

Nominal 2

Christian Urban, Stefan Berghofer, and Cezary Kaliszyk

May 26, 2024

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	3
2.1	Permutations form a (multiplicative) group	4
3	Implementation of swappings	5
4	Permutation Types	6
4.1	Permutations for atoms	7
4.2	Permutations for permutations	8
4.3	Permutations for functions	9
4.4	Permutations for booleans	9
4.5	Permutations for sets	10
4.6	Permutations for <i>unit</i>	11
4.7	Permutations for products	11
4.8	Permutations for sums	12
4.9	Permutations for <i>'a list</i>	12
4.10	Permutations for <i>'a option</i>	12
4.11	Permutations for <i>'a multiset</i>	13
4.12	Permutations for <i>'a fset</i>	13

4.13	Permutations for $('a, 'b)$ <i>finfun</i>	14
4.14	Permutations for <i>char</i> , <i>nat</i> , and <i>int</i>	14
5	Pure types	15
5.1	Types <i>char</i> , <i>nat</i> , and <i>int</i>	16
6	Infrastructure for Equivariance and <i>Perm-simp</i>	16
6.1	Basic functions about permutations	16
6.2	Eqvt infrastructure	16
6.3	<i>perm-simp</i> infrastructure	17
6.3.1	Equivariance for permutations and swapping	17
6.3.2	Equivariance of Logical Operators	18
6.3.3	Equivariance of Set operators	20
6.3.4	Equivariance for product operations	23
6.3.5	Equivariance for list operations	24
6.3.6	Equivariance for <i>'a option</i>	24
6.3.7	Equivariance for <i>'a fset</i>	24
6.3.8	Equivariance for $('a, 'b)$ <i>finfun</i>	25
7	Supp, Freshness and Supports	25
7.1	supp and fresh are equivariant	26
8	supports	27
9	Support w.r.t. relations	29
10	Finitely-supported types	29
10.1	Type <i>atom</i> is finitely-supported.	30
11	Type <i>perm</i> is finitely-supported.	30
12	Finite Support instances for other types	33
12.1	Type $'a \times 'b$ is finitely-supported.	33
12.2	Type $'a + 'b$ is finitely supported	34
12.3	Type <i>'a option</i> is finitely supported	34
12.3.1	Type <i>'a list</i> is finitely supported	35
13	Support and Freshness for Applications	36
13.1	Equivariance Predicate <i>eqvt</i> and <i>eqvt-at</i>	36
13.2	helper functions for <i>nominal-functions</i>	39
14	Support of Finite Sets of Finitely Supported Elements	40
14.1	Type <i>'a multiset</i> is finitely supported	43
14.2	Type <i>'a fset</i> is finitely supported	44
14.3	Type $('a, 'b)$ <i>finfun</i> is finitely supported	45

15 Freshness and Fresh-Star	46
16 Induction principle for permutations	49
17 Avoiding of atom sets	53
18 Renaming permutations	56
19 Concrete Atoms Types	59
20 Infrastructure for concrete atom types	62
20.1 Syntax for coercing at-elements to the atom-type	64
20.2 A lemma for proving instances of class <i>at</i>	64
21 Library functions for the nominal infrastructure	65
22 The freshness lemma according to Andy Pitts	65
23 Automation for creating concrete atom types	69
24 Automatic equivariance procedure for inductive definitions	69
25 Abstractions	69
26 Mono	70
27 Equivariance	70
28 Equivalence	71
29 General Abstractions	73
30 Strengthening the equivalence	74
31 Quotient types	79
32 Abstractions of single atoms	85
32.1 Renaming of bodies of abstractions	90
33 Infrastructure for building tuples of relations and functions	92
34 Interface for <i>nominal-datatype</i>	102
35 Preparing and parsing of the specification	113
36 <i>nat-of</i> is an example of a function without finite support	117
37 Manual instantiation of class <i>at</i>.	118

38 Automatic instantiation of class <i>at</i>.	119
39 An example for multiple-sort atoms	119
40 Tests with subtyping and automatic coercions	121

```

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 –
  have INFM i. sort-of (Atom s i) = s by simp
  moreover have inj (Atom s) by (simp add: inj-on-def)
  ultimately show INFM a. sort-of a = s by (rule INFM-inj)
qed
```

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

using *INFM-sort-of-eq* **unfolding** *INFM-iff-infinite* .

lemma *atom-infinite* [*simp*]:
shows *infinite* (*UNIV* :: *atom set*)
using *subset-UNIV infinite-sort-of-eq*
by (*rule infinite-super*)

lemma *obtain-atom*:
fixes *X* :: *atom set*
assumes *X*: *finite X*
obtains *a* where $a \notin X$ *sort-of a = s*
proof –
from *X* have *MOST a. a* $\notin X$
unfolding *MOST-iff-cofinite* by *simp*
with *INFM-sort-of-eq*
have *INFM a. sort-of a = s* $\wedge a \notin X$
by (*rule INFM-conjI*)
then obtain *a* where $a \notin X$ *sort-of a = s*
by (*auto elim: INFM-E*)
then show *?thesis ..*
qed

lemma *atom-components-eq-iff*:
fixes *a b* :: *atom*
shows $a = b \iff \text{sort-of } a = \text{sort-of } b \wedge \text{nat-of } a = \text{nat-of } b$
by (*induct a, induct b, simp*)

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 *perm* = *perm*
proof
show $id \in \text{perm}$ **unfolding** *perm-def* by *simp*
qed

lemma *permI*:
assumes *bij f* and *MOST x. f x = x* and $\bigwedge a. \text{sort-of } (f a) = \text{sort-of } a$
shows $f \in \text{perm}$
using *assms* **unfolding** *perm-def MOST-iff-cofinite* by *simp*

lemma *perm-is-bij*: $f \in \text{perm} \implies \text{bij } f$
unfolding *perm-def* by *simp*

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

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

unfolding *perm-def* **by** *simp*

lemma *perm-MOST*: $f \in \text{perm} \implies \text{MOST } x. f x = x$
unfolding *perm-def MOST-iff-cofinite* **by** *simp*

lemma *perm-id*: $\text{id} \in \text{perm}$
unfolding *perm-def* **by** *simp*

lemma *perm-comp*:
assumes $f: f \in \text{perm}$ **and** $g: g \in \text{perm}$
shows $(f \circ g) \in \text{perm}$
apply (*rule permI*)
apply (*rule bij-comp*)
apply (*rule perm-is-bij [OF g]*)
apply (*rule perm-is-bij [OF f]*)
apply (*rule MOST-rev-mp [OF perm-MOST [OF g]]*)
apply (*rule MOST-rev-mp [OF perm-MOST [OF f]]*)
apply (*simp*)
apply (*simp add: perm-is-sort-respecting [OF f]*)
apply (*simp add: perm-is-sort-respecting [OF g]*)
done

lemma *perm-inv*:
assumes $f: f \in \text{perm}$
shows $(\text{inv } f) \in \text{perm}$
apply (*rule permI*)
apply (*rule bij-imp-bij-inv*)
apply (*rule perm-is-bij [OF f]*)
apply (*rule MOST-mono [OF perm-MOST [OF f]]*)
apply (*erule subst, rule inv-f-f*)
apply (*rule bij-is-inj [OF perm-is-bij [OF f]]*)
apply (*rule perm-is-sort-respecting [OF f, THEN sym, THEN trans]*)
apply (*simp add: surj-f-inv-f [OF bij-is-surj [OF perm-is-bij [OF f]]]*)
done

lemma *bij-Rep-perm*: $\text{bij } (\text{Rep-perm } p)$
using *Rep-perm [of p]* **unfolding** *perm-def* **by** *simp*

lemma *finite-Rep-perm*: $\text{finite } \{a. \text{Rep-perm } p a \neq a\}$
using *Rep-perm [of p]* **unfolding** *perm-def* **by** *simp*

lemma *sort-of-Rep-perm*: $\text{sort-of } (\text{Rep-perm } p a) = \text{sort-of } a$
using *Rep-perm [of p]* **unfolding** *perm-def* **by** *simp*

lemma *Rep-perm-ext*:
 $\text{Rep-perm } p1 = \text{Rep-perm } p2 \implies p1 = p2$
by (*simp add: fun-eq-iff Rep-perm-inject [symmetric]*)

instance *perm* :: *size* ..

2.1 Permutations form a (multiplicative) group

instantiation *perm* :: *group-add*

begin

definition

$0 = \text{Abs-perm } id$

definition

$- p = \text{Abs-perm } (\text{inv } (\text{Rep-perm } p))$

definition

$p + q = \text{Abs-perm } (\text{Rep-perm } p \circ \text{Rep-perm } q)$

definition

$(p1 :: \text{perm}) - p2 = p1 + - p2$

lemma *Rep-perm-0*: *Rep-perm 0 = id*

unfolding *zero-perm-def*

by (*simp add: Abs-perm-inverse perm-id*)

lemma *Rep-perm-add*:

Rep-perm (p1 + p2) = Rep-perm p1 \circ Rep-perm p2

unfolding *plus-perm-def*

by (*simp add: Abs-perm-inverse perm-comp Rep-perm*)

lemma *Rep-perm-uminus*:

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

unfolding *uminus-perm-def*

by (*simp add: Abs-perm-inverse perm-inv Rep-perm*)

instance

apply *standard*

unfolding *Rep-perm-inject* [*symmetric*]

unfolding *minus-perm-def*

unfolding *Rep-perm-add*

unfolding *Rep-perm-uminus*

unfolding *Rep-perm-0*

by (*simp-all add: o-assoc inv-o-cancel* [*OF bij-is-inj* [*OF bij-Rep-perm*]])

end

3 Implementation of swappings

definition

swap :: *atom* \Rightarrow *atom* \Rightarrow *perm* (*'(- \rightleftharpoons -)'*)

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

unfolding *swap-def*

apply (rule *Abs-perm-inverse*)

apply (rule *permI*)

apply (auto simp: *bij-def inj-on-def surj-def*)[1]

apply (rule *MOST-rev-mp* [OF *MOST-neq(1)*] [of a])

apply (rule *MOST-rev-mp* [OF *MOST-neq(1)*] [of b])

apply (simp)

apply (simp)

done

lemmas *Rep-perm-simps* =

Rep-perm-0

Rep-perm-add

Rep-perm-uminus

Rep-perm-swap

lemma *swap-different-sorts* [simp]:

sort-of $a \neq \text{sort-of } b \implies (a \rightleftharpoons b) = 0$

by (rule *Rep-perm-ext*) (simp add: *Rep-perm-simps*)

lemma *swap-cancel*:

shows $(a \rightleftharpoons b) + (a \rightleftharpoons b) = 0$

and $(a \rightleftharpoons b) + (b \rightleftharpoons a) = 0$

by (rule-tac [!] *Rep-perm-ext*)

(simp-all add: *Rep-perm-simps fun-eq-iff*)

lemma *swap-self* [simp]:

$(a \rightleftharpoons a) = 0$

by (rule *Rep-perm-ext*, simp add: *Rep-perm-simps fun-eq-iff*)

lemma *minus-swap* [simp]:

$-(a \rightleftharpoons b) = (a \rightleftharpoons b)$

by (rule *minus-unique* [OF *swap-cancel(1)*])

lemma *swap-commute*:

$(a \rightleftharpoons b) = (b \rightleftharpoons a)$

by (rule *Rep-perm-ext*)

(simp add: *Rep-perm-swap fun-eq-iff*)

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  $(a \rightleftharpoons c) + (b \rightleftharpoons c) + (a \rightleftharpoons c) = (a \rightleftharpoons b)$ 
using assms
by (rule-tac Rep-perm-ext)
      (auto simp: Rep-perm-simps fun-eq-iff)

```

4 Permutation Types

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

```

class pt =
  fixes permute :: perm  $\Rightarrow$   $'a \Rightarrow 'a$  (- · - [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  $(p - q) \cdot x = p \cdot - q \cdot x$ 
  using permute-plus [of p - q x] by simp

```

```

lemma permute-minus-cancel [simp]:
  shows  $p \cdot - p \cdot x = x$ 
  and  $- p \cdot p \cdot x = x$ 
  unfolding permute-plus [symmetric] by simp-all

```

```

lemma permute-swap-cancel [simp]:
  shows  $(a \rightleftharpoons b) \cdot (a \rightleftharpoons b) \cdot x = x$ 
  unfolding permute-plus [symmetric]
  by (simp add: swap-cancel)

```

```

lemma permute-swap-cancel2 [simp]:
  shows  $(a \rightleftharpoons b) \cdot (b \rightleftharpoons a) \cdot x = x$ 
  unfolding permute-plus [symmetric]
  by (simp add: swap-commute)

```

```

lemma inj-permute [simp]:
  shows inj (permute p)
  by (rule inj-on-inverseI)
      (rule permute-minus-cancel)

```

```

lemma surj-permute [simp]:
  shows surj (permute p)
  by (rule surjI, rule permute-minus-cancel)

```

```

lemma bij-permute [simp]:
  shows bij (permute p)
  by (rule bijI [OF inj-permute surj-permute])

```

lemma *inv-permute*:
shows $\text{inv } (\text{permute } p) = \text{permute } (- p)$
by (*rule inv-equality*) (*simp-all*)

lemma *permute-minus*:
shows $\text{permute } (- p) = \text{inv } (\text{permute } p)$
by (*simp add: inv-permute*)

lemma *permute-eq-iff* [*simp*]:
shows $p \cdot x = p \cdot y \iff x = y$
by (*rule inj-permute* [*THEN inj-eq*])

end

4.1 Permutations for atoms

instantiation *atom* :: *pt*
begin

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

instance
apply *standard*
apply(*simp-all add: permute-atom-def Rep-perm-simps*)
done

end

lemma *sort-of-permute* [*simp*]:
shows $\text{sort-of } (p \cdot a) = \text{sort-of } a$
unfolding *permute-atom-def* **by** (*rule sort-of-Rep-perm*)

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)
unfolding *permute-atom-def*
by (*simp add: Rep-perm-swap*)

lemma *swap-atom-simps* [*simp*]:
 $\text{sort-of } a = \text{sort-of } b \implies (a \rightleftharpoons b) \cdot a = b$
 $\text{sort-of } a = \text{sort-of } b \implies (a \rightleftharpoons b) \cdot b = a$
 $c \neq a \implies c \neq b \implies (a \rightleftharpoons b) \cdot c = c$
unfolding *swap-atom* **by** *simp-all*

lemma *perm-eq-iff*:
fixes $p q :: \text{perm}$
shows $p = q \iff (\forall a::\text{atom. } p \cdot a = q \cdot a)$

unfolding *permute-atom-def*
by (*metis Rep-perm-ext ext*)

4.2 Permutations for permutations

instantiation *perm* :: *pt*
begin

definition

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

instance

apply *standard*
apply (*simp add: permute-perm-def*)
apply (*simp add: permute-perm-def algebra-simps*)
done

end

lemma *permute-self*:

shows $p \cdot p = p$
unfolding *permute-perm-def*
by (*simp add: add.assoc*)

lemma *permute-minus-self*:

shows $- p \cdot p = p$
unfolding *permute-perm-def*
by (*simp add: add.assoc*)

4.3 Permutations for functions

instantiation *fun* :: (*pt*, *pt*) *pt*
begin

definition

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

instance

apply *standard*
apply (*simp add: permute-fun-def*)
apply (*simp add: permute-fun-def minus-add*)
done

end

lemma *permute-fun-app-eq*:

shows $p \cdot (f x) = (p \cdot f) (p \cdot x)$
unfolding *permute-fun-def* **by** *simp*

lemma *permute-fun-comp*:

shows $p \cdot f = (\text{permute } p) \circ f \circ (\text{permute } (-p))$
by (*simp add: comp-def permute-fun-def*)

4.4 Permutations for booleans

instantiation $\text{bool} :: \text{pt}$
begin

definition $p \cdot (b::\text{bool}) = b$

instance
apply *standard*
apply(*simp-all add: permute-bool-def*)
done

end

lemma *permute-boolE*:
fixes $P::\text{bool}$
shows $p \cdot P \implies P$
by (*simp add: permute-bool-def*)

lemma *permute-boolI*:
fixes $P::\text{bool}$
shows $P \implies p \cdot P$
by(*simp add: permute-bool-def*)

4.5 Permutations for sets

instantiation $\text{set} :: (\text{pt}) \text{pt}$
begin

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

instance
apply *standard*
apply (*auto simp: permute-set-def*)
done

end

lemma *permute-set-eq*:
shows $p \cdot X = \{x. - p \cdot x \in X\}$
unfolding *permute-set-def*
by (*auto*) (*metis permute-minus-cancel(1)*)

lemma *permute-set-eq-image*:
shows $p \cdot X = \text{permute } p \text{ ` } X$
unfolding *permute-set-def* **by** *auto*

lemma *permute-set-eq-vimage*:
shows $p \cdot X = \text{permute } (- p) -' X$
unfolding *permute-set-eq vimage-def*
by *simp*

lemma *permute-finite* [*simp*]:
shows $\text{finite } (p \cdot X) = \text{finite } X$
unfolding *permute-set-eq-vimage*
using *bij-permute* **by** (*rule finite-vimage-iff*)

lemma *swap-set-not-in*:
assumes $a: a \notin S \ b \notin S$
shows $(a \rightleftharpoons b) \cdot S = S$
unfolding *permute-set-def*
using a **by** (*auto simp: swap-atom*)

lemma *swap-set-in*:
assumes $a: a \in S \ b \notin S \ \text{sort-of } a = \text{sort-of } b$
shows $(a \rightleftharpoons b) \cdot S \neq S$
unfolding *permute-set-def*
using a **by** (*auto simp: swap-atom*)

lemma *swap-set-in-eq*:
assumes $a: a \in S \ b \notin S \ \text{sort-of } a = \text{sort-of } b$
shows $(a \rightleftharpoons b) \cdot S = (S - \{a\}) \cup \{b\}$
unfolding *permute-set-def*
using a **by** (*auto simp: swap-atom*)

lemma *swap-set-both-in*:
assumes $a: a \in S \ b \in S$
shows $(a \rightleftharpoons b) \cdot S = S$
unfolding *permute-set-def*
using a **by** (*auto simp: swap-atom*)

lemma *mem-permute-iff*:
shows $(p \cdot x) \in (p \cdot X) \longleftrightarrow x \in X$
unfolding *permute-set-def*
by *auto*

lemma *empty-eqt*:
shows $p \cdot \{\} = \{\}$
unfolding *permute-set-def*
by (*simp*)

lemma *insert-eqt*:
shows $p \cdot (\text{insert } x \ A) = \text{insert } (p \cdot x) \ (p \cdot A)$
unfolding *permute-set-eq-image image-insert ..*

4.6 Permutations for *unit*

instantiation *unit* :: *pt*
begin

definition $p \cdot (u::unit) = u$

instance
 by *standard (simp-all add: permute-unit-def)*

end

4.7 Permutations for products

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

primrec
 permute-prod
where
 Pair-eqvt: $p \cdot (x, y) = (p \cdot x, p \cdot y)$

instance
 by *standard auto*

end

4.8 Permutations for sums

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

primrec
 permute-sum
where
 Inl-eqvt: $p \cdot (Inl\ x) = Inl\ (p \cdot x)$
 | *Inr-eqvt*: $p \cdot (Inr\ y) = Inr\ (p \cdot y)$

instance
 by *standard (case-tac [!] x, simp-all)*

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
by *standard* (*induct-tac* [!] *x*, *simp-all*)

end

lemma *set-eqvt*:
shows $p \cdot (\text{set } xs) = \text{set } (p \cdot xs)$
by (*induct* *xs*) (*simp-all* *add*: *empty-eqvt* *insert-eqvt*)

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
by *standard* (*induct-tac* [!] *x*, *simp-all*)

end

4.11 Permutations for 'a multiset

instantiation *multiset* :: (*pt*) *pt*
begin

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

instance
proof
fix *M* :: 'a multiset and *p* *q* :: perm
show $0 \cdot M = M$
 unfolding *permute-multiset-def*
 by (*induct-tac* *M*) (*simp-all*)
show $(p + q) \cdot M = p \cdot q \cdot M$
 unfolding *permute-multiset-def*
 by (*induct-tac* *M*) (*simp-all*)
qed

end

```

lemma permute-multiset [simp]:
  fixes  $M N :: ('a :: pt) \text{multiset}$ 
  shows  $(p \cdot \{\#\}) = (\{\#\} :: ('a :: pt) \text{multiset})$ 
  and  $(p \cdot \text{add-mset } x M) = \text{add-mset } (p \cdot x) (p \cdot M)$ 
  and  $(p \cdot (M + N)) = (p \cdot M) + (p \cdot N)$ 
  unfolding permute-multiset-def
  by (simp-all)

```

4.12 Permutations for $'a$ fset

```

instantiation fset :: (pt) pt
begin

```

```

context includes fset.lifting begin

```

```

lift-definition

```

```

  permute-fset :: perm  $\Rightarrow 'a \text{ fset} \Rightarrow 'a \text{ fset}$ 
is permute :: perm  $\Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set}$  by simp
end

```

```

context includes fset.lifting begin

```

```

instance

```

```

proof

```

```

  fix  $x :: 'a \text{ fset}$  and  $p q :: \text{perm}$ 
  show  $0 \cdot x = x$  by transfer simp
  show  $(p + q) \cdot x = p \cdot q \cdot x$  by transfer simp
qed
end

```

```

end

```

```

context includes fset.lifting

```

```

begin

```

```

lemma permute-fset [simp]:

```

```

  fixes  $S :: ('a :: pt) \text{fset}$ 
  shows  $(p \cdot \{\|\}) = (\{\|\} :: ('a :: pt) \text{fset})$ 
  and  $(p \cdot \text{finsert } x S) = \text{finsert } (p \cdot x) (p \cdot S)$ 
  apply (transfer, simp add: empty-eqvt)
  apply (transfer, simp add: insert-eqvt)
  done

```

```

lemma fset-eqvt:

```

```

  shows  $p \cdot (\text{fset } S) = \text{fset } (p \cdot S)$ 
  by transfer simp
end

```

4.13 Permutations for $('a, 'b)$ finfun

```

instantiation finfun :: (pt, pt) pt
begin

```


lift-definition

permute-funfun :: $perm \Rightarrow ('a, 'b) \text{ funfun} \Rightarrow ('a, 'b) \text{ funfun}$

is

permute :: $perm \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)$

apply(*simp add: permute-fun-comp*)

apply(*rule funfun-right-compose*)

apply(*rule funfun-left-compose*)

apply(*assumption*)

apply(*simp*)

done

instance

apply *standard*

apply(*transfer*)

apply(*simp*)

apply(*transfer*)

apply(*simp*)

done

end

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

instantiation *char* :: *pt*

begin

definition $p \cdot (c::char) = c$

instance

by *standard (simp-all add: permute-char-def)*

end

instantiation *nat* :: *pt*

begin

definition $p \cdot (n::nat) = n$

instance

by *standard (simp-all add: permute-nat-def)*

end

instantiation *int* :: *pt*

begin

definition $p \cdot (i::int) = i$

instance

by *standard* (*simp-all add: permute-int-def*)

end

5 Pure types

Pure types will have always empty support.

```
class pure = pt +
  assumes permute-pure:  $p \cdot x = x$ 
```

Types *unit* and *bool* are pure.

```
instance unit :: pure
proof qed (rule permute-unit-def)
```

```
instance bool :: pure
proof qed (rule permute-bool-def)
```

Other type constructors preserve purity.

```
instance fun :: (pure, pure) pure
by standard (simp add: permute-fun-def permute-pure)
```

```
instance set :: (pure) pure
by standard (simp add: permute-set-def permute-pure)
```

```
instance prod :: (pure, pure) pure
by standard (induct-tac x, simp add: permute-pure)
```

```
instance sum :: (pure, pure) pure
by standard (induct-tac x, simp-all add: permute-pure)
```

```
instance list :: (pure) pure
by standard (induct-tac x, simp-all add: permute-pure)
```

```
instance option :: (pure) pure
by standard (induct-tac x, simp-all add: permute-pure)
```

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

```
instance char :: pure
proof qed (rule permute-char-def)
```

```
instance nat :: pure
proof qed (rule permute-nat-def)
```

```
instance int :: pure
proof qed (rule permute-int-def)
```

6 Infrastructure for Equivariance and *Perm-simp*

6.1 Basic functions about permutations

ML-file \langle *nominal-basics.ML* \rangle

6.2 Eqvt infrastructure

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

ML-file \langle *nominal-thmdecls.ML* \rangle

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 $p \cdot (f \ x) \equiv (p \cdot f) \ (p \cdot x)$

unfolding *permute-fun-def* **by** *simp*

lemma *eqvt-lambda*:

fixes $f :: 'a::pt \Rightarrow 'b::pt$

shows $p \cdot f \equiv (\lambda x. \ p \cdot (f \ (\text{unpermute } p \ x)))$

unfolding *permute-fun-def unpermute-def* **by** *simp*

lemma *eqvt-bound*:

shows $p \cdot \text{unpermute } p \ x \equiv x$

unfolding *unpermute-def* **by** *simp*

provides *perm-simp* methods

ML-file \langle *nominal-permeq.ML* \rangle

```

method-setup perm-simp =
  ⟨Nominal-Permeq.args-parser >> Nominal-Permeq.perm-simp-meth⟩
  ⟨pushes permutations inside.⟩

method-setup perm-strict-simp =
  ⟨Nominal-Permeq.args-parser >> Nominal-Permeq.perm-strict-simp-meth⟩
  ⟨pushes permutations inside, raises an error if it cannot solve all permutations.⟩

simproc-setup perm-simproc (p · t) = ⟨fn - => fn ctxt => fn ctrm =>
  case Thm.term-of (Thm.dest-arg ctrm) of
    Free - => NONE
  | Var - => NONE
  | Const-⟨permute - for - -> => NONE
  | - =>
    let
      val thm = Nominal-Permeq.eqt-conv ctxt Nominal-Permeq.eqt-strict-config
    ctrm
      handle ERROR - => Thm.reflexive ctrm
    in
      if Thm.is-reflexive thm then NONE else SOME(thm)
    end
  ⟩

```

6.3.1 Equivariance for permutations and swapping

```

lemma permute-eqt:
  shows  $p \cdot (q \cdot x) = (p \cdot q) \cdot (p \cdot x)$ 
  unfolding permute-perm-def by simp

```

```

lemma permute-eqt-raw [eqt-raw]:
  shows  $p \cdot \text{permute} \equiv \text{permute}$ 
apply(simp add: fun-eq-iff permute-fun-def)
apply(subst permute-eqt)
apply(simp)
done

```

```

lemma zero-perm-eqt [eqt]:
  shows  $p \cdot (0::\text{perm}) = 0$ 
  unfolding permute-perm-def by simp

```

```

lemma add-perm-eqt [eqt]:
  fixes p p1 p2 :: perm
  shows  $p \cdot (p1 + p2) = p \cdot p1 + p \cdot p2$ 
  unfolding permute-perm-def
  by (simp add: perm-eq-iff)

```

```

lemma swap-eqt [eqt]:

```

shows $p \cdot (a \rightleftharpoons b) = (p \cdot a \rightleftharpoons p \cdot b)$
unfolding *permute-perm-def*
by (*auto simp: swap-atom perm-eq-iff*)

lemma *uminus-eqvt* [*eqvt*]:
fixes $p q::\text{perm}$
shows $p \cdot (- q) = - (p \cdot q)$
unfolding *permute-perm-def*
by (*simp add: diff-add-eq-diff-diff-swap*)

6.3.2 Equivariance of Logical Operators

lemma *eq-eqvt* [*eqvt*]:
shows $p \cdot (x = y) \longleftrightarrow (p \cdot x) = (p \cdot y)$
unfolding *permute-eq-iff permute-bool-def ..*

lemma *Not-eqvt* [*eqvt*]:
shows $p \cdot (\neg A) \longleftrightarrow \neg (p \cdot A)$
by (*simp add: permute-bool-def*)

lemma *conj-eqvt* [*eqvt*]:
shows $p \cdot (A \wedge B) \longleftrightarrow (p \cdot A) \wedge (p \cdot B)$
by (*simp add: permute-bool-def*)

lemma *imp-eqvt* [*eqvt*]:
shows $p \cdot (A \longrightarrow B) \longleftrightarrow (p \cdot A) \longrightarrow (p \cdot B)$
by (*simp add: permute-bool-def*)

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

lemma *all-eqvt* [*eqvt*]:
shows $p \cdot (\forall x. P x) = (\forall x. (p \cdot P) x)$
unfolding *All-def*
by (*perm-simp*) (*rule refl*)

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

lemma *ex-eqvt* [*eqvt*]:
shows $p \cdot (\exists x. P x) = (\exists x. (p \cdot P) x)$
unfolding *Ex-def*
by (*perm-simp*) (*rule refl*)

lemma *ex1-eqvt* [*eqvt*]:
shows $p \cdot (\exists!x. P x) = (\exists!x. (p \cdot P) x)$
unfolding *Ex1-def*
by (*perm-simp*) (*rule refl*)

lemma *if-eqvt* [*eqvt*]:
shows $p \cdot (\text{if } b \text{ then } x \text{ else } y) = (\text{if } p \cdot b \text{ then } p \cdot x \text{ else } p \cdot y)$

by (*simp add: permute-fun-def permute-bool-def*)

lemma *True-eqvt* [*eqvt*]:
shows $p \cdot \text{True} = \text{True}$
unfolding *permute-bool-def* ..

lemma *False-eqvt* [*eqvt*]:
shows $p \cdot \text{False} = \text{False}$
unfolding *permute-bool-def* ..

lemma *disj-eqvt* [*eqvt*]:
shows $p \cdot (A \vee B) \longleftrightarrow (p \cdot A) \vee (p \cdot B)$
by (*simp add: permute-bool-def*)

lemma *all-eqvt2*:
shows $p \cdot (\forall x. P x) = (\forall x. p \cdot P (- p \cdot x))$
by (*perm-simp add: permute-minus-cancel*) (*rule refl*)

lemma *ex-eqvt2*:
shows $p \cdot (\exists x. P x) = (\exists x. p \cdot P (- p \cdot x))$
by (*perm-simp add: permute-minus-cancel*) (*rule refl*)

lemma *ex1-eqvt2*:
shows $p \cdot (\exists!x. P x) = (\exists!x. p \cdot P (- p \cdot x))$
by (*perm-simp add: permute-minus-cancel*) (*rule refl*)

lemma *the-eqvt*:
assumes *unique*: $\exists!x. P x$
shows $(p \cdot (\text{THE } x. P x)) = (\text{THE } x. (p \cdot P) x)$
apply(*rule the1-equality [symmetric]*)
apply(*rule-tac p=-p in permute-boolE*)
apply(*perm-simp add: permute-minus-cancel*)
apply(*rule unique*)
apply(*rule-tac p=-p in permute-boolE*)
apply(*perm-simp add: permute-minus-cancel*)
apply(*rule theI'[OF unique]*)
done

lemma *the-eqvt2*:
assumes *unique*: $\exists!x. P x$
shows $(p \cdot (\text{THE } x. P x)) = (\text{THE } x. p \cdot P (- p \cdot x))$
apply(*rule the1-equality [symmetric]*)
apply(*simp only: ex1-eqvt2[symmetric]*)
apply(*simp add: permute-bool-def unique*)
apply(*simp add: permute-bool-def*)
apply(*rule theI'[OF unique]*)
done

6.3.3 Equivariance of Set operators

lemma *mem-eqvt* [eqvt]:

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

unfolding *permute-bool-def permute-set-def*

by (*auto*)

lemma *Collect-eqvt* [eqvt]:

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

unfolding *permute-set-eq permute-fun-def*

by (*auto simp: permute-bool-def*)

lemma *Bex-eqvt* [eqvt]:

shows $p \cdot (\exists x \in S. P x) = (\exists x \in (p \cdot S). (p \cdot P) x)$

unfolding *Bex-def* **by** *simp*

lemma *Ball-eqvt* [eqvt]:

shows $p \cdot (\forall x \in S. P x) = (\forall x \in (p \cdot S). (p \cdot P) x)$

unfolding *Ball-def* **by** *simp*

lemma *image-eqvt* [eqvt]:

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

unfolding *image-def* **by** *simp*

lemma *Image-eqvt* [eqvt]:

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

unfolding *Image-def* **by** *simp*

lemma *UNIV-eqvt* [eqvt]:

shows $p \cdot UNIV = UNIV$

unfolding *UNIV-def*

by (*perm-simp*) (*rule refl*)

lemma *inter-eqvt* [eqvt]:

shows $p \cdot (A \cap B) = (p \cdot A) \cap (p \cdot B)$

unfolding *Int-def* **by** *simp*

lemma *Inter-eqvt* [eqvt]:

shows $p \cdot \bigcap S = \bigcap (p \cdot S)$

unfolding *Inter-eq* **by** *simp*

lemma *union-eqvt* [eqvt]:

shows $p \cdot (A \cup B) = (p \cdot A) \cup (p \cdot B)$

unfolding *Un-def* **by** *simp*

lemma *Union-eqvt* [eqvt]:

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

unfolding *Union-eq*

by *perm-simp rule*

lemma *Diff-eqvt* [eqvt]:
fixes $A B :: 'a::pt\ set$
shows $p \cdot (A - B) = (p \cdot A) - (p \cdot B)$
unfolding *set-diff-eq* **by** *simp*

lemma *Compl-eqvt* [eqvt]:
fixes $A :: 'a::pt\ set$
shows $p \cdot (- A) = - (p \cdot A)$
unfolding *Compl-eq-Diff-UNIV* **by** *simp*

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

lemma *psubset-eqvt* [eqvt]:
shows $p \cdot (S \subset T) \longleftrightarrow (p \cdot S) \subset (p \cdot T)$
unfolding *psubset-eq* **by** *simp*

lemma *vimage-eqvt* [eqvt]:
shows $p \cdot (f -^{\cdot} A) = (p \cdot f) -^{\cdot} (p \cdot A)$
unfolding *vimage-def* **by** *simp*

lemma *foldr-eqvt*[eqvt]:
 $p \cdot \text{foldr } f \text{ } xs = \text{foldr } (p \cdot f) \text{ } (p \cdot xs)$
apply(*induct xs*)
apply(*simp-all*)
apply(*perm-simp exclude: foldr*)
apply(*simp*)
done

lemma *Sigma-eqvt*:
shows $p \cdot (X \times Y) = (p \cdot X) \times (p \cdot Y)$
unfolding *Sigma-def*
by (*perm-simp*) (*rule refl*)

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]: $p \cdot (\text{Inf } X) = \text{Inf } (p \cdot (X :: 'a :: \{complete-lattice, pt\} set))$

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


```

instance
  apply standard
  unfolding le-bool-def
  apply(perm-simp)
  apply(rule refl)
  done

end

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

  instance
    apply standard
    unfolding le-fun-def
    apply(perm-simp)
    apply(rule refl)
    done

  end

  instantiation bool :: inf-eqvt
  begin

    instance
      apply standard
      unfolding Inf-bool-def
      apply(perm-simp)
      apply(rule refl)
      done

    end

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

      instance
        apply standard
        unfolding Inf-fun-def
        apply(perm-simp)
        apply(rule refl)
        done

      end

      lemma lfp-eqvt [eqvt]:
        fixes F::('a ⇒ 'b) ⇒ ('a::pt ⇒ 'b::{inf-eqvt, le-eqvt})
        shows p · (lfp F) = lfp (p · F)
      unfolding lfp-def

```

by *simp*

lemma *finite-eqvt* [*eqvt*]:
 shows $p \cdot \text{finite } A = \text{finite } (p \cdot A)$
unfolding *finite-def*
by *simp*

lemma *fun-upd-eqvt*[*eqvt*]:
 shows $p \cdot (f(x := y)) = (p \cdot f)((p \cdot x) := (p \cdot y))$
unfolding *fun-upd-def*
by *simp*

lemma *comp-eqvt* [*eqvt*]:
 shows $p \cdot (f \circ g) = (p \cdot f) \circ (p \cdot g)$
unfolding *comp-def*
by *simp*

6.3.4 Equivariance for product operations

lemma *fst-eqvt* [*eqvt*]:
 shows $p \cdot (\text{fst } x) = \text{fst } (p \cdot x)$
 by (*cases x*) *simp*

lemma *snd-eqvt* [*eqvt*]:
 shows $p \cdot (\text{snd } x) = \text{snd } (p \cdot x)$
 by (*cases x*) *simp*

lemma *split-eqvt* [*eqvt*]:
 shows $p \cdot (\text{case-prod } P x) = \text{case-prod } (p \cdot P) (p \cdot x)$
 unfolding *split-def*
 by *simp*

6.3.5 Equivariance for list operations

lemma *append-eqvt* [*eqvt*]:
 shows $p \cdot (xs @ ys) = (p \cdot xs) @ (p \cdot ys)$
 by (*induct xs*) *auto*

lemma *rev-eqvt* [*eqvt*]:
 shows $p \cdot (\text{rev } xs) = \text{rev } (p \cdot xs)$
 by (*induct xs*) (*simp-all add: append-eqvt*)

lemma *map-eqvt* [*eqvt*]:
 shows $p \cdot (\text{map } f xs) = \text{map } (p \cdot f) (p \cdot xs)$
 by (*induct xs*) (*simp-all*)

lemma *removeAll-eqvt* [*eqvt*]:
 shows $p \cdot (\text{removeAll } x xs) = \text{removeAll } (p \cdot x) (p \cdot xs)$
 by (*induct xs*) (*auto*)

```

lemma filter-eqvt [eqvt]:
  shows  $p \cdot (\text{filter } f \text{ } xs) = \text{filter } (p \cdot f) (p \cdot xs)$ 
apply(induct xs)
apply(simp)
apply(simp only: filter.simps permute-list.simps if-eqvt)
apply(simp only: permute-fun-app-eq)
done

```

```

lemma distinct-eqvt [eqvt]:
  shows  $p \cdot (\text{distinct } xs) = \text{distinct } (p \cdot xs)$ 
apply(induct xs)
apply(simp add: permute-bool-def)
apply(simp add: conj-eqvt Not-eqvt mem-eqvt set-eqvt)
done

```

```

lemma length-eqvt [eqvt]:
  shows  $p \cdot (\text{length } xs) = \text{length } (p \cdot xs)$ 
by (induct xs) (simp-all add: permute-pure)

```

6.3.6 Equivariance for 'a option

```

lemma map-option-eqvt[eqvt]:
  shows  $p \cdot (\text{map-option } f \text{ } x) = \text{map-option } (p \cdot f) (p \cdot x)$ 
by (cases x) (simp-all)

```

6.3.7 Equivariance for 'a fset

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

```

lemma in-fset-eqvt:
  shows  $(p \cdot (x \in S)) = ((p \cdot x) \in (p \cdot S))$ 
by transfer simp

```

```

lemma union-fset-eqvt [eqvt]:
  shows  $(p \cdot (S \cup T)) = ((p \cdot S) \cup (p \cdot T))$ 
by (induct S) (simp-all)

```

```

lemma inter-fset-eqvt [eqvt]:
  shows  $(p \cdot (S \cap T)) = ((p \cdot S) \cap (p \cdot T))$ 
by transfer simp

```

```

lemma subset-fset-eqvt [eqvt]:
  shows  $(p \cdot (S \subseteq T)) = ((p \cdot S) \subseteq (p \cdot T))$ 
by transfer simp

```

```

lemma map-fset-eqvt [eqvt]:
  shows  $p \cdot (f \upharpoonright S) = (p \cdot f) \upharpoonright (p \cdot S)$ 
by transfer simp
end

```

6.3.8 Equivariance for ('a, 'b) finfun

lemma *finfun-update-eqv* [eqvt]:

shows $(p \cdot (\text{finfun-update } f \ a \ b)) = \text{finfun-update } (p \cdot f) \ (p \cdot a) \ (p \cdot b)$
by (*transfer*) (*simp*)

lemma *finfun-const-eqv* [eqvt]:

shows $(p \cdot (\text{finfun-const } b)) = \text{finfun-const } (p \cdot b)$
by (*transfer*) (*simp*)

lemma *finfun-apply-eqv* [eqvt]:

shows $(p \cdot (\text{finfun-apply } f \ b)) = \text{finfun-apply } (p \cdot f) \ (p \cdot b)$
by (*transfer*) (*simp*)

7 Supp, Freshness and Supports

context *pt*

begin

definition

supp :: 'a \Rightarrow atom set

where

supp *x* = {*a*. infinite {*b*. (*a* \rightleftharpoons *b*) \cdot *x* \neq *x*}}

definition

fresh :: atom \Rightarrow 'a \Rightarrow bool (- \sharp - [55, 55] 55)

where

a \sharp *x* \equiv *a* \notin *supp* *x*

end

lemma *supp-conv-fresh*:

shows *supp* *x* = {*a*. \neg *a* \sharp *x*}
unfolding *fresh-def* **by** *simp*

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 (*cases*)

assume *a* = *b* \vee *c* = *b*

with *assms* **show** (*a* \rightleftharpoons *b*) \cdot *x* = *x* **by** *auto*

next

assume *: \neg (*a* = *b* \vee *c* = *b*)

have ((*a* \rightleftharpoons *c*) + (*b* \rightleftharpoons *c*) + (*a* \rightleftharpoons *c*)) \cdot *x* = *x*

using *assms* **by** *simp*

also have (*a* \rightleftharpoons *c*) + (*b* \rightleftharpoons *c*) + (*a* \rightleftharpoons *c*) = (*a* \rightleftharpoons *b*)

using *assms* * by (*simp add: swap-triple*)
 finally show $(a \rightleftharpoons b) \cdot x = x$.
 qed

lemma *swap-fresh-fresh*:

assumes $a: a \# x$
 and $b: b \# x$
 shows $(a \rightleftharpoons b) \cdot x = x$
 proof (*cases*)
 assume *asm*: *sort-of* $a = \text{sort-of } b$
 have *finite* $\{c. (a \rightleftharpoons c) \cdot x \neq x\}$ *finite* $\{c. (b \rightleftharpoons c) \cdot x \neq x\}$
 using *a b unfolding fresh-def supp-def by simp-all*
 then have *finite* $(\{c. (a \rightleftharpoons c) \cdot x \neq x\} \cup \{c. (b \rightleftharpoons c) \cdot x \neq x\})$ by *simp*
 then obtain *c*
 where $(a \rightleftharpoons c) \cdot x = x$ $(b \rightleftharpoons c) \cdot x = x$ *sort-of* $c = \text{sort-of } b$
 by (*rule obtain-atom*) (*auto*)
 then show $(a \rightleftharpoons b) \cdot x = x$ using *asm* by (*rule-tac swap-rel-trans*) (*simp-all*)
 next
 assume *sort-of* $a \neq \text{sort-of } b$
 then show $(a \rightleftharpoons b) \cdot x = x$ by *simp*
 qed

7.1 supp and fresh are equivariant

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

shows $p \cdot (\text{supp } x) = \text{supp } (p \cdot x)$
 unfolding *supp-def* by *simp*

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

shows $p \cdot (a \# x) = (p \cdot a) \# (p \cdot x)$
 unfolding *fresh-def* by *simp*

lemma *fresh-permute-iff*:

shows $(p \cdot a) \# (p \cdot x) \longleftrightarrow a \# x$
 by (*simp only: fresh-eqvt[symmetric] permute-bool-def*)

lemma *fresh-permute-left*:

shows $a \# p \cdot x \longleftrightarrow - p \cdot a \# x$

proof

assume $a \# p \cdot x$

then have $- p \cdot a \# - p \cdot p \cdot x$ by (*simp only: fresh-permute-iff*)

then show $- p \cdot a \# x$ by *simp*

next

assume $- p \cdot a \# x$

then have $p \cdot - p \cdot a \# p \cdot x$ by (*simp only: fresh-permute-iff*)

then show $a \# p \cdot x$ by *simp*

qed

8 supports

definition

supports :: atom set \Rightarrow 'a::pt \Rightarrow bool (infixl supports 80)

where

S 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 :: atom set

and x :: 'a::pt

assumes $a1$: S supports x

and $a2$: finite S

shows $(\text{supp } x) \subseteq S$

proof (rule ccontr)

assume $\neg (\text{supp } x \subseteq S)$

then obtain a **where** $b1$: $a \in \text{supp } x$ **and** $b2$: $a \notin S$ **by** auto

from $a1$ $b2$ **have** $\forall b. b \notin S \longrightarrow (a \rightleftharpoons b) \cdot x = x$ **unfolding** *supports-def* **by** auto

then have $\{b. (a \rightleftharpoons b) \cdot x \neq x\} \subseteq S$ **by** auto

with $a2$ **have** finite $\{b. (a \rightleftharpoons b) \cdot x \neq x\}$ **by** (simp add: finite-subset)

then have $a \notin (\text{supp } x)$ **unfolding** *supp-def* **by** simp

with $b1$ **show** False **by** simp

qed

lemma *supports-finite*:

fixes S :: atom set

and x :: 'a::pt

assumes $a1$: S supports x

and $a2$: finite S

shows finite $(\text{supp } x)$

proof –

have $(\text{supp } x) \subseteq S$ **using** $a1$ $a2$ **by** (rule *supp-is-subset*)

then show finite $(\text{supp } x)$ **using** $a2$ **by** (simp add: finite-subset)

qed

lemma *supp-supports*:

fixes x :: 'a::pt

shows $(\text{supp } x)$ supports x

unfolding *supports-def*

proof (intro strip)

fix $a b$

assume $a \notin (\text{supp } x) \wedge b \notin (\text{supp } x)$

then have $a \not\# x$ **and** $b \not\# x$ **by** (simp-all add: fresh-def)

then show $(a \rightleftharpoons b) \cdot x = x$ **by** (simp add: swap-fresh-fresh)

qed

lemma *supports-fresh*:

fixes x :: 'a::pt

assumes $a1$: S supports x

```

and a2: finite S
and a3: a ∉ S
shows a # x
unfolding fresh-def
proof -
  have (supp x) ⊆ S using a1 a2 by (rule supp-is-subset)
  then show a ∉ (supp x) using a3 by auto
qed

```

```

lemma supp-is-least-supports:
  fixes S :: atom set
  and x :: 'a::pt
  assumes a1: S supports x
  and a2: finite S
  and a3: ⋀ S'. finite S' ⟹ (S' supports x) ⟹ S ⊆ S'
  shows (supp x) = S
proof (rule equalityI)
  show (supp x) ⊆ S using a1 a2 by (rule supp-is-subset)
  with a2 have fin: finite (supp x) by (rule rev-finite-subset)
  have (supp x) supports x by (rule supp-supports)
  with fin a3 show S ⊆ supp x by blast
qed

```

```

lemma subsetCI:
  shows (⋀ x. x ∈ A ⟹ x ∉ B ⟹ False) ⟹ A ⊆ B
  by auto

```

```

lemma finite-supp-unique:
  assumes a1: S supports x
  assumes a2: finite S
  assumes a3: ⋀ a b. [a ∈ S; b ∉ S; sort-of a = sort-of b] ⟹ (a ⇔ b) · x ≠ x
  shows (supp x) = S
  using a1 a2
proof (rule supp-is-least-supports)
  fix S'
  assume finite S' and S' supports x
  show S ⊆ S'
  proof (rule subsetCI)
    fix a
    assume a ∈ S and a ∉ S'
    have finite (S ∪ S')
      using ⟨finite S⟩ ⟨finite S'⟩ by simp
    then obtain b where b ∉ S ∪ S' and sort-of b = sort-of a
      by (rule obtain-atom)
    then have b ∉ S and b ∉ S' and sort-of a = sort-of b
      by simp-all
    then have (a ⇔ b) · x = x
      using ⟨a ∉ S'⟩ ⟨S' supports x⟩ by (simp add: supports-def)

```

```

moreover have  $(a \rightleftharpoons b) \cdot x \neq x$ 
  using  $\langle a \in S \rangle \langle b \notin S \rangle \langle \text{sort-of } a = \text{sort-of } b \rangle$ 
  by  $(\text{rule } a3)$ 
ultimately show False by simp
qed
qed

```

9 Support w.r.t. relations

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

definition

```

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

```

10 Finitely-supported types

```

class fs = pt +
  assumes finite-supp: finite (supp  $x$ )

```

lemma *pure-supp*:

```

fixes  $x::'a::\text{pure}$ 
shows  $\text{supp } x = \{\}$ 
unfolding supp-def by  $(\text{simp add: permute-pure})$ 

```

lemma *pure-fresh*:

```

fixes  $x::'a::\text{pure}$ 
shows  $a \# x$ 
unfolding fresh-def by  $(\text{simp add: pure-supp})$ 

```

instance *pure* < *fs*

```

by standard  $(\text{simp add: pure-supp})$ 

```

10.1 Type *atom* is finitely-supported.

lemma *supp-atom*:

```

shows  $\text{supp } a = \{a\}$ 
apply  $(\text{rule } \text{finite-supp-unique})$ 
apply  $(\text{clarsimp simp add: supports-def})$ 
apply simp
apply simp
done

```

lemma *fresh-atom*:

```

shows  $a \# b \longleftrightarrow a \neq b$ 
unfolding fresh-def supp-atom by simp

```

instance *atom* :: *fs*

```

by standard  $(\text{simp add: supp-atom})$ 

```


11 Type *perm* is finitely-supported.

lemma *perm-swap-eq*:

shows $(a \rightleftharpoons b) \cdot p = p \longleftrightarrow (p \cdot (a \rightleftharpoons b)) = (a \rightleftharpoons b)$

unfolding *permute-perm-def*

by (*metis add-diff-cancel minus-perm-def*)

lemma *supports-perm*:

shows $\{a. p \cdot a \neq a\}$ *supports* *p*

unfolding *supports-def*

unfolding *perm-swap-eq*

by (*simp add: swap-eqvt*)

lemma *finite-perm-lemma*:

shows *finite* $\{a::atom. p \cdot a \neq a\}$

using *finite-Rep-perm* [of *p*]

unfolding *permute-atom-def* .

lemma *supp-perm*:

shows $\text{supp } p = \{a. p \cdot a \neq a\}$

apply (*rule finite-supp-unique*)

apply (*rule supports-perm*)

apply (*rule finite-perm-lemma*)

apply (*simp add: perm-swap-eq swap-eqvt*)

apply (*auto simp: perm-eq-iff swap-atom*)

done

lemma *fresh-perm*:

shows $a \# p \longleftrightarrow p \cdot a = a$

unfolding *fresh-def*

by (*simp add: supp-perm*)

lemma *supp-swap*:

shows $\text{supp } (a \rightleftharpoons b) = (\text{if } a = b \vee \text{sort-of } a \neq \text{sort-of } b \text{ then } \{\} \text{ else } \{a, b\})$

by (*auto simp: supp-perm swap-atom*)

lemma *fresh-swap*:

shows $a \# (b \rightleftharpoons c) \longleftrightarrow (\text{sort-of } b \neq \text{sort-of } c) \vee b = c \vee (a \# b \wedge a \# c)$

by (*simp add: fresh-def supp-swap supp-atom*)

lemma *fresh-zero-perm*:

shows $a \# (0::perm)$

unfolding *fresh-perm* **by** *simp*

lemma *supp-zero-perm*:

shows $\text{supp } (0::perm) = \{\}$

unfolding *supp-perm* **by** *simp*

lemma *fresh-plus-perm*:

```

fixes p q::perm
assumes a # p a # q
shows a # (p + q)
using assms
unfolding fresh-def
by (auto simp: supp-perm)

```

```

lemma supp-plus-perm:
fixes p q::perm
shows  $\text{supp } (p + q) \subseteq \text{supp } p \cup \text{supp } q$ 
by (auto simp: supp-perm)

```

```

lemma fresh-minus-perm:
fixes p::perm
shows  $a \# (- p) \longleftrightarrow a \# p$ 
unfolding fresh-def
unfolding supp-perm
apply(simp)
apply(metis permute-minus-cancel)
done

```

```

lemma supp-minus-perm:
fixes p::perm
shows  $\text{supp } (- p) = \text{supp } p$ 
unfolding supp-conv-fresh
by (simp add: fresh-minus-perm)

```

```

lemma plus-perm-eq:
fixes p q::perm
assumes asm:  $\text{supp } p \cap \text{supp } q = \{\}$ 
shows  $p + q = q + p$ 
unfolding perm-eq-iff
proof
  fix a::atom
  show  $(p + q) \cdot a = (q + p) \cdot a$ 
  proof -
    { assume  $a \notin \text{supp } p \ a \notin \text{supp } q$ 
      then have  $(p + q) \cdot a = (q + p) \cdot a$ 
        by (simp add: supp-perm)
    }
  moreover
    { assume  $a \in \text{supp } p \ a \notin \text{supp } q$ 
      then have  $p \cdot a \in \text{supp } p$  by (simp add: supp-perm)
      then have  $p \cdot a \notin \text{supp } q$  using asm by auto
      with a have  $(p + q) \cdot a = (q + p) \cdot a$ 
        by (simp add: supp-perm)
    }
  moreover
    { assume  $a \notin \text{supp } p \ a \in \text{supp } q$ 

```

```

    then have  $q \cdot a \in \text{supp } q$  by (simp add: supp-perm)
    then have  $q \cdot a \notin \text{supp } p$  using asm by auto
    with a have  $(p + q) \cdot a = (q + p) \cdot a$ 
      by (simp add: supp-perm)
  }
  ultimately show  $(p + q) \cdot a = (q + p) \cdot a$ 
    using asm by blast
qed
qed

lemma supp-plus-perm-eq:
  fixes p q::perm
  assumes asm:  $\text{supp } p \cap \text{supp } q = \{\}$ 
  shows  $\text{supp } (p + q) = \text{supp } p \cup \text{supp } q$ 
proof -
  { fix a::atom
    assume  $a \in \text{supp } p$ 
    then have  $a \notin \text{supp } q$  using asm by auto
    then have  $a \in \text{supp } (p + q)$  using  $\langle a \in \text{supp } p \rangle$ 
      by (simp add: supp-perm)
  }
  moreover
  { fix a::atom
    assume  $a \in \text{supp } q$ 
    then have  $a \notin \text{supp } p$  using asm by auto
    then have  $a \in \text{supp } (q + p)$  using  $\langle a \in \text{supp } q \rangle$ 
      by (simp add: supp-perm)
    then have  $a \in \text{supp } (p + q)$  using asm plus-perm-eq
      by metis
  }
  ultimately have  $\text{supp } p \cup \text{supp } q \subseteq \text{supp } (p + q)$ 
    by blast
  then show  $\text{supp } (p + q) = \text{supp } p \cup \text{supp } q$  using supp-plus-perm
    by blast
qed

lemma perm-eq-iff2:
  fixes p q :: perm
  shows  $p = q \iff (\forall a::atom \in \text{supp } p \cup \text{supp } q. p \cdot a = q \cdot a)$ 
  unfolding perm-eq-iff
  apply(auto)
  apply(case-tac a # p ^ a # q)
  apply(simp add: fresh-perm)
  apply(simp add: fresh-def)
  done

instance perm :: fs
  by standard (simp add: supp-perm finite-perm-lemma)

```

12 Finite Support instances for other types

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

lemma *supp-Pair*:

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

by (*simp add: supp-def Collect-imp-eq Collect-neg-eq*)

lemma *fresh-Pair*:

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

by (*simp add: fresh-def supp-Pair*)

lemma *supp-Unit*:

shows $\text{supp } () = \{\}$

by (*simp add: supp-def*)

lemma *fresh-Unit*:

shows $a \# ()$

by (*simp add: fresh-def supp-Unit*)

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

apply *standard*

apply (*case-tac x*)

apply (*simp add: supp-Pair finite-supp*)

done

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

lemma *supp-Inl*:

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

by (*simp add: supp-def*)

lemma *supp-Inr*:

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

by (*simp add: supp-def*)

lemma *fresh-Inl*:

shows $a \# \text{Inl } x \longleftrightarrow a \# x$

by (*simp add: fresh-def supp-Inl*)

lemma *fresh-Inr*:

shows $a \# \text{Inr } y \longleftrightarrow a \# y$

by (*simp add: fresh-def supp-Inr*)

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

apply *standard*

apply (*case-tac x*)

apply (*simp-all add: supp-Inl supp-Inr finite-supp*)

done

12.3 Type 'a option is finitely supported

lemma *supp-None*:

shows $\text{supp } \text{None} = \{\}$

by (*simp add: supp-def*)

lemma *supp-Some*:

shows $\text{supp } (\text{Some } x) = \text{supp } x$

by (*simp add: supp-def*)

lemma *fresh-None*:

shows $a \# \text{None}$

by (*simp add: fresh-def supp-None*)

lemma *fresh-Some*:

shows $a \# \text{Some } x \longleftrightarrow a \# x$

by (*simp add: fresh-def supp-Some*)

instance *option* :: (*fs*) *fs*

apply *standard*

apply (*induct-tac x*)

apply (*simp-all add: supp-None supp-Some finite-supp*)

done

12.3.1 Type 'a list is finitely supported

lemma *supp-Nil*:

shows $\text{supp } [] = \{\}$

by (*simp add: supp-def*)

lemma *fresh-Nil*:

shows $a \# []$

by (*simp add: fresh-def supp-Nil*)

lemma *supp-Cons*:

shows $\text{supp } (x \# xs) = \text{supp } x \cup \text{supp } xs$

by (*simp add: supp-def Collect-imp-eq Collect-neg-eq*)

lemma *fresh-Cons*:

shows $a \# (x \# xs) \longleftrightarrow a \# x \wedge a \# xs$

by (*simp add: fresh-def supp-Cons*)

lemma *supp-append*:

shows $\text{supp } (xs @ ys) = \text{supp } xs \cup \text{supp } ys$

by (*induct xs*) (*auto simp: supp-Nil supp-Cons*)

lemma *fresh-append*:

shows $a \# (xs @ ys) \longleftrightarrow a \# xs \wedge a \# ys$

by (*induct xs*) (*simp-all add: fresh-Nil fresh-Cons*)

lemma *supp-rev*:
shows $\text{supp } (\text{rev } xs) = \text{supp } xs$
by (*induct xs*) (*auto simp: supp-append supp-Cons supp-Nil*)

lemma *fresh-rev*:
shows $a \# \text{rev } xs \longleftrightarrow a \# xs$
by (*induct xs*) (*auto simp: fresh-append fresh-Cons fresh-Nil*)

lemma *supp-removeAll*:
fixes $x::\text{atom}$
shows $\text{supp } (\text{removeAll } x \ xs) = \text{supp } xs - \{x\}$
by (*induct xs*)
(auto simp: supp-Nil supp-Cons supp-atom)

lemma *supp-of-atom-list*:
fixes $as::\text{atom list}$
shows $\text{supp } as = \text{set } as$
by (*induct as*)
(simp-all add: supp-Nil supp-Cons supp-atom)

instance *list* :: $(fs) \ fs$
apply *standard*
apply (*induct-tac x*)
apply (*simp-all add: supp-Nil supp-Cons finite-supp*)
done

13 Support and Freshness for Applications

lemma *fresh-conv-MOST*:
shows $a \# x \longleftrightarrow (\text{MOST } b. (a \rightleftharpoons b) \cdot x = x)$
unfolding *fresh-def supp-def*
unfolding *MOST-iff-cofinite* **by** *simp*

lemma *fresh-fun-app*:
assumes $a \# f$ **and** $a \# x$
shows $a \# f \ x$
using *assms*
unfolding *fresh-conv-MOST*
unfolding *permute-fun-app-eq*
by (*elim MOST-rev-mp*) (*simp*)

lemma *supp-fun-app*:
shows $\text{supp } (f \ x) \subseteq (\text{supp } f) \cup (\text{supp } x)$
using *fresh-fun-app*
unfolding *fresh-def*
by *auto*

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*

unfolding *eqvt-def* **by** (*simp add: permute-bool-def*)

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 $(\bigwedge p. p \cdot f \equiv f) \implies eqvt\ f$

unfolding *eqvt-def*

by *simp*

lemma *eqvt-at-perm*:

assumes *eqvt-at f x*

shows *eqvt-at f (q · x)*

proof –

{ **fix** *p::perm*

have $p \cdot (f\ (q \cdot x)) = p \cdot q \cdot (f\ x)$

using *assms* **by** (*simp add: eqvt-at-def*)

also have $\dots = (p + q) \cdot (f\ x)$ **by** *simp*

also have $\dots = f\ ((p + q) \cdot x)$

using *assms* **by** (*simp only: eqvt-at-def*)

finally have $p \cdot (f\ (q \cdot x)) = f\ (p \cdot q \cdot x)$ **by** *simp* }

then show *eqvt-at f (q · x)* **unfolding** *eqvt-at-def*

by *simp*

qed

lemma *supp-fun-eqvt*:

assumes *a: eqvt f*

shows $supp\ f = \{\}$

using *a*

unfolding *eqvt-def*

unfolding *supp-def*

by *simp*

lemma *fresh-fun-eqvt*:

assumes *a: eqvt f*

shows $a \# f$

using *a*

unfolding *fresh-def*

by (*simp add: supp-fun-eqvt*)

lemma *fresh-fun-eqvt-app*:

```

assumes a: eqvt f
shows  $a \# x \implies a \# f x$ 
proof –
  from a have  $\text{supp } f = \{\}$  by (simp add: supp-fun-eqvt)
  then show  $a \# x \implies a \# f x$ 
    unfolding fresh-def
    using supp-fun-app by auto
qed

```

```

lemma supp-fun-app-eqvt:
  assumes a: eqvt f
  shows  $\text{supp } (f x) \subseteq \text{supp } x$ 
  using fresh-fun-eqvt-app[OF a]
  unfolding fresh-def
  by auto

```

```

lemma supp-eqvt-at:
  assumes asm: eqvt-at f x
  and fin: finite (supp x)
  shows  $\text{supp } (f x) \subseteq \text{supp } x$ 
apply(rule supp-is-subset)
unfolding supports-def
unfolding fresh-def[symmetric]
using asm
apply(simp add: eqvt-at-def)
apply(simp add: swap-fresh-fresh)
apply(rule fin)
done

```

```

lemma finite-supp-eqvt-at:
  assumes asm: eqvt-at f x
  and fin: finite (supp x)
  shows finite (supp (f x))
apply(rule finite-subset)
apply(rule supp-eqvt-at[OF asm fin])
apply(rule fin)
done

```

```

lemma fresh-eqvt-at:
  assumes asm: eqvt-at f x
  and fin: finite (supp x)
  and fresh:  $a \# x$ 
  shows  $a \# f x$ 
using fresh
unfolding fresh-def
using supp-eqvt-at[OF asm fin]
by auto

```

for handling of freshness of functions


```

simproc-setup fresh-fun-simproc (a ‡ (f::'a::pt =>'b::pt)) = ‹fn - => fn ctxt =>
fn ctrm =>
  let
    val - $ - $ f = Thm.term-of ctrm
  in
    case (Term.add-frees f [], Term.add-vars f []) of
      ([], []) => SOME(@{thm fresh-fun-eqvt[simplified eqvt-def, THEN Eq-TrueI]})
    | (x::-, []) =>
      let
        val argx = Free x
        val absf = absfree x f
        val cty-inst =
          [SOME (Thm.ctyp-of ctxt (fastype-of argx)), SOME (Thm.ctyp-of ctxt
(fastype-of f))]
        val ctrm-inst = [NONE, SOME (Thm.cterm-of ctxt absf), SOME (Thm.cterm-of
ctxt argx)]
        val thm = Thm.instantiate' cty-inst ctrm-inst @ {thm fresh-fun-app}
      in
        SOME(thm RS @ {thm Eq-TrueI})
      end
    | (-, -) => NONE
  end
›

```

13.2 helper functions for *nominal-functions*

lemma *THE-defaultI2*:

```

assumes  $\exists!x. P x \wedge x. P x \implies Q x$ 
shows  $Q (THE-default d P)$ 

```

by (*iprover intro: assms THE-defaultI'*)

lemma *the-default-*eqvt**:

```

assumes unique:  $\exists!x. P x$ 
shows  $(p \cdot (THE-default d P)) = (THE-default (p \cdot d) (p \cdot P))$ 
apply(rule THE-default1-equality [symmetric])
apply(rule-tac  $p=-p$  in permute-boolE)
apply(simp add: ex1-eqvt)
apply(rule unique)
apply(rule-tac  $p=-p$  in permute-boolE)
apply(rule subst[OF permute-fun-app-eq])
apply(simp)
apply(rule THE-defaultI'[OF unique])
done

```

lemma *fundef-ex1-*eqvt**:

```

fixes  $x::'a::pt$ 
assumes f-def:  $f == (\lambda x::'a. THE-default (d x) (G x))$ 
assumes eqvt: eqvt G
assumes ex1:  $\exists!y. G x y$ 

```

```

shows  $(p \cdot (f x)) = f (p \cdot x)$ 
apply(simp only: f-def)
apply(subst the-default-eqvt)
apply(rule ex1)
apply(rule THE-default1-equality [symmetric])
apply(rule-tac p=-p in permute-boolE)
apply(perm-simp add: permute-minus-cancel)
using eqvt[simplified eqvt-def]
apply(simp)
apply(rule ex1)
apply(rule THE-defaultI2)
apply(rule-tac p=-p in permute-boolE)
apply(perm-simp add: permute-minus-cancel)
apply(rule ex1)
apply(perm-simp)
using eqvt[simplified eqvt-def]
apply(simp)
done

```

```

lemma fundef-ex1-eqvt-at:
  fixes  $x::'a::pt$ 
  assumes f-def:  $f == (\lambda x::'a. THE-default (d x) (G x))$ 
  assumes eqvt: eqvt G
  assumes ex1:  $\exists!y. G x y$ 
  shows eqvt-at f x
  unfolding eqvt-at-def
  using assms
  by (auto intro: fundef-ex1-eqvt)

```

```

lemma fundef-ex1-prop:
  fixes  $x::'a::pt$ 
  assumes f-def:  $f == (\lambda x::'a. THE-default (d x) (G x))$ 
  assumes P-all:  $\bigwedge x y. G x y \implies P x y$ 
  assumes ex1:  $\exists!y. G x y$ 
  shows  $P x (f x)$ 
  unfolding f-def
  using ex1
  apply(erule-tac ex1E)
  apply(rule THE-defaultI2)
  apply(blast)
  apply(rule P-all)
  apply(assumption)
  done

```

14 Support of Finite Sets of Finitely Supported Elements

support and freshness for atom sets

```

lemma supp-finite-atom-set:
  fixes  $S::atom\ set$ 
  assumes finite S
  shows  $supp\ S = S$ 
  apply(rule finite-supp-unique)
  apply(simp add: supports-def)
  apply(simp add: swap-set-not-in)
  apply(rule assms)
  apply(simp add: swap-set-in)
done

lemma supp-cofinite-atom-set:
  fixes  $S::atom\ set$ 
  assumes finite (UNIV - S)
  shows  $supp\ S = (UNIV - S)$ 
  apply(rule finite-supp-unique)
  apply(simp add: supports-def)
  apply(simp add: swap-set-both-in)
  apply(rule assms)
  apply(subst swap-commute)
  apply(simp add: swap-set-in)
done

lemma fresh-finite-atom-set:
  fixes  $S::atom\ set$ 
  assumes finite S
  shows  $a \# S \longleftrightarrow a \notin S$ 
  unfolding fresh-def
  by (simp add: supp-finite-atom-set[OF assms])

lemma fresh-minus-atom-set:
  fixes  $S::atom\ set$ 
  assumes finite S
  shows  $a \# S - T \longleftrightarrow (a \notin T \longrightarrow a \# S)$ 
  unfolding fresh-def
  by (auto simp: supp-finite-atom-set assms)

lemma Union-supports-set:
  shows  $(\bigcup x \in S. supp\ x)\ supports\ S$ 
proof -
  { fix  $a\ b$ 
    have  $\forall x \in S. (a \rightleftharpoons b) \cdot x = x \implies (a \rightleftharpoons b) \cdot S = S$ 
      unfolding permute-set-def by force
    }
  then show  $(\bigcup x \in S. supp\ x)\ supports\ S$ 
    unfolding supports-def
    by (simp add: fresh-def[symmetric] swap-fresh-fresh)
qed

```

```

lemma Union-of-finite-supp-sets:
  fixes  $S::('a::fs\ set)$ 
  assumes  $fin: finite\ S$ 
  shows  $finite\ (\bigcup_{x \in S} supp\ x)$ 
  using  $fin$  by (induct) (auto simp: finite-supp)

lemma Union-included-in-supp:
  fixes  $S::('a::fs\ set)$ 
  assumes  $fin: finite\ S$ 
  shows  $(\bigcup_{x \in S} supp\ x) \subseteq supp\ S$ 
proof –
  have  $eqvt: eqvt\ (\lambda S. \bigcup_{x \in S} supp\ x)$ 
    unfolding eqvt-def by simp
  have  $(\bigcup_{x \in S} supp\ x) = supp\ (\bigcup_{x \in S} supp\ x)$ 
    by (rule supp-finite-atom-set[symmetric]) (rule Union-of-finite-supp-sets[OF
fin])
  also have  $\dots \subseteq supp\ S$  using eqvt
    by (rule supp-fun-app-eqvt)
  finally show  $(\bigcup_{x \in S} supp\ x) \subseteq supp\ S$  .
qed

lemma supp-of-finite-sets:
  fixes  $S::('a::fs\ set)$ 
  assumes  $fin: finite\ S$ 
  shows  $(supp\ S) = (\bigcup_{x \in S} supp\ x)$ 
apply(rule subset-antisym)
apply(rule supp-is-subset)
apply(rule Union-supports-set)
apply(rule Union-of-finite-supp-sets[OF fin])
apply(rule Union-included-in-supp[OF fin])
done

lemma finite-sets-supp:
  fixes  $S::('a::fs\ set)$ 
  assumes  $finite\ S$ 
  shows  $finite\ (supp\ S)$ 
using assms
by (simp only: supp-of-finite-sets Union-of-finite-supp-sets)

lemma supp-of-finite-union:
  fixes  $S\ T::('a::fs)\ set$ 
  assumes  $fin1: finite\ S$ 
  and  $fin2: finite\ T$ 
  shows  $supp\ (S \cup T) = supp\ S \cup supp\ T$ 
  using  $fin1\ fin2$ 
  by (simp add: supp-of-finite-sets)

lemma fresh-finite-union:
  fixes  $S\ T::('a::fs)\ set$ 

```

```

assumes fin1: finite S
and fin2: finite T
shows  $a \# (S \cup T) \longleftrightarrow a \# S \wedge a \# T$ 
unfolding fresh-def
by (simp add: supp-of-finite-union[OF fin1 fin2])

```

```

lemma supp-of-finite-insert:
fixes S::('a::fs) set
assumes fin: finite S
shows  $\text{supp } (\text{insert } x \ S) = \text{supp } x \cup \text{supp } S$ 
using fin
by (simp add: supp-of-finite-sets)

```

```

lemma fresh-finite-insert:
fixes S::('a::fs) set
assumes fin: finite S
shows  $a \# (\text{insert } x \ S) \longleftrightarrow a \# x \wedge a \# S$ 
using fin unfolding fresh-def
by (simp add: supp-of-finite-insert)

```

```

lemma supp-set-empty:
shows  $\text{supp } \{\} = \{\}$ 
unfolding supp-def
by (simp add: empty-eqvt)

```

```

lemma fresh-set-empty:
shows  $a \# \{\}$ 
by (simp add: fresh-def supp-set-empty)

```

```

lemma supp-set:
fixes xs :: ('a::fs) list
shows  $\text{supp } (\text{set } xs) = \text{supp } xs$ 
apply(induct xs)
apply(simp add: supp-set-empty supp-Nil)
apply(simp add: supp-Cons supp-of-finite-insert)
done

```

```

lemma fresh-set:
fixes xs :: ('a::fs) list
shows  $a \# (\text{set } xs) \longleftrightarrow a \# xs$ 
unfolding fresh-def
by (simp add: supp-set)

```

14.1 Type 'a multiset is finitely supported

```

lemma set-mset-eqvt [eqvt]:
shows  $p \cdot (\text{set-mset } M) = \text{set-mset } (p \cdot M)$ 
by (induct M) (simp-all add: insert-eqvt empty-eqvt)

```

```

lemma supp-set-mset:
  shows  $\text{supp } (\text{set-mset } M) \subseteq \text{supp } M$ 
  apply (rule supp-fun-app-eqvt)
  unfolding eqvt-def
  apply(perm-simp)
  apply(simp)
  done

lemma Union-finite-multiset:
  fixes  $M::'a::fs \text{ multiset}$ 
  shows  $\text{finite } (\bigcup \{\text{supp } x \mid x. x \in\# M\})$ 
  proof -
    have  $\text{finite } (\bigcup (\text{supp } ` \{x. x \in\# M\}))$ 
      by (induct M) (simp-all add: Collect-imp-eq Collect-neg-eq finite-supp)
    then show  $\text{finite } (\bigcup \{\text{supp } x \mid x. x \in\# M\})$ 
      by (simp only: image-Collect)
  qed

lemma Union-supports-multiset:
  shows  $\bigcup \{\text{supp } x \mid x. x \in\# M\} \text{ supports } M$ 
  proof -
    have  $sw: \bigwedge a b. ((\bigwedge x. x \in\# M \implies (a \rightleftharpoons b) \cdot x = x) \implies (a \rightleftharpoons b) \cdot M = M)$ 
      unfolding permute-multiset-def by (induct M) simp-all
    have  $(\bigcup_{x \in \text{set-mset } M}. \text{supp } x) \text{ supports } M$ 
      by (auto intro!: sw swap-fresh-fresh simp add: fresh-def supports-def)
    also have  $(\bigcup_{x \in \text{set-mset } M}. \text{supp } x) = (\bigcup \{\text{supp } x \mid x. x \in\# M\})$ 
      by auto
    finally show  $(\bigcup \{\text{supp } x \mid x. x \in\# M\}) \text{ supports } M$  .
  qed

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 -
    have  $(\bigcup \{\text{supp } x \mid x. x \in\# M\}) = (\bigcup_{x \in \text{set-mset } M}. \text{supp } x)$  by auto
    also have  $\dots = \text{supp } (\text{set-mset } M)$ 
      by (simp add: supp-of-finite-sets)
    also have  $\dots \subseteq \text{supp } M$  by (rule supp-set-mset)
    finally show  $(\bigcup \{\text{supp } x \mid x. x \in\# M\}) \subseteq \text{supp } M$  .
  qed

lemma supp-of-multisets:
  fixes  $M::('a::fs \text{ multiset})$ 
  shows  $(\text{supp } M) = (\bigcup \{\text{supp } x \mid x. x \in\# M\})$ 
  apply(rule subset-antisym)
  apply(rule supp-is-subset)
  apply(rule Union-supports-multiset)
  apply(rule Union-finite-multiset)
  apply(rule Union-included-multiset)

```

done

lemma *multisets-supp-finite*:
 fixes $M::('a::fs\ multiset)$
 shows *finite (supp M)*
by (*simp only: supp-of-multisets Union-finite-multiset*)

lemma *supp-of-multiset-union*:
 fixes $M\ N::('a::fs)\ multiset$
 shows $supp\ (M + N) = supp\ M \cup supp\ N$
 by (*auto simp: supp-of-multisets*)

lemma *supp-empty-mset* [*simp*]:
 shows $supp\ \{\#\} = \{\}$
 unfolding *supp-def*
 by *simp*

instance *multiset* :: (*fs*) *fs*
 by *standard (rule multisets-supp-finite)*

14.2 Type 'a fset is finitely supported

lemma *supp-fset* [*simp*]:
 shows $supp\ (fset\ S) = supp\ S$
 unfolding *supp-def*
 by (*simp add: fset-eqvt fset-cong*)

lemma *supp-empty-fset* [*simp*]:
 shows $supp\ \{\|\} = \{\}$
 unfolding *supp-def*
 by *simp*

lemma *fresh-empty-fset*:
 shows $a \# \{\|\}$
unfolding *fresh-def*
by (*simp*)

lemma *supp-finsert* [*simp*]:
 fixes $x::'a::fs$
 and $S::'a\ fset$
 shows $supp\ (finsert\ x\ S) = supp\ x \cup supp\ S$
 apply(*subst supp-fset[symmetric]*)
 apply(*simp add: supp-of-finite-insert*)
 done

lemma *fresh-finsert*:
 fixes $x::'a::fs$
 and $S::'a\ fset$
 shows $a \# finsert\ x\ S \longleftrightarrow a \# x \wedge a \# S$

unfolding *fresh-def*
by *simp*

lemma *fset-finite-supp*:
fixes $S :: ('a::fs) \text{ fset}$
shows *finite* (*supp* S)
by (*induct* S) (*simp-all add: finite-supp*)

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

lemma *fresh-union-fset*:
fixes $S T :: 'a::fs \text{ fset}$
shows $a \# S \mid\cup\mid T \longleftrightarrow a \# S \wedge a \# T$
unfolding *fresh-def*
by (*simp add: supp-union-fset*)

instance *fset* :: (*fs*) *fs*
by *standard* (*rule fset-finite-supp*)

14.3 Type $('a, 'b)$ *finfun* is finitely supported

lemma *fresh-finfun-const*:
shows $a \# (\text{finfun-const } b) \longleftrightarrow a \# b$
by (*simp add: fresh-def supp-def*)

lemma *fresh-finfun-update*:
shows $\llbracket a \# f; a \# x; a \# y \rrbracket \implies a \# \text{finfun-update } f \ x \ y$
unfolding *fresh-conv-MOST*
unfolding *finfun-update-eqt*
by (*elim MOST-rev-mp*) (*simp*)

lemma *supp-finfun-const*:
shows $\text{supp } (\text{finfun-const } b) = \text{supp}(b)$
by (*simp add: supp-def*)

lemma *supp-finfun-update*:
shows $\text{supp } (\text{finfun-update } f \ x \ y) \subseteq \text{supp}(f, x, y)$
using *fresh-finfun-update*
by (*auto simp: fresh-def supp-Pair*)

instance *finfun* :: (*fs*, *fs*) *fs*
apply *standard*
apply (*induct-tac x rule: finfun-weak-induct*)
apply (*simp add: supp-finfun-const finite-supp*)
apply (*rule finite-subset*)
apply (*rule supp-finfun-update*)


```

apply(simp add: supp-Pair finite-supp)
done

```

15 Freshness and Fresh-Star

```

lemma fresh-Unit-elim:
  shows (a # ()  $\implies$  PROP C)  $\equiv$  PROP C
  by (simp add: fresh-Unit)

```

```

lemma fresh-Pair-elim:
  shows (a # (x, y)  $\implies$  PROP C)  $\equiv$  (a # x  $\implies$  a # y  $\implies$  PROP C)
  by rule (simp-all add: fresh-Pair)

```

```

lemma [simp]:
  shows a # x1  $\implies$  a # x2  $\implies$  a # (x1, x2)
  by (simp add: fresh-Pair)

```

```

lemma fresh-PairD:
  shows a # (x, y)  $\implies$  a # x
  and a # (x, y)  $\implies$  a # y
  by (simp-all add: fresh-Pair)

```

```

declaration ⟨fn - =>
  let
    val mksimps-pairs = (@{const-name Nominal2-Base.fresh}, @{thms fresh-PairD})
  :: mksimps-pairs
  in
    Simplifier.map-ss (fn ss => Simplifier.set-mksimps (mksimps mksimps-pairs) ss)
  end
  ⟩

```

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

```

definition
  fresh-star :: atom set  $\implies$  'a::pt  $\implies$  bool (- #* - [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
  by (auto simp: fresh-star-def fresh-def)

```

```

lemma fresh-star-perm-set-conv:
  fixes p::perm
  assumes fresh: as #* p
  and fin: finite as
  shows supp p #* as

```

apply(*rule fresh-star-supp-conv*)
apply(*simp add: supp-finite-atom-set fin fresh*)
done

lemma *fresh-star-atom-set-conv*:
assumes *fresh: as #* bs*
and *fin: finite as finite bs*
shows *bs #* as*
using *fresh*
unfolding *fresh-star-def fresh-def*
by (*auto simp: supp-finite-atom-set fin*)

lemma *atom-fresh-star-disjoint*:
assumes *fin: finite bs*
shows *as #* bs \longleftrightarrow (as \cap bs = {})*

unfolding *fresh-star-def fresh-def*
by (*auto simp: supp-finite-atom-set fin*)

lemma *fresh-star-Pair*:
shows *as #* (x, y) = (as #* x \wedge as #* y)*
by (*auto simp: fresh-star-def fresh-Pair*)

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 #* []*
by (*auto simp: fresh-star-def fresh-Nil fresh-Cons fresh-append*)

lemma *fresh-star-set*:
fixes *xs::('a::fs) list*
shows *as #* set xs \longleftrightarrow as #* xs*
unfolding *fresh-star-def*
by (*simp add: fresh-set*)

lemma *fresh-star-singleton*:
fixes *a::atom*
shows *as #* {a} \longleftrightarrow as #* a*
by (*simp add: fresh-star-def fresh-finite-insert fresh-set-empty*)

lemma *fresh-star-fset*:
fixes *xs::('a::fs) list*
shows *as #* fset S \longleftrightarrow as #* S*
by (*simp add: fresh-star-def fresh-def*)

lemma *fresh-star-Un*:
shows *(as \cup bs) #* x = (as #* x \wedge bs #* x)*
by (*auto simp: fresh-star-def*)

lemma *fresh-star-insert*:
shows $(\text{insert } a \text{ as}) \#* x = (a \# x \wedge \text{as} \#* x)$
by (*auto simp: fresh-star-def*)

lemma *fresh-star-Un-elim*:
 $((\text{as} \cup \text{bs}) \#* x \Longrightarrow \text{PROP } C) \equiv (\text{as} \#* x \Longrightarrow \text{bs} \#* x \Longrightarrow \text{PROP } C)$
unfolding *fresh-star-def*
apply(*rule*)
apply(*erule meta-mp*)
apply(*auto*)
done

lemma *fresh-star-insert-elim*:
 $(\text{insert } a \text{ as} \#* x \Longrightarrow \text{PROP } C) \equiv (a \# x \Longrightarrow \text{as} \#* x \Longrightarrow \text{PROP } C)$
unfolding *fresh-star-def*
by rule (*simp-all add: fresh-star-def*)

lemma *fresh-star-empty-elim*:
 $(\{\} \#* x \Longrightarrow \text{PROP } C) \equiv \text{PROP } C$
by (*simp add: fresh-star-def*)

lemma *fresh-star-Unit-elim*:
shows $(a \#* ()) \Longrightarrow \text{PROP } C) \equiv \text{PROP } C$
by (*simp add: fresh-star-def fresh-Unit*)

lemma *fresh-star-Pair-elim*:
shows $(a \#* (x, y) \Longrightarrow \text{PROP } C) \equiv (a \#* x \Longrightarrow a \#* y \Longrightarrow \text{PROP } C)$
by (*rule, simp-all add: fresh-star-Pair*)

lemma *fresh-star-zero*:
shows $\text{as} \#* (0::\text{perm})$
unfolding *fresh-star-def*
by (*simp add: fresh-zero-perm*)

lemma *fresh-star-plus*:
fixes $p \ q::\text{perm}$
shows $\llbracket a \#* p; a \#* q \rrbracket \Longrightarrow a \#* (p + q)$
unfolding *fresh-star-def*
by (*simp add: fresh-plus-perm*)

lemma *fresh-star-permute-iff*:
shows $(p \cdot a) \#* (p \cdot x) \longleftrightarrow a \#* x$
unfolding *fresh-star-def*
by (*metis mem-permute-iff permute-minus-cancel(1) fresh-permute-iff*)

lemma *fresh-star-eqvt* [*eqvt*]:
shows $p \cdot (\text{as} \#* x) \longleftrightarrow (p \cdot \text{as}) \#* (p \cdot x)$
unfolding *fresh-star-def* **by** *simp*

16 Induction principle for permutations

lemma *smaller-supp*:

assumes $a: a \in \text{supp } p$

shows $\text{supp } ((p \cdot a \rightleftharpoons a) + p) \subseteq \text{supp } p$

proof –

have $\text{supp } ((p \cdot a \rightleftharpoons a) + p) \subseteq \text{supp } p$

unfolding *supp-perm* **by** (*auto simp: swap-atom*)

moreover

have $a \notin \text{supp } ((p \cdot a \rightleftharpoons a) + p)$ **by** (*simp add: supp-perm*)

then have $\text{supp } ((p \cdot a \rightleftharpoons a) + p) \neq \text{supp } p$ **using** a **by** *auto*

ultimately

show $\text{supp } ((p \cdot a \rightleftharpoons a) + p) \subseteq \text{supp } p$ **by** *auto*

qed

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

assumes $S: \text{supp } p \subseteq S$

and $\text{zero}: P \ 0$

and $\text{swap}: \bigwedge p \ a \ b. \llbracket P \ p; \text{supp } p \subseteq S; a \in S; b \in S; a \neq b; \text{sort-of } a = \text{sort-of } b \rrbracket$
 $\implies P \ ((a \rightleftharpoons b) + p)$

shows $P \ p$

proof –

have *finite* ($\text{supp } p$) **by** (*simp add: finite-supp*)

then show $P \ p$ **using** S

proof(*induct* $A \equiv \text{supp } p$ *arbitrary: p rule: finite-psubset-induct*)

case (*psubset* p)

then have *ih*: $\bigwedge q. \text{supp } q \subseteq \text{supp } p \implies P \ q$ **by** *auto*

have *as*: $\text{supp } p \subseteq S$ **by** *fact*

{ **assume** $\text{supp } p = \{\}$

then have $p = 0$ **by** (*simp add: supp-perm perm-eq-iff*)

then have $P \ p$ **using** *zero* **by** *simp*

}

moreover

{ **assume** $\text{supp } p \neq \{\}$

then obtain a **where** $a0: a \in \text{supp } p$ **by** *blast*

then have $a1: p \cdot a \in S \ a \in S \ \text{sort-of } (p \cdot a) = \text{sort-of } a \ p \cdot a \neq a$

using *as* **by** (*auto simp: supp-atom supp-perm swap-atom*)

let $?q = (p \cdot a \rightleftharpoons a) + p$

have $a2: \text{supp } ?q \subseteq \text{supp } p$ **using** $a0$ *smaller-supp* **by** *simp*

then have $P \ ?q$ **using** *ih* **by** *simp*

moreover

have $\text{supp } ?q \subseteq S$ **using** *as* $a2$ **by** *simp*

ultimately have $P \ ((p \cdot a \rightleftharpoons a) + ?q)$ **using** *as* $a1$ *swap* **by** *simp*

moreover

have $p = (p \cdot a \rightleftharpoons a) + ?q$ **by** (*simp add: perm-eq-iff*)

ultimately have $P \ p$ **by** *simp*

}

ultimately show $P \ p$ **by** *blast*

qed
qed

lemma *perm-simple-struct-induct*[*case-names zero swap*]:
assumes *zero*: $P\ 0$
and *swap*: $\bigwedge p\ a\ b. \llbracket P\ p; a \neq b; \text{sort-of } a = \text{sort-of } b \rrbracket \implies P\ ((a \rightleftharpoons b) + p)$
shows $P\ p$
by (*rule-tac S=supp p in perm-struct-induct*)
(*auto intro: zero swap*)

lemma *perm-struct-induct2*[*consumes 1, case-names zero swap plus*]:
assumes *S*: $\text{supp } p \subseteq S$
assumes *zero*: $P\ 0$
assumes *swap*: $\bigwedge a\ b. \llbracket \text{sort-of } a = \text{sort-of } b; a \neq b; a \in S; b \in S \rrbracket \implies P\ (a \rightleftharpoons b)$
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 $P\ p$
using *S*
by (*induct p rule: perm-struct-induct*)
(*auto intro: zero plus swap simp add: supp-swap*)

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 $P\ p$
by (*rule-tac S=supp p in perm-struct-induct2*)
(*auto intro: zero swap plus*)

lemma *supp-perm-singleton*:
fixes *p*::*perm*
shows $\text{supp } p \subseteq \{b\} \longleftrightarrow p = 0$
proof –
{ **assume** $\text{supp } p \subseteq \{b\}$
then have $p = 0$
by (*induct p rule: perm-struct-induct*) (*simp-all*)
}
then show $\text{supp } p \subseteq \{b\} \longleftrightarrow p = 0$ **by** (*auto simp: supp-zero-perm*)
qed

lemma *supp-perm-pair*:
fixes *p*::*perm*
shows $\text{supp } p \subseteq \{a, b\} \longleftrightarrow p = 0 \vee p = (b \rightleftharpoons a)$
proof –
{ **assume** $\text{supp } p \subseteq \{a, b\}$
then have $p = 0 \vee p = (b \rightleftharpoons a)$
apply (*induct p rule: perm-struct-induct*)
apply (*auto simp: swap-cancel supp-zero-perm supp-swap*)
}

```

    apply (simp add: swap-commute)
  done
}
then show supp p ⊆ {a, b} ↔ p = 0 ∨ p = (b ⇌ a)
  by (auto simp: supp-zero-perm supp-swap split: if-splits)
qed

```

```

lemma supp-perm-eq:
  assumes (supp x) #* p
  shows p · x = x
proof -
  from assms have supp p ⊆ {a. a # x}
    unfolding supp-perm fresh-star-def fresh-def by auto
  then show p · x = x
  proof (induct p rule: perm-struct-induct)
    case zero
    show 0 · x = x by simp
  next
    case (swap p a b)
    then have a # x b # x p · x = x by simp-all
    then show ((a ⇌ b) + p) · x = x by (simp add: swap-fresh-fresh)
  qed
qed

```

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

```

lemma supp-perm-eq-test:
  assumes (supp x) #* p
  shows p · x = x
proof -
  from assms have supp p ⊆ {a. a # x}
    unfolding supp-perm fresh-star-def fresh-def by auto
  then show p · x = x
  proof (induct p rule: perm-struct-induct2)
    case zero
    show 0 · x = x by simp
  next
    case (swap a b)
    then have a # x b # x by simp-all
    then show (a ⇌ b) · x = x by (simp add: swap-fresh-fresh)
  next
    case (plus p1 p2)
    have p1 · x = x p2 · x = x by fact+
    then show (p1 + p2) · x = x by simp
  qed
qed

```

```

lemma perm-supp-eq:
  assumes a: (supp p) #* x
  shows p · x = x

```

```

proof –
  from assms have  $\text{supp } p \subseteq \{a. a \# x\}$ 
    unfolding supp-perm fresh-star-def fresh-def by auto
  then show  $p \cdot x = x$ 
  proof (induct p rule: perm-struct-induct2)
    case zero
      show  $0 \cdot x = x$  by simp
    next
      case (swap a b)
        then have  $a \# x \ b \# x$  by simp-all
        then show  $(a \rightleftharpoons b) \cdot x = x$  by (simp add: swap-fresh-fresh)
      next
        case (plus p1 p2)
          have  $p1 \cdot x = x \ p2 \cdot x = x$  by fact+
          then show  $(p1 + p2) \cdot x = x$  by simp
        qed
      qed
  qed

```

```

lemma supp-perm-perm-eq:
  assumes  $a: \forall a \in \text{supp } x. p \cdot a = q \cdot a$ 
  shows  $p \cdot x = q \cdot x$ 
proof –
  from  $a$  have  $\forall a \in \text{supp } x. (-q + p) \cdot a = a$  by simp
  then have  $\forall a \in \text{supp } x. a \notin \text{supp } (-q + p)$ 
    unfolding supp-perm by simp
  then have  $\text{supp } x \#^* (-q + p)$ 
    unfolding fresh-star-def fresh-def by simp
  then have  $(-q + p) \cdot x = x$  by (simp only: supp-perm-eq)
  then show  $p \cdot x = q \cdot x$ 
    by (metis permute-minus-cancel permute-plus)
qed

```

disagreement set

```

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

```

```

lemma ds-fresh:
  assumes  $dset \ p \ q \ \#^* \ x$ 
  shows  $p \cdot x = q \cdot x$ 
using assms
unfolding dset-def fresh-star-def fresh-def
by (auto intro: supp-perm-perm-eq)

```

```

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

```

```

from  $a$  have  $\text{supp } p \subseteq \{a. a \notin as\}$ 
  unfolding  $\text{supp-perm fresh-star-def fresh-def}$  by  $\text{auto}$ 
then show  $p \cdot as = as$ 
proof ( $\text{induct } p \text{ rule: perm-struct-induct}$ )
  case  $\text{zero}$ 
    show  $0 \cdot as = as$  by  $\text{simp}$ 
  next
    case ( $\text{swap } p \ a \ b$ )
      then have  $a \notin as \ b \notin as \ p \cdot as = as$  by  $\text{simp-all}$ 
      then show  $((a \rightleftharpoons b) + p) \cdot as = as$  by ( $\text{simp add: swap-set-not-in}$ )
  qed
qed

```

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 $\text{at-set-avoiding-aux}$:

```

fixes  $Xs::\text{atom set}$ 
and  $As::\text{atom set}$ 
assumes  $b: Xs \subseteq As$ 
and  $c: \text{finite } As$ 
shows  $\exists p. (p \cdot Xs) \cap As = \{\} \wedge (\text{supp } p) = (Xs \cup (p \cdot Xs))$ 
proof –
  from  $b \ c$  have  $\text{finite } Xs$  by ( $\text{rule finite-subset}$ )
  then show  $?thesis$  using  $b$ 
  proof ( $\text{induct rule: finite-subset-induct}$ )
    case  $\text{empty}$ 
      have  $0 \cdot \{\} \cap As = \{\}$  by  $\text{simp}$ 
      moreover
        have  $\text{supp } (0::\text{perm}) = \{\} \cup 0 \cdot \{\}$  by ( $\text{simp add: supp-zero-perm}$ )
        ultimately show  $?case$  by  $\text{blast}$ 
    next
      case ( $\text{insert } x \ Xs$ )
        then obtain  $p$  where
           $p1: (p \cdot Xs) \cap As = \{\}$  and
           $p2: \text{supp } p = (Xs \cup (p \cdot Xs))$  by  $\text{blast}$ 
        from  $\langle x \in As \rangle \ p1$  have  $x \notin p \cdot Xs$  by  $\text{fast}$ 
        with  $\langle x \notin Xs \rangle \ p2$  have  $x \notin \text{supp } p$  by  $\text{fast}$ 
        hence  $p \cdot x = x$  unfolding  $\text{supp-perm}$  by  $\text{simp}$ 
        have  $\text{finite } (As \cup p \cdot Xs \cup \text{supp } p)$ 
          using  $\langle \text{finite } As \rangle \ \langle \text{finite } Xs \rangle$ 
          by ( $\text{simp add: permute-set-eq-image finite-supp}$ )
        then obtain  $y$  where  $y \notin (As \cup p \cdot Xs \cup \text{supp } p)$   $\text{sort-of } y = \text{sort-of } x$ 
          by ( $\text{rule obtain-atom}$ )
        hence  $y: y \notin As \ y \notin p \cdot Xs \ y \notin \text{supp } p$   $\text{sort-of } y = \text{sort-of } x$ 
          by  $\text{simp-all}$ 

```



```

hence  $py: p \cdot y = y \ x \neq y$  using  $\langle x \in As \rangle$ 
  by (auto simp: supp-perm)
let  $?q = (x \rightleftharpoons y) + p$ 
have  $q: ?q \cdot insert\ x\ Xs = insert\ y\ (p \cdot Xs)$ 
  unfolding insert-eqt
  using  $\langle p \cdot x = x \rangle \langle sort\ of\ y = sort\ of\ x \rangle$ 
  using  $\langle x \notin p \cdot Xs \rangle \langle y \notin p \cdot Xs \rangle$ 
  by (simp add: swap-atom swap-set-not-in)
have  $?q \cdot insert\ x\ Xs \cap As = \{\}$ 
  using  $\langle y \notin As \rangle \langle p \cdot Xs \cap As = \{\} \rangle$ 
  unfolding  $q$  by simp
moreover
have  $supp\ (x \rightleftharpoons y) \cap supp\ p = \{\}$  using  $px\ py \ \langle sort\ of\ y = sort\ of\ x \rangle$ 
  unfolding supp-swap by (simp add: supp-perm)
then have  $supp\ ?q = (supp\ (x \rightleftharpoons y) \cup supp\ p)$ 
  by (simp add: supp-plus-perm-eq)
then have  $supp\ ?q = insert\ x\ Xs \cup ?q \cdot insert\ x\ Xs$ 
  using  $p2 \ \langle sort\ of\ y = sort\ of\ x \rangle \langle x \neq y \rangle$  unfolding  $q$  supp-swap
  by auto
ultimately show  $?case$  by blast
qed
qed

```

```

lemma at-set-avoiding:
  assumes  $a: finite\ Xs$ 
  and  $b: finite\ (supp\ c)$ 
  obtains  $p::perm$  where  $(p \cdot Xs) \#* c$  and  $(supp\ p) = (Xs \cup (p \cdot Xs))$ 
  using  $a\ b$  at-set-avoiding-aux [where  $Xs=Xs$  and  $As=Xs \cup supp\ c$ ]
  unfolding fresh-star-def fresh-def by blast

```

```

lemma at-set-avoiding1:
  assumes  $finite\ xs$ 
  and  $finite\ (supp\ c)$ 
  shows  $\exists p. (p \cdot xs) \#* c$ 
using assms
apply(erule-tac c=c in at-set-avoiding)
apply(auto)
done

```

```

lemma at-set-avoiding2:
  assumes  $finite\ xs$ 
  and  $finite\ (supp\ c)$   $finite\ (supp\ x)$ 
  and  $xs \#* x$ 
  shows  $\exists p. (p \cdot xs) \#* c \wedge supp\ x \#* p$ 
using assms
apply(erule-tac c=(c, x) in at-set-avoiding)
apply(simp add: supp-Pair)
apply(rule-tac x=p in exI)
apply(simp add: fresh-star-Pair)

```

```

apply(rule fresh-star-supp-conv)
apply(auto simp: fresh-star-def)
done

```

```

lemma at-set-avoiding3:
  assumes finite xs
  and     finite (supp c) finite (supp x)
  and     xs  $\#^*$  x
  shows  $\exists p. (p \cdot xs) \#^* c \wedge \text{supp } x \#^* p \wedge \text{supp } p = xs \cup (p \cdot xs)$ 
using assms
apply(erule-tac c=(c, x) in at-set-avoiding)
apply(simp add: supp-Pair)
apply(rule-tac x=p in exI)
apply(simp add: fresh-star-Pair)
apply(rule fresh-star-supp-conv)
apply(auto simp: fresh-star-def)
done

```

```

lemma at-set-avoiding2-atom:
  assumes finite (supp c) finite (supp x)
  and     b: a  $\#$  x
  shows  $\exists p. (p \cdot a) \# c \wedge \text{supp } x \#^* p$ 
proof –
  have a: {a}  $\#^*$  x unfolding fresh-star-def by (simp add: b)
  obtain p where p1: (p · {a})  $\#^*$  c and p2: supp x  $\#^*$  p
    using at-set-avoiding2[of {a} c x] assms a by blast
  have c: (p · a)  $\#$  c using p1
    unfolding fresh-star-def Ball-def
    by(erule-tac x=p · a in allE) (simp add: permute-set-def)
  hence p · a  $\#$  c  $\wedge$  supp x  $\#^*$  p using p2 by blast
  then show  $\exists p. (p \cdot a) \# c \wedge \text{supp } x \#^* p$  by blast
qed

```

18 Renaming permutations

```

lemma set-renaming-perm:
  assumes b: finite bs
  shows  $\exists q. (\forall b \in bs. q \cdot b = p \cdot b) \wedge \text{supp } q \subseteq bs \cup (p \cdot bs)$ 
using b
proof (induct)
  case empty
  have  $(\forall b \in \{\}. 0 \cdot b = p \cdot b) \wedge \text{supp } (0::\text{perm}) \subseteq \{\} \cup p \cdot \{\}$ 
    by (simp add: permute-set-def supp-perm)
  then show  $\exists q. (\forall b \in \{\}. q \cdot b = p \cdot b) \wedge \text{supp } q \subseteq \{\} \cup p \cdot \{\}$  by blast
next
  case (insert a bs)
  then have  $\exists q. (\forall b \in bs. q \cdot b = p \cdot b) \wedge \text{supp } q \subseteq bs \cup p \cdot bs$  by simp
  then obtain q where *:  $\forall b \in bs. q \cdot b = p \cdot b$  and **:  $\text{supp } q \subseteq bs \cup p \cdot bs$ 
    by (metis empty-subsetI insert(3) supp-swap)

```

```

{ assume 1:  $q \cdot a = p \cdot a$ 
  have  $\forall b \in (\text{insert } a \text{ } bs). q \cdot b = p \cdot b$  using 1 * by simp
  moreover
  have  $\text{supp } q \subseteq \text{insert } a \text{ } bs \cup p \cdot \text{insert } a \text{ } bs$ 
    using ** by (auto simp: insert-eqt)
  ultimately
  have  $\exists q. (\forall b \in \text{insert } a \text{ } bs. q \cdot b = p \cdot b) \wedge \text{supp } q \subseteq \text{insert } a \text{ } bs \cup p \cdot \text{insert}$ 
 $a \text{ } bs$  by blast
}
moreover
{ assume 2:  $q \cdot a \neq p \cdot a$ 
  define  $q'$  where  $q' = ((q \cdot a) \rightleftharpoons (p \cdot a)) + q$ 
  have  $\forall b \in \text{insert } a \text{ } bs. q' \cdot b = p \cdot b$  using 2 *  $\langle a \notin bs \rangle$  unfolding  $q'$ -def
    by (auto simp: swap-atom)
  moreover
  { have  $\{q \cdot a, p \cdot a\} \subseteq \text{insert } a \text{ } bs \cup p \cdot \text{insert } a \text{ } bs$ 
    using **
    apply (auto simp: supp-perm insert-eqt)
    apply (subgoal-tac  $q \cdot a \in bs \cup p \cdot bs$ )
    apply (auto)[1]
    apply (subgoal-tac  $q \cdot a \in \{a. q \cdot a \neq a\}$ )
    apply (blast)
    apply (simp)
    done
  then have  $\text{supp } (q \cdot a \rightleftharpoons p \cdot a) \subseteq \text{insert } a \text{ } bs \cup p \cdot \text{insert } a \text{ } bs$ 
    unfolding supp-swap by auto
  moreover
  have  $\text{supp } q \subseteq \text{insert } a \text{ } bs \cup p \cdot \text{insert } a \text{ } bs$ 
    using ** by (auto simp: insert-eqt)
  ultimately
  have  $\text{supp } q' \subseteq \text{insert } a \text{ } bs \cup p \cdot \text{insert } a \text{ } bs$ 
    unfolding  $q'$ -def using supp-plus-perm by blast
  }
  ultimately
  have  $\exists q. (\forall b \in \text{insert } a \text{ } bs. q \cdot b = p \cdot b) \wedge \text{supp } q \subseteq \text{insert } a \text{ } bs \cup p \cdot \text{insert}$ 
 $a \text{ } bs$  by blast
}
ultimately show  $\exists q. (\forall b \in \text{insert } a \text{ } bs. q \cdot b = p \cdot b) \wedge \text{supp } q \subseteq \text{insert } a \text{ } bs$ 
 $\cup p \cdot \text{insert } a \text{ } bs$ 
  by blast
qed

```

lemma *set-renaming-perm2*:

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

proof –

have *finite* ($bs \cap \text{supp } p$) by (simp add: finite-supp)

then obtain q

where *: $\forall b \in bs \cap \text{supp } p. q \cdot b = p \cdot b$ and **: $\text{supp } q \subseteq (bs \cap \text{supp } p) \cup (p \cdot (bs \cap \text{supp } p))$

```

    using set-renaming-perm by blast
  from ** have supp q ⊆ bs ∪ (p · bs) by (auto simp: inter-eqvt)
  moreover
  have ∀ b ∈ bs - supp p. q · b = p · b
    apply(auto)
    apply(subgoal-tac b ∉ supp q)
    apply(simp add: fresh-def[symmetric])
    apply(simp add: fresh-perm)
    apply(clarify)
    apply(rotate-tac 2)
    apply(drule subsetD[OF **])
    apply(simp add: inter-eqvt supp-eqvt permute-self)
  done
  ultimately have (∀ b ∈ bs. q · b = p · b) ∧ supp q ⊆ bs ∪ (p · bs) using * by
  auto
  then show ∃ q. (∀ b ∈ bs. q · b = p · b) ∧ supp q ⊆ bs ∪ (p · bs) by blast
qed

lemma list-renaming-perm:
  shows ∃ q. (∀ b ∈ set bs. q · b = p · b) ∧ supp q ⊆ set bs ∪ (p · set bs)
  proof (induct bs)
    case (Cons a bs)
    then have ∃ q. (∀ b ∈ set bs. q · b = p · b) ∧ supp q ⊆ set bs ∪ p · (set bs) by
  simp
    then obtain q where *: ∀ b ∈ set bs. q · b = p · b and **: supp q ⊆ set bs ∪ p
    · (set bs)
    by (blast)
    { assume 1: a ∈ set bs
      have q · a = p · a using * 1 by (induct bs) (auto)
      then have ∀ b ∈ set (a # bs). q · b = p · b using * by simp
      moreover
      have supp q ⊆ set (a # bs) ∪ p · (set (a # bs)) using ** by (auto simp:
  insert-eqvt)
      ultimately
      have ∃ q. (∀ b ∈ set (a # bs). q · b = p · b) ∧ supp q ⊆ set (a # bs) ∪ p · (set
  (a # bs)) by blast
    }
    moreover
    { assume 2: a ∉ set bs
      define q' where q' = ((q · a) ⇐ (p · a)) + q
      have ∀ b ∈ set (a # bs). q' · b = p · b
      unfolding q'-def using 2 * ⟨a ∉ set bs⟩ by (auto simp: swap-atom)
      moreover
      { have {q · a, p · a} ⊆ set (a # bs) ∪ p · (set (a # bs))
        using **
        apply (auto simp: supp-perm insert-eqvt)
        apply (subgoal-tac q · a ∈ set bs ∪ p · set bs)
        apply(auto)[1]
        apply(subgoal-tac q · a ∈ {a. q · a ≠ a})
      }
    }
  }

```

```

    apply(blast)
    apply(simp)
    done
  then have  $\text{supp } (q \cdot a \Rightarrow p \cdot a) \subseteq \text{set } (a \# bs) \cup p \cdot \text{set } (a \# bs)$ 
    unfolding supp-swap by auto
  moreover
  have  $\text{supp } q \subseteq \text{set } (a \# bs) \cup p \cdot (\text{set } (a \# bs))$ 
    using ** by (auto simp: insert-eqvt)
  ultimately
  have  $\text{supp } q' \subseteq \text{set } (a \# bs) \cup p \cdot (\text{set } (a \# bs))$ 
    unfolding q'-def using supp-plus-perm by blast
}
ultimately
have  $\exists q. (\forall b \in \text{set } (a \# bs). q \cdot b = p \cdot b) \wedge \text{supp } q \subseteq \text{set } (a \# bs) \cup p \cdot$ 
 $(\text{set } (a \# bs))$  by blast
}
ultimately show  $\exists q. (\forall b \in \text{set } (a \# bs). q \cdot b = p \cdot b) \wedge \text{supp } q \subseteq \text{set } (a \#$ 
 $bs) \cup p \cdot (\text{set } (a \# bs))$ 
  by blast
next
case Nil
  have  $(\forall b \in \text{set } []. 0 \cdot b = p \cdot b) \wedge \text{supp } (0::\text{perm}) \subseteq \text{set } [] \cup p \cdot \text{set } []$ 
    by (simp add: supp-zero-perm)
  then show  $\exists q. (\forall b \in \text{set } []. q \cdot b = p \cdot b) \wedge \text{supp } q \subseteq \text{set } [] \cup p \cdot (\text{set } [])$  by
blast
qed

```

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  $\cdot$  (atom a) = 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
using assms by metis

```

```

lemma supp-at-base:
  fixes a::'a::at-base

```

shows $\text{supp } a = \{\text{atom } a\}$
by (*simp* *add*: *supp-atom* [*symmetric*] *supp-def* *atom-eqvt*)

lemma *fresh-at-base*:
shows $\text{sort-of } a \neq \text{sort-of } (\text{atom } b) \implies a \# b$
and $a \# b \longleftrightarrow a \neq \text{atom } b$
unfolding *fresh-def*
apply(*simp-all* *add*: *supp-at-base*)
apply(*metis*)
done

lemma *fresh-ineq-at-base* [*simp*]:
shows $a \neq \text{atom } b \implies a \# b$
by (*simp* *add*: *fresh-at-base*)

lemma *fresh-atom-at-base* [*simp*]:
fixes $b::'a::\text{at-base}$
shows $a \# \text{atom } b \longleftrightarrow a \# b$
by (*simp* *add*: *fresh-def* *supp-at-base* *supp-atom*)

lemma *fresh-star-atom-at-base*:
fixes $b::'a::\text{at-base}$
shows $as \#* \text{atom } b \longleftrightarrow as \#* b$
by (*simp* *add*: *fresh-star-def* *fresh-atom-at-base*)

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 $\text{atom } a \# x \implies P (\text{if } x = a \text{ then } t \text{ else } s) = P s$
by (*simp-all* *add*: *fresh-at-base*)

simproc-setup *fresh-ineq* ($x \neq (y::'a::\text{at-base})$) = $\langle \text{fn } - \implies \text{fn } \text{ctxt} \implies \text{fn } \text{ctrm}$
 \implies
case *Thm.term-of* *ctrm* of **Const**- $\langle \text{Not for } \mathbf{Const}$ - $\langle \text{HOL.eq} - \text{for lhs rhs} \rangle \rangle \implies$
let
fun *first-is-neg* *lhs* *rhs* [] = *NONE*
| *first-is-neg* *lhs* *rhs* (*thm::thms*) =
(*case* *Thm.prop-of* *thm* of
- \$ **Const**- $\langle \text{Not for } \mathbf{Const}$ - $\langle \text{HOL.eq} - \text{for } l r \rangle \rangle \implies$
(*if* $l = \text{lhs}$ *andalso* $r = \text{rhs}$ *then* *SOME*(*thm*)
else *if* $r = \text{lhs}$ *andalso* $l = \text{rhs}$ *then* *SOME*(*thm* *RS* @{*thm not-sym*})
else *first-is-neg* *lhs* *rhs* *thms*)
| - \implies *first-is-neg* *lhs* *rhs* *thms*)

val *simp-thms* = @{*thms* *fresh-Pair* *fresh-at-base* *atom-eq-iff*}
val *prems* = *Simplifier.prems-of* *ctxt*
|> *filter* (*fn* *thm* \implies *case* *Thm.prop-of* *thm* of

```

- $ Const- ⟨fresh - for ⟨- $ a⟩ b⟩ =>
  (let
    val atms = a :: HOLogic.strip-tuple b
    in
      member (op =) atms lhs andalso member (op =) atms rhs
    end)
| - => false)
|> map (simplify (put-simpset HOL-basic-ss ctxt addsimps simp-thms))
|> map (HOLogic.conj-elimns ctxt)
|> flat
in
  case first-is-neg lhs rhs prems of
    SOME(thm) => SOME(thm RS @ {thm Eq-TrueI})
  | NONE => NONE
end
| - => NONE
>

```

instance *at-base* < *fs*
proof **qed** (*simp add: supp-at-base*)

lemma *at-base-infinite* [*simp*]:
 shows *infinite* (*UNIV* :: 'a::*at-base* set) (**is infinite** ?*U*)
proof
 obtain *a* :: 'a **where** *True* **by** *auto*
 assume *finite* ?*U*
 hence *finite* (*atom* ' ?*U*)
 by (*rule finite-imageI*)
 then obtain *b* **where** *b* ∉ *atom* ' ?*U* *sort-of* *b* = *sort-of* (*atom* *a*)
 by (*rule obtain-atom*)
 from *b*(2) **have** *b* = *atom* ((*atom* *a* ⇔ *b*) • *a*)
 unfolding *atom-eqvt* [*symmetric*]
 by (*simp add: swap-atom*)
 hence *b* ∈ *atom* ' ?*U* **by** *simp*
 with *b*(1) **show** *False* **by** *simp*
qed

lemma *swap-at-base-simps* [*simp*]:
 fixes *x y*::'a::*at-base*
 shows *sort-of* (*atom* *x*) = *sort-of* (*atom* *y*) ⇒ (*atom* *x* ⇔ *atom* *y*) • *x* = *y*
 and *sort-of* (*atom* *x*) = *sort-of* (*atom* *y*) ⇒ (*atom* *x* ⇔ *atom* *y*) • *y* = *x*
 and *atom* *x* ≠ *a* ⇒ *atom* *x* ≠ *b* ⇒ (*a* ⇔ *b*) • *x* = *x*
 unfolding *atom-eq-iff* [*symmetric*]
 unfolding *atom-eqvt* [*symmetric*]
 by *simp-all*

lemma *obtain-at-base*:
 assumes *X*: *finite* *X*

```

obtains  $a::'a::at-base$  where  $atom\ a \notin X$ 
proof –
  have  $inj\ (atom\ ::\ 'a \Rightarrow atom)$ 
    by  $(simp\ add:\ inj-on-def)$ 
  with  $X$  have  $finite\ (atom\ -\ 'X\ ::\ 'a\ set)$ 
    by  $(rule\ finite-vimageI)$ 
  with  $at-base-infinite$  have  $atom\ -\ 'X \neq (UNIV\ ::\ 'a\ set)$ 
    by  $auto$ 
  then obtain  $a\ ::\ 'a$  where  $atom\ a \notin X$ 
    by  $auto$ 
  thus  $?thesis\ ..$ 
qed

```

```

lemma  $obtain-fresh'$ :
  assumes  $fin:\ finite\ (supp\ x)$ 
  obtains  $a::'a::at-base$  where  $atom\ a \# x$ 
using  $obtain-at-base[where\ X=supp\ x]$ 
by  $(auto\ simp:\ fresh-def\ fin)$ 

```

```

lemma  $obtain-fresh$ :
  fixes  $x::'b::fs$ 
  obtains  $a::'a::at-base$  where  $atom\ a \# x$ 
  by  $(rule\ obtain-fresh')\ (auto\ simp:\ finite-supp)$ 

```

```

lemma  $supp-finite-set-at-base$ :
  assumes  $a:\ finite\ S$ 
  shows  $supp\ S = atom\ 'S$ 
apply $(simp\ add:\ supp-of-finite-sets[OF\ a])$ 
apply $(simp\ add:\ supp-at-base)$ 
apply $(auto)$ 
done

```

```

lemma  $fresh-finite-set-at-base$ :
  fixes  $a::'a::at-base$ 
  assumes  $a:\ finite\ S$ 
  shows  $atom\ a \# S \longleftrightarrow a \notin S$ 
  unfolding  $fresh-def$ 
apply $(simp\ add:\ supp-finite-set-at-base[OF\ a])$ 
apply $(subst\ inj-image-mem-iff)$ 
apply $(simp\ add:\ inj-on-def)$ 
apply $(simp)$ 
done

```

```

lemma  $fresh-at-base-permute-iff\ [simp]$ :
  fixes  $a::'a::at-base$ 
  shows  $atom\ (p \cdot a) \# p \cdot x \longleftrightarrow atom\ a \# x$ 
  unfolding  $atom-eqvt[symmetric]$ 

```


by (*simp only: fresh-permute-iff*)

lemma *fresh-at-base-permI*:

shows $\text{atom } a \# p \implies p \cdot a = a$

by (*simp add: fresh-def supp-perm*)

20 Infrastructure for concrete atom types

definition

flip :: $'a::\text{at-base} \Rightarrow 'a \Rightarrow \text{perm } ('(- \leftrightarrow -))$

where

$(a \leftrightarrow b) = (\text{atom } a \rightleftharpoons \text{atom } b)$

lemma *flip-fresh-fresh*:

assumes $\text{atom } a \# x \text{ atom } b \# x$

shows $(a \leftrightarrow b) \cdot x = x$

using *assms*

by (*simp add: flip-def swap-fresh-fresh*)

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

unfolding *flip-def* by (*rule swap-self*)

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

unfolding *flip-def* by (*rule swap-commute*)

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

unfolding *flip-def* by (*rule minus-swap*)

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

unfolding *flip-def* by (*rule swap-cancel*)

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

unfolding *permute-plus* [*symmetric*] *add-flip-cancel* by *simp*

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

by (*simp add: flip-commute*)

lemma *flip-eqvt* [*eqvt*]:

shows $p \cdot (a \leftrightarrow b) = (p \cdot a \leftrightarrow p \cdot b)$

unfolding *flip-def*

by (*simp add: swap-eqvt atom-eqvt*)

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 $\text{sort-of } (\text{atom } a) \neq \text{sort-of } (\text{atom } b) \implies (a \leftrightarrow b) \cdot x = x$

unfolding *flip-def*

unfolding *atom-eq-iff* [*symmetric*]

unfolding *atom-eqvt* [*symmetric*]
by *simp-all*

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::at*
assumes *a ≠ b* **and** *c ≠ b*
shows $(a \leftrightarrow c) + (b \leftrightarrow c) + (a \leftrightarrow c) = (a \leftrightarrow b)$
unfolding *flip-def*
by (*rule swap-triple*) (*simp-all add: assms*)

lemma *permute-flip-at*:
fixes *a b c::'a::at*
shows $(a \leftrightarrow b) \cdot c = (\text{if } c = a \text{ then } b \text{ else if } c = b \text{ then } a \text{ else } c)$
unfolding *flip-def*
apply (*rule atom-eq-iff* [*THEN iffD1*])
apply (*subst atom-eqvt* [*symmetric*])
apply (*simp add: swap-atom*)
done

lemma *flip-at-simps* [*simp*]:
fixes *a b::'a::at*
shows $(a \leftrightarrow b) \cdot a = b$
and $(a \leftrightarrow b) \cdot b = a$
unfolding *permute-flip-at* **by** *simp-all*

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

syntax

-atom-constrain :: *logic* \Rightarrow *type* \Rightarrow *logic* (*-::-* [*4*, *0*] *?*)

translations

-atom-constrain a t \Rightarrow *CONST atom* (*-constrain a t*)

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

setup \langle *Sign.add-const-constraint* ($\@$ {*const-name permute*}, *NONE*) \rangle

setup \langle *Sign.add-const-constraint* ($\@$ {*const-name atom*}, *NONE*) \rangle

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

lemma *exists-eq-simple-sort*:
shows $\exists a. a \in \{a. \text{sort-of } a = s\}$
by (*rule-tac x=Atom s 0 in exI, simp*)

lemma *exists-eq-sort*:
shows $\exists a. a \in \{a. \text{sort-of } a \in \text{range sort-fun}\}$
by (*rule-tac x=Atom (sort-fun x) y in exI, simp*)

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*: $\bigwedge a$. *atom* *a* = *Rep* *a*

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

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

proof

interpret *type-definition* *Rep* *Abs* {*a*. *sort-of* *a* \in *range sort-fun*} **by** (*rule type*)

have *sort-of-Rep*: $\bigwedge a$. *sort-of* (*Rep* *a*) \in *range sort-fun* **using** *Rep* **by** *simp*

fix *a b* :: 'a **and** *p p1 p2* :: *perm*

show $0 \cdot a$ = *a*

unfolding *permute-def* **by** (*simp add: Rep-inverse*)

show $(p1 + p2) \cdot a$ = $p1 \cdot p2 \cdot a$

unfolding *permute-def* **by** (*simp add: Abs-inverse sort-of-Rep*)

show *atom* *a* = *atom* *b* \longleftrightarrow *a* = *b*

unfolding *atom-def* **by** (*simp add: Rep-inject*)

show $p \cdot \text{atom } a$ = *atom* ($p \cdot a$)

unfolding *permute-def atom-def* **by** (*simp add: Abs-inverse sort-of-Rep*)

qed

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*: $\bigwedge a$. *atom* *a* = *Rep* *a*

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

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

proof

interpret *type-definition* *Rep* *Abs* {*a*. *sort-of* *a* = *s*} **by** (*rule type*)

have *sort-of-Rep*: $\bigwedge a$. *sort-of* (*Rep* *a*) = *s* **using** *Rep* **by** (*simp add: image-def*)

fix *a b* :: 'a **and** *p p1 p2* :: *perm*

show $0 \cdot a$ = *a*

unfolding *permute-def* **by** (*simp add: Rep-inverse*)

show $(p1 + p2) \cdot a$ = $p1 \cdot p2 \cdot a$

unfolding *permute-def* **by** (*simp add: Abs-inverse sort-of-Rep*)

show *sort-of* (*atom* *a*) = *sort-of* (*atom* *b*)

unfolding *atom-def* **by** (*simp add: sort-of-Rep*)

show *atom* *a* = *atom* *b* \longleftrightarrow *a* = *b*

unfolding *atom-def* **by** (*simp add: Rep-inject*)

show $p \cdot \text{atom } a$ = *atom* ($p \cdot a$)

unfolding *permute-def atom-def* **by** (*simp add: Abs-inverse sort-of-Rep*)

qed

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 $atom-def: \bigwedge a.\ atom\ a = Rep\ a$
shows $sort-of\ (atom\ a) = s$
using $atom-def\ type$
unfolding $type-definition-def$ **by** $simp$

setup $\langle Sign.add-const-constraint$
 $(@{\const-name\ permute}, SOME\ @{\typ\ perm \Rightarrow 'a::pt \Rightarrow 'a})\rangle$
setup $\langle Sign.add-const-constraint$
 $(@{\const-name\ atom}, SOME\ @{\typ\ 'a::at-base \Rightarrow atom})\rangle$

21 Library functions for the nominal infrastructure

ML-file $\langle nominal-library.ML\rangle$

22 The freshness lemma according to Andy Pitts

lemma $freshness-lemma:$

fixes $h :: 'a::at \Rightarrow 'b::pt$
assumes $a: \exists a.\ atom\ a \# (h, h\ a)$
shows $\exists x.\ \forall a.\ atom\ a \# h \longrightarrow h\ a = x$

proof –

from a **obtain** b **where** $a1: atom\ b \# h$ **and** $a2: atom\ b \# h\ b$

by $(auto\ simp: fresh-Pair)$

show $\exists x.\ \forall a.\ atom\ a \# h \longrightarrow h\ a = x$

proof $(intro\ exI\ allI\ impI)$

fix $a :: 'a$

assume $a3: atom\ a \# h$

show $h\ a = h\ b$

proof $(cases\ a = b)$

assume $a = b$

thus $h\ a = h\ b$ **by** $simp$

next

assume $a \neq b$

hence $atom\ a \# b$ **by** $(simp\ add: fresh-at-base)$

with $a3$ **have** $atom\ a \# h\ b$

by $(rule\ fresh-fun-app)$

with $a2$ **have** $d1: (atom\ b \rightleftharpoons atom\ a) \cdot (h\ b) = (h\ b)$

by $(rule\ swap-fresh-fresh)$

from $a1\ a3$ **have** $d2: (atom\ b \rightleftharpoons atom\ a) \cdot h = h$

by $(rule\ swap-fresh-fresh)$

from $d1$ **have** $h\ b = (atom\ b \rightleftharpoons atom\ a) \cdot (h\ b)$ **by** $simp$

also **have** $\dots = ((atom\ b \rightleftharpoons atom\ a) \cdot h) ((atom\ b \rightleftharpoons atom\ a) \cdot b)$

by $(rule\ permute-fun-app-eq)$

also **have** $\dots = h\ a$

```

    using d2 by simp
    finally show h a = h b by simp
  qed
qed
qed

```

```

lemma freshness-lemma-unique:
  fixes h :: 'a::at ⇒ 'b::pt
  assumes a: ∃ a. atom a # (h, h a)
  shows ∃!x. ∀ a. atom a # h ⟶ h a = x
proof (rule ex-ex1I)
  from a show ∃ x. ∀ a. atom a # h ⟶ h a = x
    by (rule freshness-lemma)
next
  fix x y
  assume x: ∀ a. atom a # h ⟶ h a = x
  assume y: ∀ a. atom a # h ⟶ h a = y
  from a x y show x = y
    by (auto simp: fresh-Pair)
qed

```

packaging the freshness lemma into a function

```

definition
  Fresh :: ('a::at ⇒ 'b::pt) ⇒ 'b
where
  Fresh h = (THE x. ∀ a. atom a # h ⟶ h a = x)

```

```

lemma Fresh-apply:
  fixes h :: 'a::at ⇒ 'b::pt
  assumes a: ∃ a. atom a # (h, h a)
  assumes b: atom a # h
  shows Fresh h = h a
unfolding Fresh-def
proof (rule the-equality)
  show ∀ a'. atom a' # h ⟶ h a' = h a
  proof (intro strip)
    fix a':: 'a
    assume c: atom a' # h
    from a have ∃ x. ∀ a. atom a # h ⟶ h a = x by (rule freshness-lemma)
    with b c show h a' = h a by auto
  qed
next
  fix fr :: 'b
  assume ∃ a. atom a # h ⟶ h a = fr
  with b show fr = h a by auto
qed

```

```

lemma Fresh-apply':
  fixes h :: 'a::at ⇒ 'b::pt

```

```

assumes a: atom a # h atom a # h a
shows Fresh h = h a
apply (rule Fresh-apply)
apply (auto simp: fresh-Pair intro: a)
done

simproc-setup Fresh-simproc (Fresh (h::'a::at ⇒ 'b::pt)) = ⟨fn - => fn ctxt =>
fn ctrm =>
  let
    val - $ h = Thm.term-of ctrm

    val atoms = Simplifier.premis-of ctxt
    |> map-filter (fn thm => case Thm.prop-of thm of
      - $ Const-⟨fresh - for Const-⟨atom - for atm⟩ -⟩ => SOME atm | - =>
NONE)
    |> distinct (op =)

    fun get-thm atm =
      let
        val goal1 = HOLogic.mk-Trueprop (mk-fresh (mk-atom atm) h)
        val goal2 = HOLogic.mk-Trueprop (mk-fresh (mk-atom atm) (h $ atm))

        val thm1 = Goal.prove ctxt [] [] goal1 (fn {context = ctxt', ...} =>
asm-simp-tac ctxt' 1)
        val thm2 = Goal.prove ctxt [] [] goal2 (fn {context = ctxt', ...} =>
asm-simp-tac ctxt' 1)
      in
        SOME (@{thm Fresh-apply'} OF [thm1, thm2] RS eq-reflection)
      end handle ERROR - => NONE
    in
      get-first get-thm atoms
    end
  ⟩

```

lemma *Fresh-eqvt*:

```

fixes h :: 'a::at ⇒ 'b::pt
assumes a: ∃ a. atom a # (h, h a)
shows p · (Fresh h) = Fresh (p · h)
proof -
  from a obtain a::'a::at where fr: atom a # h atom a # h a
  by (metis fresh-Pair)
  then have fr-p: atom (p · a) # (p · h) atom (p · a) # (p · h) (p · a)
  by (metis atom-eqvt fresh-permute-iff eqvt-apply)+
  have p · (Fresh h) = p · (h a) using fr by simp
  also have ... = (p · h) (p · a) by simp
  also have ... = Fresh (p · h) using fr-p by simp
  finally show p · (Fresh h) = Fresh (p · h) .
qed

```

lemma *Fresh-supports*:
fixes $h :: 'a::at \Rightarrow 'b::pt$
assumes $a: \exists a. \text{atom } a \# (h, h a)$
shows $(\text{supp } h) \text{ supports } (\text{Fresh } h)$
apply $(\text{simp add: supports-def fresh-def [symmetric]})$
apply $(\text{simp add: Fresh-eqvt [OF a] swap-fresh-fresh})$
done

notation *Fresh* (**binder** *FRESH* 10)

lemma *FRESH-f-iff*:
fixes $P :: 'a::at \Rightarrow 'b::pure$
fixes $f :: 'b \Rightarrow 'c::pure$
assumes $P: \text{finite } (\text{supp } P)$
shows $(\text{FRESH } x. f (P x)) = f (\text{FRESH } x. P x)$
proof –
obtain $a::'a$ **where** $\text{atom } a \# P$ **using** P **by** $(\text{rule obtain-fresh'})$
then show $(\text{FRESH } x. f (P x)) = f (\text{FRESH } x. P x)$
by $(\text{simp add: pure-fresh})$
qed

lemma *FRESH-binop-iff*:
fixes $P :: 'a::at \Rightarrow 'b::pure$
fixes $Q :: 'a::at \Rightarrow 'c::pure$
fixes $\text{binop} :: 'b \Rightarrow 'c \Rightarrow 'd::pure$
assumes $P: \text{finite } (\text{supp } P)$
and $Q: \text{finite } (\text{supp } Q)$
shows $(\text{FRESH } x. \text{binop } (P x) (Q x)) = \text{binop } (\text{FRESH } x. P x) (\text{FRESH } x. Q x)$
proof –
from assms **have** $\text{finite } (\text{supp } (P, Q))$ **by** $(\text{simp add: supp-Pair})$
then obtain $a::'a$ **where** $\text{atom } a \# (P, Q)$ **by** $(\text{rule obtain-fresh'})$
then show $?thesis$
by $(\text{simp add: pure-fresh})$
qed

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

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

23 Automation for creating concrete atom types

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

ML-file \langle *nominal-atoms.ML* \rangle

24 Automatic equivariance procedure for inductive definitions

ML-file \langle *nominal-eqvt.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 (- \approx_{set} - - - - [100, 100, 100, 100, 100] 100) **and**
alpha-res (- \approx_{res} - - - - [100, 100, 100, 100, 100] 100) **and**
alpha-lst (- \approx_{lst} - - - - [100, 100, 100, 100, 100] 100)

26 Mono

lemma [*mono*]:

shows $R1 \leq R2 \implies \text{alpha-set } bs \ R1 \leq \text{alpha-set } bs \ R2$
and $R1 \leq R2 \implies \text{alpha-res } bs \ R1 \leq \text{alpha-res } bs \ R2$
and $R1 \leq R2 \implies \text{alpha-lst } cs \ R1 \leq \text{alpha-lst } cs \ R2$
by (*case-tac* [!] *bs*, *case-tac* [!] *cs*)
(*auto simp: le-fun-def le-bool-def alphas*)

27 Equivariance

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

shows $(bs, x) \approx_{set} R \ f \ q \ (cs, y) \implies (p \cdot bs, p \cdot x) \approx_{set} (p \cdot R) \ (p \cdot f) \ (p \cdot q)$
 $(p \cdot cs, p \cdot y)$
and $(bs, x) \approx_{res} R \ f \ q \ (cs, y) \implies (p \cdot bs, p \cdot x) \approx_{res} (p \cdot R) \ (p \cdot f) \ (p \cdot q)$
 $(p \cdot cs, p \cdot y)$
and $(ds, x) \approx_{lst} R \ f \ q \ (es, y) \implies (p \cdot ds, p \cdot x) \approx_{lst} (p \cdot R) \ (p \cdot f) \ (p \cdot q) \ (p$
 $\cdot es, p \cdot y)$
unfolding *alphas*
unfolding *permute-eqvt*[*symmetric*]
unfolding *set-eqvt*[*symmetric*]
unfolding *permute-fun-app-eq*[*symmetric*]
unfolding *Diff-eqvt*[*symmetric*]
unfolding *eq-eqvt*[*symmetric*]
unfolding *fresh-star-eqvt*[*symmetric*]
by (*auto simp only: permute-bool-def*)

28 Equivalence

lemma *alpha-reft*:

assumes *a*: $R \ x \ x$
shows $(bs, x) \approx_{set} R \ f \ 0 \ (bs, x)$
and $(bs, x) \approx_{res} R \ f \ 0 \ (bs, x)$
and $(cs, x) \approx_{lst} R \ f \ 0 \ (cs, x)$
using *a*
unfolding *alphas*
unfolding *fresh-star-def*
by (*simp-all add: fresh-zero-perm*)

lemma *alpha-sym*:

assumes $a: R (p \cdot x) y \implies R (-p \cdot y) x$
shows $(bs, x) \approx_{set} R f p (cs, y) \implies (cs, y) \approx_{set} R f (-p) (bs, x)$
and $(bs, x) \approx_{res} R f p (cs, y) \implies (cs, y) \approx_{res} R f (-p) (bs, x)$
and $(ds, x) \approx_{lst} R f p (es, y) \implies (es, y) \approx_{lst} R f (-p) (ds, x)$
unfolding $alphas\ fresh-star-def$
using a
by $(auto\ simp: fresh-minus-perm)$

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 $\llbracket (bs, x) \approx_{set} R f p (cs, y); (cs, y) \approx_{set} R f q (ds, z) \rrbracket \implies (bs, x) \approx_{set} R f (q + p) (ds, z)$
and $\llbracket (bs, x) \approx_{res} R f p (cs, y); (cs, y) \approx_{res} R f q (ds, z) \rrbracket \implies (bs, x) \approx_{res} R f (q + p) (ds, z)$
and $\llbracket (es, x) \approx_{lst} R f p (gs, y); (gs, y) \approx_{lst} R f q (hs, z) \rrbracket \implies (es, x) \approx_{lst} R f (q + p) (hs, z)$
using a
unfolding $alphas\ fresh-star-def$
by $(simp-all\ add: fresh-plus-perm)$

lemma $alpha-sym-eqvt$:

assumes $a: R (p \cdot x) y \implies R y (p \cdot x)$
and $b: p \cdot R = R$
shows $(bs, x) \approx_{set} R f p (cs, y) \implies (cs, y) \approx_{set} R f (-p) (bs, x)$
and $(bs, x) \approx_{res} R f p (cs, y) \implies (cs, y) \approx_{res} R f (-p) (bs, x)$
and $(ds, x) \approx_{lst} R f p (es, y) \implies (es, y) \approx_{lst} R f (-p) (ds, x)$
apply $(auto\ intro!: alpha-sym)$
apply $(drule-tac [!]) a$
apply $(rule-tac [!] p=p\ in\ permute-boolE)$
apply $(simp-all\ add: b\ permute-self)$
done

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 $(bs, x) \approx_{set} R f (q + p) (ds, z)$
apply $(rule\ alpha-trans(1)[OF - a\ b])$
apply $(drule\ c)$
apply $(rule-tac\ p=q\ in\ permute-boolE)$
apply $(simp\ add: d\ permute-self)$
apply $(rotate-tac -1)$
apply $(drule-tac\ p=q\ in\ permute-boolI)$
apply $(simp\ add: d\ permute-self\ permute-eqvt[symmetric])$
done

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 · 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)
apply(rule alpha-trans(2)[OF - a b])
apply(drule c)
apply(rule-tac p=q in permute-boolE)
apply(simp add: d permute-self)
apply(rotate-tac -1)
apply(drule-tac p=q in permute-boolI)
apply(simp add: d permute-self permute-eqvt[symmetric])
done

```

```

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 · 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)
apply(rule alpha-trans(3)[OF - a b])
apply(drule c)
apply(rule-tac p=q in permute-boolE)
apply(simp add: d permute-self)
apply(rotate-tac -1)
apply(drule-tac p=q in permute-boolI)
apply(simp add: d permute-self permute-eqvt[symmetric])
done

```

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)  $\longleftrightarrow (\exists p. (bs, x) \approx_{set} ((=)) \text{supp } p (cs, y))$ 

```

```

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

```

```

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

```

notation

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

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

lemma *alphas-abs-refl*:

shows $(bs, x) \approx_{abs-set} (bs, x)$
and $(bs, x) \approx_{abs-res} (bs, x)$
and $(cs, x) \approx_{abs- lst } (cs, x)$
unfolding *alphas-abs*
unfolding *alphas*
unfolding *fresh-star-def*
by (*rule-tac* [!] $x=0$ **in** exI)
(simp-all add: fresh-zero-perm)

lemma *alphas-abs-sym*:

shows $(bs, x) \approx_{abs-set} (cs, y) \implies (cs, y) \approx_{abs-set} (bs, x)$
and $(bs, x) \approx_{abs-res} (cs, y) \implies (cs, y) \approx_{abs-res} (bs, x)$
and $(ds, x) \approx_{abs- lst } (es, y) \implies (es, y) \approx_{abs- lst } (ds, x)$
unfolding *alphas-abs*
unfolding *alphas*
unfolding *fresh-star-def*
by (*erule-tac* [!] exE , *rule-tac* [!] $x=-p$ **in** exI)
(auto simp: fresh-minus-perm)

lemma *alphas-abs-trans*:

shows $\llbracket (bs, x) \approx_{abs-set} (cs, y); (cs, y) \approx_{abs-set} (ds, z) \rrbracket \implies (bs, x) \approx_{abs-set} (ds, z)$
and $\llbracket (bs, x) \approx_{abs-res} (cs, y); (cs, y) \approx_{abs-res} (ds, z) \rrbracket \implies (bs, x) \approx_{abs-res} (ds, z)$
and $\llbracket (es, x) \approx_{abs- lst } (gs, y); (gs, y) \approx_{abs- lst } (hs, z) \rrbracket \implies (es, x) \approx_{abs- lst } (hs, z)$
unfolding *alphas-abs*
unfolding *alphas*
unfolding *fresh-star-def*
apply(*erule-tac* [!] exE , *erule-tac* [!] exE)
apply(*rule-tac* [!] $x=pa + p$ **in** exI)
by (*simp-all add: fresh-plus-perm*)

lemma *alphas-abs-eqvt*:

shows $(bs, x) \approx_{abs-set} (cs, y) \implies (p \cdot bs, p \cdot x) \approx_{abs-set} (p \cdot cs, p \cdot y)$
and $(bs, x) \approx_{abs-res} (cs, y) \implies (p \cdot bs, p \cdot x) \approx_{abs-res} (p \cdot cs, p \cdot y)$
and $(ds, x) \approx_{abs- lst } (es, y) \implies (p \cdot ds, p \cdot x) \approx_{abs- lst } (p \cdot es, p \cdot y)$
unfolding *alphas-abs*
unfolding *alphas*

unfolding *set-eqvt*[*symmetric*]
unfolding *supp-eqvt*[*symmetric*]
unfolding *Diff-eqvt*[*symmetric*]
apply(*erule-tac* [!] *exE*)
apply(*rule-tac* [!] *x=p · pa in exI*)
by (*auto simp only: fresh-star-permute-iff permute-eqvt*[*symmetric*])

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 $B1 = B2$

using *a b*

by (*metis Int-Un-distrib2 Int-absorb2 Int-commute Un-upper2*)

lemma *supp-property-res*:

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

proof –

from *a* **have** $(supp\ x - as) \#* p$ **by** (*auto simp only: alphas*)

then have $*$: $p \cdot (supp\ x - as) = (supp\ x - as)$

by (*simp add: atom-set-perm-eq*)

have $(supp\ x' - as') \cup (supp\ x' \cap as') = supp\ x'$ **by** *auto*

also have $\dots = supp\ (p \cdot x)$ **using** *a* **by** (*simp add: alphas*)

also have $\dots = p \cdot (supp\ x)$ **by** (*simp add: supp-eqvt*)

also have $\dots = p \cdot ((supp\ x - as) \cup (supp\ x \cap as))$ **by** *auto*

also have $\dots = (p \cdot (supp\ x - as)) \cup (p \cdot (supp\ x \cap as))$ **by** (*simp add: union-eqvt*)

also have $\dots = (supp\ x - as) \cup (p \cdot (supp\ x \cap as))$ **using** $*$ **by** *simp*

also have $\dots = (supp\ x' - as') \cup (p \cdot (supp\ x \cap as))$ **using** *a* **by** (*simp add: alphas*)

finally have $(supp\ x' - as') \cup (supp\ x' \cap as') = (supp\ x' - as') \cup (p \cdot (supp\ x \cap as))$.

moreover

have $(supp\ x' - as') \cap (supp\ x' \cap as') = \{\}$ **by** *auto*

moreover

have $(supp\ x - as) \cap (supp\ x \cap as) = \{\}$ **by** *auto*

then have $p \cdot ((supp\ x - as) \cap (supp\ x \cap as) = \{\})$ **by** (*simp add: permute-bool-def*)

then have $(p \cdot (supp\ x - as)) \cap (p \cdot (supp\ x \cap as)) = \{\}$ **by** (*perm-simp*) (*simp*)

then have $(supp\ x - as) \cap (p \cdot (supp\ x \cap as)) = \{\}$ **using** $*$ **by** *simp*

then have $(supp\ x' - as') \cap (p \cdot (supp\ x \cap as)) = \{\}$ **using** *a* **by** (*simp add: alphas*)

ultimately show $p \cdot (supp\ x \cap as) = supp\ x' \cap as'$

by (*auto dest: disjoint-right-eq*)

qed

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

assumes $asm: (as, x) \approx_{res} (=) supp\ p' (as', x')$
shows $\exists p. (as, x) \approx_{res} (=) supp\ p (as', x') \wedge supp\ p \subseteq (supp\ x \cap as) \cup (supp\ x' \cap as')$
proof –
from asm **have** $0: (supp\ x - as) \#^* p'$ **by** $(auto\ simp\ only: alphas)$
then have $\#:$ $p' \cdot (supp\ x - as) = (supp\ x - as)$
by $(simp\ add: atom-set-perm-eq)$
obtain p **where** $*$: $\forall b \in supp\ x. p \cdot b = p' \cdot b$ **and** $**:$ $supp\ p \subseteq supp\ x \cup p' \cdot supp\ x$
using $set-renaming-perm2$ **by** $blast$
from $*$ **have** $a:$ $p \cdot x = p' \cdot x$ **using** $supp-perm-perm-eq$ **by** $auto$
from 0 **have** $1:$ $(supp\ x - as) \#^* p$ **using** $*$
by $(auto\ simp: fresh-star-def\ fresh-perm)$
then have $2:$ $(supp\ x - as) \cap supp\ p = \{\}$
by $(auto\ simp: fresh-star-def\ fresh-def)$
have $b:$ $supp\ x = (supp\ x - as) \cup (supp\ x \cap as)$ **by** $auto$
have $supp\ p \subseteq supp\ x \cup p' \cdot supp\ x$ **using** $**$ **by** $simp$
also have $\dots = (supp\ x - as) \cup (supp\ x \cap as) \cup (p' \cdot ((supp\ x - as) \cup (supp\ x \cap as)))$
using b **by** $simp$
also have $\dots = (supp\ x - as) \cup (supp\ x \cap as) \cup ((p' \cdot (supp\ x - as)) \cup (p' \cdot (supp\ x \cap as)))$
by $(simp\ add: union-eqvt)$
also have $\dots = (supp\ x - as) \cup (supp\ x \cap as) \cup (p' \cdot (supp\ x \cap as))$
using $\#$ **by** $auto$
also have $\dots = (supp\ x - as) \cup (supp\ x \cap as) \cup (supp\ x' \cap as')$ **using** asm
by $(simp\ add: supp-property-res)$
finally have $supp\ p \subseteq (supp\ x - as) \cup (supp\ x \cap as) \cup (supp\ x' \cap as')$.
then
have $supp\ p \subseteq (supp\ x \cap as) \cup (supp\ x' \cap as')$ **using** 2 **by** $auto$
moreover
have $(as, x) \approx_{res} (=) supp\ p (as', x')$ **using** $asm\ 1\ a$ **by** $(simp\ add: alphas)$
ultimately
show $\exists p. (as, x) \approx_{res} (=) supp\ p (as', x') \wedge supp\ p \subseteq (supp\ x \cap as) \cup (supp\ x' \cap as')$ **by** $blast$
qed

lemma $alpha-abs-res-minimal:$

assumes $asm: (as, x) \approx_{res} (=) supp\ p (as', x')$
shows $(as \cap supp\ x, x) \approx_{res} (=) supp\ p (as' \cap supp\ x', x')$
using asm **unfolding** $alpha-res$ **by** $(auto\ simp: Diff-Int)$

lemma $alpha-abs-res-abs-set:$

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

proof –

have $c:$ $p \cdot x = x'$
using $alpha-abs-res-minimal[OF\ asm]$ **unfolding** $alpha-res$ **by** $clarify$
then have $a:$ $supp\ x - as \cap supp\ x = supp\ (p \cdot x) - as' \cap supp\ (p \cdot x)$

using *alpha-abs-res-minimal*[*OF asm*] **by** (*simp add: alpha-res*)
have *b*: (*supp x - as ∩ supp x*) $\#^*$ *p*
using *alpha-abs-res-minimal*[*OF asm*] **unfolding** *alpha-res* **by** *clarify*
have *p* · (*as ∩ supp x*) = *as'* ∩ *supp (p · x)*
by (*metis Int-commute asm c supp-property-res*)
then show *?thesis* **using** *a b c* **unfolding** *alpha-set* **by** *simp*
qed

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

assumes *asm*: (*as ∩ supp x, x*) \approx_{set} (=) *supp p (as' ∩ supp x', x')*
shows (*as, x*) \approx_{res} (=) *supp p (as', x')*
using *asm* **unfolding** *alphas* **by** (*auto simp: Diff-Int*)

lemma *alpha-abs-res-stronger1*:

assumes *asm*: (*as, x*) \approx_{res} (=) *supp p' (as', x')*
shows $\exists p. (as, x) \approx_{res} (=) \text{supp } p (as', x') \wedge \text{supp } p \subseteq as \cup as'$
using *alpha-abs-res-stronger1-aux*[*OF asm*] **by** *auto*

lemma *alpha-abs-set-stronger1*:

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

from *asm* **have** *0*: (*supp x - as*) $\#^*$ *p'* **by** (*auto simp only: alphas*)
then have *#*: *p'* · (*supp x - as*) = (*supp x - as*)
by (*simp add: atom-set-perm-eq*)
obtain *p* **where** ***: $\forall b \in (supp\ x \cup as). p \cdot b = p' \cdot b$
and ****: *supp p* $\subseteq (supp\ x \cup as) \cup p' \cdot (supp\ x \cup as)$
using *set-renaming-perm2* **by** *blast*
from *** **have** $\forall b \in supp\ x. p \cdot b = p' \cdot b$ **by** *blast*
then have *a*: *p* · *x* = *p'* · *x* **using** *supp-perm-perm-eq* **by** *auto*
from *** **have** $\forall b \in as. p \cdot b = p' \cdot b$ **by** *blast*
then have *zb*: *p* · *as* = *p'* · *as*
apply(*auto simp: permute-set-def*)
apply(*rule-tac x=xa in exI*)
apply(*simp*)
done
have *zc*: *p'* · *as* = *as'* **using** *asm* **by** (*simp add: alphas*)
from *0* **have** *1*: (*supp x - as*) $\#^*$ *p* **using** ***
by (*auto simp: fresh-star-def fresh-perm*)
then have *2*: (*supp x - as*) ∩ *supp p* = {}
by (*auto simp: fresh-star-def fresh-def*)
have *b*: *supp x* = (*supp x - as*) ∪ (*supp x ∩ as*) **by** *auto*
have *supp p* $\subseteq supp\ x \cup as \cup p' \cdot supp\ x \cup p' \cdot as$ **using** **** **using** *union-eqvt*
by *blast*
also have $\dots = (supp\ x - as) \cup (supp\ x \cap as) \cup as \cup (p' \cdot ((supp\ x - as) \cup (supp\ x \cap as))) \cup p' \cdot as$
using *b* **by** *simp*
also have $\dots = (supp\ x - as) \cup (supp\ x \cap as) \cup as \cup ((p' \cdot (supp\ x - as)) \cup (p' \cdot (supp\ x \cap as))) \cup p' \cdot as$ **by** (*simp add: union-eqvt*)

also have $\dots = (\text{supp } x - \text{as}) \cup (\text{supp } x \cap \text{as}) \cup \text{as} \cup (p' \cdot (\text{supp } x \cap \text{as})) \cup p' \cdot \text{as}$
using $\#$ **by** *auto*
also have $\dots = (\text{supp } x - \text{as}) \cup (\text{supp } x \cap \text{as}) \cup \text{as} \cup p' \cdot ((\text{supp } x \cap \text{as}) \cup \text{as})$
using *union-eqvt*
by *auto*
also have $\dots = (\text{supp } x - \text{as}) \cup (\text{supp } x \cap \text{as}) \cup \text{as} \cup p' \cdot \text{as}$
by (*metis Int-commute Un-commute sup-inf-absorb*)
also have $\dots = (\text{supp } x - \text{as}) \cup \text{as} \cup p' \cdot \text{as}$ **by** *blast*
finally have $\text{supp } p \subseteq (\text{supp } x - \text{as}) \cup \text{as} \cup p' \cdot \text{as}$.
then have $\text{supp } p \subseteq \text{as} \cup p' \cdot \text{as}$ **using** \mathcal{Q} **by** *blast*
moreover
have $(\text{as}, x) \approx_{\text{set}} (=) \text{supp } p (\text{as}', x')$ **using** *asm 1 a zb* **by** (*simp add: alphas*)
ultimately
show $\exists p. (\text{as}, x) \approx_{\text{set}} (=) \text{supp } p (\text{as}', x') \wedge \text{supp } p \subseteq \text{as} \cup \text{as}'$ **using** *zc* **by**
blast
qed

lemma *alpha-abs-lst-stronger1*:

assumes *asm*: $(\text{as}, x) \approx_{\text{lst}} (=) \text{supp } p' (\text{as}', x')$
shows $\exists p. (\text{as}, x) \approx_{\text{lst}} (=) \text{supp } p (\text{as}', x') \wedge \text{supp } p \subseteq \text{set } \text{as} \cup \text{set } \text{as}'$
proof –
from *asm* **have** 0 : $(\text{supp } x - \text{set } \text{as}) \#* p'$ **by** (*auto simp only: alphas*)
then have $\#$: $p' \cdot (\text{supp } x - \text{set } \text{as}) = (\text{supp } x - \text{set } \text{as})$
by (*simp add: atom-set-perm-eq*)
obtain p **where** $*$: $\forall b \in (\text{supp } x \cup \text{set } \text{as}). p \cdot b = p' \cdot b$
and $**$: $\text{supp } p \subseteq (\text{supp } x \cup \text{set } \text{as}) \cup p' \cdot (\text{supp } x \cup \text{set } \text{as})$
using *set-renaming-perm2* **by** *blast*
from $*$ **have** $\forall b \in \text{supp } x. p \cdot b = p' \cdot b$ **by** *blast*
then have a : $p \cdot x = p' \cdot x$ **using** *supp-perm-perm-eq* **by** *auto*
from $*$ **have** $\forall b \in \text{set } \text{as}. p \cdot b = p' \cdot b$ **by** *blast*
then have zb : $p \cdot \text{as} = p' \cdot \text{as}$ **by** (*induct as*) (*auto*)
have zc : $p' \cdot \text{set } \text{as} = \text{set } \text{as}'$ **using** *asm* **by** (*simp add: alphas set-eqvt*)
from 0 **have** 1 : $(\text{supp } x - \text{set } \text{as}) \#* p$ **using** $*$
by (*auto simp: fresh-star-def fresh-perm*)
then have 2 : $(\text{supp } x - \text{set } \text{as}) \cap \text{supp } p = \{\}$
by (*auto simp: fresh-star-def fresh-def*)
have b : $\text{supp } x = (\text{supp } x - \text{set } \text{as}) \cup (\text{supp } x \cap \text{set } \text{as})$ **by** *auto*
have $\text{supp } p \subseteq \text{supp } x \cup \text{set } \text{as} \cup p' \cdot \text{supp } x \cup p' \cdot \text{set } \text{as}$ **using** $**$ **using**
union-eqvt **by** *blast*
also have $\dots = (\text{supp } x - \text{set } \text{as}) \cup (\text{supp } x \cap \text{set } \text{as}) \cup \text{set } \text{as} \cup$
 $(p' \cdot ((\text{supp } x - \text{set } \text{as}) \cup (\text{supp } x \cap \text{set } \text{as}))) \cup p' \cdot \text{set } \text{as}$ **using** b **by** *simp*
also have $\dots = (\text{supp } x - \text{set } \text{as}) \cup (\text{supp } x \cap \text{set } \text{as}) \cup \text{set } \text{as} \cup$
 $((p' \cdot (\text{supp } x - \text{set } \text{as})) \cup (p' \cdot (\text{supp } x \cap \text{set } \text{as}))) \cup p' \cdot \text{set } \text{as}$ **by** (*simp add:*
union-eqvt)
also have $\dots = (\text{supp } x - \text{set } \text{as}) \cup (\text{supp } x \cap \text{set } \text{as}) \cup \text{set } \text{as} \cup$
 $(p' \cdot (\text{supp } x \cap \text{set } \text{as})) \cup p' \cdot \text{set } \text{as}$ **using** $\#$ **by** *auto*
also have $\dots = (\text{supp } x - \text{set } \text{as}) \cup (\text{supp } x \cap \text{set } \text{as}) \cup \text{set } \text{as} \cup p' \cdot ((\text{supp } x$
 $\cap \text{set } \text{as}) \cup \text{set } \text{as})$

using *union-eqt* **by** *auto*
also have $\dots = (\text{supp } x - \text{set } as) \cup (\text{supp } x \cap \text{set } as) \cup \text{set } as \cup p' \cdot \text{set } as$
by (*metis Int-commute Un-commute sup-inf-absorb*)
also have $\dots = (\text{supp } x - \text{set } as) \cup \text{set } as \cup p' \cdot \text{set } as$ **by** *blast*
finally have $\text{supp } p \subseteq (\text{supp } x - \text{set } as) \cup \text{set } as \cup p' \cdot \text{set } as$.
then have $\text{supp } p \subseteq \text{set } as \cup p' \cdot \text{set } as$ **using** 2 **by** *blast*
moreover
have $(as, x) \approx_{\text{lst}} (=) \text{supp } p (as', x')$ **using** *asm 1 a zb* **by** (*simp add: alphas*)
ultimately
show $\exists p. (as, x) \approx_{\text{lst}} (=) \text{supp } p (as', x') \wedge \text{supp } p \subseteq \text{set } as \cup \text{set } as'$ **using** *zc*
by *blast*
qed

lemma *alphas-abs-stronger*:

shows $(as, x) \approx_{\text{abs-set}} (as', x') \longleftrightarrow (\exists p. (as, x) \approx_{\text{set}} (=) \text{supp } p (as', x') \wedge \text{supp } p \subseteq as \cup as')$
and $(as, x) \approx_{\text{abs-res}} (as', x') \longleftrightarrow (\exists p. (as, x) \approx_{\text{res}} (=) \text{supp } p (as', x') \wedge \text{supp } p \subseteq as \cup as')$
and $(bs, x) \approx_{\text{abs-lst}} (bs', x') \longleftrightarrow$
 $(\exists p. (bs, x) \approx_{\text{lst}} (=) \text{supp } p (bs', x') \wedge \text{supp } p \subseteq \text{set } bs \cup \text{set } bs')$
apply(*rule iffI*)
apply(*auto simp: alphas-abs alpha-abs-set-stronger1*)[1]
apply(*auto simp: alphas-abs*)[1]
apply(*rule iffI*)
apply(*auto simp: alphas-abs alpha-abs-res-stronger1*)[1]
apply(*auto simp: alphas-abs*)[1]
apply(*rule iffI*)
apply(*auto simp: alphas-abs alpha-abs-lst-stronger1*)[1]
apply(*auto simp: alphas-abs*)[1]
done

lemma *alpha-res-alpha-set*:

$(bs, x) \approx_{\text{res}} (=) \text{supp } p (cs, y) \longleftrightarrow (bs \cap \text{supp } x, x) \approx_{\text{set}} (=) \text{supp } p (cs \cap \text{supp } y, y)$
using *alpha-abs-set-abs-res alpha-abs-res-abs-set* **by** *blast*

31 Quotient types

quotient-type

$'a \text{ abs-set} = (\text{atom set} \times 'a::\text{pt}) / \text{alpha-abs-set}$

apply(*rule equivpI*)

unfolding *reflp-def refl-on-def symp-def sym-def transp-def trans-def*

by (*auto intro: alphas-abs-sym alphas-abs-refl alphas-abs-trans simp only*.)

quotient-type

$'b \text{ abs-res} = (\text{atom set} \times 'b::\text{pt}) / \text{alpha-abs-res}$

apply(*rule equivpI*)

unfolding *reflp-def refl-on-def symp-def sym-def transp-def trans-def*

by (*auto intro: alphas-abs-sym alphas-abs-refl alphas-abs-trans simp only*.)

quotient-type

'c abs-*lst* = (atom *list* × 'c::pt) / alpha-abs-*lst*
apply(rule-tac [!] equivpI)
unfolding reflp-def refl-on-def symp-def sym-def transp-def trans-def
by (auto intro: alphas-abs-sym alphas-abs-refl alphas-abs-trans simp only:)

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) .

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) .

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) .

lemma [quot-respect]:

shows ((=) ==> (=) ==> alpha-abs-set) Pair Pair
and ((=) ==> (=) ==> alpha-abs-res) Pair Pair
and ((=) ==> (=) ==> alpha-abs-*lst*) Pair Pair
unfolding rel-fun-def
by (auto intro: alphas-abs-refl)

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
unfolding rel-fun-def
by (auto intro: alphas-abs-*eqvt* simp only: Pair-*eqvt*)

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))
by (lifting alphas-abs)

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 $[cs]lst. x = [cs']lst. y \longleftrightarrow (\exists p. (cs, x) \approx_{lst} (=) supp\ p (cs', y) \wedge supp\ p \subseteq set\ cs \cup set\ cs')$
by (*lifting alphas-abs-stronger*)

lemma *Abs-eq-res-set*:

shows $[bs]res. x = [cs]res. y \longleftrightarrow [bs \cap supp\ x]set. x = [cs \cap supp\ y]set. y$
unfolding *Abs-eq-iff alpha-res-alpha-set* **by** *rule*

lemma *Abs-eq-res-supp*:

assumes *asm*: $supp\ x \subseteq bs$
shows $[as]res. x = [as \cap bs]res. x$
unfolding *Abs-eq-iff alphas*
apply (*rule-tac x=0::perm in exI*)
apply (*simp add: fresh-star-zero*)
using *asm* **by** *blast*

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 $(\bigwedge bs (x::'a::pt). y3 = [bs]lst. x \implies P3) \implies P3$
by (*lifting prod.exhaust[where 'a=atom set and 'b='a]*
prod.exhaust[where 'a=atom set and 'b='a]
prod.exhaust[where 'a=atom list and 'b='a])

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

begin

quotient-definition

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

is

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

by (*auto intro: alphas-abs-eqvt simp only: Pair-eqvt*)

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

fixes $x::'a::pt$

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

by (*lifting permute-prod.simps[where 'a=atom set and 'b='a]*)

instance

apply *standard*

apply (*case-tac [!] x*)

apply (*simp-all*)

done

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)
by (auto intro: *alphas-abs-eqv* simp only: *Pair-eqv*)

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

fixes *x*::'a::pt

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

by (*lifting permute-prod.simps*[**where** 'a=atom set **and** 'b='a])

instance

apply *standard*

apply(*case-tac* [!] *x*)

apply(*simp-all*)

done

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)
by (auto intro: *alphas-abs-eqv* simp only: *Pair-eqv*)

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

fixes *x*::'a::pt

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

by (*lifting permute-prod.simps*[**where** 'a=atom list **and** 'b='a])

instance

apply *standard*

apply(*case-tac* [!] *x*)

apply(*simp-all*)

done

end

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

lemma *Abs-swap1*:

assumes $a1: a \notin (\text{supp } x) - bs$
and $a2: b \notin (\text{supp } x) - bs$
shows $[bs]set. x = [(a \rightleftharpoons b) \cdot bs]set. ((a \rightleftharpoons b) \cdot x)$
and $[bs]res. x = [(a \rightleftharpoons b) \cdot bs]res. ((a \rightleftharpoons b) \cdot x)$
unfolding *Abs-eq-iff*
unfolding *alphas*
unfolding *supp-eqvt[symmetric] Diff-eqvt[symmetric]*
unfolding *fresh-star-def fresh-def*
unfolding *swap-set-not-in[OF a1 a2]*
using $a1 a2$
by (*rule-tac* [!] $x=(a \rightleftharpoons b)$ **in** exI)
(*auto simp: supp-perm swap-atom*)

lemma *Abs-swap2*:

assumes $a1: a \notin (\text{supp } x) - (\text{set } bs)$
and $a2: b \notin (\text{supp } x) - (\text{set } bs)$
shows $[bs]lst. x = [(a \rightleftharpoons b) \cdot bs]lst. ((a \rightleftharpoons b) \cdot x)$
unfolding *Abs-eq-iff*
unfolding *alphas*
unfolding *supp-eqvt[symmetric] Diff-eqvt[symmetric] set-eqvt[symmetric]*
unfolding *fresh-star-def fresh-def*
unfolding *swap-set-not-in[OF a1 a2]*
using $a1 a2$
by (*rule-tac* [!] $x=(a \rightleftharpoons b)$ **in** exI)
(*auto simp: supp-perm swap-atom*)

lemma *Abs-supports*:

shows $((\text{supp } x) - as) \text{ supports } ([as]set. x)$
and $((\text{supp } x) - as) \text{ supports } ([as]res. x)$
and $((\text{supp } x) - \text{set } bs) \text{ supports } ([bs]lst. x)$
unfolding *supports-def*
unfolding *permute-Abs*
by (*simp-all add: Abs-swap1[symmetric] Abs-swap2[symmetric]*)

function

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

where

$\text{supp-set } ([as]set. x) = \text{supp } x - as$
 $| \text{supp-res } ([as]res. x) = \text{supp } x - as$
 $| \text{supp-lst } (\text{Abs-lst } cs \ x) = (\text{supp } x) - (\text{set } cs)$
apply (*simp-all add: Abs-eq-iff alphas-abs alphas*)
apply (*case-tac x*)
apply (*case-tac a*)
apply (*simp*)
apply (*case-tac b*)
apply (*case-tac a*)

apply(*simp*)
apply(*case-tac ba*)
apply(*simp*)
done

termination
by *lexicographic-order*

lemma *supp-funs-eqvt*[*eqvt*]:
shows $(p \cdot \text{supp-set } x) = \text{supp-set } (p \cdot x)$
and $(p \cdot \text{supp-res } y) = \text{supp-res } (p \cdot y)$
and $(p \cdot \text{supp-lst } z) = \text{supp-lst } (p \cdot z)$
apply(*case-tac x*)
apply(*simp*)
apply(*case-tac y*)
apply(*simp*)
apply(*case-tac z*)
apply(*simp*)
done

lemma *Abs-fresh-aux*:
shows $a \# [\text{bs}] \text{set. } x \implies a \# \text{supp-set } ([\text{bs}] \text{set. } x)$
and $a \# [\text{bs}] \text{res. } x \implies a \# \text{supp-res } ([\text{bs}] \text{res. } x)$
and $a \# [\text{cs}] \text{lst. } x \implies a \# \text{supp-lst } ([\text{cs}] \text{lst. } x)$
by (*rule-tac* [!] *fresh-fun-eqvt-app*)
(auto simp only: eqvt-def eqvts-raw)

lemma *Abs-supp-subset1*:
assumes $a: \text{finite } (\text{supp } x)$
shows $(\text{supp } x) - \text{as} \subseteq \text{supp } ([\text{as}] \text{set. } x)$
and $(\text{supp } x) - \text{as} \subseteq \text{supp } ([\text{as}] \text{res. } x)$
and $(\text{supp } x) - (\text{set } \text{bs}) \subseteq \text{supp } ([\text{bs}] \text{lst. } x)$
unfolding *supp-conv-fresh*
by (*auto dest!*: *Abs-fresh-aux*)
(simp-all add: fresh-def supp-finite-atom-set a)

lemma *Abs-supp-subset2*:
assumes $a: \text{finite } (\text{supp } x)$
shows $\text{supp } ([\text{as}] \text{set. } x) \subseteq (\text{supp } x) - \text{as}$
and $\text{supp } ([\text{as}] \text{res. } x) \subseteq (\text{supp } x) - \text{as}$
and $\text{supp } ([\text{bs}] \text{lst. } x) \subseteq (\text{supp } x) - (\text{set } \text{bs})$
by (*rule-tac* [!] *supp-is-subset*)
(simp-all add: Abs-supports a)

lemma *Abs-finite-supp*:
assumes $a: \text{finite } (\text{supp } x)$
shows $\text{supp } ([\text{as}] \text{set. } x) = (\text{supp } x) - \text{as}$
and $\text{supp } ([\text{as}] \text{res. } x) = (\text{supp } x) - \text{as}$
and $\text{supp } ([\text{bs}] \text{lst. } x) = (\text{supp } x) - (\text{set } \text{bs})$

using *Abs-supp-subset1*[*OF a*] *Abs-supp-subset2*[*OF a*]
by *blast+*

lemma *supp-Abs*:
fixes $x::'a::fs$
shows $supp ([as]set. x) = (supp x) - as$
and $supp ([as]res. x) = (supp x) - as$
and $supp ([bs]lst. x) = (supp x) - (set bs)$
by (*simp-all add: Abs-finite-supp finite-supp*)

instance *abs-set* :: (*fs*) *fs*
apply *standard*
apply(*case-tac x*)
apply(*simp add: supp-Abs finite-supp*)
done

instance *abs-res* :: (*fs*) *fs*
apply *standard*
apply(*case-tac x*)
apply(*simp add: supp-Abs finite-supp*)
done

instance *abs-lst* :: (*fs*) *fs*
apply *standard*
apply(*case-tac x*)
apply(*simp add: supp-Abs finite-supp*)
done

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 $a \# [cs]lst. x \longleftrightarrow a \in (set cs) \vee (a \notin (set cs) \wedge a \# x)$
unfolding *fresh-def*
unfolding *supp-Abs*
by *auto*

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 $as \#* ([cs]lst. x) \longleftrightarrow (as - set cs) \#* x$
unfolding *fresh-star-def*
by (*auto simp: Abs-fresh-iff*)

lemma *Abs-fresh-star*:
fixes $x::'a::fs$
shows $as \subseteq as' \implies as \#* ([as']set. x)$
and $as \subseteq as' \implies as \#* ([as']res. x)$

and $bs \subseteq \text{set } bs' \implies bs \#* ([bs]lst. x)$
unfolding *fresh-star-def*
by(*auto simp: Abs-fresh-iff*)

lemma *Abs-fresh-star2*:

fixes $x::'a::fs$
shows $as \cap bs = \{\} \implies as \#* ([bs]set. x) \longleftrightarrow as \#* x$
and $as \cap bs = \{\} \implies as \#* ([bs]res. x) \longleftrightarrow as \#* x$
and $cs \cap \text{set } ds = \{\} \implies cs \#* ([ds]lst. x) \longleftrightarrow cs \#* x$
unfolding *fresh-star-def Abs-fresh-iff*
by *auto*

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 $[[atom\ a]]lst. x = [[atom\ a]]lst. y \longleftrightarrow x = y$
unfolding *Abs-eq-iff2 alphas*
by (*auto simp: supp-perm-singleton fresh-star-def fresh-zero-perm*)

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 –
have $[\{atom\ a\}]set. x = (a \leftrightarrow c) \cdot ([\{atom\ a\}]set. x)$
by (*rule-tac flip-fresh-fresh[symmetric]*) (*simp-all add: Abs-fresh-iff assms*)
then have $[\{atom\ a\}]set. x = [\{atom\ c\}]set. ((a \leftrightarrow c) \cdot x)$ **by** *simp*
moreover
have $[\{atom\ b\}]set. y = (b \leftrightarrow c) \cdot ([\{atom\ b\}]set. y)$
by (*rule-tac flip-fresh-fresh[symmetric]*) (*simp-all add: Abs-fresh-iff assms*)
then have $[\{atom\ b\}]set. y = [\{atom\ c\}]set. ((b \leftrightarrow c) \cdot y)$ **by** *simp*
ultimately
show $[\{atom\ a\}]set. x = [\{atom\ b\}]set. y \longleftrightarrow (a \leftrightarrow c) \cdot x = (b \leftrightarrow c) \cdot y$
by (*simp add: Abs1-eq*)
next
have $[\{atom\ a\}]res. x = (a \leftrightarrow c) \cdot ([\{atom\ a\}]res. x)$
by (*rule-tac flip-fresh-fresh[symmetric]*) (*simp-all add: Abs-fresh-iff assms*)
then have $[\{atom\ a\}]res. x = [\{atom\ c\}]res. ((a \leftrightarrow c) \cdot x)$ **by** *simp*
moreover
have $[\{atom\ b\}]res. y = (b \leftrightarrow c) \cdot ([\{atom\ b\}]res. y)$
by (*rule-tac flip-fresh-fresh[symmetric]*) (*simp-all add: Abs-fresh-iff assms*)
then have $[\{atom\ b\}]res. y = [\{atom\ c\}]res. ((b \leftrightarrow c) \cdot y)$ **by** *simp*
ultimately


```

show [{atom a}]res. x = [{atom b}]res. y  $\longleftrightarrow$  (a  $\leftrightarrow$  c)  $\cdot$  x = (b  $\leftrightarrow$  c)  $\cdot$  y
  by (simp add: Abs1-eq)
next
have [[atom a]]lst. x = (a  $\leftrightarrow$  c)  $\cdot$  ([[atom a]]lst. x)
  by (rule-tac flip-fresh-fresh[symmetric]) (simp-all add: Abs-fresh-iff assms)
then have [[atom a]]lst. x = [[atom c]]lst. ((a  $\leftrightarrow$  c)  $\cdot$  x) by simp
moreover
have [[atom b]]lst. y = (b  $\leftrightarrow$  c)  $\cdot$  ([[atom b]]lst. y)
  by (rule-tac flip-fresh-fresh[symmetric]) (simp-all add: Abs-fresh-iff assms)
then have [[atom b]]lst. y = [[atom c]]lst. ((b  $\leftrightarrow$  c)  $\cdot$  y) by simp
ultimately
show [[atom a]]lst. x = [[atom b]]lst. y  $\longleftrightarrow$  (a  $\leftrightarrow$  c)  $\cdot$  x = (b  $\leftrightarrow$  c)  $\cdot$  y
  by (simp add: Abs1-eq)
qed

```

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)
apply(auto)
apply(simp add: Abs1-eq-iff-fresh(1)[symmetric])
apply(rule-tac ?'a='b::at and x=(a, b, x, y, z) in obtain-fresh)
apply(drule-tac x=aa in spec)
apply(simp)
apply(subst Abs1-eq-iff-fresh(1))
apply(auto simp: fresh-Pair)[2]
apply(simp add: Abs1-eq-iff-fresh(2)[symmetric])
apply(rule-tac ?'a='b::at and x=(a, b, x, y, z) in obtain-fresh)
apply(drule-tac x=aa in spec)
apply(simp)
apply(subst Abs1-eq-iff-fresh(2))
apply(auto simp: fresh-Pair)[2]
apply(simp add: Abs1-eq-iff-fresh(3)[symmetric])
apply(rule-tac ?'a='b::at and x=(a, b, x, y, z) in obtain-fresh)
apply(drule-tac x=aa in spec)
apply(simp)
apply(subst Abs1-eq-iff-fresh(3))
apply(auto simp: fresh-Pair)[2]
done

```

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 –
 { **assume** $a = b$
 then have $[\{atom\ a\}]set. x = [\{atom\ b\}]set. y \longleftrightarrow (a = b \wedge x = y)$ **by** (*simp add: Abs1-eq*)
 }
 moreover
 { **assume** $*$: $a \neq b$ **and** $**$: $[\{atom\ a\}]set. x = [\{atom\ b\}]set. y$
 have $\#$: $atom\ a \# [\{atom\ b\}]set. y$ **by** (*simp add: **[symmetric] Abs-fresh-iff*)
 have $[\{atom\ a\}]set. ((a \leftrightarrow b) \cdot y) = (a \leftrightarrow b) \cdot ([\{atom\ b\}]set. y)$ **by** (*simp*)
 also have $\dots = [\{atom\ b\}]set. y$
 by (*rule flip-fresh-fresh*) (*simp add: #, simp add: Abs-fresh-iff*)
 also have $\dots = [\{atom\ a\}]set. x$ **using** $**$ **by** *simp*
 finally have $a \neq b \wedge x = (a \leftrightarrow b) \cdot y \wedge atom\ a \# y$ **using** $\# *$ **by** (*simp add: Abs1-eq Abs-fresh-iff*)
 }
 moreover
 { **assume** $*$: $a \neq b$ **and** $**$: $x = (a \leftrightarrow b) \cdot y \wedge atom\ a \# y$
 have $[\{atom\ a\}]set. x = [\{atom\ a\}]set. ((a \leftrightarrow b) \cdot y)$ **using** $**$ **by** *simp*
 also have $\dots = (a \leftrightarrow b) \cdot ([\{atom\ b\}]set. y)$ **by** (*simp add: permute-set-def*)
 also have $\dots = [\{atom\ b\}]set. y$
 by (*rule flip-fresh-fresh*) (*simp add: Abs-fresh-iff **, simp add: Abs-fresh-iff*)
 finally have $[\{atom\ a\}]set. x = [\{atom\ b\}]set. y$.
 }
 ultimately
 show $[\{atom\ a\}]set. x = [\{atom\ b\}]set. y \longleftrightarrow (a = b \wedge x = y) \vee (a \neq b \wedge x = (a \leftrightarrow b) \cdot y \wedge atom\ a \# y)$
 by *blast*
next
 { **assume** $a = b$
 then have $Abs-res\ \{atom\ a\}\ x = Abs-res\ \{atom\ b\}\ y \longleftrightarrow (a = b \wedge x = y)$ **by** (*simp add: Abs1-eq*)
 }
 moreover
 { **assume** $*$: $a \neq b$ **and** $**$: $Abs-res\ \{atom\ a\}\ x = Abs-res\ \{atom\ b\}\ y$
 have $\#$: $atom\ a \# Abs-res\ \{atom\ b\}\ y$ **by** (*simp add: **[symmetric] Abs-fresh-iff*)
 have $Abs-res\ \{atom\ a\}\ ((a \leftrightarrow b) \cdot y) = (a \leftrightarrow b) \cdot (Abs-res\ \{atom\ b\}\ y)$ **by** *simp*
 also have $\dots = Abs-res\ \{atom\ b\}\ y$
 by (*rule flip-fresh-fresh*) (*simp add: #, simp add: Abs-fresh-iff*)
 also have $\dots = Abs-res\ \{atom\ a\}\ x$ **using** $**$ **by** *simp*
 finally have $a \neq b \wedge x = (a \leftrightarrow b) \cdot y \wedge atom\ a \# y$ **using** $\# *$ **by** (*simp add: Abs1-eq Abs-fresh-iff*)
 }
}

```

moreover
{ assume *:  $a \neq b$  and **:  $x = (a \leftrightarrow b) \cdot y \wedge \text{atom } a \# y$ 
  have  $\text{Abs-res } \{ \text{atom } a \} x = \text{Abs-res } \{ \text{atom } a \} ((a \leftrightarrow b) \cdot y)$  using ** by simp
  also have  $\dots = (a \leftrightarrow b) \cdot \text{Abs-res } \{ \text{atom } b \} y$  by (simp add: permute-set-def)
  also have  $\dots = \text{Abs-res } \{ \text{atom } b \} y$ 
    by (rule flip-fresh-fresh) (simp add: Abs-fresh-iff **, simp add: Abs-fresh-iff)
  finally have  $\text{Abs-res } \{ \text{atom } a \} x = \text{Abs-res } \{ \text{atom } b \} y$  .
}
ultimately
show  $\text{Abs-res } \{ \text{atom } a \} x = \text{Abs-res } \{ \text{atom } b \} y \longleftrightarrow (a = b \wedge x = y) \vee (a \neq b \wedge x = (a \leftrightarrow b) \cdot y \wedge \text{atom } a \# y)$ 
  by blast
next
{ assume  $a = b$ 
  then have  $\text{Abs-lst } [\text{atom } a] x = \text{Abs-lst } [\text{atom } b] y \longleftrightarrow (a = b \wedge x = y)$  by
(simp add: Abs1-eq)
}
moreover
{ assume *:  $a \neq b$  and **:  $\text{Abs-lst } [\text{atom } a] x = \text{Abs-lst } [\text{atom } b] y$ 
  have #:  $\text{atom } a \# \text{Abs-lst } [\text{atom } b] y$  by (simp add: **[symmetric] Abs-fresh-iff)
  have  $\text{Abs-lst } [\text{atom } a] ((a \leftrightarrow b) \cdot y) = (a \leftrightarrow b) \cdot (\text{Abs-lst } [\text{atom } b] y)$  by simp
  also have  $\dots = \text{Abs-lst } [\text{atom } b] y$ 
    by (rule flip-fresh-fresh) (simp add: #, simp add: Abs-fresh-iff)
  also have  $\dots = \text{Abs-lst } [\text{atom } a] x$  using ** by simp
  finally have  $a \neq b \wedge x = (a \leftrightarrow b) \cdot y \wedge \text{atom } a \# y$  using # * by (simp add: Abs1-eq Abs-fresh-iff)
}
moreover
{ assume *:  $a \neq b$  and **:  $x = (a \leftrightarrow b) \cdot y \wedge \text{atom } a \# y$ 
  have  $\text{Abs-lst } [\text{atom } a] x = \text{Abs-lst } [\text{atom } a] ((a \leftrightarrow b) \cdot y)$  using ** by simp
  also have  $\dots = (a \leftrightarrow b) \cdot \text{Abs-lst } [\text{atom } b] y$  by simp
  also have  $\dots = \text{Abs-lst } [\text{atom } b] y$ 
    by (rule flip-fresh-fresh) (simp add: Abs-fresh-iff **, simp add: Abs-fresh-iff)
  finally have  $\text{Abs-lst } [\text{atom } a] x = \text{Abs-lst } [\text{atom } b] y$  .
}
ultimately
show  $\text{Abs-lst } [\text{atom } a] x = \text{Abs-lst } [\text{atom } b] y \longleftrightarrow (a = b \wedge x = y) \vee (a \neq b \wedge x = (a \leftrightarrow b) \cdot y \wedge \text{atom } a \# y)$ 
  by blast
qed

```

lemma *Abs1-eq-iff'*:

```

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

```

$a) \cdot x = y \wedge \text{atom } b \# x$
by (*auto simp: Abs1-eq-iff fresh-permute-left*)

ML \langle

```

fun alpha-single-simproc thm - ctxt ctrm =
  let
    val thy = Proof-Context.theory-of ctxt
    val - $ (- $ x) $ (- $ y) = Thm.term-of ctrm
    val cvrs = union (op =) (Term.add-frees x []) (Term.add-frees y [])
      |> filter (fn (-, ty) => Sign.of-sort thy (ty, @ {sort fs}))
      |> map Free
      |> HOLogic.mk-tuple
      |> Thm.cterm-of ctxt
    val cvrs-ty = Thm.ctyp-of-cterm cvrs
    val thm' = thm
      |> Thm.instantiate' [NONE, NONE, SOME cvrs-ty] [NONE, NONE, NONE,
NONE, SOME cvrs]
    in
      SOME thm'
    end
  >

```

simproc-setup *alpha-set* ($[\{\text{atom } a\}]_{\text{set}}. x = [\{\text{atom } b\}]_{\text{set}}. y$) =
 $\langle \text{alpha-single-simproc } @ \{ \text{thm Abs1-eq-iff-all}(1) [\text{THEN eq-reflection}] \} \rangle$

simproc-setup *alpha-res* ($[\{\text{atom } a\}]_{\text{res}}. x = [\{\text{atom } b\}]_{\text{res}}. y$) =
 $\langle \text{alpha-single-simproc } @ \{ \text{thm Abs1-eq-iff-all}(2) [\text{THEN eq-reflection}] \} \rangle$

simproc-setup *alpha-lst* ($[[\text{atom } a]]_{\text{lst}}. x = [[\text{atom } b]]_{\text{lst}}. y$) =
 $\langle \text{alpha-single-simproc } @ \{ \text{thm Abs1-eq-iff-all}(3) [\text{THEN eq-reflection}] \} \rangle$

32.1 Renaming of bodies of abstractions

lemma *Abs-rename-set*:

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

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

proof –

from *set-renaming-perm2*

obtain q **where** $*$: $\forall b \in bs. q \cdot b = p \cdot b$ **and** $**$: $\text{supp } q \subseteq bs \cup (p \cdot bs)$ **by**
blast

have $**$: $q \cdot bs = p \cdot bs$ **using** $*$

unfolding *permute-set-eq-image image-def* **by** *auto*

have $[bs]_{\text{set}}. x = q \cdot ([bs]_{\text{set}}. x)$

apply (*rule perm-supp-eq[symmetric]*)

using $a **$

unfolding *Abs-fresh-star-iff*

unfolding *fresh-star-def*
by *auto*
also have $\dots = [q \cdot bs]set. (q \cdot x)$ **by** *simp*
finally have $[bs]set. x = [p \cdot bs]set. (q \cdot x)$ **by** (*simp add: ****)
then show $\exists q. [bs]set. x = [p \cdot bs]set. (q \cdot x) \wedge q \cdot bs = p \cdot bs$ **using** ***** **by**
metis
qed

lemma *Abs-rename-res:*

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

shows $\exists q. [bs]res. x = [p \cdot bs]res. (q \cdot x) \wedge q \cdot bs = p \cdot bs$

proof –

from *set-renaming-perm2*

obtain q **where** $*$: $\forall b \in bs. q \cdot b = p \cdot b$ **and** $**$: $supp\ q \subseteq bs \cup (p \cdot bs)$ **by**

blast

have $***$: $q \cdot bs = p \cdot bs$ **using** $*$

unfolding *permute-set-eq-image image-def* **by** *auto*

have $[bs]res. x = q \cdot ([bs]res. x)$

apply(*rule perm-supp-eq[symmetric]*)

using a $**$

unfolding *Abs-fresh-star-iff*

unfolding *fresh-star-def*

by *auto*

also have $\dots = [q \cdot bs]res. (q \cdot x)$ **by** *simp*

finally have $[bs]res. x = [p \cdot bs]res. (q \cdot x)$ **by** (*simp add: ****)

then show $\exists q. [bs]res. x = [p \cdot bs]res. (q \cdot x) \wedge q \cdot bs = p \cdot bs$ **using** ***** **by**

metis

qed

lemma *Abs-rename-lst:*

fixes $x::'a::fs$

assumes $a: (p \cdot (set\ bs)) \#* x$

shows $\exists q. [bs]lst. x = [p \cdot bs]lst. (q \cdot x) \wedge q \cdot bs = p \cdot bs$

proof –

from *list-renaming-perm*

obtain q **where** $*$: $\forall b \in set\ bs. q \cdot b = p \cdot b$ **and** $**$: $supp\ q \subseteq set\ bs \cup (p \cdot set\ bs)$ **by** *blast*

have $***$: $q \cdot bs = p \cdot bs$ **using** $*$ **by** (*induct bs*) (*simp-all add: insert-eqt*)

have $[bs]lst. x = q \cdot ([bs]lst. x)$

apply(*rule perm-supp-eq[symmetric]*)

using a $**$

unfolding *Abs-fresh-star-iff*

unfolding *fresh-star-def*

by *auto*

also have $\dots = [q \cdot bs]lst. (q \cdot x)$ **by** *simp*

finally have $[bs]lst. x = [p \cdot bs]lst. (q \cdot x)$ **by** (*simp add: ****)

then show $\exists q. [bs]lst. x = [p \cdot bs]lst. (q \cdot x) \wedge q \cdot bs = p \cdot bs$ **using** ***** **by**

metis

qed

for deep recursive binders

lemma *Abs-rename-set'*:

fixes $x::'a::fs$

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

shows $\exists q. [bs]set. x = [q \cdot bs]set. (q \cdot x) \wedge q \cdot bs = p \cdot bs$

using *Abs-rename-set[OF a]* **by** *metis*

lemma *Abs-rename-res'*:

fixes $x::'a::fs$

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

shows $\exists q. [bs]res. x = [q \cdot bs]res. (q \cdot x) \wedge q \cdot bs = p \cdot bs$

using *Abs-rename-res[OF a]* **by** *metis*

lemma *Abs-rename-lst'*:

fixes $x::'a::fs$

assumes $a: (p \cdot (set\ bs)) \#* x$

shows $\exists q. [bs]lst. x = [q \cdot bs]lst. (q \cdot x) \wedge q \cdot bs = p \cdot bs$

using *Abs-rename-lst[OF a]* **by** *metis*

33 Infrastructure for building tuples of relations and functions

fun

prod-fv :: $('a \Rightarrow atom\ set) \Rightarrow ('b \Rightarrow atom\ set) \Rightarrow ('a \times 'b) \Rightarrow atom\ set$

where

prod-fv *fv1* *fv2* $(x, y) = fv1\ x \cup fv2\ y$

definition

prod-alpha :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \times 'b \Rightarrow 'a \times 'b \Rightarrow bool)$

where

prod-alpha = *rel-prod*

lemma [*quot-respect*]:

shows $((R1 \implies (=)) \implies (R2 \implies (=)) \implies rel\text{-}prod\ R1\ R2 \implies (=))\ prod\text{-}fv\ prod\text{-}fv$

unfolding *rel-fun-def*

by *auto*

lemma [*quot-preserve*]:

assumes $q1: Quotient3\ R1\ abs1\ rep1$

and $q2: Quotient3\ R2\ abs2\ rep2$

shows $((abs1 \dashrightarrow id) \dashrightarrow (abs2 \dashrightarrow id) \dashrightarrow map\text{-}prod\ rep1\ rep2$

```

---> id) prod-fv = prod-fv
  by (simp add: fun-eq-iff Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2])

```

```

lemma [mono]:
  shows  $A \leq B \implies C \leq D \implies \text{prod-alpha } A \ C \leq \text{prod-alpha } B \ D$ 
  unfolding prod-alpha-def
  by auto

```

```

lemma [eqvt]:
  shows  $p \cdot \text{prod-alpha } A \ B \ x \ y = \text{prod-alpha } (p \cdot A) \ (p \cdot B) \ (p \cdot x) \ (p \cdot y)$ 
  unfolding prod-alpha-def
  unfolding rel-prod-conv
  by (perm-simp) (rule refl)

```

```

lemma [eqvt]:
  shows  $p \cdot \text{prod-fv } A \ B \ (x, y) = \text{prod-fv } (p \cdot A) \ (p \cdot B) \ (p \cdot x, p \cdot y)$ 
  unfolding prod-fv.simps
  by (perm-simp) (rule refl)

```

```

lemma prod-fv-supp:
  shows  $\text{prod-fv } \text{supp } \text{supp} = \text{supp}$ 
  by (rule ext)
  (auto simp: supp-Pair)

```

```

lemma prod-alpha-eq:
  shows  $\text{prod-alpha } ((=)) \ ((=)) = ((=))$ 
  unfolding prod-alpha-def
  by (auto intro!: ext)

```

```

end
theory Nominal2-FCB
imports Nominal2-Abs
begin

```

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

```

ML <
val all-trivials : (Proof.context -> Proof.method) context-parser =
Scan.succeed (fn ctxt =>
  let
    val tac = TRYALL (SOLVED' (full-simp-tac ctxt))
  in
    Method.SIMPLE-METHOD' (K tac)
  end)
>

```

```

method-setup all-trivials = <all-trivials> <solves trivial goals>

```

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

using e

apply(*case-tac* $atom\ x \# S$)

apply(*simp* *add: Abs1-eq-iff'*)

apply(*elim* *conjE* *disjE*)

apply(*simp*)

apply(*rule* *trans*)

apply(*rule-tac* $p=(x \leftrightarrow y)$ **in** *supp-perm-eq[symmetric]*)

apply(*rule* *fresh-star-supp-conv*)

apply(*simp* *add: flip-def* *supp-swap* *fresh-star-def* $f1\ f2$)

apply(*simp* *add: flip-commute* p)

apply(*simp* *add: Abs1-eq-iff*)

done

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

using e **apply** $-$

apply(*subst* (*asm*) *Abs-eq-iff2*)

apply(*simp* *add: alphas*)

apply(*elim* *exE* *conjE*)

apply(*rule* *trans*)

apply(*rule-tac* $p=p$ **in** *supp-perm-eq[symmetric]*)

apply(*rule* *fresh-star-supp-conv*)

apply(*drule* *fresh-star-perm-set-conv*)

apply(*rule* *finite-Diff*)

apply(*rule* *finite-supp*)

apply(*subgoal-tac* ($set\ (ba\ xs) \cup set\ (ba\ ys)$) $\#* f\ xs\ T$)

apply(*metis* *Un-absorb2* *fresh-star-Un*)

apply(*subst* *fresh-star-Un*)

apply(*rule* *conjI*)

apply(*simp* *add: fresh-star-def* $f1$)

apply(*simp* *add: fresh-star-def* $f2$)

apply(*simp* *add: eqv*)

done

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 $f1: \bigwedge x. x \in ba\ xs \implies x \# f\ xs\ T$
and $f2: \bigwedge x. \llbracket supp\ T - ba\ xs = supp\ S - ba\ ys; x \in ba\ ys \rrbracket \implies x \# f\ xs\ T$
and $eqv: \bigwedge p. \llbracket p \cdot T = S; p \cdot ba\ xs = ba\ ys; supp\ p \subseteq ba\ xs \cup ba\ ys \rrbracket \implies p \cdot (f\ xs\ T) = f\ ys\ S$
shows $f\ xs\ T = f\ ys\ S$
using e apply –
apply(subst (asm) Abs-eq-iff2)
apply(simp add: alphas)
apply(elim exE conjE)
apply(rule trans)
apply(rule-tac $p=p$ in supp-perm-eq[symmetric])
apply(rule fresh-star-supp-conv)
apply(drule fresh-star-perm-set-conv)
apply(rule finite-Diff)
apply(rule finite-supp)
apply(subgoal-tac $(ba\ xs \cup ba\ ys) \#* f\ xs\ T$)
apply(metis Un-absorb2 fresh-star-Un)
apply(subst fresh-star-Un)
apply(rule conjI)
apply(simp add: fresh-star-def f1)
apply(simp add: fresh-star-def f2)
apply(simp add: eqv)
done

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 $f1: \bigwedge x. x \in atom\ 'xs \implies x \in supp\ T \implies x \# f\ xs\ T$
and $f2: \bigwedge x. \llbracket supp\ T - atom\ 'xs = supp\ S - atom\ 'ys; x \in atom\ 'ys; x \in supp\ S \rrbracket \implies x \# f\ xs\ T$
and $eqv: \bigwedge p. \llbracket p \cdot T = S; supp\ p \subseteq atom\ 'xs \cap supp\ T \cup atom\ 'ys \cap supp\ S; p \cdot (atom\ 'xs \cap supp\ T) = atom\ 'ys \cap supp\ S \rrbracket \implies p \cdot (f\ xs\ T) = f\ ys\ S$
shows $f\ xs\ T = f\ ys\ S$
using e apply –
apply(subst (asm) Abs-eq-res-set)
apply(subst (asm) Abs-eq-iff2)
apply(simp add: alphas)
apply(elim exE conjE)
apply(rule trans)
apply(rule-tac $p=p$ in supp-perm-eq[symmetric])
apply(rule fresh-star-supp-conv)
apply(drule fresh-star-perm-set-conv)

```

apply(rule finite-Diff)
apply(rule finite-supp)
apply(subgoal-tac (atom ' xs  $\cap$  supp T  $\cup$  atom ' ys  $\cap$  supp S)  $\#^*$  f xs T)
apply(metis Un-absorb2 fresh-star-Un)
apply(subst fresh-star-Un)
apply(rule conjI)
apply(simp add: fresh-star-def f1)
apply(subgoal-tac supp T - atom ' xs = supp S - atom ' ys)
apply(simp add: fresh-star-def f2)
apply(blast)
apply(simp add: eqv)
done

```

lemma Abs-set-fcb2:

```

fixes as bs :: atom set
  and x y :: 'b :: fs
  and c::'c::fs
assumes eq: [as]set. x = [bs]set. y
and fin: finite as finite bs
and fcb1: as  $\#^*$  f as x c
and fresh1: as  $\#^*$  c
and fresh2: bs  $\#^*$  c
and perm1:  $\bigwedge p. \text{supp } p \ \#^* \ c \implies p \cdot (f \text{ as } x \ c) = f (p \cdot \text{as}) (p \cdot x) \ c$ 
and perm2:  $\bigwedge p. \text{supp } p \ \#^* \ c \implies p \cdot (f \text{ bs } y \ c) = f (p \cdot \text{bs}) (p \cdot y) \ c$ 
shows f as x c = f bs y c
proof -
  have supp (as, x, c) supports (f as x c)
    unfolding supports-def fresh-def[symmetric]
    by (simp add: fresh-Pair perm1 fresh-star-def supp-swap swap-fresh-fresh)
  then have fin1: finite (supp (f as x c))
    using fin by (auto intro: supports-finite simp add: finite-supp supp-of-finite-sets
supp-Pair)
  have supp (bs, y, c) supports (f bs y c)
    unfolding supports-def fresh-def[symmetric]
    by (simp add: fresh-Pair perm2 fresh-star-def supp-swap swap-fresh-fresh)
  then have fin2: finite (supp (f bs y c))
    using fin by (auto intro: supports-finite simp add: finite-supp supp-of-finite-sets
supp-Pair)
  obtain q::perm where
    fr1: (q  $\cdot$  as)  $\#^*$  (x, c, f as x c, f bs y c) and
    fr2: supp q  $\#^*$  ([as]set. x) and
    inc: supp q  $\subseteq$  as  $\cup$  (q  $\cdot$  as)
    using at-set-avoiding3[where xs=as and c=(x, c, f as x c, f bs y c) and
x=[as]set. x]
    fin1 fin2 fin
  by (auto simp add: supp-Pair finite-supp Abs-fresh-star dest: fresh-star-supp-conv)
  have [q  $\cdot$  as]set. (q  $\cdot$  x) = q  $\cdot$  ([as]set. x) by simp

```

```

also have ... = [as]set. x
  by (simp only: fr2 perm-supp-eq)
finally have [q · as]set. (q · x) = [bs]set. y using eq by simp
then obtain r::perm where
  qq1: q · x = r · y and
  qq2: q · as = r · bs and
  qq3: supp r ⊆ (q · as) ∪ bs
  apply(drule-tac sym)
  apply(simp only: Abs-eq-iff2 alphas)
  apply(erule exE)
  apply(erule conjE)+
  apply(drule-tac x=p in meta-spec)
  apply(simp add: set-eqvt)
  apply(blast)
  done
have as #* f as x c by (rule fcb1)
then have q · (as #* f as x c)
  by (simp add: permute-bool-def)
then have (q · as) #* f (q · as) (q · x) c
  apply(simp only: fresh-star-eqvt set-eqvt)
  apply(subst (asm) perm1)
  using inc fresh1 fr1
  apply(auto simp add: fresh-star-def fresh-Pair)
  done
then have (r · bs) #* f (r · bs) (r · y) c using qq1 qq2 by simp
then have r · (bs #* f bs y c)
  apply(simp only: fresh-star-eqvt set-eqvt)
  apply(subst (asm) perm2[symmetric])
  using qq3 fresh2 fr1
  apply(auto simp add: set-eqvt fresh-star-def fresh-Pair)
  done
then have fcb2: bs #* f bs y c by (simp add: permute-bool-def)
have f as x c = q · (f as x c)
  apply(rule perm-supp-eq[symmetric])
  using inc fcb1 fr1 by (auto simp add: fresh-star-def)
also have ... = f (q · as) (q · x) c
  apply(rule perm1)
  using inc fresh1 fr1 by (auto simp add: fresh-star-def)
also have ... = f (r · bs) (r · y) c using qq1 qq2 by simp
also have ... = r · (f bs y c)
  apply(rule perm2[symmetric])
  using qq3 fresh2 fr1 by (auto simp add: fresh-star-def)
also have ... = f bs y c
  apply(rule perm-supp-eq)
  using qq3 fr1 fcb2 by (auto simp add: fresh-star-def)
finally show ?thesis by simp
qed

```

```

lemma Abs-res-fcb2:
  fixes as bs :: atom set
    and x y :: 'b :: fs
    and c::'c::fs
  assumes eq: [as]res. x = [bs]res. y
  and fin: finite as finite bs
  and fcb1: (as ∩ supp x) #* f (as ∩ 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 –
  have supp (as, x, c) supports (f (as ∩ supp x) x c)
    unfolding supports-def fresh-def[symmetric]
    by (simp add: fresh-Pair perm1 fresh-star-def supp-swap swap-fresh-fresh inter-eqvt supp-eqvt)
  then have fin1: finite (supp (f (as ∩ supp x) x c))
    using fin by (auto intro: supports-finite simp add: finite-supp supp-of-finite-sets supp-Pair)
  have supp (bs, y, c) supports (f (bs ∩ supp y) y c)
    unfolding supports-def fresh-def[symmetric]
    by (simp add: fresh-Pair perm2 fresh-star-def supp-swap swap-fresh-fresh inter-eqvt supp-eqvt)
  then have fin2: finite (supp (f (bs ∩ supp y) y c))
    using fin by (auto intro: supports-finite simp add: finite-supp supp-of-finite-sets supp-Pair)
  obtain q::perm where
    fr1: (q ∙ (as ∩ supp x)) #* (x, c, f (as ∩ supp x) x c, f (bs ∩ supp y) y c) and
    fr2: supp q #* ([as ∩ supp x]set. x) and
    inc: supp q ⊆ (as ∩ supp x) ∪ (q ∙ (as ∩ supp x))
    using at-set-avoiding3[where xs=as ∩ supp x and c=(x, c, f (as ∩ supp x) x c, f (bs ∩ supp y) y c)
      and x=[as ∩ supp x]set. x
      fin1 fin2 fin
  apply (auto simp add: supp-Pair finite-supp Abs-fresh-star dest: fresh-star-supp-conv)
  done
  have [q ∙ (as ∩ supp x)]set. (q ∙ x) = q ∙ ([as ∩ supp x]set. x) by simp
  also have ... = [as ∩ supp x]set. x
    by (simp only: fr2 perm-supp-eq)
  finally have [q ∙ (as ∩ supp x)]set. (q ∙ x) = [bs ∩ supp y]set. y using eq
    by(simp add: Abs-eq-res-set)
  then obtain r::perm where
    qq1: q ∙ x = r ∙ y and
    qq2: (q ∙ as ∩ supp (q ∙ x)) = r ∙ (bs ∩ supp y) and
    qq3: supp r ⊆ (bs ∩ supp y) ∪ q ∙ (as ∩ supp x)
    apply(drule-tac sym)

```

```

apply(simp only: Abs-eq-iff2 alphas)
apply(erule exE)
apply(erule conjE)+
apply(drule-tac x=p in meta-spec)
apply(simp add: set-eqvt inter-eqvt supp-eqvt)
done
have ( $as \cap supp\ x$ )  $\#*$   $f$  ( $as \cap supp\ x$ )  $x\ c$  by (rule fcb1)
then have  $q \cdot ((as \cap supp\ x) \#* f (as \cap supp\ x) x\ c)$ 
  by (simp add: permute-bool-def)
then have ( $q \cdot (as \cap supp\ x)$ )  $\#*$   $f$  ( $q \cdot (as \cap supp\ x)$ ) ( $q \cdot x$ )  $c$ 
  apply(simp only: fresh-star-eqvt set-eqvt)
  apply(subst (asm) perm1)
  using inc fresh1 fr1
  apply(auto simp add: fresh-star-def fresh-Pair)
  done
then have ( $r \cdot (bs \cap supp\ y)$ )  $\#*$   $f$  ( $r \cdot (bs \cap supp\ y)$ ) ( $r \cdot y$ )  $c$  using qq1 qq2
  apply(perm-simp)
  apply simp
  done
then have  $r \cdot ((bs \cap supp\ y) \#* f (bs \cap supp\ y) y\ c)$ 
  apply(simp only: fresh-star-eqvt set-eqvt)
  apply(subst (asm) perm2[symmetric])
  using qq3 fresh2 fr1
  apply(auto simp add: set-eqvt fresh-star-def fresh-Pair)
  done
then have fc2: ( $bs \cap supp\ y$ )  $\#*$   $f$  ( $bs \cap supp\ y$ )  $y\ c$  by (simp add: permute-bool-def)
have  $f (as \cap supp\ x) x\ c = q \cdot (f (as \cap supp\ x) x\ c)$ 
  apply(rule perm-supp-eq[symmetric])
  using inc fcb1 fr1
  apply (auto simp add: fresh-star-def)
  done
also have  $\dots = f (q \cdot (as \cap supp\ x)) (q \cdot x) c$ 
  apply(rule perm1)
  using inc fresh1 fr1 by (auto simp add: fresh-star-def)
also have  $\dots = f (r \cdot (bs \cap supp\ y)) (r \cdot y) c$  using qq1 qq2
  apply(perm-simp)
  apply simp
  done
also have  $\dots = r \cdot (f (bs \cap supp\ y) y\ c)$ 
  apply(rule perm2[symmetric])
  using qq3 fresh2 fr1 by (auto simp add: fresh-star-def)
also have  $\dots = f (bs \cap supp\ y) y\ c$ 
  apply(rule perm-supp-eq)
  using qq3 fr1 fcb2 by (auto simp add: fresh-star-def)
finally show ?thesis by simp
qed

```

lemma *Abs-lst-fcb2*:

```

fixes as bs :: atom list
  and x y :: 'b :: fs
  and c::'c::fs
assumes eq: [as]lst. x = [bs]lst. y
and fc1: (set as) #* f as x c
and fresh1: set as #* c
and fresh2: set bs #* c
and perm1:  $\bigwedge p. \text{supp } p \#* c \implies p \cdot (f \text{ as } x \ c) = f (p \cdot \text{as}) (p \cdot x) \ c$ 
and perm2:  $\bigwedge p. \text{supp } p \#* c \implies p \cdot (f \text{ bs } y \ c) = f (p \cdot \text{bs}) (p \cdot y) \ c$ 
shows f as x c = f bs y c
proof –
  have supp (as, x, c) supports (f as x c)
    unfolding supports-def fresh-def[symmetric]
    by (simp add: fresh-Pair perm1 fresh-star-def supp-swap swap-fresh-fresh)
  then have fin1: finite (supp (f as x c))
    by (auto intro: supports-finite simp add: finite-supp)
  have supp (bs, y, c) supports (f bs y c)
    unfolding supports-def fresh-def[symmetric]
    by (simp add: fresh-Pair perm2 fresh-star-def supp-swap swap-fresh-fresh)
  then have fin2: finite (supp (f bs y c))
    by (auto intro: supports-finite simp add: finite-supp)
  obtain q::perm where
    fr1: (q · (set as)) #* (x, c, f as x c, f bs y c) and
    fr2: supp q #* Abs-lst as x and
    inc: supp q  $\subseteq$  (set as)  $\cup$  q · (set as)
    using at-set-avoiding3[where xs=set as and c=(x, c, f as x c, f bs y c) and
x=[as]lst. x]
    fin1 fin2
    by (auto simp add: supp-Pair finite-supp Abs-fresh-star dest: fresh-star-supp-conv)
  have Abs-lst (q · as) (q · x) = q · Abs-lst as x by simp
  also have ... = Abs-lst as x
    by (simp only: fr2 perm-supp-eq)
  finally have Abs-lst (q · as) (q · x) = Abs-lst bs y using eq by simp
  then obtain r::perm where
    qq1: q · x = r · y and
    qq2: q · as = r · bs and
    qq3: supp r  $\subseteq$  (q · (set as))  $\cup$  set bs
    apply(drule-tac sym)
    apply(simp only: Abs-eq-iff2 alphas)
    apply(erule exE)
    apply(erule conjE)+
    apply(drule-tac x=p in meta-spec)
    apply(simp add: set-eqvt)
    apply(blast)
  done
  have (set as) #* f as x c by (rule fc1)
  then have q · ((set as) #* f as x c)
    by (simp add: permute-bool-def)
  then have set (q · as) #* f (q · as) (q · x) c

```

```

apply(simp only: fresh-star-eqvt set-eqvt)
apply(subst (asm) perm1)
using inc fresh1 fr1
apply(auto simp add: fresh-star-def fresh-Pair)
done
then have set (r · bs) #* f (r · bs) (r · y) c using qq1 qq2 by simp
then have r · ((set bs) #* f bs y c)
  apply(simp only: fresh-star-eqvt set-eqvt)
  apply(subst (asm) perm2[symmetric])
  using qq3 fresh2 fr1
  apply(auto simp add: set-eqvt fresh-star-def fresh-Pair)
  done
then have fc2: (set bs) #* f bs y c by (simp add: permute-bool-def)
have f as x c = q · (f as x c)
  apply(rule perm-supp-eq[symmetric])
  using inc fc1 fr1 by (auto simp add: fresh-star-def)
also have ... = f (q · as) (q · x) c
  apply(rule perm1)
  using inc fresh1 fr1 by (auto simp add: fresh-star-def)
also have ... = f (r · bs) (r · y) c using qq1 qq2 by simp
also have ... = r · (f bs y c)
  apply(rule perm2[symmetric])
  using qq3 fresh2 fr1 by (auto simp add: fresh-star-def)
also have ... = f bs y c
  apply(rule perm-supp-eq)
  using qq3 fr1 fc2 by (auto simp add: fresh-star-def)
finally show ?thesis by simp
qed

```

```

lemma Abs-lst1-fcb2:
  fixes a b :: atom
    and x y :: 'b :: fs
    and c::'c :: fs
  assumes e: [[a]]lst. x = [[b]]lst. y
  and fc1: a #* f a x c
  and fresh: {a, b} #* c
  and perm1:  $\bigwedge p. \text{supp } p \#* c \implies p \cdot (f a x c) = f (p \cdot a) (p \cdot x) c$ 
  and 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
using e
apply(drule-tac Abs-lst-fcb2[where c=c and f= $\lambda(as::atom list) . f (hd as)$ ])
apply(simp-all)
using fc1 fresh perm1 perm2
apply(simp-all add: fresh-star-def)
done

```

```

lemma Abs-lst1-fcb2':
  fixes a b :: 'a::at-base
    and x y :: 'b :: fs

```

```

    and c::'c :: fs
  assumes e: [[atom a]]lst. x = [[atom b]]lst. y
  and fcb1: atom a # f a x c
  and fresh: {atom a, atom b} #* c
  and perm1:  $\bigwedge p. \text{supp } p \#* c \implies p \cdot (f a x c) = f (p \cdot a) (p \cdot x) c$ 
  and 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
using e
apply(drule-tac Abs-lst1-fcb2[where c=c and f= $\lambda a . f ((inv \text{atom}) a)$ ])
using fcb1 fresh perm1 perm2
apply(simp-all add: fresh-star-def inv-f-f inj-on-def atom-eqvt)
done

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

ML-file <nominal-dt-data.ML>
ML <open Nominal-Dt-Data>

ML-file <nominal-dt-rawfuns.ML>
ML <open Nominal-Dt-RawFuns>

ML-file <nominal-dt-alpha.ML>
ML <open Nominal-Dt-Alpha>

ML-file <nominal-dt-quot.ML>
ML <open Nominal-Dt-Quot>

ML-file <nominal-induct.ML>
method-setup nominal-induct =
  <NominalInduct.nominal-induct-method>
  <nominal induction>

ML-file <nominal-inductive.ML>

```


ML-file \langle nominal-function-common.ML \rangle
ML-file \langle nominal-function-core.ML \rangle
ML-file \langle nominal-mutual.ML \rangle
ML-file \langle nominal-function.ML \rangle
ML-file \langle nominal-termination.ML \rangle

34 Interface for *nominal-datatype*

```

ML  $\langle$ 
  fun get-cnstrs dts =
    map snd dts

  fun get-typed-cnstrs dts =
    flat (map (fn ((bn, -, -), constrs) =>
      (map (fn (bn', -, -) => (Binding.name-of bn, Binding.name-of bn')) constrs))
    dts)

  fun get-cnstr-strs dts =
    map (fn (bn, -, -) => Binding.name-of bn) (flat (get-cnstrs dts))

  fun get-bn-fun-strs bn-funs =
    map (fn (bn-fun, -, -) => Binding.name-of bn-fun) bn-funs
 $\rangle$ 

```

Infrastructure for adding *-raw* to types and terms

```

ML  $\langle$ 
  fun add-raw s = s ^ -raw
  fun add-raws ss = map add-raw ss
  fun raw-bind bn = Binding.suffix-name -raw bn

  fun replace-str ss s =
    case (AList.lookup (op =) ss s) of
      SOME s' => s'
    | NONE => s

  fun replace-typ ty-ss (Type (a, Ts)) = Type (replace-str ty-ss a, map (replace-typ
  ty-ss) Ts)
    | replace-typ ty-ss T = T

  fun raw-dts ty-ss dts =
  let
    fun raw-dts-aux1 (bind, tys, -) =
      (raw-bind bind, map (replace-typ ty-ss) tys, NoSyn)

    fun raw-dts-aux2 ((bind, ty-args, -), constrs) =
      ((raw-bind bind, ty-args, NoSyn), map raw-dts-aux1 constrs)
  in
    map raw-dts-aux2 dts

```

end

```
fun replace-aterm trm-ss (Const (a, T)) = Const (replace-str trm-ss a, T)  
  | replace-aterm trm-ss (Free (a, T)) = Free (replace-str trm-ss a, T)  
  | replace-aterm trm-ss trm = trm
```

```
fun replace-term trm-ss ty-ss trm =  
  trm |> Term.map-aterms (replace-aterm trm-ss) |> map-types (replace-typ ty-ss)  
>
```

ML <

```
fun rawify-dts dts dts-env = raw-dts dts-env dts  
>
```

ML <

```
fun rawify-bn-funs dts-env cnstrs-env bn-fun-env bn-funs bn-eqs =  
  let  
    val bn-funs' = map (fn (bn, ty, -) =>  
      (raw-bind bn, SOME (replace-typ dts-env ty), NoSyn)) bn-funs  
  
    val bn-eqs' = map (fn (attr, trm) =>  
      ((attr, replace-term (cnstrs-env @ bn-fun-env) dts-env trm), [], [])) bn-eqs  
  in  
    (bn-funs', bn-eqs')  
end  
>
```

ML <

```
fun rawify-bclauses dts-env cnstrs-env bn-fun-env bclauses =  
  let  
    fun rawify-bnds bnds =  
      map (apfst (Option.map (replace-term (cnstrs-env @ bn-fun-env) dts-env)))  
    bnds  
  
    fun rawify-bclause (BC (mode, bnds, bdys)) = BC (mode, rawify-bnds bnds, bdys)  
  in  
    (map o map o map) rawify-bclause bclauses  
end  
>
```

ML <

(* definition of the raw datatype *)

```
fun define-raw-dts dts cnstr-names cnstr-tys bn-funs bn-eqs bclauses lthy =  
  let  
    val thy = Local-Theory.exit-global lthy  
    val thy-name = Context.theory-base-name thy
```

```

val dt-names = map (fn ((s, -, -), -) => Binding.name-of s) dts
val dt-full-names = map (Long-Name.qualify thy-name) dt-names
val dt-full-names' = add-rows dt-full-names
val dts-env = dt-full-names ~~ dt-full-names'

val cnstr-full-names = map (Long-Name.qualify thy-name) cnstr-names
val cnstr-full-names' = map (fn (x, y) => Long-Name.qualify thy-name
  (Long-Name.qualify (add-row x) (add-row y))) cnstr-tys
val cnstrs-env = cnstr-full-names ~~ cnstr-full-names'

val bn-fun-strings = get-bn-fun-strings bn-funs
val bn-fun-strings' = add-rows bn-fun-strings
val bn-fun-env = bn-fun-strings ~~ bn-fun-strings'
val bn-fun-full-env = map (apply2 (Long-Name.qualify thy-name))
  (bn-fun-strings ~~ bn-fun-strings')

val raw-dts = rawify-dts dts dts-env
val (raw-bn-funs, raw-bn-egs) = rawify-bn-funs dts-env cnstrs-env bn-fun-env
bn-funs bn-egs
val raw-bclauses = rawify-bclauses dts-env cnstrs-env bn-fun-full-env bclauses

val (raw-full-dt-names', thy1) =
  BNF-LFP-Compat.add-datatype [BNF-LFP-Compat.Kill-Type-Args] raw-dts thy

val lthy1 = Named-Target.theory-init thy1

val dtinfos = map (Old-Datatype-Data.the-info (Proof-Context.theory-of lthy1))
raw-full-dt-names'
val raw-fp-sugars = map (the o BNF-FP-Def-Sugar.fp-sugar-of lthy1) raw-full-dt-names'
val {descr, ...} = hd dtinfos

val raw-ty-args = hd (Old-Datatype-Aux.get-rec-types descr)
  |> snd o dest-Type
  |> map dest-TFree
val raw-schematic-ty-args = (snd o dest-Type o #T o hd) raw-fp-sugars
val typ-subst = raw-schematic-ty-args ~~ map TFree raw-ty-args
val freezeT = Term.typ-subst-atomic typ-subst
val freeze = Term.subst-atomic-types typ-subst
val raw-tys = map (freezeT o #T) raw-fp-sugars

val raw-cns-info = all-dtyp-constrs-types descr
val raw-all-cns = map (map freeze o #ctrs o #ctr-sugar o #fp-ctr-sugar) raw-fp-sugars

val raw-inject-thms = flat (map #inject dtinfos)
val raw-distinct-thms = flat (map #distinct dtinfos)
val raw-induct-thm = (hd o #common-co-inducts o the o #fp-co-induct-sugar o
hd) raw-fp-sugars
val raw-induct-thms = map (the-single o #co-inducts o the o #fp-co-induct-sugar)
raw-fp-sugars

```

```

val raw-exhaust-thms = map #exhaust dtinfos
val raw-size-trms = map HOLogic.size-const raw-tys
val raw-size-thms = these (Option.map (#2 o #2)
  (BNF-LFP-Size.size-of lthy1 (hd raw-full-dt-names')))

val raw-result = RawDtInfo
  {raw-dt-names = raw-full-dt-names',
   raw-fp-sugars = raw-fp-sugars,
   raw-dts = raw-dts,
   raw-tys = raw-tys,
   raw-ty-args = raw-ty-args,
   raw-cns-info = raw-cns-info,
   raw-all-cns = raw-all-cns,
   raw-inject-thms = raw-inject-thms,
   raw-distinct-thms = raw-distinct-thms,
   raw-induct-thm = raw-induct-thm,
   raw-induct-thms = raw-induct-thms,
   raw-exhaust-thms = raw-exhaust-thms,
   raw-size-trms = raw-size-trms,
   raw-size-thms = raw-size-thms}
in
  (raw-bclauses, raw-bn-funs, raw-bn-egs, raw-result, lthy1)
end
>

```

ML <

```

fun nominal-datatype2 opt-thms-name dts bn-funs bn-egs bclauses lthy =
let
  val cnstr-names = get-cnstr-strs dts
  val cnstr-tys = get-typed-cnstrs dts

  val - = trace-msg (K Defining raw datatypes...)
  val (raw-bclauses, raw-bn-funs, raw-bn-egs, raw-dt-info, lthy0) =
    define-raw-dts dts cnstr-names cnstr-tys bn-funs bn-egs bclauses lthy

  val RawDtInfo
    {raw-dt-names,
     raw-tys,
     raw-ty-args,
     raw-fp-sugars,
     raw-all-cns,
     raw-inject-thms,
     raw-distinct-thms,
     raw-induct-thm,
     raw-induct-thms,
     raw-exhaust-thms,
     raw-size-trms,
     raw-size-thms, ...} = raw-dt-info

```

```

val - = trace-msg (K Defining raw permutations...)
val ((raw-perm-funs, raw-perm-simps, raw-perm-laws), lthy2a) = define-raw-perms
raw-dt-info lthy0

(* noting the raw permutations as eqvt theorems *)
val lthy3 = snd (Local-Theory.note ((Binding.empty, @{attributes [eqvt]}), raw-perm-simps)
lthy2a)

val - = trace-msg (K Defining raw fv- and bn-functions...)
val (raw-bns, raw-bn-defs, raw-bn-info, raw-bn-inducts, lthy3a) =
  define-raw-bns raw-dt-info raw-bn-funs raw-bn-egs lthy3

(* defining the permute-bn functions *)
val (raw-perm-bns, raw-perm-bn-simps, lthy3b) =
  define-raw-bn-perms raw-dt-info raw-bn-info lthy3a

val (raw-fvs, raw-fv-bns, raw-fv-defs, raw-fv-bns-induct, lthy3c) =
  define-raw-fvs raw-dt-info raw-bn-info raw-bclauses lthy3b

val - = trace-msg (K Defining alpha relations...)
val (alpha-result, lthy4) =
  define-raw-alpha raw-dt-info raw-bn-info raw-bclauses raw-fvs lthy3c

val - = trace-msg (K Proving distinct theorems...)
val alpha-distincts = raw-prove-alpha-distincts lthy4 alpha-result raw-dt-info

val - = trace-msg (K Proving eq-iff theorems...)
val alpha-eq-iff = raw-prove-alpha-eq-iff lthy4 alpha-result raw-dt-info

val - = trace-msg (K Proving equivariance of bns, fvs, size and alpha...)
val raw-bn-eqvt =
  raw-prove-eqvt raw-bns raw-bn-inducts (raw-bn-defs @ raw-perm-simps) lthy4

(* noting the raw-bn-eqvt lemmas in a temporary theory *)
val lthy-tmp =
  lthy4
  |> Local-Theory.begin-nested
  |> snd
  |> Local-Theory.note ((Binding.empty, @{attributes [eqvt]}), raw-bn-eqvt)
  |> snd
  |> Local-Theory.end-nested

val raw-fv-eqvt =
  raw-prove-eqvt (raw-fvs @ raw-fv-bns) raw-fv-bns-induct (raw-fv-defs @ raw-perm-simps)
  lthy-tmp

val raw-size-eqvt =
  let

```

```

val RawDtInfo {raw-size-trms, raw-size-thms, raw-induct-thms, ...} = raw-dt-info
in
raw-prove-eqvt raw-size-trms raw-induct-thms (raw-size-thms @ raw-perm-simps)
  lthy-tmp
  |> map (rewrite-rule lthy-tmp
    @{thms permute-nat-def[THEN eq-reflection]})
  |> map (fn thm => thm RS @{thm sym})
end

val lthy5 = snd (Local-Theory.note ((Binding.empty, @{attributes [eqvt]}), raw-fv-eqvt)
lthy-tmp)

val alpha-eqvt =
  let
    val AlphaResult {alpha-trms, alpha-bn-trms, alpha-raw-induct, alpha-intros,
...} = alpha-result
  in
    Nominal-Eqvt.raw-equivariance lthy5 (alpha-trms @ alpha-bn-trms) alpha-raw-induct
alpha-intros
  end

val alpha-eqvt-norm = map (Nominal-ThmDecls.eqvt-transform lthy5) alpha-eqvt

val - = trace-msg (K Proving equivalence of alpha...)
val alpha-refl-thms = raw-prove-refl lthy5 alpha-result raw-induct-thm
val alpha-sym-thms = raw-prove-sym lthy5 alpha-result alpha-eqvt-norm
val alpha-trans-thms =
  raw-prove-trans lthy5 alpha-result (raw-distinct-thms @ raw-inject-thms) al-
pha-eqvt-norm

val (alpha-equivp-thms, alpha-bn-equivp-thms) =
  raw-prove-equivp lthy5 alpha-result alpha-refl-thms alpha-sym-thms alpha-trans-thms

val - = trace-msg (K Proving alpha implies bn...)
val alpha-bn-imp-thms = raw-prove-bn-imp lthy5 alpha-result

val - = trace-msg (K Proving respectfulness...)
val raw-funs-rsp-aux =
  raw-fv-bn-rsp-aux lthy5 alpha-result raw-fvs raw-bns raw-fv-bns (raw-bn-defs @
raw-fv-defs)

val raw-funs-rsp = map (Drule.eta-contraction-rule o mk-funs-rsp lthy5) raw-funs-rsp-aux

fun match-const cst th =
  (fst o dest-Const o snd o dest-comb o HOLogic.dest-Trueprop o Thm.prop-of)
th =
  fst (dest-Const cst);
fun find-matching-rsp cst =
  hd (filter (fn th => match-const cst th) raw-funs-rsp);

```

```

val raw-fv-rsp = map find-matching-rsp raw-fvs;
val raw-bn-rsp = map find-matching-rsp raw-bns;
val raw-fv-bn-rsp = map find-matching-rsp raw-fv-bns;

val raw-size-rsp =
  raw-size-rsp-aux lthy5 alpha-result (raw-size-thms @ raw-size-eqvt)
  |> map (mk-funs-rsp lthy5)

val raw-constrs-rsp =
  raw-constrs-rsp lthy5 alpha-result raw-all-cns (alpha-bn-imp-thms @ raw-funs-rsp-aux)

val alpha-permute-rsp = map (mk-alpha-permute-rsp lthy5) alpha-eqvt

val alpha-bn-rsp =
  raw-alpha-bn-rsp alpha-result alpha-bn-equivp-thms alpha-bn-imp-thms

val raw-perm-bn-rsp = raw-perm-bn-rsp lthy5 alpha-result raw-perm-bns raw-perm-bn-simps

val - = trace-msg (K Defining the quotient types...)
val qty-descr = map (fn ((bind, vs, mx), -) => (map fst vs, bind, mx)) dts

val (qty-infos, lthy7) =
  let
    val AlphaResult {alpha-trms, alpha-tys, ...} = alpha-result
  in
    define-qtypes qty-descr alpha-tys alpha-trms alpha-equivp-thms lthy5
  end

val qtys = map #qtyp qty-infos
val qty-full-names = map (fst o dest-Type) qtys
val qty-names = map Long-Name.base-name qty-full-names

val - = trace-msg (K Defining the quotient constants...)
val qconstrs-descrs =
  (map2 o map2) (fn (b, -, mx) => fn (t, th) => (Variable.check-name b, t, mx,
th))
  (get-cnstrs dts) (map (op ~~~) (raw-all-cns ~~~ raw-constrs-rsp))

val qbns-descr =
  map2 (fn (b, -, mx) => fn (t, th) => (Variable.check-name b, t, mx, th))
bn-funs (raw-bns ~~~ raw-bn-rsp)

val qfvs-descr =
  map2 (fn n => fn (t, th) => (fv- ^ n, t, NoSyn, th)) qty-names (raw-fvs ~~~
raw-fv-rsp)

val qfv-bns-descr =
  map2 (fn (b, -, -) => fn (t, th) => (fv- ^ Variable.check-name b, t, NoSyn,
th))

```

```

    bn-funs (raw-fv-bns ~~~ raw-fv-bn-rsp)

val qalpha-bns-descr =
  let
    val AlphaResult {alpha-bn-trms, ...} = alpha-result
  in
    map2 (fn (b, -, -) => fn (t, th) => (alpha- ^ Variable.check-name b, t, NoSyn,
th))
      bn-funs (alpha-bn-trms ~~~ alpha-bn-rsp)
  end

val qperm-descr =
  map2 (fn n => fn (t, th) => (permute- ^ n, Type.legacy-freeze t, NoSyn, th))
    qty-names (raw-perm-funs ~~~ (take (length raw-perm-funs) alpha-permute-rsp))

val qsize-descr =
  map2 (fn n => fn (t, th) => (size- ^ n, t, NoSyn, th)) qty-names
    (raw-size-trms ~~~ (take (length raw-size-trms) raw-size-rsp))

val qperm-bn-descr =
  map2 (fn (b, -, -) => fn (t, th) => (permute- ^ Variable.check-name b, t,
NoSyn, th))
    bn-funs (raw-perm-bns ~~~ raw-perm-bn-rsp)

val ((((((qconstrs-infos, qbns-info), qfvs-info), qfv-bns-info), qalpha-bns-info), qperm-bns-info),
lthy8) =
  lthy7
  |> fold-map (define-qconstrs qtys) qconstrs-descrs
  ||>> define-qconstrs qtys qbns-descr
  ||>> define-qconstrs qtys qfvs-descr
  ||>> define-qconstrs qtys qfv-bns-descr
  ||>> define-qconstrs qtys qalpha-bns-descr
  ||>> define-qconstrs qtys qperm-bn-descr

val lthy9 =
  define-qperms qtys qty-full-names raw-ty-args qperm-descr raw-perm-laws lthy8

val lthy9a =
  define-qsizes qtys qty-full-names raw-ty-args qsize-descr lthy9

val qtrms = (map o map) #qconst qconstrs-infos
val qbns = map #qconst qbns-info
val qfvs = map #qconst qfvs-info
val qfv-bns = map #qconst qfv-bns-info
val qalpha-bns = map #qconst qalpha-bns-info
val qperm-bns = map #qconst qperm-bns-info

val - = trace-msg (K Lifting of theorems...)
val eq-iff-simps = @{thms alphas permute-prod.simps prod-fv.simps prod-alpha-def

```



```

rel-prod-sel
  prod.case}

val ([ qdistincts, qeq-iffs, qfv-defs, qbn-defs, qperm-simps, qfv-qbn-equts,
      qbn-inducts, qsize-eqvt, [qinduct], qexhausts, qsize-simps, qperm-bn-simps,
      qalpha-refl-thms, qalpha-sym-thms, qalpha-trans-thms ], lthyB) =
  lthy9a
|>>> lift-thms qtys [] alpha-distincts
|>>> lift-thms qtys eq-iff-simps alpha-eq-iff
|>>> lift-thms qtys [] raw-fv-defs
|>>> lift-thms qtys [] raw-bn-defs
|>>> lift-thms qtys [] raw-perm-simps
|>>> lift-thms qtys [] (raw-fv-eqvt @ raw-bn-eqvt)
|>>> lift-thms qtys [] raw-bn-inducts
|>>> lift-thms qtys [] raw-size-eqvt
|>>> lift-thms qtys [] [raw-induct-thm]
|>>> lift-thms qtys [] raw-exhaust-thms
|>>> lift-thms qtys [] raw-size-thms
|>>> lift-thms qtys [] raw-perm-bn-simps
|>>> lift-thms qtys [] alpha-refl-thms
|>>> lift-thms qtys [] alpha-sym-thms
|>>> lift-thms qtys [] alpha-trans-thms

val qinducts = Project-Rule.projections lthyB qinduct

val - = trace-msg (K Proving supp lemmas and fs-instances...)
val qsupports-thms = prove-supports lthyB qperm-simps (flat qtrms)

(* finite supp lemmas *)
val qfsupp-thms = prove-fsupp lthyB qtys qinduct qsupports-thms

(* fs instances *)
val lthyC = fs-instance qtys qty-full-names raw-ty-args qfsupp-thms lthyB

val - = trace-msg (K Proving equality between fv and supp...)
val qfv-supp-thms =
  prove-fv-supp qtys (flat qtrms) qfvs qfv-bns qalpha-bns qfv-defs qeq-iffs
  qperm-simps qfv-qbn-equts qinduct (flat raw-bclauses) lthyC
|> map Drule.eta-contraction-rule

(* postprocessing of eq and fv theorems *)
val qeq-iffs' = qeq-iffs
|> map (simplify (put-simpset HOL-basic-ss lthyC addsimps qfv-supp-thms))
|> map (simplify (put-simpset HOL-basic-ss lthyC
  addsimps @ {thms prod-fv-supp prod-alpha-eq Abs-eq-iff[symmetric]}))

(* filters the theorems that are of the form qfv = supp *)
val qfv-names = map (fst o dest-Const) qfvs
fun is-qfv-thm Const-<Trueprop for Const-<HOL.eq - for <Const (lhs, -)> -> =

```

```

    member (op =) qfv-names lhs
  | is-qfv-thm - = false

val qsupp-constrs = qfv-defs
  |> map (simplify (put-simpset HOL-basic-ss lthyC
    addsimps (filter (is-qfv-thm o Thm.prop-of) qfv-supp-thms)))

val transform-thm = @{lemma x = y  $\implies$  a  $\notin$  x  $\longleftrightarrow$  a  $\notin$  y by simp}
val transform-thms =
  [ @{lemma a  $\notin$  (S  $\cup$  T)  $\longleftrightarrow$  a  $\notin$  S  $\wedge$  a  $\notin$  T by simp},
    @{lemma a  $\notin$  (S - T)  $\longleftrightarrow$  a  $\notin$  S  $\vee$  a  $\in$  T by simp},
    @{lemma (lhs = (a  $\notin$  { }))  $\longleftrightarrow$  lhs by simp},
    @{thm fresh-def[symmetric]}]

val qfresh-constrs = qsupp-constrs
  |> map (fn thm => thm RS transform-thm)
  |> map (simplify (put-simpset HOL-basic-ss lthyC addsimps transform-thms))

(* proving that the qbn result is finite *)
val qbn-finite-thms = prove-bns-finite qtys qbns qinduct qbn-defs lthyC

(* proving that perm-bns preserve alpha *)
val qperm-bn-alpha-thms =
  prove-perm-bn-alpha-thms qtys qperm-bns qalpha-bns qinduct qperm-bn-simps
  qeq-iffs'
  qalpha-refl-thms lthyC

(* proving the relationship of bn and permute-bn *)
val qpermute-bn-thms =
  prove-permute-bn-thms qtys qbns qperm-bns qinduct qperm-bn-simps qbn-defs
  qfv-qbn-eqvs lthyC

val - = trace-msg (K Proving strong exhaust lemmas...)
val qstrong-exhaust-thms = prove-strong-exhausts lthyC qexhausts bclauses qbn-finite-thms
  qeq-iffs'
  qfv-qbn-eqvs qpermute-bn-thms qperm-bn-alpha-thms

val - = trace-msg (K Proving strong induct lemmas...)
val qstrong-induct-thms = prove-strong-induct lthyC qinduct qstrong-exhaust-thms
  qsize-simps bclauses

(* noting the theorems *)

(* generating the prefix for the theorem names *)
val thms-name =
  the-default (Binding.name (space-implode - qty-names)) opt-thms-name
fun thms-suffix s = Binding.qualify-name true thms-name s
val case-names-attr = Attrib.internal here (K (Rule-Cases.case-names cnstr-names))

```

val infos = mk-infos qty-full-names qeq-iffs' qdistincts qstrong-exhaust-thms qstrong-induct-thms

```

val (-, lthy9') = lthyC
  |> Local-Theory.declaration {syntax = false, pervasive = false, pos = here}
(K (fold register-info infos))
  |> Local-Theory.note ((thms-suffix distinct, @{attributes [induct-simp, simp]}) ,
qdistincts)
  ||>> Local-Theory.note ((thms-suffix eq-iff, @{attributes [induct-simp, simp]}) ,
qeq-iffs')
  ||>> Local-Theory.note ((thms-suffix fv-defs, []), qfv-defs)
  ||>> Local-Theory.note ((thms-suffix bn-defs, []), qbn-defs)
  ||>> Local-Theory.note ((thms-suffix bn-inducts, []), qbn-inducts)
  ||>> Local-Theory.note ((thms-suffix perm-simps, @{attributes [eqvt, simp]}) ,
qperm-simps)
  ||>> Local-Theory.note ((thms-suffix fv-bn-eqvt, @{attributes [eqvt]}) , qfv-qbn-eqvt)
  ||>> Local-Theory.note ((thms-suffix size, @{attributes [simp]}) , qsize-simps)
  ||>> Local-Theory.note ((thms-suffix size-eqvt, []), qsize-eqvt)
  ||>> Local-Theory.note ((thms-suffix induct, [case-names-attr]), [qinduct])
  ||>> Local-Theory.note ((thms-suffix inducts, [case-names-attr]), [qinducts])
  ||>> Local-Theory.note ((thms-suffix exhaust, [case-names-attr]), [qexhausts])
  ||>> Local-Theory.note ((thms-suffix strong-exhaust, [case-names-attr]), [qstrong-exhaust-thms])
  ||>> Local-Theory.note ((thms-suffix strong-induct, [case-names-attr]), [qstrong-induct-thms])
  ||>> Local-Theory.note ((thms-suffix supports, []), [qsupports-thms])
  ||>> Local-Theory.note ((thms-suffix fsupp, []), [qfsupp-thms])
  ||>> Local-Theory.note ((thms-suffix supp, []), [qsupp-constrs])
  ||>> Local-Theory.note ((thms-suffix fresh, @{attributes [simp]}) , [qfresh-constrs])
  ||>> Local-Theory.note ((thms-suffix perm-bn-simps, []), [qperm-bn-simps])
  ||>> Local-Theory.note ((thms-suffix bn-finite, []), [qbn-finite-thms])
  ||>> Local-Theory.note ((thms-suffix perm-bn-alpha, []), [qperm-bn-alpha-thms])
  ||>> Local-Theory.note ((thms-suffix permute-bn, []), [qpermute-bn-thms])
  ||>> Local-Theory.note ((thms-suffix alpha-refl, []), [qalpha-refl-thms])
  ||>> Local-Theory.note ((thms-suffix alpha-sym, []), [qalpha-sym-thms])
  ||>> Local-Theory.note ((thms-suffix alpha-trans, []), [qalpha-trans-thms])

in
  lthy9'
end

```

35 Preparing and parsing of the specification

ML <

(* adds the default sort @{sort fs} to nominal specifications *)

fun *augment-sort thy S* = *Sign.inter-sort thy* (@{sort fs}, *S*)

```

fun augment-sort-typ thy =
  map-type-tfree (fn (s, S) => TFree (s, augment-sort thy S))

```

```

ML <
(* generates the parsed datatypes and declares the constructors *)

fun prepare-dts dt-strs thy =
let
  fun prep-spec ((tname, tvs, mx), constra) =
    ((tname, tvs, mx), constra |> map (fn (c, atys, mx', -) => (c, map snd atys,
mx')))

  val (dts, spec-ctxt) =
    Old-Datatype.read-specs (map prep-spec dt-strs) thy

  fun augment ((tname, tvs, mx), constra) =
    ((tname, map (apsnd (augment-sort thy)) tvs, mx),
    constra |> map (fn (c, tys, mx') => (c, map (augment-sort-typ thy) tys,
mx')))

  val dts' = map augment dts

  fun mk-constr-trms ((tname, tvs, -), constra) =
    let
      val ty = Type (Sign.full-name thy tname, map TFree tvs)
    in
      map (fn (c, tys, mx) => (c, (tys ----> ty), mx)) constra
    end

  val constr-trms = flat (map mk-constr-trms dts')

  (* FIXME: local version *)
  (* val (-, spec-ctxt') = Proof-Context.add-fixes constr-trms spec-ctxt *)

  val thy' = Sign.add-consts constr-trms (Proof-Context.theory-of spec-ctxt)
in
  (dts', thy')
end
>

```

```

ML <
(* parsing the binding function specifications and *)
(* declaring the function constants *)
fun prepare-bn-funs bn-fun-strs bn-eq-strs thy =
let
  val lthy = Named-Target.theory-init thy

  val ((bn-funs, bn-eqs), lthy') =
    Specification.read-multi-specs bn-fun-strs bn-eq-strs lthy

  fun prep-bn-fun ((bn, T), mx) = (bn, T, mx)

```

```

val bn-funs' = map prep-bn-fun bn-funs

in
  (Local-Theory.exit-global lthy')
  |> Sign.add-consts bn-funs'
  |> pair (bn-funs', bn-egs)
end
>

```

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

```

ML <
fun indexify xs =
  let
    fun mapp - [] = []
      | mapp i (NONE :: xs) = mapp (i + 1) xs
      | mapp i (SOME x :: xs) = (x, i) :: mapp (i + 1) xs
  in
    mapp 0 xs
  end

fun index-lookup xs x =
  case AList.lookup (op =) xs x of
    SOME x => x
  | NONE => error (Cannot find ^x^ as argument annotation.);
>

```

```

ML <
fun prepare-bclauses dt-strs thy =
  let
    val annos-bclauses =
      get-cnstrs dt-strs
      |> (map o map) (fn (-, antys, -, bns) => (map fst antys, bns))

    fun prep-binder env bn-str =
      case (Syntax.read-term-global thy bn-str) of
        Free (x, -) => (NONE, index-lookup env x)
      | Const (a, T) $ Free (x, -) => (SOME (Const (a, T)), index-lookup env x)
      | - => error (The term ^bn-str^ is not allowed as binding function.)
  in
    fun prep-body env bn-str = index-lookup env bn-str

    fun prep-bclause env (mode, binders, bodies) =
      let
        val binders' = map (prep-binder env) binders
        val bodies' = map (prep-body env) bodies
      in
        BC (mode, binders', bodies')
      end
  end

```

```

fun prep-bclauses (annos, bclause-strs) =
  let
    val env = indexify annos (* for every label, associate the index *)
  in
    map (prep-bclause env) bclause-strs
  end
in
  ((map o map) prep-bclauses annos-bclauses, thy)
end
>

```

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

```

ML <
fun included i bcs =
  let
    fun incl (BC (-, bns, bds)) =
      member (op =) (map snd bns) i orelse member (op =) bds i
  in
    exists incl bcs
  end
>

```

```

ML <
fun complete dt-strs bclauses =
  let
    val args =
      get-cnstrs dt-strs
      |> (map o map) (fn (-, antys, -, -) => length antys)

    fun complt n bcs =
      let
        fun add bcs i = (if included i bcs then [] else [BC (Lst, [], [i])])
      in
        bcs @ (flat (map-range (add bcs) n))
      end
  in
    (map2 o map2) complt args bclauses
  end
>

```

```

ML <
fun nominal-datatype2-cmd (opt-thms-name, dt-strs, bn-fun-strs, bn-eq-strs) lthy =
  let
    (* this theory is used just for parsing *)
    val thy = Proof-Context.theory-of lthy

    val (((dts, (bn-funs, bn-eqs)), bclauses), -) =

```

```

    thy
    |> prepare-dts dt-strings
    ||>> prepare-bn-funs bn-fun-strings bn-eq-strings
    ||>> prepare-bclauses dt-strings

    val bclauses' = complete dt-strings bclauses
in
    nominal-datatype2 opt-thms-name dts bn-funs bn-eqs bclauses' lthy
end
}

ML {
(* nominal datatype parser *)
local
    fun triple1 ((x, y), z) = (x, y, z)
    fun triple2 ((x, y), z) = (y, x, z)
    fun tuple2 (((x, y), z), u) = (x, y, u, z)
    fun tuple3 ((x, y), (z, u)) = (x, y, z, u)
in

    val opt-name = Scan.option (Parse.binding --| Args.colon)

    val anno-typ = Scan.option (Parse.name --| @ {keyword ::}) -- Parse.typ

    val bind-mode = @ {keyword binds} |--
        Scan.optional (Args.parens
            (Args.$$$ list >> K Lst || (Args.$$$ set -- Args.$$$ +) >> K Res || Args.$$$
            set >> K Set)) Lst

    val bind-clauses =
        Parse.enum , (bind-mode -- Scan.repeat1 Parse.term -- (@ {keyword in} |--
        Scan.repeat1 Parse.name) >> triple1)

    val cnstr-parser =
        Parse.binding -- Scan.repeat anno-typ -- bind-clauses -- Parse.opt-mixfix
        >> tuple2

    (* datatype parser *)
    val dt-parser =
        (Parse.type-args-constrained -- Parse.binding -- Parse.opt-mixfix >> triple2)
        --
        (@ {keyword =} |-- Parse.enum1 | cnstr-parser)

    (* binding function parser *)
    val bnfun-parser =
        Scan.optional (@ {keyword binder} |-- Parse.Spec.specification) ([], [])

    (* main parser *)
    val main-parser =

```

```

    opt-name -- Parse.and-list1 dt-parser -- bnfun-parser >> tuple3
end

(* Command Keyword *)
val - = Outer-Syntax.local-theory @ {command-keyword nominal-datatype}
    declaration of nominal datatypes
    (main-parser >> nominal-datatype2-cmd)
>

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  $\neg a \# \text{nat-of}$ 
unfolding fresh-def supp-def
proof (clarsimp)
  assume finite {b. (a  $\rightleftharpoons$  b)  $\cdot$  nat-of  $\neq$  nat-of}
  hence finite ({a}  $\cup$  {b. (a  $\rightleftharpoons$  b)  $\cdot$  nat-of  $\neq$  nat-of})
    by simp
  then obtain b where
    b1: b  $\neq$  a and
    b2: sort-of b = sort-of a and
    b3: (a  $\rightleftharpoons$  b)  $\cdot$  nat-of = nat-of
    by (rule obtain-atom) auto
  have nat-of a = (a  $\rightleftharpoons$  b)  $\cdot$  (nat-of a) by (simp add: permute-nat-def)
  also have ... = ((a  $\rightleftharpoons$  b)  $\cdot$  nat-of) ((a  $\rightleftharpoons$  b)  $\cdot$  a) by (simp add: permute-fun-app-eq)
  also have ... = nat-of ((a  $\rightleftharpoons$  b)  $\cdot$  a) using b3 by simp
  also have ... = nat-of b using b2 by simp
  finally have nat-of a = nat-of b by simp
  with b2 have a = b by (simp add: atom-components-eq-iff)
  with b1 show False by simp
qed

```

```

lemma supp-nat-of:
  shows supp nat-of = UNIV
  using not-fresh-nat-of [unfolded fresh-def] by auto

```

37 Manual instantiation of class *at*.

```

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

```


by (*rule exists-eq-simple-sort*)

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

apply (*rule at-class*)
apply (*rule type-definition-name*)
apply (*rule atom-name-def*)
apply (*rule permute-name-def*)
done

end

lemma *sort-of-atom-name*:

shows $\text{sort-of } (\text{atom } (a::\text{name})) = \text{Sort } \text{"name"} []$
by (*simp add: atom-name-def Rep-name[simplified]*)

Custom syntax for concrete atoms of type *at*

term *a::name*

38 Automatic instantiation of class *at*.

atom-decl *name2*

lemma

$\text{sort-of } (\text{atom } (a::\text{name2})) \neq \text{sort-of } (\text{atom } (b::\text{name}))$
by (*simp add: sort-of-atom-name*)

example swappings

lemma

fixes *a b::atom*
assumes $\text{sort-of } a = \text{sort-of } b$
shows $(a \rightleftharpoons b) \cdot (a, b) = (b, a)$

using *assms*

by *simp*

lemma

fixes *a b::name2*
shows $(a \leftrightarrow b) \cdot (a, b) = (b, a)$
by *simp*

39 An example for multiple-sort atoms

```

datatype ty =
  TVar string
| Fun ty ty (- → -)

primrec
  sort-of-ty::ty ⇒ 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 ⟷ x = y
apply(induct x arbitrary: y)
apply(case-tac [!] y)
apply(simp-all)
done

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

typedef var = {a. sort-of a ∈ range sort-of-ty}
  by (rule-tac x=Atom (sort-of-ty x) y in exI, simp)

instantiation var :: at-base
begin

definition
  p · a = Abs-var (p · Rep-var a)

definition
  atom a = Rep-var a

instance
apply (rule at-base-class)
apply (rule type-definition-var)
apply (rule atom-var-def)
apply (rule permute-var-def)
done

end

  Constructor for variables.

definition
  Var :: nat ⇒ ty ⇒ var
where
  Var x t = Abs-var (Atom (sort-of-ty t) x)

lemma Var-eq-iff [simp]:
  shows Var x s = Var y t ⟷ x = y ∧ s = t

```

unfolding *Var-def*
by (*auto simp add: Abs-var-inject sort-of-ty-eq-iff*)

lemma *sort-of-atom-var* [*simp*]:
sort-of (atom (Var n ty)) = sort-of-ty ty
unfolding *atom-var-def Var-def*
by (*simp add: Abs-var-inverse*)

lemma
assumes $\alpha \neq \beta$
shows $(Var\ x\ \alpha \leftrightarrow Var\ y\ \alpha) \cdot (Var\ x\ \alpha, Var\ x\ \beta) = (Var\ y\ \alpha, Var\ x\ \beta)$
using *assms* **by** *simp*

Projecting out the type component of a variable.

definition
ty-of :: *var* \Rightarrow *ty*
where
ty-of *x* = *inv sort-of-ty (sort-of (atom x))*

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

lemma *ty-of-Var* [*simp*]:
shows *ty-of (Var x t) = t*
unfolding *ty-of-def*
unfolding *sort-of-atom-var*
apply (*rule inv-f-f*)
apply (*simp add: inj-on-def sort-of-ty-eq-iff*)
done

lemma *ty-of-permute* [*simp*]:
shows *ty-of (p · x) = ty-of x*
unfolding *ty-of-def*
unfolding *atom-eqvt* [*symmetric*]
by (*simp only: sort-of-permute*)

40 Tests with subtyping and automatic coercions

declare [[*coercion-enabled*]]

atom-decl *var1*
atom-decl *var2*

declare [[*coercion atom::var1* \Rightarrow *atom*]]

declare [[*coercion atom::var2* \Rightarrow *atom*]]

lemma
fixes *a::var1* **and** *b::var2*
shows $a \# t \wedge b \# t$
oops

```

lemma
  fixes  $as::var1\ set$ 
  shows  $atom\ 'a\ \#*\ t$ 

oops

end

theory Eqvt
imports Nominal2-Base
begin

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

lemma
  fixes  $B::'a::pt$ 
  shows  $p \cdot (B = C)$ 
  apply(perm-simp)
  oops

lemma
  fixes  $B::bool$ 
  shows  $p \cdot (B = C)$ 
  apply(perm-simp)
  oops

lemma
  fixes  $B::bool$ 
  shows  $p \cdot (A \longrightarrow B = C)$ 
  apply(perm-simp)
  oops

lemma
  shows  $p \cdot (\lambda(x::'a::pt). A \longrightarrow (B::'a \Rightarrow bool)\ x = C) = foo$ 
  apply(perm-simp)
  oops

lemma
  shows  $p \cdot (\lambda B::bool. A \longrightarrow (B = C)) = foo$ 
  apply(perm-simp)
  oops

lemma

```

```

  shows  $p \cdot (\lambda x y. \exists z. x = z \wedge x = y \longrightarrow z \neq x) = foo$ 
apply (perm-simp)
oops

```

```

lemma
  shows  $p \cdot (\lambda f x. f (g (f x))) = foo$ 
apply (perm-simp)
oops

```

```

lemma
  fixes  $p q::perm$ 
  and  $x::'a::pt$ 
  shows  $p \cdot (q \cdot x) = foo$ 
apply(perm-simp)
oops

```

```

lemma
  fixes  $p q r::perm$ 
  and  $x::'a::pt$ 
  shows  $p \cdot (q \cdot r \cdot x) = foo$ 
apply(perm-simp)
oops

```

```

lemma
  fixes  $p r::perm$ 
  shows  $p \cdot (\lambda q::perm. q \cdot (r \cdot x)) = foo$ 
apply (perm-simp)
oops

```

```

lemma
  fixes  $C D::bool$ 
  shows  $B (p \cdot (C = D))$ 
apply(perm-simp)
oops

```

```

declare [[trace-eqvt = false]]

```

there is no raw *eqvt*-rule for *The*

```

lemma  $p \cdot (THE x. P x) = foo$ 
apply(perm-strict-simp exclude: The)
apply(perm-simp exclude: The)
oops

```

```

lemma
  fixes  $P :: (('b \Rightarrow bool) \Rightarrow ('b::pt)) \Rightarrow ('a::pt)$ 
  shows  $p \cdot (P The) = foo$ 
apply(perm-simp exclude: The)
oops

```

```

lemma
  fixes  $P :: ('a::pt) \Rightarrow ('b::pt) \Rightarrow bool$ 
  shows  $p \cdot (\lambda(a, b). P a b) = (\lambda(a, b). (p \cdot P) a b)$ 
apply(perm-simp)
oops

thm eqvts
thm eqvts-raw

ML  $\langle \text{Nominal-ThmDecls.is-eqvt } @\{\text{context}\} @\{\text{term supp}\} \rangle$ 

end

```