

Nominal 2

Christian Urban, Stefan Berghofer, and Cezary Kaliszyk

February 6, 2026

Abstract

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 Barendregts variable convention already built in.

This entry can be used as a more advanced replacement for HOL/Nominal in the Isabelle distribution.

Contents

1	Atoms and Sorts	1
2	Sort-Respecting Permutations	2
2.1	Permutations form a (multiplicative) group	3
3	Implementation of swappings	4
4	Permutation Types	5
4.1	Permutations for atoms	6
4.2	Permutations for permutations	7
4.3	Permutations for functions	7
4.4	Permutations for booleans	8
4.5	Permutations for sets	8
4.6	Permutations for <i>unit</i>	9
4.7	Permutations for products	10
4.8	Permutations for sums	10
4.9	Permutations for <i>'a list</i>	10
4.10	Permutations for <i>'a option</i>	11
4.11	Permutations for <i>'a multiset</i>	11
4.12	Permutations for <i>'a fset</i>	11

4.13	Permutations for $('a, 'b)$ <i>finfun</i>	12
4.14	Permutations for <i>char</i> , <i>nat</i> , and <i>int</i>	12
5	Pure types	13
5.1	Types <i>char</i> , <i>nat</i> , and <i>int</i>	14
6	Infrastructure for Equivariance and <i>Perm-simp</i>	14
6.1	Basic functions about permutations	14
6.2	Eqvt infrastructure	14
6.3	<i>perm-simp</i> infrastructure	14
6.3.1	Equivariance for permutations and swapping	15
6.3.2	Equivariance of Logical Operators	15
6.3.3	Equivariance of Set operators	17
6.3.4	Equivariance for product operations	20
6.3.5	Equivariance for list operations	20
6.3.6	Equivariance for <i>'a option</i>	21
6.3.7	Equivariance for <i>'a fset</i>	21
6.3.8	Equivariance for $('a, 'b)$ <i>finfun</i>	21
7	Supp, Freshness and Supports	21
7.1	supp and fresh are equivariant	22
8	supports	23
9	Support w.r.t. relations	24
10	Finitely-supported types	24
10.1	Type <i>atom</i> is finitely-supported.	24
11	Type <i>perm</i> is finitely-supported.	25
12	Finite Support instances for other types	26
12.1	Type $'a \times 'b$ is finitely-supported.	26
12.2	Type $'a + 'b$ is finitely supported	27
12.3	Type <i>'a option</i> is finitely supported	27
12.3.1	Type <i>'a list</i> is finitely supported	27
13	Support and Freshness for Applications	28
13.1	Equivariance Predicate <i>eqvt</i> and <i>eqvt-at</i>	29
13.2	helper functions for <i>nominal-functions</i>	30
14	Support of Finite Sets of Finitely Supported Elements	31
14.1	Type <i>'a multiset</i> is finitely supported	33
14.2	Type <i>'a fset</i> is finitely supported	34
14.3	Type $('a, 'b)$ <i>finfun</i> is finitely supported	35

15 Freshness and Fresh-Star	35
16 Induction principle for permutations	38
17 Avoiding of atom sets	39
18 Renaming permutations	40
19 Concrete Atoms Types	41
20 Infrastructure for concrete atom types	43
20.1 Syntax for coercing at-elements to the atom-type	44
20.2 A lemma for proving instances of class <i>at</i>	44
21 Library functions for the nominal infrastructure	45
22 The freshness lemma according to Andy Pitts	45
23 Automation for creating concrete atom types	47
24 Automatic equivariance procedure for inductive definitions	47
25 Abstractions	47
26 Mono	48
27 Equivariance	48
28 Equivalence	48
29 General Abstractions	50
30 Strengthening the equivalence	51
31 Quotient types	52
32 Abstractions of single atoms	57
32.1 Renaming of bodies of abstractions	58
33 Infrastructure for building tuples of relations and functions	59
34 Interface for <i>nominal-datatype</i>	63
35 Preparing and parsing of the specification	63
36 <i>nat-of</i> is an example of a function without finite support	63
37 Manual instantiation of class <i>at</i>.	64

38 Automatic instantiation of class <i>at</i>.	64
39 An example for multiple-sort atoms	65
40 Tests with subtyping and automatic coercions	66

```

theory Nominal2-Base
imports HOL-Library.Infinite-Set
         HOL-Library.Multiset
         HOL-Library.FSet
         FinFun.FinFun
keywords
  atom-decl equivariance :: thy-decl
begin

declare [[typedef-overloaded]]

```

1 Atoms and Sorts

A simple implementation for *atom-sorts* is strings.

To deal with Church-like binding we use trees of strings as sorts.

```
datatype atom-sort = Sort string atom-sort list
```

```
datatype atom = Atom atom-sort nat
```

Basic projection function.

```
primrec
  sort-of :: atom  $\Rightarrow$  atom-sort
where
  sort-of (Atom s n) = s
```

```
primrec
  nat-of :: atom  $\Rightarrow$  nat
where
  nat-of (Atom s n) = n
```

There are infinitely many atoms of each sort.

```
lemma INFM-sort-of-eq:
shows INFM a. sort-of a = s
<proof>
```

```
lemma infinite-sort-of-eq:
shows infinite {a. sort-of a = s}
<proof>
```

```
lemma atom-infinite [simp]:
shows infinite (UNIV :: atom set)
```

<proof>

lemma *obtain-atom*:

fixes $X :: \text{atom set}$

assumes $X: \text{finite } X$

obtains a **where** $a \notin X$ $\text{sort-of } a = s$

<proof>

lemma *atom-components-eq-iff*:

fixes $a b :: \text{atom}$

shows $a = b \iff \text{sort-of } a = \text{sort-of } b \wedge \text{nat-of } a = \text{nat-of } b$

<proof>

2 Sort-Respecting Permutations

definition

$\text{perm} \equiv \{f. \text{bij } f \wedge \text{finite } \{a. f a \neq a\} \wedge (\forall a. \text{sort-of } (f a) = \text{sort-of } a)\}$

typedef $\text{perm} = \text{perm}$

<proof>

lemma *permI*:

assumes $\text{bij } f$ **and** $\text{MOST } x. f x = x$ **and** $\bigwedge a. \text{sort-of } (f a) = \text{sort-of } a$

shows $f \in \text{perm}$

<proof>

lemma *perm-is-bij*: $f \in \text{perm} \implies \text{bij } f$

<proof>

lemma *perm-is-finite*: $f \in \text{perm} \implies \text{finite } \{a. f a \neq a\}$

<proof>

lemma *perm-is-sort-respecting*: $f \in \text{perm} \implies \text{sort-of } (f a) = \text{sort-of } a$

<proof>

lemma *perm-MOST*: $f \in \text{perm} \implies \text{MOST } x. f x = x$

<proof>

lemma *perm-id*: $\text{id} \in \text{perm}$

<proof>

lemma *perm-comp*:

assumes $f: f \in \text{perm}$ **and** $g: g \in \text{perm}$

shows $(f \circ g) \in \text{perm}$

<proof>

lemma *perm-inv*:

assumes $f: f \in \text{perm}$

shows $(\text{inv } f) \in \text{perm}$

<proof>

lemma *bij-Rep-perm*: *bij (Rep-perm p)*
<proof>

lemma *finite-Rep-perm*: *finite {a. Rep-perm p a ≠ a}*
<proof>

lemma *sort-of-Rep-perm*: *sort-of (Rep-perm p a) = sort-of a*
<proof>

lemma *Rep-perm-ext*:
Rep-perm p1 = Rep-perm p2 ⇒ p1 = p2
<proof>

instance *perm :: size <proof>*

2.1 Permutations form a (multiplicative) group

instantiation *perm :: group-add*
begin

definition
0 = Abs-perm id

definition
- p = Abs-perm (inv (Rep-perm p))

definition
p + q = Abs-perm (Rep-perm p ∘ Rep-perm q)

definition
(p1::perm) - p2 = p1 + - p2

lemma *Rep-perm-0*: *Rep-perm 0 = id*
<proof>

lemma *Rep-perm-add*:
Rep-perm (p1 + p2) = Rep-perm p1 ∘ Rep-perm p2
<proof>

lemma *Rep-perm-uminus*:
Rep-perm (- p) = inv (Rep-perm p)
<proof>

instance
<proof>

end

3 Implementation of swappings

definition

$swap :: atom \Rightarrow atom \Rightarrow perm \langle '(- \rightleftharpoons -)' \rangle$

where

$(a \rightleftharpoons b) =$
Abs-perm (if sort-of a = sort-of b
then $(\lambda c. \text{if } a = c \text{ then } b \text{ else if } b = c \text{ then } a \text{ else } c)$
else id)

lemma Rep-perm-swap:

Rep-perm $(a \rightleftharpoons b) =$
(if sort-of a = sort-of b
then $(\lambda c. \text{if } a = c \text{ then } b \text{ else if } b = c \text{ then } a \text{ else } c)$
else id)

$\langle proof \rangle$

lemmas Rep-perm-simps =

Rep-perm-0
Rep-perm-add
Rep-perm-uminus
Rep-perm-swap

lemma swap-different-sorts [simp]:

sort-of a \neq sort-of b $\implies (a \rightleftharpoons b) = 0$
 $\langle proof \rangle$

lemma swap-cancel:

shows $(a \rightleftharpoons b) + (a \rightleftharpoons b) = 0$
and $(a \rightleftharpoons b) + (b \rightleftharpoons a) = 0$
 $\langle proof \rangle$

lemma swap-self [simp]:

$(a \rightleftharpoons a) = 0$
 $\langle proof \rangle$

lemma minus-swap [simp]:

$-(a \rightleftharpoons b) = (a \rightleftharpoons b)$
 $\langle proof \rangle$

lemma swap-commute:

$(a \rightleftharpoons b) = (b \rightleftharpoons a)$
 $\langle proof \rangle$

lemma swap-triple:

assumes $a \neq b$ and $c \neq b$
assumes sort-of a = sort-of b sort-of b = sort-of c
shows $(a \rightleftharpoons c) + (b \rightleftharpoons c) + (a \rightleftharpoons c) = (a \rightleftharpoons b)$
 $\langle proof \rangle$

4 Permutation Types

Infix syntax for *permute* has higher precedence than addition, but lower than unary minus.

```
class pt =  
  fixes permute :: perm  $\Rightarrow$  'a  $\Rightarrow$  'a ( $\langle \cdot \cdot \cdot \rangle$  [76, 75] 75)  
  assumes permute-zero [simp]:  $0 \cdot x = x$   
  assumes permute-plus [simp]:  $(p + q) \cdot x = p \cdot (q \cdot x)$   
begin
```

```
lemma permute-diff [simp]:  
  shows  $(p - q) \cdot x = p \cdot - q \cdot x$   
   $\langle$ proof $\rangle$ 
```

```
lemma permute-minus-cancel [simp]:  
  shows  $p \cdot - p \cdot x = x$   
  and  $- p \cdot p \cdot x = x$   
   $\langle$ proof $\rangle$ 
```

```
lemma permute-swap-cancel [simp]:  
  shows  $(a \rightleftharpoons b) \cdot (a \rightleftharpoons b) \cdot x = x$   
   $\langle$ proof $\rangle$ 
```

```
lemma permute-swap-cancel2 [simp]:  
  shows  $(a \rightleftharpoons b) \cdot (b \rightleftharpoons a) \cdot x = x$   
   $\langle$ proof $\rangle$ 
```

```
lemma inj-permute [simp]:  
  shows inj (permute p)  
   $\langle$ proof $\rangle$ 
```

```
lemma surj-permute [simp]:  
  shows surj (permute p)  
   $\langle$ proof $\rangle$ 
```

```
lemma bij-permute [simp]:  
  shows bij (permute p)  
   $\langle$ proof $\rangle$ 
```

```
lemma inv-permute:  
  shows inv (permute p) = permute (- p)  
   $\langle$ proof $\rangle$ 
```

```
lemma permute-minus:  
  shows permute (- p) = inv (permute p)  
   $\langle$ proof $\rangle$ 
```

```
lemma permute-eq-iff [simp]:  
  shows  $p \cdot x = p \cdot y \iff x = y$ 
```

<proof>

end

4.1 Permutations for atoms

instantiation *atom* :: *pt*

begin

definition

$$p \cdot a = (\text{Rep-perm } p) a$$

instance

<proof>

end

lemma *sort-of-permute* [*simp*]:

shows *sort-of* ($p \cdot a$) = *sort-of* a

<proof>

lemma *swap-atom*:

shows ($a \rightleftharpoons b$) \cdot $c =$

(*if* *sort-of* $a =$ *sort-of* b

then (*if* $c = a$ *then* b *else if* $c = b$ *then* a *else* c) *else* c)

<proof>

lemma *swap-atom-simps* [*simp*]:

sort-of $a =$ *sort-of* $b \implies (a \rightleftharpoons b) \cdot a = b$

sort-of $a =$ *sort-of* $b \implies (a \rightleftharpoons b) \cdot b = a$

$c \neq a \implies c \neq b \implies (a \rightleftharpoons b) \cdot c = c$

<proof>

lemma *perm-eq-iff*:

fixes p q :: *perm*

shows $p = q \iff (\forall a::\text{atom}. p \cdot a = q \cdot a)$

<proof>

4.2 Permutations for permutations

instantiation *perm* :: *pt*

begin

definition

$$p \cdot q = p + q - p$$

instance

<proof>

end

lemma *permute-self*:

shows $p \cdot p = p$

\langle *proof* \rangle

lemma *permute-minus-self*:

shows $- p \cdot p = p$

\langle *proof* \rangle

4.3 Permutations for functions

instantiation *fun* :: $(pt, pt) \Rightarrow pt$

begin

definition

$p \cdot f = (\lambda x. p \cdot (f (- p \cdot x)))$

instance

\langle *proof* \rangle

end

lemma *permute-fun-app-eq*:

shows $p \cdot (f x) = (p \cdot f) (p \cdot x)$

\langle *proof* \rangle

lemma *permute-fun-comp*:

shows $p \cdot f = (permute\ p) \circ f \circ (permute\ (-p))$

\langle *proof* \rangle

4.4 Permutations for booleans

instantiation *bool* :: pt

begin

definition $p \cdot (b::bool) = b$

instance

\langle *proof* \rangle

end

lemma *permute-boolE*:

fixes $P::bool$

shows $p \cdot P \implies P$

\langle *proof* \rangle

lemma *permute-boolI*:

fixes $P::bool$

shows $P \implies p \cdot P$

<proof>

4.5 Permutations for sets

instantiation *set* :: (*pt*) *pt*
begin

definition

$p \cdot X = \{p \cdot x \mid x. x \in X\}$

instance

<proof>

end

lemma *permute-set-eq*:

shows $p \cdot X = \{x. \neg p \cdot x \in X\}$

<proof>

lemma *permute-set-eq-image*:

shows $p \cdot X = \text{permute } p \text{ ` } X$

<proof>

lemma *permute-set-eq-vimage*:

shows $p \cdot X = \text{permute } (\neg p) \text{ - ` } X$

<proof>

lemma *permute-finite* [*simp*]:

shows $\text{finite } (p \cdot X) = \text{finite } X$

<proof>

lemma *swap-set-not-in*:

assumes $a \notin S \ b \notin S$

shows $(a \rightleftharpoons b) \cdot S = S$

<proof>

lemma *swap-set-in*:

assumes $a \in S \ b \notin S \ \text{sort-of } a = \text{sort-of } b$

shows $(a \rightleftharpoons b) \cdot S \neq S$

<proof>

lemma *swap-set-in-eq*:

assumes $a \in S \ b \notin S \ \text{sort-of } a = \text{sort-of } b$

shows $(a \rightleftharpoons b) \cdot S = (S - \{a\}) \cup \{b\}$

<proof>

lemma *swap-set-both-in*:

assumes $a \in S \ b \in S$

shows $(a \rightleftharpoons b) \cdot S = S$

<proof>

lemma *mem-permute-iff*:

shows $(p \cdot x) \in (p \cdot X) \longleftrightarrow x \in X$

<proof>

lemma *empty-eqt*:

shows $p \cdot \{\} = \{\}$

<proof>

lemma *insert-eqt*:

shows $p \cdot (\text{insert } x \ A) = \text{insert } (p \cdot x) \ (p \cdot A)$

<proof>

4.6 Permutations for *unit*

instantiation *unit* :: *pt*

begin

definition $p \cdot (u::\text{unit}) = u$

instance

<proof>

end

4.7 Permutations for products

instantiation *prod* :: (*pt*, *pt*) *pt*

begin

primrec

permute-prod

where

Pair-eqt: $p \cdot (x, y) = (p \cdot x, p \cdot y)$

instance

<proof>

end

4.8 Permutations for sums

instantiation *sum* :: (*pt*, *pt*) *pt*

begin

primrec

permute-sum

where

Inl-eqt: $p \cdot (\text{Inl } x) = \text{Inl } (p \cdot x)$

| *Inr-eqvt*: $p \cdot (\text{Inr } y) = \text{Inr } (p \cdot y)$

instance

<proof>

end

4.9 Permutations for 'a list

instantiation *list* :: (pt) pt

begin

primrec

permute-list

where

Nil-eqvt: $p \cdot [] = []$

| *Cons-eqvt*: $p \cdot (x \# xs) = p \cdot x \# p \cdot xs$

instance

<proof>

end

lemma *set-eqvt*:

shows $p \cdot (\text{set } xs) = \text{set } (p \cdot xs)$

<proof>

4.10 Permutations for 'a option

instantiation *option* :: (pt) pt

begin

primrec

permute-option

where

None-eqvt: $p \cdot \text{None} = \text{None}$

| *Some-eqvt*: $p \cdot (\text{Some } x) = \text{Some } (p \cdot x)$

instance

<proof>

end

4.11 Permutations for 'a multiset

instantiation *multiset* :: (pt) pt

begin

definition

$p \cdot M = \{\# p \cdot x. x \# M \#\}$

```

instance
  ⟨proof⟩

end

lemma permutate-multiset [simp]:
  fixes  $M\ N::('a::pt)\ multiset$ 
  shows  $(p \cdot \{\#\}) = (\{\#\} ::('a::pt)\ multiset)$ 
  and  $(p \cdot \text{add-mset } x\ M) = \text{add-mset } (p \cdot x)\ (p \cdot M)$ 
  and  $(p \cdot (M + N)) = (p \cdot M) + (p \cdot N)$ 
  ⟨proof⟩

```

4.12 Permutations for 'a fset

```

instantiation fset :: (pt) pt
begin

context includes fset.lifting begin
lift-definition
  permutate-fset :: perm ⇒ 'a fset ⇒ 'a fset
is permutate :: perm ⇒ 'a set ⇒ 'a set ⟨proof⟩
end

```

```

context includes fset.lifting begin
instance
  ⟨proof⟩
end

```

end

```

context includes fset.lifting
begin
lemma permutate-fset [simp]:
  fixes  $S::('a::pt)\ fset$ 
  shows  $(p \cdot \{\|\}) = (\{\|\} ::('a::pt)\ fset)$ 
  and  $(p \cdot \text{finsert } x\ S) = \text{finsert } (p \cdot x)\ (p \cdot S)$ 
  ⟨proof⟩

```

```

lemma fset-eqvt:
  shows  $p \cdot (\text{fset } S) = \text{fset } (p \cdot S)$ 
  ⟨proof⟩
end

```

4.13 Permutations for ('a, 'b) finfun

```

instantiation finfun :: (pt, pt) pt
begin

```

```

lift-definition

```

```

    permute-funfun :: perm ⇒ ('a, 'b) funfun ⇒ ('a, 'b) funfun
  is
    permute :: perm ⇒ ('a ⇒ 'b) ⇒ ('a ⇒ 'b)
    ⟨proof⟩

instance
  ⟨proof⟩

end

```

4.14 Permutations for *char*, *nat*, and *int*

```

instantiation char :: pt
begin

definition p · (c::char) = c

instance
  ⟨proof⟩

end

instantiation nat :: pt
begin

definition p · (n::nat) = n

instance
  ⟨proof⟩

end

instantiation int :: pt
begin

definition p · (i::int) = i

instance
  ⟨proof⟩

end

```

5 Pure types

Pure types will have always empty support.

```

class pure = pt +
  assumes permute-pure: p · x = x

  Types unit and bool are pure.

```

instance *unit* :: *pure*
⟨*proof*⟩

instance *bool* :: *pure*
⟨*proof*⟩

Other type constructors preserve purity.

instance *fun* :: (*pure*, *pure*) *pure*
⟨*proof*⟩

instance *set* :: (*pure*) *pure*
⟨*proof*⟩

instance *prod* :: (*pure*, *pure*) *pure*
⟨*proof*⟩

instance *sum* :: (*pure*, *pure*) *pure*
⟨*proof*⟩

instance *list* :: (*pure*) *pure*
⟨*proof*⟩

instance *option* :: (*pure*) *pure*
⟨*proof*⟩

5.1 Types *char*, *nat*, and *int*

instance *char* :: *pure*
⟨*proof*⟩

instance *nat* :: *pure*
⟨*proof*⟩

instance *int* :: *pure*
⟨*proof*⟩

6 Infrastructure for Equivariance and *Perm-simp*

6.1 Basic functions about permutations

⟨*ML*⟩

6.2 Eqvt infrastructure

Setup of the theorem attributes *eqvt* and *eqvt-raw*.

⟨*ML*⟩

lemmas [*eqvt*] =

permute-prod.simps
permute-list.simps
permute-option.simps
permute-sum.simps

empty-eqvt insert-eqvt set-eqvt

permute-fset fset-eqvt

permute-multiset

6.3 *perm-simp* infrastructure

definition

unpermute $p = \text{permute } (- \ p)$

lemma *eqvt-apply*:

fixes $f :: 'a::pt \Rightarrow 'b::pt$
and $x :: 'a::pt$
shows $p \cdot (f \ x) \equiv (p \cdot f) \ (p \cdot x)$
<proof>

lemma *eqvt-lambda*:

fixes $f :: 'a::pt \Rightarrow 'b::pt$
shows $p \cdot f \equiv (\lambda x. \ p \cdot (f \ (\text{unpermute } p \ x)))$
<proof>

lemma *eqvt-bound*:

shows $p \cdot \text{unpermute } p \ x \equiv x$
<proof>

provides *perm-simp* methods

<ML>

6.3.1 Equivariance for permutations and swapping

lemma *permute-eqvt*:

shows $p \cdot (q \cdot x) = (p \cdot q) \cdot (p \cdot x)$
<proof>

lemma *permute-eqvt-raw* [*eqvt-raw*]:

shows $p \cdot \text{permute} \equiv \text{permute}$
<proof>

lemma *zero-perm-eqvt* [*eqvt*]:

shows $p \cdot (0::perm) = 0$
<proof>

lemma *add-perm-eqvt* [eqvt]:
fixes $p\ p1\ p2 :: perm$
shows $p \cdot (p1 + p2) = p \cdot p1 + p \cdot p2$
<proof>

lemma *swap-eqvt* [eqvt]:
shows $p \cdot (a \rightleftharpoons b) = (p \cdot a \rightleftharpoons p \cdot b)$
<proof>

lemma *uminus-eqvt* [eqvt]:
fixes $p\ q::perm$
shows $p \cdot (-\ q) = -(p \cdot q)$
<proof>

6.3.2 Equivariance of Logical Operators

lemma *eq-eqvt* [eqvt]:
shows $p \cdot (x = y) \longleftrightarrow (p \cdot x) = (p \cdot y)$
<proof>

lemma *Not-eqvt* [eqvt]:
shows $p \cdot (\neg A) \longleftrightarrow \neg (p \cdot A)$
<proof>

lemma *conj-eqvt* [eqvt]:
shows $p \cdot (A \wedge B) \longleftrightarrow (p \cdot A) \wedge (p \cdot B)$
<proof>

lemma *imp-eqvt* [eqvt]:
shows $p \cdot (A \longrightarrow B) \longleftrightarrow (p \cdot A) \longrightarrow (p \cdot B)$
<proof>

declare *imp-eqvt*[folded *HOL.induct-implies-def*, eqvt]

lemma *all-eqvt* [eqvt]:
shows $p \cdot (\forall x. P\ x) = (\forall x. (p \cdot P)\ x)$
<proof>

declare *all-eqvt*[folded *HOL.induct-forall-def*, eqvt]

lemma *ex-eqvt* [eqvt]:
shows $p \cdot (\exists x. P\ x) = (\exists x. (p \cdot P)\ x)$
<proof>

lemma *ex1-eqvt* [eqvt]:
shows $p \cdot (\exists!x. P\ x) = (\exists!x. (p \cdot P)\ x)$

$\langle proof \rangle$

lemma *if-eqvt* [eqvt]:

shows $p \cdot (\text{if } b \text{ then } x \text{ else } y) = (\text{if } p \cdot b \text{ then } p \cdot x \text{ else } p \cdot y)$
 $\langle proof \rangle$

lemma *True-eqvt* [eqvt]:

shows $p \cdot \text{True} = \text{True}$
 $\langle proof \rangle$

lemma *False-eqvt* [eqvt]:

shows $p \cdot \text{False} = \text{False}$
 $\langle proof \rangle$

lemma *disj-eqvt* [eqvt]:

shows $p \cdot (A \vee B) \longleftrightarrow (p \cdot A) \vee (p \cdot B)$
 $\langle proof \rangle$

lemma *all-eqvt2*:

shows $p \cdot (\forall x. P x) = (\forall x. p \cdot P (- p \cdot x))$
 $\langle proof \rangle$

lemma *ex-eqvt2*:

shows $p \cdot (\exists x. P x) = (\exists x. p \cdot P (- p \cdot x))$
 $\langle proof \rangle$

lemma *ex1-eqvt2*:

shows $p \cdot (\exists!x. P x) = (\exists!x. p \cdot P (- p \cdot x))$
 $\langle proof \rangle$

lemma *the-eqvt*:

assumes *unique*: $\exists!x. P x$
shows $(p \cdot (\text{THE } x. P x)) = (\text{THE } x. (p \cdot P) x)$
 $\langle proof \rangle$

lemma *the-eqvt2*:

assumes *unique*: $\exists!x. P x$
shows $(p \cdot (\text{THE } x. P x)) = (\text{THE } x. p \cdot P (- p \cdot x))$
 $\langle proof \rangle$

6.3.3 Equivariance of Set operators

lemma *mem-eqvt* [eqvt]:

shows $p \cdot (x \in A) \longleftrightarrow (p \cdot x) \in (p \cdot A)$
 $\langle proof \rangle$

lemma *Collect-eqvt* [eqvt]:

shows $p \cdot \{x. P x\} = \{x. (p \cdot P) x\}$
 $\langle proof \rangle$

lemma *Bex-eqvt* [eqvt]:
shows $p \cdot (\exists x \in S. P x) = (\exists x \in (p \cdot S). (p \cdot P) x)$
 ⟨proof⟩

lemma *Ball-eqvt* [eqvt]:
shows $p \cdot (\forall x \in S. P x) = (\forall x \in (p \cdot S). (p \cdot P) x)$
 ⟨proof⟩

lemma *image-eqvt* [eqvt]:
shows $p \cdot (f \cdot A) = (p \cdot f) \cdot (p \cdot A)$
 ⟨proof⟩

lemma *Image-eqvt* [eqvt]:
shows $p \cdot (R \cdot A) = (p \cdot R) \cdot (p \cdot A)$
 ⟨proof⟩

lemma *UNIV-eqvt* [eqvt]:
shows $p \cdot UNIV = UNIV$
 ⟨proof⟩

lemma *inter-eqvt* [eqvt]:
shows $p \cdot (A \cap B) = (p \cdot A) \cap (p \cdot B)$
 ⟨proof⟩

lemma *Inter-eqvt* [eqvt]:
shows $p \cdot \bigcap S = \bigcap (p \cdot S)$
 ⟨proof⟩

lemma *union-eqvt* [eqvt]:
shows $p \cdot (A \cup B) = (p \cdot A) \cup (p \cdot B)$
 ⟨proof⟩

lemma *Union-eqvt* [eqvt]:
shows $p \cdot \bigcup A = \bigcup (p \cdot A)$
 ⟨proof⟩

lemma *Diff-eqvt* [eqvt]:
fixes $A B :: 'a::pt\ set$
shows $p \cdot (A - B) = (p \cdot A) - (p \cdot B)$
 ⟨proof⟩

lemma *Compl-eqvt* [eqvt]:
fixes $A :: 'a::pt\ set$
shows $p \cdot (- A) = - (p \cdot A)$
 ⟨proof⟩

lemma *subset-eqvt* [eqvt]:
shows $p \cdot (S \subseteq T) \longleftrightarrow (p \cdot S) \subseteq (p \cdot T)$

<proof>

lemma *psubset-eqvt* [*eqvt*]:
shows $p \cdot (S \subset T) \longleftrightarrow (p \cdot S) \subset (p \cdot T)$
<proof>

lemma *vimage-eqvt* [*eqvt*]:
shows $p \cdot (f -' A) = (p \cdot f) -' (p \cdot A)$
<proof>

lemma *foldr-eqvt*[*eqvt*]:
 $p \cdot \text{foldr } f \text{ } xs = \text{foldr } (p \cdot f) (p \cdot xs)$
<proof>

lemma *Sigma-eqvt*:
shows $(p \cdot (X \times Y)) = (p \cdot X) \times (p \cdot Y)$
<proof>

In order to prove that *lfp* is equivariant we need two auxiliary classes which specify that (*<=>*) and *Inf* are equivariant. Instances for *bool* and *fun* are given.

class *le-eqvt* = *pt* +
assumes *le-eqvt* [*eqvt*]: $p \cdot (x \leq y) = ((p \cdot x) \leq (p \cdot (y :: 'a :: \{order, pt\})))$

class *inf-eqvt* = *pt* +
assumes *inf-eqvt* [*eqvt*]: $p \cdot (\text{Inf } X) = \text{Inf } (p \cdot (X :: 'a :: \{complete-lattice, pt\} \text{ set}))$

instantiation *bool* :: *le-eqvt*
begin

instance
<proof>

end

instantiation *fun* :: (*pt*, *le-eqvt*) *le-eqvt*
begin

instance
<proof>

end

instantiation *bool* :: *inf-eqvt*
begin

instance

<proof>

end

instantiation *fun* :: (*pt*, *inf-eqvt*) *inf-eqvt*
begin

instance

<proof>

end

lemma *lfp-eqvt* [*eqvt*]:

fixes $F :: ('a \Rightarrow 'b) \Rightarrow ('a :: pt \Rightarrow 'b :: \{inf-eqvt, le-eqvt\})$

shows $p \cdot (lfp F) = lfp (p \cdot F)$

<proof>

lemma *finite-eqvt* [*eqvt*]:

shows $p \cdot finite A = finite (p \cdot A)$

<proof>

lemma *fun-upd-eqvt* [*eqvt*]:

shows $p \cdot (f(x := y)) = (p \cdot f)((p \cdot x) := (p \cdot y))$

<proof>

lemma *comp-eqvt* [*eqvt*]:

shows $p \cdot (f \circ g) = (p \cdot f) \circ (p \cdot g)$

<proof>

6.3.4 Equivariance for product operations

lemma *fst-eqvt* [*eqvt*]:

shows $p \cdot (fst x) = fst (p \cdot x)$

<proof>

lemma *snd-eqvt* [*eqvt*]:

shows $p \cdot (snd x) = snd (p \cdot x)$

<proof>

lemma *split-eqvt* [*eqvt*]:

shows $p \cdot (case-prod P x) = case-prod (p \cdot P) (p \cdot x)$

<proof>

6.3.5 Equivariance for list operations

lemma *append-eqvt* [*eqvt*]:

shows $p \cdot (xs @ ys) = (p \cdot xs) @ (p \cdot ys)$

<proof>

lemma *rev-eqvt* [*eqvt*]:

shows $p \cdot (\text{rev } xs) = \text{rev } (p \cdot xs)$
<proof>

lemma *map-eqvt* [eqvt]:
shows $p \cdot (\text{map } f \text{ } xs) = \text{map } (p \cdot f) (p \cdot xs)$
<proof>

lemma *removeAll-eqvt* [eqvt]:
shows $p \cdot (\text{removeAll } x \text{ } xs) = \text{removeAll } (p \cdot x) (p \cdot xs)$
<proof>

lemma *filter-eqvt* [eqvt]:
shows $p \cdot (\text{filter } f \text{ } xs) = \text{filter } (p \cdot f) (p \cdot xs)$
<proof>

lemma *distinct-eqvt* [eqvt]:
shows $p \cdot (\text{distinct } xs) = \text{distinct } (p \cdot xs)$
<proof>

lemma *length-eqvt* [eqvt]:
shows $p \cdot (\text{length } xs) = \text{length } (p \cdot xs)$
<proof>

6.3.6 Equivariance for 'a option

lemma *map-option-eqvt*[eqvt]:
shows $p \cdot (\text{map-option } f \text{ } x) = \text{map-option } (p \cdot f) (p \cdot x)$
<proof>

6.3.7 Equivariance for 'a fset

context includes *fset.lifting* **begin**

lemma *in-fset-eqvt*:
shows $(p \cdot (x \mid \in \mid S)) = ((p \cdot x) \mid \in \mid (p \cdot S))$
<proof>

lemma *union-fset-eqvt* [eqvt]:
shows $(p \cdot (S \mid \cup \mid T)) = ((p \cdot S) \mid \cup \mid (p \cdot T))$
<proof>

lemma *inter-fset-eqvt* [eqvt]:
shows $(p \cdot (S \mid \cap \mid T)) = ((p \cdot S) \mid \cap \mid (p \cdot T))$
<proof>

lemma *subset-fset-eqvt* [eqvt]:
shows $(p \cdot (S \mid \subseteq \mid T)) = ((p \cdot S) \mid \subseteq \mid (p \cdot T))$
<proof>

lemma *map-fset-eqvt* [eqvt]:
shows $p \cdot (f \mid \uparrow \mid S) = (p \cdot f) \mid \uparrow \mid (p \cdot S)$

<proof>
end

6.3.8 Equivariance for (*'a, 'b*) *finfun*

lemma *finfun-update-eqv* [*eqvt*]:
 shows $(p \cdot (\text{finfun-update } f \ a \ b)) = \text{finfun-update } (p \cdot f) \ (p \cdot a) \ (p \cdot b)$
<proof>

lemma *finfun-const-eqv* [*eqvt*]:
 shows $(p \cdot (\text{finfun-const } b)) = \text{finfun-const } (p \cdot b)$
<proof>

lemma *finfun-apply-eqv* [*eqvt*]:
 shows $(p \cdot (\text{finfun-apply } f \ b)) = \text{finfun-apply } (p \cdot f) \ (p \cdot b)$
<proof>

7 Supp, Freshness and Supports

context *pt*
begin

definition
 supp :: *'a* \Rightarrow *atom set*
where
 supp *x* = $\{a. \text{infinite } \{b. (a \rightleftharpoons b) \cdot x \neq x\}\}$

definition
 fresh :: *atom* \Rightarrow *'a* \Rightarrow *bool* (*↔* $\#$ \rightarrow [55, 55] 55)
where
 a $\#$ *x* \equiv $a \notin \text{supp } x$

end

lemma *supp-conv-fresh*:
 shows $\text{supp } x = \{a. \neg a \# x\}$
<proof>

lemma *swap-rel-trans*:
 assumes *sort-of* *a* = *sort-of* *b*
 assumes *sort-of* *b* = *sort-of* *c*
 assumes $(a \rightleftharpoons c) \cdot x = x$
 assumes $(b \rightleftharpoons c) \cdot x = x$
 shows $(a \rightleftharpoons b) \cdot x = x$
<proof>

lemma *swap-fresh-fresh*:
 assumes *a*: *a* $\#$ *x*
 and *b*: *b* $\#$ *x*

shows $(a \rightleftharpoons b) \cdot x = x$
<proof>

7.1 supp and fresh are equivariant

lemma *supp-eqvt* [*eqvt*]:
shows $p \cdot (\text{supp } x) = \text{supp } (p \cdot x)$
<proof>

lemma *fresh-eqvt* [*eqvt*]:
shows $p \cdot (a \# x) = (p \cdot a) \# (p \cdot x)$
<proof>

lemma *fresh-permute-iff*:
shows $(p \cdot a) \# (p \cdot x) \longleftrightarrow a \# x$
<proof>

lemma *fresh-permute-left*:
shows $a \# p \cdot x \longleftrightarrow - p \cdot a \# x$
<proof>

8 supports

definition
supports :: *atom set* \Rightarrow *'a::pt* \Rightarrow *bool* (**infixl** *<supports>* 80)
where
 $S \text{ supports } x \equiv \forall a b. (a \notin S \wedge b \notin S \longrightarrow (a \rightleftharpoons b) \cdot x = x)$

lemma *supp-is-subset*:
fixes $S :: \text{atom set}$
and $x :: \text{'a::pt}$
assumes $a1: S \text{ supports } x$
and $a2: \text{finite } S$
shows $(\text{supp } x) \subseteq S$
<proof>

lemma *supports-finite*:
fixes $S :: \text{atom set}$
and $x :: \text{'a::pt}$
assumes $a1: S \text{ supports } x$
and $a2: \text{finite } S$
shows $\text{finite } (\text{supp } x)$
<proof>

lemma *supp-supports*:
fixes $x :: \text{'a::pt}$
shows $(\text{supp } x) \text{ supports } x$
<proof>

lemma *supports-fresh*:
fixes $x :: 'a::pt$
assumes $a1: S \text{ supports } x$
and $a2: \text{finite } S$
and $a3: a \notin S$
shows $a \# x$
 $\langle \text{proof} \rangle$

lemma *supp-is-least-supports*:
fixes $S :: \text{atom set}$
and $x :: 'a::pt$
assumes $a1: S \text{ supports } x$
and $a2: \text{finite } S$
and $a3: \bigwedge S'. \text{finite } S' \implies (S' \text{ supports } x) \implies S \subseteq S'$
shows $(\text{supp } x) = S$
 $\langle \text{proof} \rangle$

lemma *subsetCI*:
shows $(\bigwedge x. x \in A \implies x \notin B \implies \text{False}) \implies A \subseteq B$
 $\langle \text{proof} \rangle$

lemma *finite-supp-unique*:
assumes $a1: S \text{ supports } x$
assumes $a2: \text{finite } S$
assumes $a3: \bigwedge a b. \llbracket a \in S; b \notin S; \text{sort-of } a = \text{sort-of } b \rrbracket \implies (a \rightleftharpoons b) \cdot x \neq x$
shows $(\text{supp } x) = S$
 $\langle \text{proof} \rangle$

9 Support w.r.t. relations

This definition is used for unquotient types, where alpha-equivalence does not coincide with equality.

definition
 $\text{supp-rel } R \ x = \{a. \text{infinite } \{b. \neg(R ((a \rightleftharpoons b) \cdot x) x)\}\}$

10 Finitely-supported types

class $fs = pt +$
assumes $\text{finite-supp}: \text{finite } (\text{supp } x)$

lemma *pure-supp*:
fixes $x::'a::\text{pure}$
shows $\text{supp } x = \{\}$
 $\langle \text{proof} \rangle$

lemma *pure-fresh*:
fixes $x::'a::\text{pure}$

shows $a \# x$
 $\langle proof \rangle$

instance $pure < fs$
 $\langle proof \rangle$

10.1 Type *atom* is finitely-supported.

lemma *supp-atom*:
shows $supp\ a = \{a\}$
 $\langle proof \rangle$

lemma *fresh-atom*:
shows $a \# b \longleftrightarrow a \neq b$
 $\langle proof \rangle$

instance $atom :: fs$
 $\langle proof \rangle$

11 Type *perm* is finitely-supported.

lemma *perm-swap-eq*:
shows $(a \rightleftharpoons b) \cdot p = p \longleftrightarrow (p \cdot (a \rightleftharpoons b)) = (a \rightleftharpoons b)$
 $\langle proof \rangle$

lemma *supports-perm*:
shows $\{a. p \cdot a \neq a\}$ supports p
 $\langle proof \rangle$

lemma *finite-perm-lemma*:
shows *finite* $\{a::atom. p \cdot a \neq a\}$
 $\langle proof \rangle$

lemma *supp-perm*:
shows $supp\ p = \{a. p \cdot a \neq a\}$
 $\langle proof \rangle$

lemma *fresh-perm*:
shows $a \# p \longleftrightarrow p \cdot a = a$
 $\langle proof \rangle$

lemma *supp-swap*:
shows $supp\ (a \rightleftharpoons b) = (if\ a = b \vee sort-of\ a \neq sort-of\ b\ then\ \{\}\ else\ \{a, b\})$
 $\langle proof \rangle$

lemma *fresh-swap*:
shows $a \# (b \rightleftharpoons c) \longleftrightarrow (sort-of\ b \neq sort-of\ c) \vee b = c \vee (a \# b \wedge a \# c)$
 $\langle proof \rangle$

lemma *fresh-zero-perm*:
shows $a \# (0::perm)$
 $\langle proof \rangle$

lemma *supp-zero-perm*:
shows $supp (0::perm) = \{\}$
 $\langle proof \rangle$

lemma *fresh-plus-perm*:
fixes $p q::perm$
assumes $a \# p \ a \# q$
shows $a \# (p + q)$
 $\langle proof \rangle$

lemma *supp-plus-perm*:
fixes $p q::perm$
shows $supp (p + q) \subseteq supp\ p \cup supp\ q$
 $\langle proof \rangle$

lemma *fresh-minus-perm*:
fixes $p::perm$
shows $a \# (-\ p) \longleftrightarrow a \# p$
 $\langle proof \rangle$

lemma *supp-minus-perm*:
fixes $p::perm$
shows $supp (-\ p) = supp\ p$
 $\langle proof \rangle$

lemma *plus-perm-eq*:
fixes $p q::perm$
assumes $asm: supp\ p \cap supp\ q = \{\}$
shows $p + q = q + p$
 $\langle proof \rangle$

lemma *supp-plus-perm-eq*:
fixes $p q::perm$
assumes $asm: supp\ p \cap supp\ q = \{\}$
shows $supp (p + q) = supp\ p \cup supp\ q$
 $\langle proof \rangle$

lemma *perm-eq-iff2*:
fixes $p\ q :: perm$
shows $p = q \longleftrightarrow (\forall a::atom \in supp\ p \cup supp\ q. p \cdot a = q \cdot a)$
 $\langle proof \rangle$

instance *perm* :: *fs*
 $\langle proof \rangle$

12 Finite Support instances for other types

12.1 Type $'a \times 'b$ is finitely-supported.

lemma *supp-Pair*:

shows $\text{supp } (x, y) = \text{supp } x \cup \text{supp } y$
<proof>

lemma *fresh-Pair*:

shows $a \# (x, y) \longleftrightarrow a \# x \wedge a \# y$
<proof>

lemma *supp-Unit*:

shows $\text{supp } () = \{\}$
<proof>

lemma *fresh-Unit*:

shows $a \# ()$
<proof>

instance *prod* :: $(fs, fs) fs$

<proof>

12.2 Type $'a + 'b$ is finitely supported

lemma *supp-Inl*:

shows $\text{supp } (\text{Inl } x) = \text{supp } x$
<proof>

lemma *supp-Inr*:

shows $\text{supp } (\text{Inr } x) = \text{supp } x$
<proof>

lemma *fresh-Inl*:

shows $a \# \text{Inl } x \longleftrightarrow a \# x$
<proof>

lemma *fresh-Inr*:

shows $a \# \text{Inr } y \longleftrightarrow a \# y$
<proof>

instance *sum* :: $(fs, fs) fs$

<proof>

12.3 Type $'a \text{ option}$ is finitely supported

lemma *supp-None*:

shows $\text{supp } \text{None} = \{\}$
<proof>

lemma *supp-Some*:
shows $\text{supp } (\text{Some } x) = \text{supp } x$
<proof>

lemma *fresh-None*:
shows $a \# \text{None}$
<proof>

lemma *fresh-Some*:
shows $a \# \text{Some } x \longleftrightarrow a \# x$
<proof>

instance *option* :: (fs) fs
<proof>

12.3.1 Type 'a list is finitely supported

lemma *supp-Nil*:
shows $\text{supp } [] = \{\}$
<proof>

lemma *fresh-Nil*:
shows $a \# []$
<proof>

lemma *supp-Cons*:
shows $\text{supp } (x \# xs) = \text{supp } x \cup \text{supp } xs$
<proof>

lemma *fresh-Cons*:
shows $a \# (x \# xs) \longleftrightarrow a \# x \wedge a \# xs$
<proof>

lemma *supp-append*:
shows $\text{supp } (xs @ ys) = \text{supp } xs \cup \text{supp } ys$
<proof>

lemma *fresh-append*:
shows $a \# (xs @ ys) \longleftrightarrow a \# xs \wedge a \# ys$
<proof>

lemma *supp-rev*:
shows $\text{supp } (\text{rev } xs) = \text{supp } xs$
<proof>

lemma *fresh-rev*:
shows $a \# \text{rev } xs \longleftrightarrow a \# xs$
<proof>

lemma *supp-removeAll*:
fixes $x::atom$
shows $supp (removeAll\ x\ xs) = supp\ xs - \{x\}$
 $\langle proof \rangle$

lemma *supp-of-atom-list*:
fixes $as::atom\ list$
shows $supp\ as = set\ as$
 $\langle proof \rangle$

instance $list :: (fs)\ fs$
 $\langle proof \rangle$

13 Support and Freshness for Applications

lemma *fresh-conv-MOST*:
shows $a \# x \longleftrightarrow (MOST\ b.\ (a \rightleftharpoons b) \cdot x = x)$
 $\langle proof \rangle$

lemma *fresh-fun-app*:
assumes $a \# f$ **and** $a \# x$
shows $a \# f\ x$
 $\langle proof \rangle$

lemma *supp-fun-app*:
shows $supp\ (f\ x) \subseteq (supp\ f) \cup (supp\ x)$
 $\langle proof \rangle$

13.1 Equivariance Predicate *eqvt* and *eqvt-at*

definition
 $eqvt\ f \equiv \forall p.\ p \cdot f = f$

lemma *eqvt-boolI*:
fixes $f::bool$
shows $eqvt\ f$
 $\langle proof \rangle$

equivariance of a function at a given argument

definition
 $eqvt-at\ f\ x \equiv \forall p.\ p \cdot (f\ x) = f\ (p \cdot x)$

lemma *eqvtI*:
shows $(\bigwedge p.\ p \cdot f \equiv f) \implies eqvt\ f$
 $\langle proof \rangle$

lemma *eqvt-at-perm*:
assumes $eqvt-at\ f\ x$
shows $eqvt-at\ f\ (q \cdot x)$

<proof>

lemma *supp-fun-eqvt*:

assumes *a*: *eqvt f*
shows $\text{supp } f = \{\}$
<proof>

lemma *fresh-fun-eqvt*:

assumes *a*: *eqvt f*
shows $a \# f$
<proof>

lemma *fresh-fun-eqvt-app*:

assumes *a*: *eqvt f*
shows $a \# x \implies a \# f x$
<proof>

lemma *supp-fun-app-eqvt*:

assumes *a*: *eqvt f*
shows $\text{supp } (f x) \subseteq \text{supp } x$
<proof>

lemma *supp-eqvt-at*:

assumes *asm*: *eqvt-at f x*
and *fin*: *finite (supp x)*
shows $\text{supp } (f x) \subseteq \text{supp } x$
<proof>

lemma *finite-supp-eqvt-at*:

assumes *asm*: *eqvt-at f x*
and *fin*: *finite (supp x)*
shows *finite (supp (f x))*
<proof>

lemma *fresh-eqvt-at*:

assumes *asm*: *eqvt-at f x*
and *fin*: *finite (supp x)*
and *fresh*: $a \# x$
shows $a \# f x$
<proof>

for handling of freshness of functions

<ML>

13.2 helper functions for *nominal-functions*

lemma *THE-defaultI2*:

assumes $\exists!x. P x \wedge x. P x \implies Q x$
shows $Q (\text{THE-default } d P)$
<proof>

lemma *the-default-eqvt*:
assumes *unique*: $\exists!x. P x$
shows $(p \cdot (\text{THE-default } d P)) = (\text{THE-default } (p \cdot d) (p \cdot P))$
 $\langle \text{proof} \rangle$

lemma *fundef-ex1-eqvt*:
fixes $x::'a::pt$
assumes *f-def*: $f == (\lambda x::'a. \text{THE-default } (d x) (G x))$
assumes *eqvt*: *eqvt* G
assumes *ex1*: $\exists!y. G x y$
shows $(p \cdot (f x)) = f (p \cdot x)$
 $\langle \text{proof} \rangle$

lemma *fundef-ex1-eqvt-at*:
fixes $x::'a::pt$
assumes *f-def*: $f == (\lambda x::'a. \text{THE-default } (d x) (G x))$
assumes *eqvt*: *eqvt* G
assumes *ex1*: $\exists!y. G x y$
shows *eqvt-at* $f x$
 $\langle \text{proof} \rangle$

lemma *fundef-ex1-prop*:
fixes $x::'a::pt$
assumes *f-def*: $f == (\lambda x::'a. \text{THE-default } (d x) (G x))$
assumes *P-all*: $\bigwedge x y. G x y \implies P x y$
assumes *ex1*: $\exists!y. G x y$
shows $P x (f x)$
 $\langle \text{proof} \rangle$

14 Support of Finite Sets of Finitely Supported Elements

support and freshness for atom sets

lemma *supp-finite-atom-set*:
fixes $S::\text{atom set}$
assumes *finite* S
shows *supp* $S = S$
 $\langle \text{proof} \rangle$

lemma *supp-cofinite-atom-set*:
fixes $S::\text{atom set}$
assumes *finite* $(UNIV - S)$
shows *supp* $S = (UNIV - S)$
 $\langle \text{proof} \rangle$

lemma *fresh-finite-atom-set*:
fixes $S::\text{atom set}$

assumes *finite S*
shows $a \# S \longleftrightarrow a \notin S$
<proof>

lemma *fresh-minus-atom-set*:
fixes $S::\text{atom set}$
assumes *finite S*
shows $a \# S - T \longleftrightarrow (a \notin T \longrightarrow a \# S)$
<proof>

lemma *Union-supports-set*:
shows $(\bigcup x \in S. \text{supp } x) \text{ supports } S$
<proof>

lemma *Union-of-finite-supp-sets*:
fixes $S::('a::\text{fs set})$
assumes *fin: finite S*
shows *finite* $(\bigcup x \in S. \text{supp } x)$
<proof>

lemma *Union-included-in-supp*:
fixes $S::('a::\text{fs set})$
assumes *fin: finite S*
shows $(\bigcup x \in S. \text{supp } x) \subseteq \text{supp } S$
<proof>

lemma *supp-of-finite-sets*:
fixes $S::('a::\text{fs set})$
assumes *fin: finite S*
shows $(\text{supp } S) = (\bigcup x \in S. \text{supp } x)$
<proof>

lemma *finite-sets-supp*:
fixes $S::('a::\text{fs set})$
assumes *finite S*
shows *finite* $(\text{supp } S)$
<proof>

lemma *supp-of-finite-union*:
fixes $S T::('a::\text{fs set})$
assumes *fin1: finite S*
and *fin2: finite T*
shows $\text{supp } (S \cup T) = \text{supp } S \cup \text{supp } T$
<proof>

lemma *fresh-finite-union*:
fixes $S T::('a::\text{fs set})$
assumes *fin1: finite S*
and *fin2: finite T*

shows $a \# (S \cup T) \longleftrightarrow a \# S \wedge a \# T$
<proof>

lemma *supp-of-finite-insert*:
fixes $S::('a::fs) \text{ set}$
assumes $fin: \text{finite } S$
shows $\text{supp } (\text{insert } x \ S) = \text{supp } x \cup \text{supp } S$
<proof>

lemma *fresh-finite-insert*:
fixes $S::('a::fs) \text{ set}$
assumes $fin: \text{finite } S$
shows $a \# (\text{insert } x \ S) \longleftrightarrow a \# x \wedge a \# S$
<proof>

lemma *supp-set-empty*:
shows $\text{supp } \{\} = \{\}$
<proof>

lemma *fresh-set-empty*:
shows $a \# \{\}$
<proof>

lemma *supp-set*:
fixes $xs :: ('a::fs) \text{ list}$
shows $\text{supp } (\text{set } xs) = \text{supp } xs$
<proof>

lemma *fresh-set*:
fixes $xs :: ('a::fs) \text{ list}$
shows $a \# (\text{set } xs) \longleftrightarrow a \# xs$
<proof>

14.1 Type 'a multiset is finitely supported

lemma *set-mset-eqvt [eqvt]*:
shows $p \cdot (\text{set-mset } M) = \text{set-mset } (p \cdot M)$
<proof>

lemma *supp-set-mset*:
shows $\text{supp } (\text{set-mset } M) \subseteq \text{supp } M$
<proof>

lemma *Union-finite-multiset*:
fixes $M::'a::fs \text{ multiset}$
shows $\text{finite } (\bigcup \{\text{supp } x \mid x. x \in\# M\})$
<proof>

lemma *Union-supports-multiset*:

shows $\bigcup \{ \text{supp } x \mid x. x \in \# M \}$ *supports* M
<proof>

lemma *Union-included-multiset*:
fixes $M :: ('a :: fs) \text{ multiset}$
shows $(\bigcup \{ \text{supp } x \mid x. x \in \# M \}) \subseteq \text{supp } M$
<proof>

lemma *supp-of-multisets*:
fixes $M :: ('a :: fs) \text{ multiset}$
shows $(\text{supp } M) = (\bigcup \{ \text{supp } x \mid x. x \in \# M \})$
<proof>

lemma *multisets-supp-finite*:
fixes $M :: ('a :: fs) \text{ multiset}$
shows *finite* $(\text{supp } M)$
<proof>

lemma *supp-of-multiset-union*:
fixes $M N :: ('a :: fs) \text{ multiset}$
shows $\text{supp } (M + N) = \text{supp } M \cup \text{supp } N$
<proof>

lemma *supp-empty-mset* [*simp*]:
shows $\text{supp } \{ \# \} = \{ \}$
<proof>

instance *multiset* :: $(fs) fs$
<proof>

14.2 Type $'a fset$ is finitely supported

lemma *supp-fset* [*simp*]:
shows $\text{supp } (fset S) = \text{supp } S$
<proof>

lemma *supp-empty-fset* [*simp*]:
shows $\text{supp } \{ \{ \} \} = \{ \}$
<proof>

lemma *fresh-empty-fset*:
shows $a \# \{ \{ \} \}$
<proof>

lemma *supp-finsert* [*simp*]:
fixes $x :: 'a :: fs$
and $S :: 'a fset$
shows $\text{supp } (finsert x S) = \text{supp } x \cup \text{supp } S$
<proof>

lemma *fresh-finsert*:
fixes $x::'a::fs$
and $S::'a\ fset$
shows $a \# \text{finsert } x\ S \longleftrightarrow a \# x \wedge a \# S$
 $\langle proof \rangle$

lemma *fset-finite-supp*:
fixes $S::('a::fs)\ fset$
shows $\text{finite } (\text{supp } S)$
 $\langle proof \rangle$

lemma *supp-union-fset*:
fixes $S\ T::'a::fs\ fset$
shows $\text{supp } (S \mid\cup\mid T) = \text{supp } S \cup \text{supp } T$
 $\langle proof \rangle$

lemma *fresh-union-fset*:
fixes $S\ T::'a::fs\ fset$
shows $a \# S \mid\cup\mid T \longleftrightarrow a \# S \wedge a \# T$
 $\langle proof \rangle$

instance *fset* :: $(fs)\ fs$
 $\langle proof \rangle$

14.3 Type $('a, 'b)\ \text{finfun}$ is finitely supported

lemma *fresh-finfun-const*:
shows $a \# (\text{finfun-const } b) \longleftrightarrow a \# b$
 $\langle proof \rangle$

lemma *fresh-finfun-update*:
shows $\llbracket a \# f; a \# x; a \# y \rrbracket \implies a \# \text{finfun-update } f\ x\ y$
 $\langle proof \rangle$

lemma *supp-finfun-const*:
shows $\text{supp } (\text{finfun-const } b) = \text{supp}(b)$
 $\langle proof \rangle$

lemma *supp-finfun-update*:
shows $\text{supp } (\text{finfun-update } f\ x\ y) \subseteq \text{supp}(f, x, y)$
 $\langle proof \rangle$

instance *finfun* :: $(fs, fs)\ fs$
 $\langle proof \rangle$

15 Freshness and Fresh-Star

lemma *fresh-Unit-elim*:

shows $(a \# () \implies PROP C) \equiv PROP C$
 $\langle proof \rangle$

lemma *fresh-Pair-elim*:

shows $(a \# (x, y) \implies PROP C) \equiv (a \# x \implies a \# y \implies PROP C)$
 $\langle proof \rangle$

lemma [*simp*]:

shows $a \# x1 \implies a \# x2 \implies a \# (x1, x2)$
 $\langle proof \rangle$

lemma *fresh-PairD*:

shows $a \# (x, y) \implies a \# x$
and $a \# (x, y) \implies a \# y$
 $\langle proof \rangle$

$\langle ML \rangle$

The fresh-star generalisation of fresh is used in strong induction principles.

definition

fresh-star :: *atom set* \Rightarrow *'a::pt* \Rightarrow *bool* ($\langle \cdot \#* \cdot \rangle [80,80] 80$)

where

$as \#* x \equiv \forall a \in as. a \# x$

lemma *fresh-star-supp-conv*:

shows $supp x \#* y \implies supp y \#* x$
 $\langle proof \rangle$

lemma *fresh-star-perm-set-conv*:

fixes *p::perm*
assumes *fresh*: $as \#* p$
and *fn*: *finite as*
shows $supp p \#* as$
 $\langle proof \rangle$

lemma *fresh-star-atom-set-conv*:

assumes *fresh*: $as \#* bs$
and *fn*: *finite as finite bs*
shows $bs \#* as$
 $\langle proof \rangle$

lemma *atom-fresh-star-disjoint*:

assumes *fn*: *finite bs*
shows $as \#* bs \longleftrightarrow (as \cap bs = \{\})$

$\langle proof \rangle$

lemma *fresh-star-Pair*:

shows $as \#* (x, y) = (as \#* x \wedge as \#* y)$
<proof>

lemma *fresh-star-list*:

shows $as \#* (xs @ ys) \longleftrightarrow as \#* xs \wedge as \#* ys$
and $as \#* (x \# xs) \longleftrightarrow as \#* x \wedge as \#* xs$
and $as \#* []$
<proof>

lemma *fresh-star-set*:

fixes $xs::('a::fs) list$
shows $as \#* set xs \longleftrightarrow as \#* xs$
<proof>

lemma *fresh-star-singleton*:

fixes $a::atom$
shows $as \#* \{a\} \longleftrightarrow as \#* a$
<proof>

lemma *fresh-star-fset*:

fixes $xs::('a::fs) list$
shows $as \#* fset S \longleftrightarrow as \#* S$
<proof>

lemma *fresh-star-Un*:

shows $(as \cup bs) \#* x = (as \#* x \wedge bs \#* x)$
<proof>

lemma *fresh-star-insert*:

shows $(insert a as) \#* x = (a \# x \wedge as \#* x)$
<proof>

lemma *fresh-star-Un-elim*:

$((as \cup bs) \#* x \Longrightarrow PROP C) \equiv (as \#* x \Longrightarrow bs \#* x \Longrightarrow PROP C)$
<proof>

lemma *fresh-star-insert-elim*:

$(insert a as \#* x \Longrightarrow PROP C) \equiv (a \# x \Longrightarrow as \#* x \Longrightarrow PROP C)$
<proof>

lemma *fresh-star-empty-elim*:

$(\{\} \#* x \Longrightarrow PROP C) \equiv PROP C$
<proof>

lemma *fresh-star-Unit-elim*:

shows $(a \#* ()) \Longrightarrow PROP C \equiv PROP C$

$\langle proof \rangle$

lemma *fresh-star-Pair-elim*:

shows $(a \#* (x, y) \Longrightarrow PROP C) \equiv (a \#* x \Longrightarrow a \#* y \Longrightarrow PROP C)$
 $\langle proof \rangle$

lemma *fresh-star-zero*:

shows $as \#* (0::perm)$
 $\langle proof \rangle$

lemma *fresh-star-plus*:

fixes $p q::perm$
shows $\llbracket a \#* p; a \#* q \rrbracket \Longrightarrow a \#* (p + q)$
 $\langle proof \rangle$

lemma *fresh-star-permute-iff*:

shows $(p \cdot a) \#* (p \cdot x) \longleftrightarrow a \#* x$
 $\langle proof \rangle$

lemma *fresh-star-eqvt* [eqvt]:

shows $p \cdot (as \#* x) \longleftrightarrow (p \cdot as) \#* (p \cdot x)$
 $\langle proof \rangle$

16 Induction principle for permutations

lemma *smaller-supp*:

assumes $a: a \in supp\ p$
shows $supp\ ((p \cdot a \rightleftharpoons a) + p) \subset supp\ p$
 $\langle proof \rangle$

lemma *perm-struct-induct*[consumes 1, case-names zero swap]:

assumes $S: supp\ p \subseteq S$
and $zero: P\ 0$
and $swap: \bigwedge p\ a\ b. \llbracket P\ p; supp\ p \subseteq S; a \in S; b \in S; a \neq b; sort-of\ a = sort-of\ b \rrbracket$
 $\Longrightarrow P\ ((a \rightleftharpoons b) + p)$
shows $P\ p$
 $\langle proof \rangle$

lemma *perm-simple-struct-induct*[case-names zero swap]:

assumes $zero: P\ 0$
and $swap: \bigwedge p\ a\ b. \llbracket P\ p; a \neq b; sort-of\ a = sort-of\ b \rrbracket \Longrightarrow P\ ((a \rightleftharpoons b) + p)$
shows $P\ p$
 $\langle proof \rangle$

lemma *perm-struct-induct2*[consumes 1, case-names zero swap plus]:

assumes $S: supp\ p \subseteq S$
assumes $zero: P\ 0$
assumes $swap: \bigwedge a\ b. \llbracket sort-of\ a = sort-of\ b; a \neq b; a \in S; b \in S \rrbracket \Longrightarrow P\ (a \rightleftharpoons$

b)
assumes *plus*: $\bigwedge p1\ p2. \llbracket P\ p1; P\ p2; \text{supp}\ p1 \subseteq S; \text{supp}\ p2 \subseteq S \rrbracket \implies P\ (p1 + p2)$
shows $P\ p$
 $\langle \text{proof} \rangle$

lemma *perm-simple-struct-induct2*[*case-names zero swap plus*]:
assumes *zero*: $P\ 0$
assumes *swap*: $\bigwedge a\ b. \llbracket \text{sort-of}\ a = \text{sort-of}\ b; a \neq b \rrbracket \implies P\ (a \rightleftharpoons b)$
assumes *plus*: $\bigwedge p1\ p2. \llbracket P\ p1; P\ p2 \rrbracket \implies P\ (p1 + p2)$
shows $P\ p$
 $\langle \text{proof} \rangle$

lemma *supp-perm-singleton*:
fixes $p::\text{perm}$
shows $\text{supp}\ p \subseteq \{b\} \longleftrightarrow p = 0$
 $\langle \text{proof} \rangle$

lemma *supp-perm-pair*:
fixes $p::\text{perm}$
shows $\text{supp}\ p \subseteq \{a, b\} \longleftrightarrow p = 0 \vee p = (b \rightleftharpoons a)$
 $\langle \text{proof} \rangle$

lemma *supp-perm-eq*:
assumes $(\text{supp}\ x) \#* p$
shows $p \cdot x = x$
 $\langle \text{proof} \rangle$

same lemma as above, but proved with a different induction principle

lemma *supp-perm-eq-test*:
assumes $(\text{supp}\ x) \#* p$
shows $p \cdot x = x$
 $\langle \text{proof} \rangle$

lemma *perm-supp-eq*:
assumes $a: (\text{supp}\ p) \#* x$
shows $p \cdot x = x$
 $\langle \text{proof} \rangle$

lemma *supp-perm-perm-eq*:
assumes $a: \forall a \in \text{supp}\ x. p \cdot a = q \cdot a$
shows $p \cdot x = q \cdot x$
 $\langle \text{proof} \rangle$

disagreement set

definition
 $dset :: \text{perm} \Rightarrow \text{perm} \Rightarrow \text{atom}\ \text{set}$
where
 $dset\ p\ q = \{a::\text{atom}. p \cdot a \neq q \cdot a\}$

lemma *ds-fresh*:
assumes $dset\ p\ q\ \#* x$
shows $p \cdot x = q \cdot x$
 $\langle proof \rangle$

lemma *atom-set-perm-eq*:
assumes $a: as\ \#* p$
shows $p \cdot as = as$
 $\langle proof \rangle$

17 Avoiding of atom sets

For every set of atoms, there is another set of atoms avoiding a finitely supported c and there is a permutation which 'translates' between both sets.

lemma *at-set-avoiding-aux*:
fixes $Xs::atom\ set$
and $As::atom\ set$
assumes $b: Xs \subseteq As$
and $c: finite\ As$
shows $\exists p. (p \cdot Xs) \cap As = \{\} \wedge (supp\ p) = (Xs \cup (p \cdot Xs))$
 $\langle proof \rangle$

lemma *at-set-avoiding*:
assumes $a: finite\ Xs$
and $b: finite\ (supp\ c)$
obtains $p::perm$ **where** $(p \cdot Xs)\ \#* c$ **and** $(supp\ p) = (Xs \cup (p \cdot Xs))$
 $\langle proof \rangle$

lemma *at-set-avoiding1*:
assumes $finite\ xs$
and $finite\ (supp\ c)$
shows $\exists p. (p \cdot xs)\ \#* c$
 $\langle proof \rangle$

lemma *at-set-avoiding2*:
assumes $finite\ xs$
and $finite\ (supp\ c)\ finite\ (supp\ x)$
and $xs\ \#* x$
shows $\exists p. (p \cdot xs)\ \#* c \wedge supp\ x\ \#* p$
 $\langle proof \rangle$

lemma *at-set-avoiding3*:
assumes $finite\ xs$
and $finite\ (supp\ c)\ finite\ (supp\ x)$
and $xs\ \#* x$
shows $\exists p. (p \cdot xs)\ \#* c \wedge supp\ x\ \#* p \wedge supp\ p = xs \cup (p \cdot xs)$

$\langle proof \rangle$

lemma *at-set-avoiding2-atom*:

assumes *finite* (*supp* *c*) *finite* (*supp* *x*)

and $b: a \# x$

shows $\exists p. (p \cdot a) \# c \wedge \text{supp } x \#^* p$

$\langle proof \rangle$

18 Renaming permutations

lemma *set-renaming-perm*:

assumes *b*: *finite* *bs*

shows $\exists q. (\forall b \in \text{bs}. q \cdot b = p \cdot b) \wedge \text{supp } q \subseteq \text{bs} \cup (p \cdot \text{bs})$

$\langle proof \rangle$

lemma *set-renaming-perm2*:

shows $\exists q. (\forall b \in \text{bs}. q \cdot b = p \cdot b) \wedge \text{supp } q \subseteq \text{bs} \cup (p \cdot \text{bs})$

$\langle proof \rangle$

lemma *list-renaming-perm*:

shows $\exists q. (\forall b \in \text{set } \text{bs}. q \cdot b = p \cdot b) \wedge \text{supp } q \subseteq \text{set } \text{bs} \cup (p \cdot \text{set } \text{bs})$

$\langle proof \rangle$

19 Concrete Atoms Types

Class *at-base* allows types containing multiple sorts of atoms. Class *at* only allows types with a single sort.

class *at-base* = *pt* +

fixes *atom* :: 'a \Rightarrow *atom*

assumes *atom-eq-iff* [*simp*]: *atom* *a* = *atom* *b* \longleftrightarrow *a* = *b*

assumes *atom-eqvt*: $p \cdot (\text{atom } a) = \text{atom } (p \cdot a)$

declare *atom-eqvt* [*eqvt*]

class *at* = *at-base* +

assumes *sort-of-atom-eq* [*simp*]: *sort-of* (*atom* *a*) = *sort-of* (*atom* *b*)

lemma *sort-ineq* [*simp*]:

assumes *sort-of* (*atom* *a*) \neq *sort-of* (*atom* *b*)

shows *atom* *a* \neq *atom* *b*

$\langle proof \rangle$

lemma *supp-at-base*:

fixes *a*::'a::*at-base*

shows *supp* *a* = {*atom* *a*}

$\langle proof \rangle$

lemma *fresh-at-base*:

shows $\text{sort-of } a \neq \text{sort-of } (\text{atom } b) \implies a \# b$
and $a \# b \longleftrightarrow a \neq \text{atom } b$
 ⟨proof⟩

lemma *fresh-ineq-at-base* [simp]:
shows $a \neq \text{atom } b \implies a \# b$
 ⟨proof⟩

lemma *fresh-atom-at-base* [simp]:
fixes $b::'a::\text{at-base}$
shows $a \# \text{atom } b \longleftrightarrow a \# b$
 ⟨proof⟩

lemma *fresh-star-atom-at-base*:
fixes $b::'a::\text{at-base}$
shows $as \#* \text{atom } b \longleftrightarrow as \#* b$
 ⟨proof⟩

lemma *if-fresh-at-base* [simp]:
shows $\text{atom } a \# x \implies P (\text{if } a = x \text{ then } t \text{ else } s) = P s$
and $\text{atom } a \# x \implies P (\text{if } x = a \text{ then } t \text{ else } s) = P s$
 ⟨proof⟩

⟨ML⟩

instance *at-base < fs*
 ⟨proof⟩

lemma *at-base-infinite* [simp]:
shows $\text{infinite } (\text{UNIV} :: 'a::\text{at-base set}) \text{ (is infinite ?U)}$
 ⟨proof⟩

lemma *swap-at-base-simps* [simp]:
fixes $x y::'a::\text{at-base}$
shows $\text{sort-of } (\text{atom } x) = \text{sort-of } (\text{atom } y) \implies (\text{atom } x \rightleftharpoons \text{atom } y) \cdot x = y$
and $\text{sort-of } (\text{atom } x) = \text{sort-of } (\text{atom } y) \implies (\text{atom } x \rightleftharpoons \text{atom } y) \cdot y = x$
and $\text{atom } x \neq a \implies \text{atom } x \neq b \implies (a \rightleftharpoons b) \cdot x = x$
 ⟨proof⟩

lemma *obtain-at-base*:
assumes $X: \text{finite } X$
obtains $a::'a::\text{at-base}$ **where** $\text{atom } a \notin X$
 ⟨proof⟩

lemma *obtain-fresh'*:

assumes $fin: finite (supp\ x)$
obtains $a::'a::at-base$ **where** $atom\ a \# x$
 $\langle proof \rangle$

lemma *obtain-fresh*:
fixes $x::'b::fs$
obtains $a::'a::at-base$ **where** $atom\ a \# x$
 $\langle proof \rangle$

lemma *supp-finite-set-at-base*:
assumes $a: finite\ S$
shows $supp\ S = atom\ 'S$
 $\langle proof \rangle$

lemma *fresh-finite-set-at-base*:
fixes $a::'a::at-base$
assumes $a: finite\ S$
shows $atom\ a \# S \longleftrightarrow a \notin S$
 $\langle proof \rangle$

lemma *fresh-at-base-permute-iff* [*simp*]:
fixes $a::'a::at-base$
shows $atom\ (p \cdot a) \# p \cdot x \longleftrightarrow atom\ a \# x$
 $\langle proof \rangle$

lemma *fresh-at-base-permI*:
shows $atom\ a \# p \implies p \cdot a = a$
 $\langle proof \rangle$

20 Infrastructure for concrete atom types

definition
 $flip :: 'a::at-base \Rightarrow 'a \Rightarrow perm\ (\lambda'(- \leftrightarrow -)')$
where
 $(a \leftrightarrow b) = (atom\ a \iff atom\ b)$

lemma *flip-fresh-fresh*:
assumes $atom\ a \# x\ atom\ b \# x$
shows $(a \leftrightarrow b) \cdot x = x$
 $\langle proof \rangle$

lemma *flip-self* [*simp*]: $(a \leftrightarrow a) = 0$
 $\langle proof \rangle$

lemma *flip-commute*: $(a \leftrightarrow b) = (b \leftrightarrow a)$
 $\langle proof \rangle$

lemma *minus-flip* [simp]: $-(a \leftrightarrow b) = (a \leftrightarrow b)$
 ⟨proof⟩

lemma *add-flip-cancel*: $(a \leftrightarrow b) + (a \leftrightarrow b) = 0$
 ⟨proof⟩

lemma *permute-flip-cancel* [simp]: $(a \leftrightarrow b) \cdot (a \leftrightarrow b) \cdot x = x$
 ⟨proof⟩

lemma *permute-flip-cancel2* [simp]: $(a \leftrightarrow b) \cdot (b \leftrightarrow a) \cdot x = x$
 ⟨proof⟩

lemma *flip-eqvt* [eqvt]:
 shows $p \cdot (a \leftrightarrow b) = (p \cdot a \leftrightarrow p \cdot b)$
 ⟨proof⟩

lemma *flip-at-base-simps* [simp]:
 shows $\text{sort-of } (\text{atom } a) = \text{sort-of } (\text{atom } b) \implies (a \leftrightarrow b) \cdot a = b$
 and $\text{sort-of } (\text{atom } a) = \text{sort-of } (\text{atom } b) \implies (a \leftrightarrow b) \cdot b = a$
 and $\llbracket a \neq c; b \neq c \rrbracket \implies (a \leftrightarrow b) \cdot c = c$
 and $\text{sort-of } (\text{atom } a) \neq \text{sort-of } (\text{atom } b) \implies (a \leftrightarrow b) \cdot x = x$
 ⟨proof⟩

the following two lemmas do not hold for *at-base*, only for single sort atoms from *at*

lemma *flip-triple*:
 fixes $a b c :: 'a :: \text{at}$
 assumes $a \neq b$ and $c \neq b$
 shows $(a \leftrightarrow c) + (b \leftrightarrow c) + (a \leftrightarrow c) = (a \leftrightarrow b)$
 ⟨proof⟩

lemma *permute-flip-at*:
 fixes $a b c :: 'a :: \text{at}$
 shows $(a \leftrightarrow b) \cdot c = (\text{if } c = a \text{ then } b \text{ else if } c = b \text{ then } a \text{ else } c)$
 ⟨proof⟩

lemma *flip-at-simps* [simp]:
 fixes $a b :: 'a :: \text{at}$
 shows $(a \leftrightarrow b) \cdot a = b$
 and $(a \leftrightarrow b) \cdot b = a$
 ⟨proof⟩

20.1 Syntax for coercing at-elements to the atom-type

syntax
 $-\text{atom-constrain} :: \text{logic} \Rightarrow \text{type} \Rightarrow \text{logic} \ (\langle \cdot :: \cdot \rangle [4, 0] \ 3)$

syntax-consts
 $-\text{atom-constrain} == \text{atom}$

translations

-atom-constrain a t => CONST atom (-constrain a t)

20.2 A lemma for proving instances of class *at*.

<ML>

New atom types are defined as subtypes of *atom*.

lemma *exists-eq-simple-sort*:

shows $\exists a. a \in \{a. \text{sort-of } a = s\}$

<proof>

lemma *exists-eq-sort*:

shows $\exists a. a \in \{a. \text{sort-of } a \in \text{range sort-fun}\}$

<proof>

lemma *at-base-class*:

fixes *sort-fun* :: 'b \Rightarrow *atom-sort*

fixes *Rep* :: 'a \Rightarrow *atom* **and** *Abs* :: *atom* \Rightarrow 'a

assumes *type*: *type-definition Rep Abs* {*a. sort-of a* \in *range sort-fun*}

assumes *atom-def*: $\bigwedge a. \text{atom } a = \text{Rep } a$

assumes *permute-def*: $\bigwedge p a. p \cdot a = \text{Abs } (p \cdot \text{Rep } a)$

shows *OFCLASS*('a, *at-base-class*)

<proof>

lemma *at-class*:

fixes *s* :: *atom-sort*

fixes *Rep* :: 'a \Rightarrow *atom* **and** *Abs* :: *atom* \Rightarrow 'a

assumes *type*: *type-definition Rep Abs* {*a. sort-of a* = *s*}

assumes *atom-def*: $\bigwedge a. \text{atom } a = \text{Rep } a$

assumes *permute-def*: $\bigwedge p a. p \cdot a = \text{Abs } (p \cdot \text{Rep } a)$

shows *OFCLASS*('a, *at-class*)

<proof>

lemma *at-class-sort*:

fixes *s* :: *atom-sort*

fixes *Rep* :: 'a \Rightarrow *atom* **and** *Abs* :: *atom* \Rightarrow 'a

fixes *a*::'a

assumes *type*: *type-definition Rep Abs* {*a. sort-of a* = *s*}

assumes *atom-def*: $\bigwedge a. \text{atom } a = \text{Rep } a$

shows *sort-of* (*atom a*) = *s*

<proof>

<ML>

21 Library functions for the nominal infrastructure

$\langle ML \rangle$

22 The freshness lemma according to Andy Pitts

lemma *freshness-lemma*:

fixes $h :: 'a::at \Rightarrow 'b::pt$

assumes $a: \exists a. \text{atom } a \# (h, h \ a)$

shows $\exists x. \forall a. \text{atom } a \# h \longrightarrow h \ a = x$

$\langle proof \rangle$

lemma *freshness-lemma-unique*:

fixes $h :: 'a::at \Rightarrow 'b::pt$

assumes $a: \exists a. \text{atom } a \# (h, h \ a)$

shows $\exists! x. \forall a. \text{atom } a \# h \longrightarrow h \ a = x$

$\langle proof \rangle$

packaging the freshness lemma into a function

definition

$Fresh :: ('a::at \Rightarrow 'b::pt) \Rightarrow 'b$

where

$Fresh \ h = (THE \ x. \forall a. \text{atom } a \# h \longrightarrow h \ a = x)$

lemma *Fresh-apply*:

fixes $h :: 'a::at \Rightarrow 'b::pt$

assumes $a: \exists a. \text{atom } a \# (h, h \ a)$

assumes $b: \text{atom } a \# h$

shows $Fresh \ h = h \ a$

$\langle proof \rangle$

lemma *Fresh-apply'*:

fixes $h :: 'a::at \Rightarrow 'b::pt$

assumes $a: \text{atom } a \# h \ \text{atom } a \# h \ a$

shows $Fresh \ h = h \ a$

$\langle proof \rangle$

$\langle ML \rangle$

lemma *Fresh-eqvt*:

fixes $h :: 'a::at \Rightarrow 'b::pt$

assumes $a: \exists a. \text{atom } a \# (h, h \ a)$

shows $p \cdot (Fresh \ h) = Fresh \ (p \cdot h)$

$\langle proof \rangle$

lemma *Fresh-supports*:

fixes $h :: 'a::at \Rightarrow 'b::pt$

assumes $a: \exists a. \text{atom } a \# (h, h a)$
shows $(\text{supp } h) \text{ supports } (\text{Fresh } h)$
 $\langle \text{proof} \rangle$

notation Fresh (**binder** $\langle \text{FRESH} \rangle 10$)

lemma FRESH-f-iff :
fixes $P :: 'a::\text{at} \Rightarrow 'b::\text{pure}$
fixes $f :: 'b \Rightarrow 'c::\text{pure}$
assumes $P: \text{finite } (\text{supp } P)$
shows $(\text{FRESH } x. f (P x)) = f (\text{FRESH } x. P x)$
 $\langle \text{proof} \rangle$

lemma FRESH-binop-iff :
fixes $P :: 'a::\text{at} \Rightarrow 'b::\text{pure}$
fixes $Q :: 'a::\text{at} \Rightarrow 'c::\text{pure}$
fixes $\text{binop} :: 'b \Rightarrow 'c \Rightarrow 'd::\text{pure}$
assumes $P: \text{finite } (\text{supp } P)$
and $Q: \text{finite } (\text{supp } Q)$
shows $(\text{FRESH } x. \text{binop } (P x) (Q x)) = \text{binop } (\text{FRESH } x. P x) (\text{FRESH } x. Q x)$
 $\langle \text{proof} \rangle$

lemma FRESH-conj-iff :
fixes $P Q :: 'a::\text{at} \Rightarrow \text{bool}$
assumes $P: \text{finite } (\text{supp } P)$ **and** $Q: \text{finite } (\text{supp } Q)$
shows $(\text{FRESH } x. P x \wedge Q x) \longleftrightarrow (\text{FRESH } x. P x) \wedge (\text{FRESH } x. Q x)$
 $\langle \text{proof} \rangle$

lemma FRESH-disj-iff :
fixes $P Q :: 'a::\text{at} \Rightarrow \text{bool}$
assumes $P: \text{finite } (\text{supp } P)$ **and** $Q: \text{finite } (\text{supp } Q)$
shows $(\text{FRESH } x. P x \vee Q x) \longleftrightarrow (\text{FRESH } x. P x) \vee (\text{FRESH } x. Q x)$
 $\langle \text{proof} \rangle$

23 Automation for creating concrete atom types

At the moment only single-sort concrete atoms are supported.

$\langle \text{ML} \rangle$

24 Automatic equivariance procedure for inductive definitions

$\langle \text{ML} \rangle$

end
theory Nominal2-Abs

```

imports Nominal2-Base
          HOL-Library.Quotient-List
          HOL-Library.Quotient-Product
begin

```

25 Abstractions

```

fun
  alpha-set
where
  alpha-set[simp del]:
  alpha-set (bs, x) R f p (cs, y)  $\longleftrightarrow$ 
     $f\ x - bs = f\ y - cs \wedge$ 
     $(f\ x - bs) \#* p \wedge$ 
     $R\ (p \cdot x)\ y \wedge$ 
     $p \cdot bs = cs$ 

```

```

fun
  alpha-res
where
  alpha-res[simp del]:
  alpha-res (bs, x) R f p (cs, y)  $\longleftrightarrow$ 
     $f\ x - bs = f\ y - cs \wedge$ 
     $(f\ x - bs) \#* p \wedge$ 
     $R\ (p \cdot x)\ y$ 

```

```

fun
  alpha-lst
where
  alpha-lst[simp del]:
  alpha-lst (bs, x) R f p (cs, y)  $\longleftrightarrow$ 
     $f\ x - set\ bs = f\ y - set\ cs \wedge$ 
     $(f\ x - set\ bs) \#* p \wedge$ 
     $R\ (p \cdot x)\ y \wedge$ 
     $p \cdot bs = cs$ 

```

lemmas *alphas* = *alpha-set.simps alpha-res.simps alpha-lst.simps*

notation

alpha-set ($\langle - \approx_{set} - - - \rangle [100, 100, 100, 100, 100] 100$) **and**
alpha-res ($\langle - \approx_{res} - - - \rangle [100, 100, 100, 100, 100] 100$) **and**
alpha-lst ($\langle - \approx_{lst} - - - \rangle [100, 100, 100, 100, 100] 100$)

26 Mono

```

lemma [mono]:
  shows  $R1 \leq R2 \implies \alpha\text{-set}\ bs\ R1 \leq \alpha\text{-set}\ bs\ R2$ 
  and  $R1 \leq R2 \implies \alpha\text{-res}\ bs\ R1 \leq \alpha\text{-res}\ bs\ R2$ 

```

and $R1 \leq R2 \implies \text{alpha-lst cs } R1 \leq \text{alpha-lst cs } R2$
 ⟨proof⟩

27 Equivariance

lemma *alpha-eqvt*[*eqvt*]:

shows $(bs, x) \approx_{\text{set}} R f q (cs, y) \implies (p \cdot bs, p \cdot x) \approx_{\text{set}} (p \cdot R) (p \cdot f) (p \cdot q)$
 $(p \cdot cs, p \cdot y)$
and $(bs, x) \approx_{\text{res}} R f q (cs, y) \implies (p \cdot bs, p \cdot x) \approx_{\text{res}} (p \cdot R) (p \cdot f) (p \cdot q)$
 $(p \cdot cs, p \cdot y)$
and $(ds, x) \approx_{\text{lst}} R f q (es, y) \implies (p \cdot ds, p \cdot x) \approx_{\text{lst}} (p \cdot R) (p \cdot f) (p \cdot q) (p$
 $\cdot es, p \cdot y)$
 ⟨proof⟩

28 Equivalence

lemma *alpha-refl*:

assumes $a: R x x$
shows $(bs, x) \approx_{\text{set}} R f 0 (bs, x)$
and $(bs, x) \approx_{\text{res}} R f 0 (bs, x)$
and $(cs, x) \approx_{\text{lst}} R f 0 (cs, x)$
 ⟨proof⟩

lemma *alpha-sym*:

assumes $a: R (p \cdot x) y \implies R (- p \cdot y) x$
shows $(bs, x) \approx_{\text{set}} R f p (cs, y) \implies (cs, y) \approx_{\text{set}} R f (- p) (bs, x)$
and $(bs, x) \approx_{\text{res}} R f p (cs, y) \implies (cs, y) \approx_{\text{res}} R f (- p) (bs, x)$
and $(ds, x) \approx_{\text{lst}} R f p (es, y) \implies (es, y) \approx_{\text{lst}} R f (- p) (ds, x)$
 ⟨proof⟩

lemma *alpha-trans*:

assumes $a: \llbracket R (p \cdot x) y; R (q \cdot y) z \rrbracket \implies R ((q + p) \cdot x) z$
shows $\llbracket (bs, x) \approx_{\text{set}} R f p (cs, y); (cs, y) \approx_{\text{set}} R f q (ds, z) \rrbracket \implies (bs, x) \approx_{\text{set}} R$
 $f (q + p) (ds, z)$
and $\llbracket (bs, x) \approx_{\text{res}} R f p (cs, y); (cs, y) \approx_{\text{res}} R f q (ds, z) \rrbracket \implies (bs, x) \approx_{\text{res}} R$
 $f (q + p) (ds, z)$
and $\llbracket (es, x) \approx_{\text{lst}} R f p (gs, y); (gs, y) \approx_{\text{lst}} R f q (hs, z) \rrbracket \implies (es, x) \approx_{\text{lst}} R f$
 $(q + p) (hs, z)$
 ⟨proof⟩

lemma *alpha-sym-eqvt*:

assumes $a: R (p \cdot x) y \implies R y (p \cdot x)$
and $b: p \cdot R = R$
shows $(bs, x) \approx_{\text{set}} R f p (cs, y) \implies (cs, y) \approx_{\text{set}} R f (- p) (bs, x)$
and $(bs, x) \approx_{\text{res}} R f p (cs, y) \implies (cs, y) \approx_{\text{res}} R f (- p) (bs, x)$
and $(ds, x) \approx_{\text{lst}} R f p (es, y) \implies (es, y) \approx_{\text{lst}} R f (- p) (ds, x)$
 ⟨proof⟩

lemma *alpha-set-trans-eqvt*:
assumes $b: (cs, y) \approx_{set} R f q (ds, z)$
and $a: (bs, x) \approx_{set} R f p (cs, y)$
and $d: q \cdot R = R$
and $c: \llbracket R (p \cdot x) y; R y (- q \cdot z) \rrbracket \implies R (p \cdot x) (- q \cdot z)$
shows $(bs, x) \approx_{set} R f (q + p) (ds, z)$
 $\langle proof \rangle$

lemma *alpha-res-trans-eqvt*:
assumes $b: (cs, y) \approx_{res} R f q (ds, z)$
and $a: (bs, x) \approx_{res} R f p (cs, y)$
and $d: q \cdot R = R$
and $c: \llbracket R (p \cdot x) y; R y (- q \cdot z) \rrbracket \implies R (p \cdot x) (- q \cdot z)$
shows $(bs, x) \approx_{res} R f (q + p) (ds, z)$
 $\langle proof \rangle$

lemma *alpha-lst-trans-eqvt*:
assumes $b: (cs, y) \approx_{lst} R f q (ds, z)$
and $a: (bs, x) \approx_{lst} R f p (cs, y)$
and $d: q \cdot R = R$
and $c: \llbracket R (p \cdot x) y; R y (- q \cdot z) \rrbracket \implies R (p \cdot x) (- q \cdot z)$
shows $(bs, x) \approx_{lst} R f (q + p) (ds, z)$
 $\langle proof \rangle$

lemmas *alpha-trans-eqvt = alpha-set-trans-eqvt alpha-res-trans-eqvt alpha-lst-trans-eqvt*

29 General Abstractions

fun
alpha-abs-set
where
 $[simp\ del]:$
 $alpha-abs-set (bs, x) (cs, y) \longleftrightarrow (\exists p. (bs, x) \approx_{set} ((=)) supp p (cs, y))$

fun
alpha-abs-lst
where
 $[simp\ del]:$
 $alpha-abs-lst (bs, x) (cs, y) \longleftrightarrow (\exists p. (bs, x) \approx_{lst} ((=)) supp p (cs, y))$

fun
alpha-abs-res
where
 $[simp\ del]:$
 $alpha-abs-res (bs, x) (cs, y) \longleftrightarrow (\exists p. (bs, x) \approx_{res} ((=)) supp p (cs, y))$

notation
 $alpha-abs-set$ (**infix** $\langle \approx_{abs'-set} \rangle$ 50) **and**
 $alpha-abs-lst$ (**infix** $\langle \approx_{abs'-lst} \rangle$ 50) **and**

alpha-abs-res (**infix** $\approx_{abs\text{-}res}$ 50)

lemmas *alphas-abs* = *alpha-abs-set.simps alpha-abs-res.simps alpha-abs-lst.simps*

lemma *alphas-abs-refl*:

shows $(bs, x) \approx_{abs\text{-}set} (bs, x)$
and $(bs, x) \approx_{abs\text{-}res} (bs, x)$
and $(cs, x) \approx_{abs\text{-}lst} (cs, x)$
<proof>

lemma *alphas-abs-sym*:

shows $(bs, x) \approx_{abs\text{-}set} (cs, y) \implies (cs, y) \approx_{abs\text{-}set} (bs, x)$
and $(bs, x) \approx_{abs\text{-}res} (cs, y) \implies (cs, y) \approx_{abs\text{-}res} (bs, x)$
and $(ds, x) \approx_{abs\text{-}lst} (es, y) \implies (es, y) \approx_{abs\text{-}lst} (ds, x)$
<proof>

lemma *alphas-abs-trans*:

shows $\llbracket (bs, x) \approx_{abs\text{-}set} (cs, y); (cs, y) \approx_{abs\text{-}set} (ds, z) \rrbracket \implies (bs, x) \approx_{abs\text{-}set} (ds, z)$
and $\llbracket (bs, x) \approx_{abs\text{-}res} (cs, y); (cs, y) \approx_{abs\text{-}res} (ds, z) \rrbracket \implies (bs, x) \approx_{abs\text{-}res} (ds, z)$
and $\llbracket (es, x) \approx_{abs\text{-}lst} (gs, y); (gs, y) \approx_{abs\text{-}lst} (hs, z) \rrbracket \implies (es, x) \approx_{abs\text{-}lst} (hs, z)$
<proof>

lemma *alphas-abs-eqvt*:

shows $(bs, x) \approx_{abs\text{-}set} (cs, y) \implies (p \cdot bs, p \cdot x) \approx_{abs\text{-}set} (p \cdot cs, p \cdot y)$
and $(bs, x) \approx_{abs\text{-}res} (cs, y) \implies (p \cdot bs, p \cdot x) \approx_{abs\text{-}res} (p \cdot cs, p \cdot y)$
and $(ds, x) \approx_{abs\text{-}lst} (es, y) \implies (p \cdot ds, p \cdot x) \approx_{abs\text{-}lst} (p \cdot es, p \cdot y)$
<proof>

30 Strengthening the equivalence

lemma *disjoint-right-eq*:

assumes $a: A \cup B1 = A \cup B2$
and $b: A \cap B1 = \{\} \ A \cap B2 = \{\}$
shows $B1 = B2$
<proof>

lemma *supp-property-res*:

assumes $a: (as, x) \approx_{res} (=) \text{supp } p (as', x')$
shows $p \cdot (\text{supp } x \cap as) = \text{supp } x' \cap as'$
<proof>

lemma *alpha-abs-res-stronger1-aux*:

assumes $asm: (as, x) \approx_{res} (=) \text{supp } p' (as', x')$
shows $\exists p. (as, x) \approx_{res} (=) \text{supp } p (as', x') \wedge \text{supp } p \subseteq (\text{supp } x \cap as) \cup (\text{supp } x' \cap as')$

$\langle proof \rangle$

lemma *alpha-abs-res-minimal*:

assumes *asm*: $(as, x) \approx_{res} (=) supp\ p\ (as', x')$

shows $(as \cap supp\ x, x) \approx_{res} (=) supp\ p\ (as' \cap supp\ x', x')$

$\langle proof \rangle$

lemma *alpha-abs-res-abs-set*:

assumes *asm*: $(as, x) \approx_{res} (=) supp\ p\ (as', x')$

shows $(as \cap supp\ x, x) \approx_{set} (=) supp\ p\ (as' \cap supp\ x', x')$

$\langle proof \rangle$

lemma *alpha-abs-set-abs-res*:

assumes *asm*: $(as \cap supp\ x, x) \approx_{set} (=) supp\ p\ (as' \cap supp\ x', x')$

shows $(as, x) \approx_{res} (=) supp\ p\ (as', x')$

$\langle proof \rangle$

lemma *alpha-abs-res-stronger1*:

assumes *asm*: $(as, x) \approx_{res} (=) supp\ p'\ (as', x')$

shows $\exists p. (as, x) \approx_{res} (=) supp\ p\ (as', x') \wedge supp\ p \subseteq as \cup as'$

$\langle proof \rangle$

lemma *alpha-abs-set-stronger1*:

assumes *asm*: $(as, x) \approx_{set} (=) supp\ p'\ (as', x')$

shows $\exists p. (as, x) \approx_{set} (=) supp\ p\ (as', x') \wedge supp\ p \subseteq as \cup as'$

$\langle proof \rangle$

lemma *alpha-abs-lst-stronger1*:

assumes *asm*: $(as, x) \approx_{lst} (=) supp\ p'\ (as', x')$

shows $\exists p. (as, x) \approx_{lst} (=) supp\ p\ (as', x') \wedge supp\ p \subseteq set\ as \cup set\ as'$

$\langle proof \rangle$

lemma *alphas-abs-stronger*:

shows $(as, x) \approx_{abs-set} (as', x') \longleftrightarrow (\exists p. (as, x) \approx_{set} (=) supp\ p\ (as', x') \wedge supp\ p \subseteq as \cup as')$

and $(as, x) \approx_{abs-res} (as', x') \longleftrightarrow (\exists p. (as, x) \approx_{res} (=) supp\ p\ (as', x') \wedge supp\ p \subseteq as \cup as')$

and $(bs, x) \approx_{abs-lst} (bs', x') \longleftrightarrow$

$(\exists p. (bs, x) \approx_{lst} (=) supp\ p\ (bs', x') \wedge supp\ p \subseteq set\ bs \cup set\ bs')$

$\langle proof \rangle$

lemma *alpha-res-alpha-set*:

$(bs, x) \approx_{res} (=) supp\ p\ (cs, y) \longleftrightarrow (bs \cap supp\ x, x) \approx_{set} (=) supp\ p\ (cs \cap supp\ y, y)$

$\langle proof \rangle$

31 Quotient types

quotient-type

'a abs-set = (atom set × 'a::pt) / alpha-abs-set
 ⟨proof⟩

quotient-type

'b abs-res = (atom set × 'b::pt) / alpha-abs-res
 ⟨proof⟩

quotient-type

'c abs-lst = (atom list × 'c::pt) / alpha-abs-lst
 ⟨proof⟩

quotient-definition

Abs-set (⟨[-]set. → [60, 60] 60)

where

Abs-set::atom set ⇒ ('a::pt) ⇒ 'a abs-set

is

Pair::atom set ⇒ ('a::pt) ⇒ (atom set × 'a) ⟨proof⟩

quotient-definition

Abs-res (⟨[-]res. → [60, 60] 60)

where

Abs-res::atom set ⇒ ('a::pt) ⇒ 'a abs-res

is

Pair::atom set ⇒ ('a::pt) ⇒ (atom set × 'a) ⟨proof⟩

quotient-definition

Abs-lst (⟨[-]lst. → [60, 60] 60)

where

Abs-lst::atom list ⇒ ('a::pt) ⇒ 'a abs-lst

is

Pair::atom list ⇒ ('a::pt) ⇒ (atom list × 'a) ⟨proof⟩

lemma [quot-respect]:

shows ((=) ==> (=) ==> alpha-abs-set) Pair Pair

and ((=) ==> (=) ==> alpha-abs-res) Pair Pair

and ((=) ==> (=) ==> alpha-abs-lst) Pair Pair

⟨proof⟩

lemma [quot-respect]:

shows ((=) ==> alpha-abs-set ==> alpha-abs-set) permute permute

and ((=) ==> alpha-abs-res ==> alpha-abs-res) permute permute

and ((=) ==> alpha-abs-lst ==> alpha-abs-lst) permute permute

⟨proof⟩

lemma Abs-eq-iff:

shows [bs]set. x = [bs']set. y ⟷ (∃ p. (bs, x) ≈set (=) supp p (bs', y))

and [bs]res. x = [bs']res. y ⟷ (∃ p. (bs, x) ≈res (=) supp p (bs', y))

and [cs]lst. x = [cs']lst. y ⟷ (∃ p. (cs, x) ≈lst (=) supp p (cs', y))

⟨proof⟩

lemma *Abs-eq-iff2*:

shows $[bs]set. x = [bs']set. y \longleftrightarrow (\exists p. (bs, x) \approx_{set} (=) supp\ p (bs', y) \wedge supp\ p \subseteq bs \cup bs')$

and $[bs]res. x = [bs']res. y \longleftrightarrow (\exists p. (bs, x) \approx_{res} (=) supp\ p (bs', y) \wedge supp\ p \subseteq bs \cup bs')$

and $[cs]lst. x = [cs']lst. y \longleftrightarrow (\exists p. (cs, x) \approx_{lst} (=) supp\ p (cs', y) \wedge supp\ p \subseteq set\ cs \cup set\ cs')$
<proof>

lemma *Abs-eq-res-set*:

shows $[bs]res. x = [cs]res. y \longleftrightarrow [bs \cap supp\ x]set. x = [cs \cap supp\ y]set. y$
<proof>

lemma *Abs-eq-res-supp*:

assumes *asm*: $supp\ x \subseteq bs$

shows $[as]res. x = [as \cap bs]res. x$
<proof>

lemma *Abs-exhausts*[*cases type*]:

shows $(\bigwedge as\ (x::'a::pt). y1 = [as]set. x \implies P1) \implies P1$

and $(\bigwedge as\ (x::'a::pt). y2 = [as]res. x \implies P2) \implies P2$

and $(\bigwedge bs\ (x::'a::pt). y3 = [bs]lst. x \implies P3) \implies P3$

<proof>

instantiation *abs-set* :: (*pt*) *pt*

begin

quotient-definition

permute-abs-set::*perm* \Rightarrow (*'a*::*pt abs-set*) \Rightarrow *'a abs-set*

is

permute::*perm* \Rightarrow (*atom set* \times *'a::pt*) \Rightarrow (*atom set* \times *'a::pt*)

<proof>

lemma *permute-Abs-set*[*simp*]:

fixes *x*::*'a::pt*

shows $(p \cdot ([as]set. x)) = [p \cdot as]set. (p \cdot x)$

<proof>

instance

<proof>

end

instantiation *abs-res* :: (*pt*) *pt*

begin

quotient-definition

permute-abs-res::perm \Rightarrow (*'a::pt abs-res*) \Rightarrow *'a abs-res*

is

permute:: perm \Rightarrow (*atom set* \times *'a::pt*) \Rightarrow (*atom set* \times *'a::pt*)
<proof>

lemma *permute-Abs-res[simp]*:

fixes *x::'a::pt*

shows (*p* \cdot (*[as]res. x*)) = [*p* \cdot *as*]*res. (p* \cdot *x*)

<proof>

instance

<proof>

end

instantiation *abs-lst* :: (*pt*) *pt*

begin

quotient-definition

permute-abs-lst::perm \Rightarrow (*'a::pt abs-lst*) \Rightarrow *'a abs-lst*

is

permute:: perm \Rightarrow (*atom list* \times *'a::pt*) \Rightarrow (*atom list* \times *'a::pt*)
<proof>

lemma *permute-Abs-lst[simp]*:

fixes *x::'a::pt*

shows (*p* \cdot (*[as]lst. x*)) = [*p* \cdot *as*]*lst. (p* \cdot *x*)

<proof>

instance

<proof>

end

lemmas *permute-Abs[eqvt]* = *permute-Abs-set permute-Abs-res permute-Abs-lst*

lemma *Abs-swap1*:

assumes *a1*: *a* \notin (*supp x*) - *bs*

and *a2*: *b* \notin (*supp x*) - *bs*

shows [*bs*]*set. x* = [(*a* \rightleftharpoons *b*) \cdot *bs*]*set. ((a* \rightleftharpoons *b*) \cdot *x*)

and [*bs*]*res. x* = [(*a* \rightleftharpoons *b*) \cdot *bs*]*res. ((a* \rightleftharpoons *b*) \cdot *x*)

<proof>

lemma *Abs-swap2*:

assumes *a1*: *a* \notin (*supp x*) - (*set bs*)

and *a2*: *b* \notin (*supp x*) - (*set bs*)

shows [*bs*]*lst. x* = [(*a* \rightleftharpoons *b*) \cdot *bs*]*lst. ((a* \rightleftharpoons *b*) \cdot *x*)

<proof>

lemma *Abs-supports*:

shows $((supp\ x) - as)\ supports\ ([as]set.\ x)$
and $((supp\ x) - as)\ supports\ ([as]res.\ x)$
and $((supp\ x) - set\ bs)\ supports\ ([bs]lst.\ x)$
<proof>

function

supp-set :: $('a::pt)\ abs-set \Rightarrow atom\ set$ **and**
supp-res :: $('a::pt)\ abs-res \Rightarrow atom\ set$ **and**
supp-lst :: $('a::pt)\ abs-lst \Rightarrow atom\ set$

where

supp-set $([as]set.\ x) = supp\ x - as$
supp-res $([as]res.\ x) = supp\ x - as$
supp-lst $(Abs-lst\ cs\ x) = (supp\ x) - (set\ cs)$
<proof>

termination

<proof>

lemma *supp-funs-eqvt*[*eqvt*]:

shows $(p \cdot supp-set\ x) = supp-set\ (p \cdot x)$
and $(p \cdot supp-res\ y) = supp-res\ (p \cdot y)$
and $(p \cdot supp-lst\ z) = supp-lst\ (p \cdot z)$
<proof>

lemma *Abs-fresh-aux*:

shows $a \# [bs]set.\ x \Longrightarrow a \# supp-set\ ([bs]set.\ x)$
and $a \# [bs]res.\ x \Longrightarrow a \# supp-res\ ([bs]res.\ x)$
and $a \# [cs]lst.\ x \Longrightarrow a \# supp-lst\ ([cs]lst.\ x)$
<proof>

lemma *Abs-supp-subset1*:

assumes $a:\ finite\ (supp\ x)$
shows $(supp\ x) - as \subseteq supp\ ([as]set.\ x)$
and $(supp\ x) - as \subseteq supp\ ([as]res.\ x)$
and $(supp\ x) - (set\ bs) \subseteq supp\ ([bs]lst.\ x)$
<proof>

lemma *Abs-supp-subset2*:

assumes $a:\ finite\ (supp\ x)$
shows $supp\ ([as]set.\ x) \subseteq (supp\ x) - as$
and $supp\ ([as]res.\ x) \subseteq (supp\ x) - as$
and $supp\ ([bs]lst.\ x) \subseteq (supp\ x) - (set\ bs)$
<proof>

lemma *Abs-finite-supp*:

assumes $a:\ finite\ (supp\ x)$
shows $supp\ ([as]set.\ x) = (supp\ x) - as$

and $\text{supp } ([as]res. x) = (\text{supp } x) - as$
and $\text{supp } ([bs]lst. x) = (\text{supp } x) - (\text{set } bs)$
 <proof>

lemma *supp-Abs*:
fixes $x::'a::fs$
shows $\text{supp } ([as]set. x) = (\text{supp } x) - as$
and $\text{supp } ([as]res. x) = (\text{supp } x) - as$
and $\text{supp } ([bs]lst. x) = (\text{supp } x) - (\text{set } bs)$
 <proof>

instance *abs-set* :: $(fs) fs$
 <proof>

instance *abs-res* :: $(fs) fs$
 <proof>

instance *abs-lst* :: $(fs) fs$
 <proof>

lemma *Abs-fresh-iff*:
fixes $x::'a::fs$
shows $a \# [bs]set. x \longleftrightarrow a \in bs \vee (a \notin bs \wedge a \# x)$
and $a \# [bs]res. x \longleftrightarrow a \in bs \vee (a \notin bs \wedge a \# x)$
and $a \# [cs]lst. x \longleftrightarrow a \in (\text{set } cs) \vee (a \notin (\text{set } cs) \wedge a \# x)$
 <proof>

lemma *Abs-fresh-star-iff*:
fixes $x::'a::fs$
shows $as \#* ([bs]set. x) \longleftrightarrow (as - bs) \#* x$
and $as \#* ([bs]res. x) \longleftrightarrow (as - bs) \#* x$
and $as \#* ([cs]lst. x) \longleftrightarrow (as - \text{set } cs) \#* x$
 <proof>

lemma *Abs-fresh-star*:
fixes $x::'a::fs$
shows $as \subseteq as' \Longrightarrow as \#* ([as']set. x)$
and $as \subseteq as' \Longrightarrow as \#* ([as']res. x)$
and $bs \subseteq \text{set } bs' \Longrightarrow bs \#* ([bs']lst. x)$
 <proof>

lemma *Abs-fresh-star2*:
fixes $x::'a::fs$
shows $as \cap bs = \{\} \Longrightarrow as \#* ([bs]set. x) \longleftrightarrow as \#* x$
and $as \cap bs = \{\} \Longrightarrow as \#* ([bs]res. x) \longleftrightarrow as \#* x$
and $cs \cap \text{set } ds = \{\} \Longrightarrow cs \#* ([ds]lst. x) \longleftrightarrow cs \#* x$
 <proof>

32 Abstractions of single atoms

lemma *Abs1-eq:*

fixes $x\ y::'a::fs$
shows $[\{atom\ a\}]set.\ x = [\{atom\ a\}]set.\ y \longleftrightarrow x = y$
and $[\{atom\ a\}]res.\ x = [\{atom\ a\}]res.\ y \longleftrightarrow x = y$
and $[[atom\ a]]lst.\ x = [[atom\ a]]lst.\ y \longleftrightarrow x = y$
<proof>

lemma *Abs1-eq-iff-fresh:*

fixes $x\ y::'a::fs$
and $a\ b\ c::'b::at$
assumes $atom\ c \# (a, b, x, y)$
shows $[\{atom\ a\}]set.\ x = [\{atom\ b\}]set.\ y \longleftrightarrow (a \leftrightarrow c) \cdot x = (b \leftrightarrow c) \cdot y$
and $[\{atom\ a\}]res.\ x = [\{atom\ b\}]res.\ y \longleftrightarrow (a \leftrightarrow c) \cdot x = (b \leftrightarrow c) \cdot y$
and $[[atom\ a]]lst.\ x = [[atom\ b]]lst.\ y \longleftrightarrow (a \leftrightarrow c) \cdot x = (b \leftrightarrow c) \cdot y$
<proof>

lemma *Abs1-eq-iff-all:*

fixes $x\ y::'a::fs$
and $z::'c::fs$
and $a\ b::'b::at$
shows $[\{atom\ a\}]set.\ x = [\{atom\ b\}]set.\ y \longleftrightarrow (\forall c.\ atom\ c \# z \longrightarrow atom\ c \# (a, b, x, y) \longrightarrow (a \leftrightarrow c) \cdot x = (b \leftrightarrow c) \cdot y)$
and $[\{atom\ a\}]res.\ x = [\{atom\ b\}]res.\ y \longleftrightarrow (\forall c.\ atom\ c \# z \longrightarrow atom\ c \# (a, b, x, y) \longrightarrow (a \leftrightarrow c) \cdot x = (b \leftrightarrow c) \cdot y)$
and $[[atom\ a]]lst.\ x = [[atom\ b]]lst.\ y \longleftrightarrow (\forall c.\ atom\ c \# z \longrightarrow atom\ c \# (a, b, x, y) \longrightarrow (a \leftrightarrow c) \cdot x = (b \leftrightarrow c) \cdot y)$
<proof>

lemma *Abs1-eq-iff:*

fixes $x\ y::'a::fs$
and $a\ b::'b::at$
shows $[\{atom\ a\}]set.\ x = [\{atom\ b\}]set.\ y \longleftrightarrow (a = b \wedge x = y) \vee (a \neq b \wedge x = (a \leftrightarrow b) \cdot y \wedge atom\ a \# y)$
and $[\{atom\ a\}]res.\ x = [\{atom\ b\}]res.\ y \longleftrightarrow (a = b \wedge x = y) \vee (a \neq b \wedge x = (a \leftrightarrow b) \cdot y \wedge atom\ a \# y)$
and $[[atom\ a]]lst.\ x = [[atom\ b]]lst.\ y \longleftrightarrow (a = b \wedge x = y) \vee (a \neq b \wedge x = (a \leftrightarrow b) \cdot y \wedge atom\ a \# y)$
<proof>

lemma *Abs1-eq-iff':*

fixes $x::'a::fs$
and $a\ b::'b::at$
shows $[\{atom\ a\}]set.\ x = [\{atom\ b\}]set.\ y \longleftrightarrow (a = b \wedge x = y) \vee (a \neq b \wedge (b \leftrightarrow a) \cdot x = y \wedge atom\ b \# x)$
and $[\{atom\ a\}]res.\ x = [\{atom\ b\}]res.\ y \longleftrightarrow (a = b \wedge x = y) \vee (a \neq b \wedge (b \leftrightarrow a) \cdot x = y \wedge atom\ b \# x)$
and $[[atom\ a]]lst.\ x = [[atom\ b]]lst.\ y \longleftrightarrow (a = b \wedge x = y) \vee (a \neq b \wedge (b \leftrightarrow a) \cdot x = y \wedge atom\ b \# x)$

$a) \cdot x = y \wedge \text{atom } b \# x$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

32.1 Renaming of bodies of abstractions

lemma *Abs-rename-set*:

fixes $x::'a::fs$
assumes $a: (p \cdot bs) \#* x$

shows $\exists q. [bs]set. x = [p \cdot bs]set. (q \cdot x) \wedge q \cdot bs = p \cdot bs$
 $\langle \text{proof} \rangle$

lemma *Abs-rename-res*:

fixes $x::'a::fs$
assumes $a: (p \cdot bs) \#* x$

shows $\exists q. [bs]res. x = [p \cdot bs]res. (q \cdot x) \wedge q \cdot bs = p \cdot bs$
 $\langle \text{proof} \rangle$

lemma *Abs-rename-lst*:

fixes $x::'a::fs$
assumes $a: (p \cdot (\text{set } bs)) \#* x$
shows $\exists q. [bs]lst. x = [p \cdot bs]lst. (q \cdot x) \wedge q \cdot bs = p \cdot bs$
 $\langle \text{proof} \rangle$

for deep recursive binders

lemma *Abs-rename-set'*:

fixes $x::'a::fs$
assumes $a: (p \cdot bs) \#* x$

shows $\exists q. [bs]set. x = [q \cdot bs]set. (q \cdot x) \wedge q \cdot bs = p \cdot bs$
 $\langle \text{proof} \rangle$

lemma *Abs-rename-res'*:

fixes $x::'a::fs$
assumes $a: (p \cdot bs) \#* x$

shows $\exists q. [bs]res. x = [q \cdot bs]res. (q \cdot x) \wedge q \cdot bs = p \cdot bs$
 $\langle \text{proof} \rangle$

lemma *Abs-rename-lst'*:

fixes $x::'a::fs$
assumes $a: (p \cdot (\text{set } bs)) \#* x$
shows $\exists q. [bs]lst. x = [q \cdot bs]lst. (q \cdot x) \wedge q \cdot bs = p \cdot bs$
 $\langle \text{proof} \rangle$

33 Infrastructure for building tuples of relations and functions

fun

prod-fv :: ('a ⇒ atom set) ⇒ ('b ⇒ atom set) ⇒ ('a × 'b) ⇒ atom set

where

prod-fv *fv1* *fv2* (x, y) = *fv1* x ∪ *fv2* y

definition

prod-alpha :: ('a ⇒ 'a ⇒ bool) ⇒ ('b ⇒ 'b ⇒ bool) ⇒ ('a × 'b ⇒ 'a × 'b ⇒ bool)

where

prod-alpha = *rel-prod*

lemma [*quot-respect*]:

shows ((*R1* ==> (=)) ==> (*R2* ==> (=)) ==> *rel-prod* *R1* *R2* ==> (=)) *prod-fv* *prod-fv*
 ⟨*proof*⟩

lemma [*quot-preserve*]:

assumes *q1*: *Quotient3* *R1* *abs1* *rep1*
and *q2*: *Quotient3* *R2* *abs2* *rep2*
shows ((*abs1* ----> *id*) ----> (*abs2* ----> *id*) ----> *map-prod* *rep1* *rep2* ----> *id*) *prod-fv* = *prod-fv*
 ⟨*proof*⟩

lemma [*mono*]:

shows *A* <= *B* ==> *C* <= *D* ==> *prod-alpha* *A* *C* <= *prod-alpha* *B* *D*
 ⟨*proof*⟩

lemma [*eqvt*]:

shows *p* · *prod-alpha* *A* *B* *x* *y* = *prod-alpha* (*p* · *A*) (*p* · *B*) (*p* · *x*) (*p* · *y*)
 ⟨*proof*⟩

lemma [*eqvt*]:

shows *p* · *prod-fv* *A* *B* (*x*, *y*) = *prod-fv* (*p* · *A*) (*p* · *B*) (*p* · *x*, *p* · *y*)
 ⟨*proof*⟩

lemma *prod-fv-supp*:

shows *prod-fv* *supp* *supp* = *supp*
 ⟨*proof*⟩

lemma *prod-alpha-eq*:

shows *prod-alpha* ((=)) ((=)) = ((=))
 ⟨*proof*⟩

end

theory *Nominal2-FCB*

imports *Nominal2-Abs*

begin

A tactic which solves all trivial cases in function definitions, and leaves the others unchanged.

$\langle ML \rangle$

lemma *Abs-lst1-fcb*:

fixes $x\ y :: 'a :: at$

and $S\ T :: 'b :: fs$

assumes $e: [[atom\ x]]lst.\ T = [[atom\ y]]lst.\ S$

and $f1: \llbracket x \neq y; atom\ y \# T; atom\ x \# (y \leftrightarrow x) \cdot T \rrbracket \Longrightarrow atom\ x \# f\ x\ T$

and $f2: \llbracket x \neq y; atom\ y \# T; atom\ x \# (y \leftrightarrow x) \cdot T \rrbracket \Longrightarrow atom\ y \# f\ x\ T$

and $p: \llbracket S = (x \leftrightarrow y) \cdot T; x \neq y; atom\ y \# T; atom\ x \# S \rrbracket$

$\Longrightarrow (x \leftrightarrow y) \cdot (f\ x\ T) = f\ y\ S$

shows $f\ x\ T = f\ y\ S$

$\langle proof \rangle$

lemma *Abs-lst-fcb*:

fixes $xs\ ys :: 'a :: fs$

and $S\ T :: 'b :: fs$

assumes $e: (Abs-lst\ (ba\ xs)\ T) = (Abs-lst\ (ba\ ys)\ S)$

and $f1: \bigwedge x. x \in set\ (ba\ xs) \Longrightarrow x \# f\ xs\ T$

and $f2: \bigwedge x. \llbracket supp\ T - set\ (ba\ xs) = supp\ S - set\ (ba\ ys); x \in set\ (ba\ ys) \rrbracket \Longrightarrow x \# f\ xs\ T$

and $eqv: \bigwedge p. \llbracket p \cdot T = S; p \cdot ba\ xs = ba\ ys; supp\ p \subseteq set\ (ba\ xs) \cup set\ (ba\ ys) \rrbracket$

$\Longrightarrow p \cdot (f\ xs\ T) = f\ ys\ S$

shows $f\ xs\ T = f\ ys\ S$

$\langle proof \rangle$

lemma *Abs-set-fcb*:

fixes $xs\ ys :: 'a :: fs$

and $S\ T :: 'b :: fs$

assumes $e: (Abs-set\ (ba\ xs)\ T) = (Abs-set\ (ba\ ys)\ S)$

and $f1: \bigwedge x. x \in ba\ xs \Longrightarrow x \# f\ xs\ T$

and $f2: \bigwedge x. \llbracket supp\ T - ba\ xs = supp\ S - ba\ ys; x \in ba\ ys \rrbracket \Longrightarrow x \# f\ xs\ T$

and $eqv: \bigwedge p. \llbracket p \cdot T = S; p \cdot ba\ xs = ba\ ys; supp\ p \subseteq ba\ xs \cup ba\ ys \rrbracket \Longrightarrow p \cdot (f\ xs\ T) = f\ ys\ S$

shows $f\ xs\ T = f\ ys\ S$

$\langle proof \rangle$

lemma *Abs-res-fcb*:

fixes $xs\ ys :: ('a :: at-base)\ set$

and $S\ T :: 'b :: fs$

assumes $e: (Abs-res\ (atom\ 'xs)\ T) = (Abs-res\ (atom\ 'ys)\ S)$

and $f1: \bigwedge x. x \in atom\ 'xs \Longrightarrow x \in supp\ T \Longrightarrow x \# f\ xs\ T$

and $f2: \bigwedge x. \llbracket supp\ T - atom\ 'xs = supp\ S - atom\ 'ys; x \in atom\ 'ys; x \in supp\ S \rrbracket \Longrightarrow x \# f\ xs\ T$

and $eqv: \bigwedge p. \llbracket p \cdot T = S; supp\ p \subseteq atom\ 'xs \cap supp\ T \cup atom\ 'ys \cap supp\ S; \rrbracket$

$p \cdot (\text{atom } \langle xs \cap \text{supp } T \rangle = \text{atom } \langle ys \cap \text{supp } S \rangle) \implies p \cdot (f \text{ xs } T) = f \text{ ys } S$
shows $f \text{ xs } T = f \text{ ys } S$
 <proof>

lemma *Abs-set-fcb2*:

fixes $as \ bs :: \text{atom set}$
and $x \ y :: 'b :: fs$
and $c :: 'c :: fs$
assumes $eq: [as]set. x = [bs]set. y$
and $fin: \text{finite } as \ \text{finite } bs$
and $fc1: as \#* f \ as \ x \ c$
and $fresh1: as \#* c$
and $fresh2: bs \#* c$
and $perm1: \bigwedge p. \text{supp } p \ \#* \ c \implies p \cdot (f \ as \ x \ c) = f (p \cdot as) (p \cdot x) \ c$
and $perm2: \bigwedge p. \text{supp } p \ \#* \ c \implies p \cdot (f \ bs \ y \ c) = f (p \cdot bs) (p \cdot y) \ c$
shows $f \ as \ x \ c = f \ bs \ y \ c$
 <proof>

lemma *Abs-res-fcb2*:

fixes $as \ bs :: \text{atom set}$
and $x \ y :: 'b :: fs$
and $c :: 'c :: fs$
assumes $eq: [as]res. x = [bs]res. y$
and $fin: \text{finite } as \ \text{finite } bs$
and $fc1: (as \cap \text{supp } x) \#* f (as \cap \text{supp } x) \ x \ c$
and $fresh1: as \#* c$
and $fresh2: bs \#* c$
and $perm1: \bigwedge p. \text{supp } p \ \#* \ c \implies p \cdot (f (as \cap \text{supp } x) \ x \ c) = f (p \cdot (as \cap \text{supp } x)) (p \cdot x) \ c$
and $perm2: \bigwedge p. \text{supp } p \ \#* \ c \implies p \cdot (f (bs \cap \text{supp } y) \ y \ c) = f (p \cdot (bs \cap \text{supp } y)) (p \cdot y) \ c$
shows $f (as \cap \text{supp } x) \ x \ c = f (bs \cap \text{supp } y) \ y \ c$
 <proof>

lemma *Abs-lst-fcb2*:

fixes $as \ bs :: \text{atom list}$
and $x \ y :: 'b :: fs$
and $c :: 'c :: fs$
assumes $eq: [as]lst. x = [bs]lst. y$
and $fc1: (\text{set } as) \#* f \ as \ x \ c$
and $fresh1: \text{set } as \ \#* \ c$
and $fresh2: \text{set } bs \ \#* \ c$
and $perm1: \bigwedge p. \text{supp } p \ \#* \ c \implies p \cdot (f \ as \ x \ c) = f (p \cdot as) (p \cdot x) \ c$
and $perm2: \bigwedge p. \text{supp } p \ \#* \ c \implies p \cdot (f \ bs \ y \ c) = f (p \cdot bs) (p \cdot y) \ c$
shows $f \ as \ x \ c = f \ bs \ y \ c$
 <proof>

lemma *Abs-lst1-fcb2*:
fixes $a\ b :: \text{atom}$
and $x\ y :: 'b :: \text{fs}$
and $c :: 'c :: \text{fs}$
assumes $e: [[a]]\text{lst. } x = [[b]]\text{lst. } y$
and $\text{fcb1}: a \# f\ a\ x\ c$
and $\text{fresh}: \{a, b\} \#* c$
and $\text{perm1}: \bigwedge p. \text{supp } p \#* c \implies p \cdot (f\ a\ x\ c) = f\ (p \cdot a)\ (p \cdot x)\ c$
and $\text{perm2}: \bigwedge p. \text{supp } p \#* c \implies p \cdot (f\ b\ y\ c) = f\ (p \cdot b)\ (p \cdot y)\ c$
shows $f\ a\ x\ c = f\ b\ y\ c$
 $\langle \text{proof} \rangle$

lemma *Abs-lst1-fcb2'*:
fixes $a\ b :: 'a :: \text{at-base}$
and $x\ y :: 'b :: \text{fs}$
and $c :: 'c :: \text{fs}$
assumes $e: [[\text{atom } a]]\text{lst. } x = [[\text{atom } b]]\text{lst. } y$
and $\text{fcb1}: \text{atom } a \# f\ a\ x\ c$
and $\text{fresh}: \{\text{atom } a, \text{atom } b\} \#* c$
and $\text{perm1}: \bigwedge p. \text{supp } p \#* c \implies p \cdot (f\ a\ x\ c) = f\ (p \cdot a)\ (p \cdot x)\ c$
and $\text{perm2}: \bigwedge p. \text{supp } p \#* c \implies p \cdot (f\ b\ y\ c) = f\ (p \cdot b)\ (p \cdot y)\ c$
shows $f\ a\ x\ c = f\ b\ y\ c$
 $\langle \text{proof} \rangle$

end

theory *Nominal2*

imports

Nominal2-Base Nominal2-Abs Nominal2-FCB

keywords

nominal-datatype :: thy-defn and

nominal-function nominal-inductive nominal-termination :: thy-goal-defn and

avoids binds

begin

$\langle \text{ML} \rangle$

34 Interface for *nominal-datatype*

$\langle \text{ML} \rangle$

Infrastructure for adding *-raw* to types and terms

$\langle \text{ML} \rangle$

35 Preparing and parsing of the specification

$\langle \text{ML} \rangle$

associates every SOME with the index in the list; drops NONEs

<ML>

adds an empty binding clause for every argument that is not already part of a binding clause

<ML>

end

```
theory Atoms
imports Nominal2-Base
begin
```

36 *nat-of* is an example of a function without finite support

```
lemma not-fresh-nat-of:
```

```
  shows  $\neg a \# \text{nat-of}$ 
```

```
<proof>
```

```
lemma supp-nat-of:
```

```
  shows supp nat-of = UNIV
```

```
<proof>
```

37 Manual instantiation of class *at*.

```
typedef name = {a. sort-of a = Sort "name" []}
```

```
<proof>
```

```
instantiation name :: at
```

```
begin
```

```
definition
```

```
   $p \cdot a = \text{Abs-name } (p \cdot \text{Rep-name } a)$ 
```

```
definition
```

```
   $\text{atom } a = \text{Rep-name } a$ 
```

```
instance
```

```
<proof>
```

```
end
```

```
lemma sort-of-atom-name:
```

```
  shows sort-of (atom (a::name)) = Sort "name" []
```

```
<proof>
```

Custom syntax for concrete atoms of type *at*

term $a::name$

38 Automatic instantiation of class *at*.

atom-decl $name2$

lemma

$sort-of (atom (a::name2)) \neq sort-of (atom (b::name))$
 $\langle proof \rangle$

example swappings

lemma

fixes $a b::atom$
assumes $sort-of a = sort-of b$
shows $(a \rightleftharpoons b) \cdot (a, b) = (b, a)$
 $\langle proof \rangle$

lemma

fixes $a b::name2$
shows $(a \leftrightarrow b) \cdot (a, b) = (b, a)$
 $\langle proof \rangle$

39 An example for multiple-sort atoms

datatype $ty =$

$TVar\ string$
 $| Fun\ ty\ ty\ (\leftarrow \rightarrow \rightarrow)$

primrec

$sort-of-ty::ty \Rightarrow atom-sort$

where

$sort-of-ty (TVar\ s) = Sort\ "TVar"\ [Sort\ s\ []]$
 $| sort-of-ty (Fun\ ty1\ ty2) = Sort\ "Fun"\ [sort-of-ty\ ty1,\ sort-of-ty\ ty2]$

lemma $sort-of-ty-eq-iff:$

shows $sort-of-ty\ x = sort-of-ty\ y \longleftrightarrow x = y$
 $\langle proof \rangle$

declare $sort-of-ty.simps [simp\ del]$

typedef $var = \{a.\ sort-of\ a \in range\ sort-of-ty\}$
 $\langle proof \rangle$

instantiation $var :: at-base$

begin

definition

$p \cdot a = Abs-var (p \cdot Rep-var\ a)$

definition

$atom\ a = Rep\text{-}var\ a$

instance

$\langle proof \rangle$

end

Constructor for variables.

definition

$Var :: nat \Rightarrow ty \Rightarrow var$

where

$Var\ x\ t = Abs\text{-}var\ (Atom\ (sort\text{-}of\text{-}ty\ t)\ x)$

lemma *Var-eq-iff* [simp]:

shows $Var\ x\ s = Var\ y\ t \iff x = y \wedge s = t$

$\langle proof \rangle$

lemma *sort-of-atom-var* [simp]:

$sort\text{-}of\ (atom\ (Var\ n\ ty)) = sort\text{-}of\text{-}ty\ ty$

$\langle proof \rangle$

lemma

assumes $\alpha \neq \beta$

shows $(Var\ x\ \alpha \leftrightarrow Var\ y\ \alpha) \cdot (Var\ x\ \alpha, Var\ x\ \beta) = (Var\ y\ \alpha, Var\ x\ \beta)$

$\langle proof \rangle$

Projecting out the type component of a variable.

definition

$ty\text{-}of :: var \Rightarrow ty$

where

$ty\text{-}of\ x = inv\ sort\text{-}of\text{-}ty\ (sort\text{-}of\ (atom\ x))$

Functions *Var/ty-of* satisfy many of the same properties as *Atom/sort-of*.

lemma *ty-of-Var* [simp]:

shows $ty\text{-}of\ (Var\ x\ t) = t$

$\langle proof \rangle$

lemma *ty-of-permute* [simp]:

shows $ty\text{-}of\ (p \cdot x) = ty\text{-}of\ x$

$\langle proof \rangle$

40 Tests with subtyping and automatic coercions

declare $[[coercion\text{-}enabled]]$

atom-decl *var1*

atom-decl *var2*

declare $[[\text{coercion } atom::var1 \Rightarrow atom]]$

declare $[[\text{coercion } atom::var2 \Rightarrow atom]]$

lemma

fixes $a::var1$ **and** $b::var2$

shows $a \# t \wedge b \# t$

$\langle proof \rangle$

lemma

fixes $as::var1$ *set*

shows $atom \text{ ' } as \#* t$

$\langle proof \rangle$

end

theory *Eqvt*

imports *Nominal2-Base*

begin

declare $[[\text{trace-eqvt} = \text{false}]]$

lemma

fixes $B::'a::pt$

shows $p \cdot (B = C)$

$\langle proof \rangle$

lemma

fixes $B::bool$

shows $p \cdot (B = C)$

$\langle proof \rangle$

lemma

fixes $B::bool$

shows $p \cdot (A \longrightarrow B = C)$

$\langle proof \rangle$

lemma

shows $p \cdot (\lambda(x::'a::pt). A \longrightarrow (B::'a \Rightarrow bool) x = C) = \text{foo}$

$\langle proof \rangle$

lemma

shows $p \cdot (\lambda B::bool. A \longrightarrow (B = C)) = foo$
 $\langle proof \rangle$

lemma
shows $p \cdot (\lambda x y. \exists z. x = z \wedge x = y \longrightarrow z \neq x) = foo$
 $\langle proof \rangle$

lemma
shows $p \cdot (\lambda f x. f (g (f x))) = foo$
 $\langle proof \rangle$

lemma
fixes $p q::perm$
and $x::'a::pt$
shows $p \cdot (q \cdot x) = foo$
 $\langle proof \rangle$

lemma
fixes $p q r::perm$
and $x::'a::pt$
shows $p \cdot (q \cdot r \cdot x) = foo$
 $\langle proof \rangle$

lemma
fixes $p r::perm$
shows $p \cdot (\lambda q::perm. q \cdot (r \cdot x)) = foo$
 $\langle proof \rangle$

lemma
fixes $C D::bool$
shows $B (p \cdot (C = D))$
 $\langle proof \rangle$

declare $[[trace-eqvt = false]]$

there is no raw eqvt-rule for The

lemma $p \cdot (THE x. P x) = foo$
 $\langle proof \rangle$

lemma
fixes $P :: ('b \Rightarrow bool) \Rightarrow ('b::pt) \Rightarrow ('a::pt)$
shows $p \cdot (P The) = foo$
 $\langle proof \rangle$

lemma
fixes $P :: ('a::pt) \Rightarrow ('b::pt) \Rightarrow bool$
shows $p \cdot (\lambda(a, b). P a b) = (\lambda(a, b). (p \cdot P) a b)$
 $\langle proof \rangle$

thm *eqvts*
thm *eqvts-raw*

$\langle ML \rangle$

end