)
            (fresh_fun (λn. App (App (Var m) (Var n)) (Var k))))))))"
```

However, it is often a bit awkward to reason about fresh_fun because one can only replace fresh_fun h with a freshly chosen name, say a, if one can determine that a is fresh for the function h and the application h a (see above).

To simplify the reasoning involving fresh_fun, the tactic fresh_fun_simp has been introduced. To use it, one has to first generate a fresh name using the generate_fresh tactic and then can apply this tactic. It uses the generated fresh name to instantiate one instance of the fresh_fun-term. In doing so, it attempts to solve all conditions required by the main property of fresh_fun. If there are unsolved goals, then they are returned to the user. Using the option (no_asm) one can direct the tactic to only search for instances of fresh_fun in the conclusion of the current goal. Possible uses are:

- fresh_fun_simp
- fresh_fun_simp (no_asm)

Chapter 6

Examples

Some examples are included in:

`[ISABELLE_HOME]/src/HOL/Nominal/Examples`:

`Weakening.thy`	a proof for weakening in the simply-typed lambda-calculus
`Fsub.thy`	System F with subtyping (Part 1A of the PoplMmark Challenge [Aydemir et al., 2005, PoplMark Challenge])
`Lam_Funs.thy`	contains the definitions of important functions in the lambda-calculus
`CR.thy`	Church-Rosser proof from Barendregt's lambda-calculus book [1981]
`CR_Takahashi.thy`	a much simpler proof for the Church-Rosser property taken from the work of Takahashi [1995] and Pollack [1995]
`CK_Machine.thy`	formalisation of soundness and completeness for a CBV CK-machine; also type-preservation and progress are shown
`SN.thy`	strong normalisation from the Proofs and Types book by Girard et al. [1989]
`Height.thy`	a simple example suggested by D. Wang about the height of lambda-terms
`Lambda_mu.thy`	datatype declaration for the lambda-mu calculus
`Class.thy`	a verification of the main result from Urban [2000] about cut-elimination in classical logic
`SOS.thy`	some typical SOS-proofs about the simply-typed lambda-calculus
`Crary.thy`	a formalisation of the chapter on logical relations by Crary [2005]
`VC_Compatible.thy`	gives two examples of faulty lemmas derived with the help of the variable convention about binders
`Support.thy`	includes some calculations of the support in non-trivial instances

`Contexts.thy`	shows the equivalence of beta-reduction defined in the Plotkin-style and Felleisen-Hieb-style
`Type_Preservation.thy`	a type preservation proof for beta-reduction in the simply-typed lambda-calculus
`Standardization.thy`	a standardization proof by Matthes formalised by Berghofer

Appendix A

Frequently Asked Questions

A.1 The atom_decl command does not work.

If you get the error message:

```
*** Outer syntax error: end of input expected,
*** but identifier "atom_decl" was found
```

while stepping through a theory-file, then the most likely problem is that the key-word file for Nominal Isabelle has not been loaded. Make sure you start Isabelle with the "-L HOL-Nominal" option.

```
isabelle emacs -L HOL-Nominal file.thy &
```

A.2 Can one avoid typing \<guillemotleft>?

If your X-Symbols are correctly installed and enabled in the menu "Proof-General → Options", then you can just type "<<" (that is two consecutive less-thans). X-Symbols will automatically expand this to \<guillemotleft>. Similarly with \<guillemotright>.

A.3 I want to prove something about support or freshness, but theorems which should be trivial cannot be proved.

Attempting to prove the following lemma

lemma support_trivial_fails:
shows "supp (App t_1 t_2) = (supp t_1) $\cup$ (supp t_2)"

will fail. The problem is that supp is a polymorphic constant and this equation is only provable with respect to a concrete atom type. The solution is to give explicit type-annotations, for example

lemma support_trivial_fails:
shows "supp (App t_1 t_2) = (supp t_1) $\cup$ ((supp t_2)::var set)"
by (simp add: lam.supp)

Similar type-annotations are necessary in lemmas involving freshness.

Appendix B

Infrastructure

B.1 Equivariance Lemmas

By tagging theorems with the [eqvt]-attribute they are included in the collection of theorems named eqvts. The following general theorems have been tagged by default:

- numbers $0, 1, 2\ldots$ of types int and nat are equivariant

  $\pi \bullet \text{number\_of n} = \text{number\_of n}$
  $\pi \bullet 0 = 0$
  $\pi \bullet 1 = 1$

- logical connectives are equivariant

  $\pi \bullet (A \wedge B) = (\pi \bullet A \wedge \pi \bullet B)$
  $\pi \bullet (A \vee B) = (\pi \bullet A \vee \pi \bullet B)$
  $\pi \bullet (A \longrightarrow B) = (\pi \bullet A \longrightarrow \pi \bullet B)$
  $\pi \bullet (\neg A) = (\neg \pi \bullet A)$
  § $\pi \bullet (x = y) = (\pi \bullet x = \pi \bullet y)$
  $\pi \bullet (\text{if b then } e_1 \text{ else } e_2) = (\text{if } \pi \bullet b \text{ then } \pi \bullet e_1 \text{ else } \pi \bullet e_2)$
  § $\pi \bullet a \sharp x = \pi \bullet a \sharp \pi \bullet x$
  $\pi \bullet \text{True} = \text{True}$
  $\pi \bullet \text{False} = \text{False}$

- units, products, lists and options are equivariant

$\pi \bullet () = ()$
$\pi \bullet (x, y) = (\pi \bullet x, \pi \bullet y)$
$\pi \bullet \text{fst } x = \text{fst } (\pi \bullet x)$
$\pi \bullet \text{snd } x = \text{snd } (\pi \bullet x)$
$\pi \bullet [] = []$
$\pi \bullet (x \# l) = \pi \bullet x \# \pi \bullet l$
$\pi \bullet (l_1 @ l_2) = \pi \bullet l_1 @ \pi \bullet l_2$
$\pi \bullet \text{Some } x = \text{Some } (\pi \bullet x)$
$\pi \bullet \text{None} = \text{None}$

- the following operators on sets are equivariant

$\pi \bullet \{\} = \{\}$
$\pi \bullet (X \cup Y) = \pi \bullet X \cup \pi \bullet Y$
§ $\pi \bullet \text{insert } x\ X = \text{insert } (\pi \bullet x)\ (\pi \bullet X)$
§ $\pi \bullet (X - Y) = \pi \bullet X - \pi \bullet Y$
§ $\pi \bullet (x \in X) = (\pi \bullet x \in \pi \bullet X)$
$\pi \bullet \text{set } l = \text{set } (\pi \bullet l)$

- the following operators on ints and nats are equivariant

$\pi \bullet \text{Suc } x = \text{Suc } (\pi \bullet x)$
$\pi \bullet \text{min } x\ y = \text{min } (\pi \bullet x)\ (\pi \bullet y)$
$\pi \bullet \text{max } x\ y = \text{max } (\pi \bullet x)\ (\pi \bullet y)$
$\pi \bullet (x + y) = \pi \bullet x + \pi \bullet y$
$\pi \bullet (x - y) = \pi \bullet x - \pi \bullet y$
$\pi \bullet (x * y) = \pi \bullet x * \pi \bullet y$
$\pi \bullet (x \text{ div } y) = \pi \bullet x \text{ div } \pi \bullet y$

Bibliography

Aydemir, B. E., Bohannon, A., Fairbairn, M., Foster, J. N., Pierce, B. C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., and Zdancewic, S. [2005]. Mechanized Metatheory for the Masses: The PoplMark Challenge. In *Proc. of the 18th International Conference on Theorem Proving in Higher-Order Logics (TPHOLs)*, volume 3603 of *LNCS*, pages 50–65.

Barendregt, H. [1981]. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland.

Crary, K. [2005]. Logical Relations and a Case Study in Equivalence Checking. In Pierce, B. C., editor, *Advanced Topics in Types and Programming Languages*, pages 139–160. MIT Press.

Girard, J.-Y., Lafont, Y., and Taylor, P. [1989]. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press.

Nipkow, T., Paulson, L. C., and Wenzel, M. [2002]. *Isabelle HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag.

Pitts, A. M. [2003]. Nominal Logic, A First Order Theory of Names and Binding. *Information and Computation*, 186:165–193.

Pitts, A. M. [2006]. Alpha-Structural Recursion and Induction. *Journal of the ACM*, 53:459–506.

Pollack, R. [1995]. Polishing Up the Tait–Martin-Löf Proof of the Church-Rosser Theorem. In *Proc. of De Wintermöte '95*. Department of Computing Science, Chalmers Univ. of Technology, Göteborg, Sweden.

PoplMark Challenge. http://www.cis.upenn.edu/group/proj/plclub/mmm/.

Takahashi, M. [1995]. Parallel Reductions in Lambda-Calculus. *Information and Computation*, 118(1):120–127.

Urban, C. [2000]. *Classical Logic and Computation*. PhD thesis, Cambridge University.

Wenzel, M. [2004]. *Using Axiomatic Type Classes in Isabelle*. Manual in the Isabelle distribution.

Wenzel, M. [2007]. *The Isabelle/Isar Implementation*.

Index