Computational p-Adics

Jeremy Sylvestre University of Alberta (Augustana Campus)

July 7, 2025

Abstract

We develop a framework for reasoning computationally about p-adic fields via formal Laurent series with integer coefficients. We define a ring-class quotient type consisting of equivalence classes of prime-indexed sequences of formal Laurent series, in which every p-adic field is contained as a subring. Via a further quotient of the ring of p-adic integers (that is, the elements of depth at least 0) by the prime ideal (that is, the subring of elements of depth at least 1), we define a ring-class type in which every prime-order finite field is contained as a subring. Finally, some topological properties are developed using depth as a proxy for a metric, Hensel's Lemma is proved, and the compactness of local rings of integers is established.

Contents

theory *Distinguished-Quotients*

imports *Main*

begin

1 Distinguished Algebraic Quotients

We define quotient groups and quotient rings modulo distinguished normal subgroups and ideals, respectively, leading to the construction of a field as a commutative ring with 1 modulo a maximal ideal.

class *distinguished-subset* =
  fixes *distinguished-subset* :: *'a set* ($\mathcal{N}$)

hide-const *distinguished-subset*

1.1 Quotient groups

1.1.1 Class definitions

Here we set up subclasses of *group-add* with a distinguished subgroup.

class *group-add-with-distinguished-subgroup* = *group-add* + *distinguished-subset* +
  assumes *distset-nonempty*: $\mathcal{N} \neq \{\}$
    and *distset-diff-closed*:
    $g \in \mathcal{N} \Longrightarrow h \in \mathcal{N} \Longrightarrow g - h \in \mathcal{N}$
begin

lemma *distset-zero-closed* : $0 \in \mathcal{N}$
proof−
  from *distset-nonempty* **obtain** g::*'a* **where** $g \in \mathcal{N}$ **by** *fast*
  hence $g - g \in \mathcal{N}$ **using** *distset-diff-closed* **by** *fast*
  thus *?thesis* **by** *simp*
qed

lemma *distset-uminus-closed*: $a \in \mathcal{N} \Longrightarrow -a \in \mathcal{N}$
  using *distset-zero-closed distset-diff-closed* **by** *force*

lemma *distset-add-closed*:
  $a \in \mathcal{N} \Longrightarrow b \in \mathcal{N} \Longrightarrow a + b \in \mathcal{N}$
  using *distset-uminus-closed*[*of b*] *distset-diff-closed*[*of a* $-b$] **by** *simp*

lemma *distset-uminus-plus-closed*:

$a \in \mathcal{N} \Longrightarrow b \in \mathcal{N} \Longrightarrow -a + b \in \mathcal{N}$
using *distset-uminus-closed distset-add-closed* **by** *simp*

lemma *distset-lcoset-closed1*:
$a \in \mathcal{N} \Longrightarrow -a + b \in \mathcal{N} \Longrightarrow b \in \mathcal{N}$
using *add.assoc*[*of* a $-a$ b] *distset-add-closed* **by** *fastforce*

lemma *distset-lcoset-closed2*:
$b \in \mathcal{N} \Longrightarrow -a + b \in \mathcal{N} \Longrightarrow a \in \mathcal{N}$
proof−
assume *ab*: $b \in \mathcal{N}$ $-a + b \in \mathcal{N}$
from *ab*(*2*) **obtain** z **where** z: $z \in \mathcal{N}$ $-a + b = z$ **by** *fast*
from z(*2*) **have** $a = -(z - b)$ **using** *add.assoc*[*of* $-a$ b $-b$] **by** *simp*
with *ab*(*1*) z(*1*) **show** $a \in \mathcal{N}$ **using** *distset-diff-closed* **by** *simp*
qed

lemma *reflp-lcosetrel*: $-\ x + x \in \mathcal{N}$
using *distset-zero-closed* **by** *simp*

lemma *symp-lcosetrel*: $-\ a + b \in \mathcal{N} \Longrightarrow -\ b + a \in \mathcal{N}$
using *distset-uminus-closed* **by** (*fastforce simp add*: *minus-add*)

lemma *transp-lcosetrel*:
$-x + z \in \mathcal{N}$ **if** $-x + y \in \mathcal{N}$ **and** $-y + z \in \mathcal{N}$
proof−
from *that* **obtain** a a' :: $'a$
where $a \in \mathcal{N}$ $-x + y = a$ $a' \in \mathcal{N}$ $-y + z = a'$ **by** *fast*
thus *?thesis* **using** *add.assoc*[*of* a $-y$ z] *distset-add-closed* **by** *fastforce*
qed

end

class *group-add-with-distinguished-normal-subgroup* = *group-add-with-distinguished-subgroup*
+
assumes *distset-normal*: $((+)\ g)$ ‘ $\mathcal{N} = (\lambda x.\ x + g)$ ‘ $\mathcal{N}$

class *ab-group-add-with-distinguished-subgroup* =
ab-group-add + *group-add-with-distinguished-subgroup*
begin

subclass *group-add-with-distinguished-normal-subgroup*
proof (*unfold-locales*, *rule distset-nonempty*, *rule distset-diff-closed*)
show $\bigwedge g.\ ((+)\ g)$ ‘ $\mathcal{N} = (\lambda x.\ x + g)$ ‘ $\mathcal{N}$ **by** (*force simp add*: *add.commute*)
qed

end

1.1.2 The quotient group type

Here we define a quotient type relative to the left-coset relation, and instantiate it as a *group-add*.

quotient-type (**overloaded**) *'a coset-by-dist-sbgrp* =
  'a::group-add-with-distinguished-subgroup / $\lambda g\ h$ :: *'a*. $-g + h \in (\mathcal{N}$::*'a set*)
  morphisms *distinguished-quotient-coset-rep distinguished-quotient-coset*
proof (*rule equivpI*)
  show *reflp* ($\lambda g\ h$ :: *'a*. $-g + h \in \mathcal{N}$) **using** *reflp-lcosetrel* **by** (*fast intro*: *reflpI*)
  show *symp* ($\lambda g\ h$ :: *'a*. $-g + h \in \mathcal{N}$) **using** *symp-lcosetrel* **by** (*fast intro*: *sympI*)
  show *transp* ($\lambda g\ h$ :: *'a*. $-g + h \in \mathcal{N}$) **using** *transp-lcosetrel*
    by (*fast intro*: *transpI*)
qed

lemmas *distinguished-quotient-coset-rep-inverse* =
    Quotient-abs-rep[*OF Quotient-coset-by-dist-sbgrp*]
  and *coset-by-dist-sbgrp-eq-iff* =
    Quotient-rel-rep[*OF Quotient-coset-by-dist-sbgrp*]
  and *distinguished-quotient-coset-eqI* =
    Quotient-rel-abs[*OF Quotient-coset-by-dist-sbgrp*]
  and *eq-to-distinguished-quotient-cosetI* =
    Quotient-rel-abs2[*OF Quotient-coset-by-dist-sbgrp*]

lemma *coset-coset-by-dist-sbgrp-cases*
  [*case-names coset*, *cases type*: *coset-by-dist-sbgrp*]:
  ($\bigwedge y$. z = *distinguished-quotient-coset* $y \Longrightarrow P) \Longrightarrow P$
  by (*induct z*) *auto*

lemmas *distinguished-quotient-coset-eq-iff* = *coset-by-dist-sbgrp.abs-eq-iff*

lemma *distinguished-quotient-coset-rep-coset-equiv*:
  $-$ *distinguished-quotient-coset-rep* (*distinguished-quotient-coset* r) + $r \in \mathcal{N}$
  using *Quotient-rep-abs*[*OF Quotient-coset-by-dist-sbgrp*]
  by (*simp add*: *distset-zero-closed*)

instantiation *coset-by-dist-sbgrp* ::
  (*group-add-with-distinguished-normal-subgroup*) *group-add*
begin

lift-definition *zero-coset-by-dist-sbgrp* :: *'a coset-by-dist-sbgrp*
  is *0*::*'a* **.**

lemmas *zero-distinguished-quotient* [*simp*] = *zero-coset-by-dist-sbgrp.abs-eq*[*symmetric*]

lemma *zero-distinguished-quotient-eq-iffs*:
  (*distinguished-quotient-coset* x = *0*) = ($x \in \mathcal{N}$)
  (X = *0*) = (*distinguished-quotient-coset-rep* $X \in \mathcal{N}$)
proof−
  have *distinguished-quotient-coset* x = *0* $\Longrightarrow x \in \mathcal{N}$

proof *transfer*
  fix x::$'a$ **show** $- x + 0 \in \mathcal{N} \Longrightarrow x \in \mathcal{N}$
    using *distset-uminus-closed*[*of* $-x$] **by** *simp*
qed
moreover have $x \in \mathcal{N} \Longrightarrow$ *distinguished-quotient-coset* $x = 0$
  by *transfer* (*simp add*: *distset-uminus-closed*)
ultimately show (*distinguished-quotient-coset* $x = 0$) $=$ ($x \in \mathcal{N}$) **by** *fast*
have *distinguished-quotient-coset-rep* $0 \in \mathcal{N}$
  using *distinguished-quotient-coset-rep-coset-equiv*[*of* 0::$'a$] *distset-uminus-closed*
  by *fastforce*
moreover have *distinguished-quotient-coset-rep* $X \in \mathcal{N} \Longrightarrow X = 0$
  using *eq-to-distinguished-quotient-cosetI*[*of* X 0] **by** (*simp add*: *distset-uminus-closed*)
ultimately show $(X = 0) =$ (*distinguished-quotient-coset-rep* $X \in \mathcal{N}$) **by** *fast*
qed

lift-definition *plus-coset-by-dist-sbgrp* ::
  $'a$ *coset-by-dist-sbgrp* $\Rightarrow$ $'a$ *coset-by-dist-sbgrp* $\Rightarrow$
    $'a$ *coset-by-dist-sbgrp*
  is $\lambda x\ y$::$'a.\ x + y$
proof−
  fix $w\ x\ y\ z$ :: $'a$ **assume** *wx*: $-w+x \in \mathcal{N}$ **and** *yz*: $-y+z \in \mathcal{N}$
  from *this* **obtain** *a1 a2*
    where *a1*: $a1 \in \mathcal{N}$ $-w + x = a1$
    and *a2*: $a2 \in \mathcal{N}$ $-y + z = a2$
    by *fast*
  from *a1*(*1*) **obtain** *a3* **where** *a3*: $a3 \in \mathcal{N}$ $-y + a1 = a3 + -y$ **using** *distset-normal* **by** *fast*
  from *a1*(*2*) *a2*(*2*) *a3*(*2*) **have** $-(w + y) + (x + z) = a3 + a2$
    using *minus-add add.assoc*[*of* $-y + -w$ x z] *add.assoc*[*of* $-y$ $-w$ x] *add.assoc*[*of* $a3$ $-y$ z]
    by *simp*
  with *a2*(*1*) *a3*(*1*) **show** $-(w + y) + (x + z) \in \mathcal{N}$ **using** *distset-add-closed* **by** *simp*
qed

lemmas *plus-coset-by-dist-sbgrp* [*simp*] = *plus-coset-by-dist-sbgrp.abs-eq*

lift-definition *uminus-coset-by-dist-sbgrp* ::
  $'a$ *coset-by-dist-sbgrp* $\Rightarrow$ $'a$ *coset-by-dist-sbgrp*
  is λx::$'a.\ -x$
proof−
  fix $x\ y$ :: $'a$ **assume** *xy*: $-x + y \in \mathcal{N}$
  from *this* **obtain** a **where** a: $a \in \mathcal{N}$ $-x + y = a$ **by** *fast*
  from a(*1*) **obtain** a' **where** a': $a' \in \mathcal{N}$ $a + -y = -y + a'$ **using** *distset-normal* **by** *fast*
  from a(*2*) a'(*2*) **have** $- (-x) + -y = -(- y + a') + -y$
    using *add.assoc*[*of* $-x$ y $-y$] **by** *simp*
  hence $- (-x) + -y = -a'$ **by** (*simp add*: *minus-add*)
  with a'(*1*) **show** $- (-x) + -y \in \mathcal{N}$ **using** *distset-uminus-closed* **by** *simp*

qed

lemmas *uminus-coset-by-dist-sbgrp* [*simp*] =
uminus-coset-by-dist-sbgrp.abs-eq

lift-definition *minus-coset-by-dist-sbgrp* ::
$'a$ *coset-by-dist-sbgrp* $\Rightarrow$ $'a$ *coset-by-dist-sbgrp* $\Rightarrow$
$'a$ *coset-by-dist-sbgrp*
is $\lambda x\ y::'a.\ x - y$
proof−
fix $w\ x\ y\ z$ :: $'a$ **assume** *wx*: $-w + x \in \mathcal{N}$ **and** *yz*: $-y + z \in \mathcal{N}$
from *this* **obtain** *a1 a2*
where *a1*: $a1 \in \mathcal{N}$ $x = w + a1$
and *a2*: $a2 \in \mathcal{N}$ $z = y + a2$
by *simp*
from *a1*(*1*) **obtain** *a3* **where** *a3*: $a3 \in \mathcal{N}$ $y + a1 = a3 + y$ **using** *distset-normal* **by** *auto*
from *a2*(*1*) **obtain** *a4* **where** *a4*: $a4 \in \mathcal{N}$ $y + a2 = a4 + y$ **using** *distset-normal* **by** *auto*
from *a3*(*2*) *a4*(*2*) *a1*(*2*) *a2*(*2*) **have** $x - z = w - y + a3 + y - y - a4$
using *add.assoc*[*of* $w - y\ y\ a1$] *add.assoc*[*of* $w - y\ a3\ y$]
add.assoc[*of* $w - y + a3\ y\ -y$]
by (*simp add*: *algebra-simps*)
hence $x - z = (w - y) + (a3 - a4)$
using *add.assoc*[*of* $w - y + a3\ y\ -y$] *add.assoc*[*of* $w - y\ a3\ -a4$] **by** *simp*
hence $-(w - y) + (x - z) = a3 - a4$
using *add.assoc*[*of* $-(w - y)\ w - y\ a3 - a4$, *symmetric*]
left-minus[*of* $w - y$]
by *simp*
with *a3*(*1*) *a4*(*1*) **show** $-(w - y) + (x - z) \in \mathcal{N}$ **using** *distset-diff-closed* **by** *auto*
qed

lemmas *minus-coset-by-dist-sbgrp* [*simp*] = *minus-coset-by-dist-sbgrp.abs-eq*

instance proof
fix $x\ y\ z$ :: $'a$ *coset-by-dist-sbgrp*
show $x + y + z = x + (y + z)$ **by** *transfer* (*simp add*: *distset-zero-closed algebra-simps*)
show $0 + x = x$ **by** *transfer* (*simp add*: *distset-zero-closed*)
show $x + 0 = x$ **by** *transfer* (*simp add*: *distset-zero-closed*)
show $-x + x = 0$ **by** *transfer* (*simp add*: *distset-zero-closed*)
show $x + - y = x - y$ **by** *transfer* (*simp add*: *distset-zero-closed algebra-simps*)
qed

end

instance *coset-by-dist-sbgrp* ::
(*ab-group-add-with-distinguished-subgroup*) *ab-group-add*

proof
  show $\bigwedge x\ y :: 'a\ \textit{coset-by-dist-sbgrp}.\ x + y = y + x$
    by *transfer* (*simp add*: *distset-zero-closed*)
qed *auto*

1.2 Quotient rings

1.2.1 Class definitions

Now we set up subclasses of *ring* with a distinguished ideal.

class *ring-with-distinguished-ideal* = *ring* + *ab-group-add-with-distinguished-subgroup* +
  assumes *distset-leftideal* : $a \in \mathcal{N} \Longrightarrow r * a \in \mathcal{N}$
  and *distset-rightideal*: $a \in \mathcal{N} \Longrightarrow a * r \in \mathcal{N}$

class *ring-1-with-distinguished-ideal* = *ring-1* + *ring-with-distinguished-ideal* +
  assumes *distset-no1*: $1 \notin \mathcal{N}$
  — For unitary rings we do not allow the quotient to be trivial to ensure that *0* and *1* will be different.

1.2.2 The quotient ring type

We just reuse the *'a coset-by-dist-sbgrp* type to define a quotient ring, and instantiate it as a *ring*.

type-synonym *'a distinguished-quotient-ring* = *'a coset-by-dist-sbgrp*
— This is intended to be used with *ring-with-distinguished-ideal* or one of its subclasses.

instantiation *coset-by-dist-sbgrp* :: (*ring-with-distinguished-ideal*) *ring*
begin

lift-definition *times-coset-by-dist-sbgrp* ::
  'a distinguished-quotient-ring ⇒ *'a distinguished-quotient-ring* ⇒
    'a distinguished-quotient-ring
  is $\lambda x\ y{::}'a.\ x * y$
proof−
  fix *w x y z* :: *'a* **assume** *wx*: $-w + x \in \mathcal{N}$ **and** *yz*: $-y + z \in \mathcal{N}$
  from *this* **obtain** *a1 a2*
    where *a1*: $a1 \in \mathcal{N}$ $-w + x = a1$
    and *a2*: $a2 \in \mathcal{N}$ $-y + z = a2$
    by *fast*
  from *a1*(*2*) *a2*(*2*) **have** $-(w * y) + (x * z) = w * a2 + a1 * z$
    by (*simp add*: *algebra-simps*)
  with *a1*(*1*) *a2*(*1*) **show** $-(w * y) + (x * z) \in \mathcal{N}$
    using *distset-leftideal*[*of - w*] *distset-rightideal*[*of - z*] *distset-add-closed* **by** *auto*
qed

lemmas *times-coset-by-dist-sbgrp* [*simp*] = *times-coset-by-dist-sbgrp.abs-eq*

instance proof
  fix *x y z* :: *'a distinguished-quotient-ring*
  show $x * y * z = x * (y * z)$ **by** *transfer* (*simp add*: *distset-zero-closed algebra-simps*)
  show $(x + y) * z = x * z + y * z$ **by** *transfer* (*simp add*: *distset-zero-closed algebra-simps*)
  show $x * (y + z) = x * y + x * z$ **by** *transfer* (*simp add*: *distset-zero-closed algebra-simps*)
qed

end

instantiation *coset-by-dist-sbgrp* :: (*ring-1-with-distinguished-ideal*) *ring-1*
begin

lift-definition *one-coset-by-dist-sbgrp* :: *'a distinguished-quotient-ring*
  is *1*::*'a* **.**

lemmas *one-coset-by-dist-sbgrp* [*simp*] = *one-coset-by-dist-sbgrp.abs-eq*[*symmetric*]

instance proof
  show (*0*::*'a distinguished-quotient-ring*) $\neq$ (*1*::*'a distinguished-quotient-ring*)
  proof *transfer* **qed** (*simp add*: *distset-no1*)
next
  fix *x* :: *'a distinguished-quotient-ring*
  show $1 * x = x$ **by** *transfer* (*simp add*: *distset-zero-closed algebra-simps*)
  show $x * 1 = x$ **by** *transfer* (*simp add*: *distset-zero-closed algebra-simps*)
qed

end

1.3 Quotients of commutative rings

For a commutative ring, the quotient modulo a prime ideal is an integral domain, and the quotient modulo a maximal ideal is a field.

1.3.1 Class definitions

class *comm-ring-with-distinguished-ideal* = *comm-ring* + *ab-group-add-with-distinguished-subgroup* +
  assumes *distset-ideal* : $a \in \mathcal{N} \Longrightarrow r * a \in \mathcal{N}$
begin

subclass *ring-with-distinguished-ideal*
  using *distset-ideal* **by** *unfold-locales* (*auto simp add*: *mult.commute*)

definition *distideal-extension* :: $'a \Rightarrow 'a$ *set*
  where *distideal-extension* $a \equiv \{x.\ \exists\, r\ z.\ z \in \mathcal{N} \wedge x = r * a + z\}$

lemma *distideal-extension*: $\mathcal{N} \subseteq$ *distideal-extension a*
  unfolding *distideal-extension-def*
proof *clarify*
  fix *x* **assume** $x{\in}\mathcal{N}$
  hence $x{\in}\mathcal{N} \wedge x = 0{*}a + x$ **by** *simp*
  thus $\exists\, r\ z.\ z \in \mathcal{N} \wedge x = r{*}a + z$ **by** *fast*
qed

lemma *distideal-extension-diff-closed*:
  $r \in$ *distideal-extension a* $\Longrightarrow s \in$ *distideal-extension a* $\Longrightarrow$
    $r - s \in$ *distideal-extension a*
  unfolding *distideal-extension-def*
proof *clarify*
  fix $r\ r'\ z\ z'$
  assume $z \in \mathcal{N}\ z' \in \mathcal{N}$
  moreover have $r * a + z - (r' * a + z') = (r - r') * a + (z - z')$
    by (*simp add*: *algebra-simps*)
  ultimately show
    $\exists\, r''\ z''.\ z'' \in \mathcal{N} \wedge r * a + z - (r' * a + z') = r'' * a + z''$
    using *distset-diff-closed* **by** *fast*
qed

lemma *distideal-extension-ideal*:
  $x \in$ *distideal-extension a* $\Longrightarrow r * x \in$ *distideal-extension a*
  unfolding *distideal-extension-def*
proof *clarify*
  fix *s r z* **assume** $z \in \mathcal{N}$
  thus $\exists\, r'\ z'.\ z' \in \mathcal{N} \wedge s * (r * a + z) = r' * a + z'$
    using *distset-ideal* **by** (*fastforce simp add*: *algebra-simps*)
qed

end

class *comm-ring-1-with-distinguished-ideal* = *ring-1* + *comm-ring-with-distinguished-ideal* +
  assumes *distset-no1*:
    $1{\notin}\mathcal{N}$
  — Don't allow the quotient to be trivial.
begin

subclass *ring-1-with-distinguished-ideal* **using** *distset-no1* **by** *unfold-locales*

lemma *distideal-extension-by*: $a \in$ *distideal-extension a*
  unfolding *distideal-extension-def*
proof
  have $0 \in \mathcal{N} \wedge a = 1 * a + 0$ **using** *distset-zero-closed* **by** *simp*
  thus $\exists\, r\ z.\ z \in \mathcal{N} \wedge a = r * a + z$ **by** *fastforce*
qed

end

class *comm-ring-1-with-distinguished-pideal* = *comm-ring-1-with-distinguished-ideal*
+
assumes *distset-pideal*: $r * s \in \mathcal{N} \Longrightarrow r \in \mathcal{N} \vee s \in \mathcal{N}$

class *comm-ring-1-with-distinguished-maxideal* = *comm-ring-1-with-distinguished-ideal*
+
assumes *distset-maxideal*:
$\llbracket$
$\mathcal{N} \subseteq I$; $1 \notin I$;
$\forall r\ s.\ r \in I \wedge s \in I \longrightarrow r - s \in I$;
$\forall a\ r.\ a \in I \longrightarrow r * a \in I$
$\rrbracket \Longrightarrow I = \mathcal{N}$
begin

lemma *distset-maxideal-vs-distideal-extension*:
$1 \notin$ *distideal-extension* $a \Longrightarrow$ *distideal-extension* $a = \mathcal{N}$
using *distideal-extension distideal-extension-diff-closed distideal-extension-ideal distset-maxideal*
by *force*

subclass *comm-ring-1-with-distinguished-pideal*
proof
fix $a\ b :: 'a$
assume *ab*: $a * b \in \mathcal{N}$
have $\neg (a \notin \mathcal{N} \wedge b \notin \mathcal{N})$
proof *clarify*
assume *a*: $a \notin \mathcal{N}$ **and** *b*: $b \notin \mathcal{N}$
define *I* **where** $I \equiv$ *distideal-extension* b
have $1 \notin I$
unfolding *distideal-extension-def I-def*
proof *clarify*
fix $r\ z$ **assume** *rz*: $z \in \mathcal{N}$ $1 = r * b + z$
from *rz*(2) **have** $a = r * (a * b) + a * z$ **using** *mult-1-right* **by** (*simp add*: *algebra-simps*)
moreover from *ab rz*(1) **have** $r * (a * b) \in \mathcal{N}$ **and** $a * z \in \mathcal{N}$
using *distset-ideal* **by** *auto*
ultimately show *False* **using** *a distset-add-closed* **by** *fastforce*
qed
with *I-def b* **show** *False*
using *distset-maxideal-vs-distideal-extension distideal-extension-by* **by** *fast*
qed
thus $a \in \mathcal{N} \vee b \in \mathcal{N}$ **by** *fast*
qed

end

1.3.2 Instances and instantiations

instance *coset-by-dist-sbgrp* :: (*comm-ring-with-distinguished-ideal*) *comm-ring*
proof
  fix $x\ y$ :: $'a$ *distinguished-quotient-ring*
  show $x * y = y * x$ **by** *transfer* (*simp add*: *distset-zero-closed algebra-simps*)
qed (*simp add*: *algebra-simps*)

instance *coset-by-dist-sbgrp* :: (*comm-ring-1-with-distinguished-ideal*) *comm-ring-1*
 by *standard* (*auto simp add*: *algebra-simps*)

instance *coset-by-dist-sbgrp* :: (*comm-ring-1-with-distinguished-pideal*) *idom*
proof (*standard*, *transfer*)
  fix $a\ b$ :: $'a$ **show**
    $-a + 0 \notin \mathcal{N} \Longrightarrow -b + 0 \notin \mathcal{N} \Longrightarrow$
      $-(a * b) + 0 \notin \mathcal{N}$
    using *distset-pideal distset-uminus-closed* **by** *fastforce*
qed

lemma *inverse-in-quotient-mod-maxideal*:
  $\exists\, y.\ -(y * x) + 1 \in \mathcal{N}$ **if** $x \notin \mathcal{N}$
  for x :: $'a$::*comm-ring-1-with-distinguished-maxideal*
proof−
  from *that* **have** $1 \notin$ *distideal-extension* $x \Longrightarrow$ *False*
   using *distset-uminus-closed distset-maxideal-vs-distideal-extension distideal-extension-by*
    by *auto*
  from *this* **obtain** $y\ z$ **where** *yz*: $z \in \mathcal{N}$ $1 = y * x + z$
    using *distideal-extension-def* **by** *fast*
  from *yz*(*2*) **have** $-\ (y * x) + 1 = z$ **by** (*simp add*: *algebra-simps*)
  with *yz*(*1*) **show** $\exists\, y.\ -(y * x) + 1 \in \mathcal{N}$ **by** *fast*
qed

instantiation *coset-by-dist-sbgrp* :: (*comm-ring-1-with-distinguished-maxideal*) *field*
begin

lift-definition *inverse-coset-by-dist-sbgrp* ::
  $'a$ *distinguished-quotient-ring* $\Rightarrow$ $'a$ *distinguished-quotient-ring*
  is $\lambda x{::}'a.$ *if* $x{\in}\mathcal{N}$ *then* 0 *else* (*SOME* $y{::}'a.\ -(y{*}x) + 1 \in \mathcal{N}$)
proof−
  fix $x\ x'$ :: $'a$
  assume xx': $-x + x' \in \mathcal{N}$
  define $ix\ ix'$
    where $ix \equiv$ *if* $x \in \mathcal{N}$ *then* 0 *else* (*SOME* $y{::}'a.\ -(y * x\ ) + 1 \in \mathcal{N}$)
    and $ix' \equiv$ *if* $x' \in \mathcal{N}$ *then* 0 *else* (*SOME* $y{::}'a.\ -(y * x') + 1 \in \mathcal{N}$)
  show $-ix + ix' \in \mathcal{N}$
  proof (*cases* $x{\in}\mathcal{N}$)
    case *True* **with** xx' *ix-def* ix'*-def* **show** *?thesis*
      using *distset-lcoset-closed1*[*of* x] *distset-zero-closed* **by** *auto*
  next
    case *False*

with xx' **have** $x' \notin \mathcal{N}$ **using** *distset-lcoset-closed2* **by** *fast*
with *ix-def ix'-def False* **have** $-((-ix + ix') * x) + ix' * (x + - x') \in \mathcal{N}$
using *someI-ex*[*OF inverse-in-quotient-mod-maxideal*]
distset-uminus-plus-closed[*of* $-(ix * x) + 1$ $-(ix' * x') + 1$]
by (*force simp add*: *algebra-simps*)
moreover have $ix' * (x + -x') \in \mathcal{N}$
using *distset-uminus-closed*[*OF xx'*] *distset-ideal* **by** *auto*
ultimately have $(-ix + ix') * x \in \mathcal{N}$ **using** *distset-lcoset-closed2* **by** *force*
with *False* **show** *?thesis* **using** *distset-pideal* **by** *fast*
qed
qed

lemma *inverse-coset-by-dist-sbgrp-eq0* [*simp*]:
$x \in \mathcal{N} \Longrightarrow$ *inverse* (*distinguished-quotient-coset* x) $= 0$
by *transfer* (*simp add*: *distset-zero-closed*)

lemma *coset-by-dist-sbgrp-inverse-rep*:
$x \notin \mathcal{N} \Longrightarrow -(y * x) + 1 \in \mathcal{N} \Longrightarrow$
inverse (*distinguished-quotient-coset* x) $=$ *distinguished-quotient-coset* y
proof *transfer*
fix $x\ y :: 'a$
assume xy: $x \notin \mathcal{N}$ $-(y * x) + 1 \in \mathcal{N}$
define $z :: 'a$ **where** $z \equiv (SOME\ y.\ - (y * x) + 1 \in \mathcal{N})$
with xy **have** $(-z + y) * x \in \mathcal{N}$
using *someI-ex*[*OF inverse-in-quotient-mod-maxideal*]
distset-diff-closed[*of* $-(z * x) + 1$ $-(y * x) + 1$]
by (*fastforce simp add*: *algebra-simps*)
with $xy(1)$ **show** $-$ (*if* $x \in \mathcal{N}$ *then* 0 *else* z) $+ y \in \mathcal{N}$
using *distset-pideal* **by** *auto*
qed

definition *divide-coset-by-dist-sbgrp-def* [*simp*]:
x *div* $y = (x{::}'a$ *distinguished-quotient-ring*$) *$ *inverse* y

lemma *coset-by-dist-sbgrp-divide-rep*:
$y \notin \mathcal{N} \Longrightarrow -(z * y) + 1 \in \mathcal{N} \Longrightarrow$
distinguished-quotient-coset x *div* *distinguished-quotient-coset* y
$=$ *distinguished-quotient-coset* $(x * z)$
by (*simp add*: *coset-by-dist-sbgrp-inverse-rep*)

instance proof
show *inverse* ($0{::}'a$ *distinguished-quotient-ring*) $= 0$
by *transfer* (*simp add*: *distset-zero-closed*)
next
fix $x :: 'a$ *distinguished-quotient-ring*
show $x \neq 0 \Longrightarrow$ *inverse* $x * x = 1$
proof *transfer*
fix $a{::}'a$
assume $-a + 0 \notin \mathcal{N}$

```
    moreover define ia
      where ia ≡ if a ∈ 𝒩 then 0 else (SOME b. − (b ∗ a) + 1 ∈ 𝒩)
    ultimately show −(ia ∗ a) + 1 ∈ 𝒩
       using distset-uminus-closed someI-ex[OF inverse-in-quotient-mod-maxideal]
by auto
  qed
qed (rule divide-coset-by-dist-sbgrp-def)

end


end
```

```
theory Polynomial-Extras

imports HOL−Computational-Algebra.Polynomial

begin
```

2 Additional facts about polynomials

2.1 General facts

```
lemma set-strip-while: set (strip-while P xs) ⊆ set xs
  using split-strip-while-append[of P xs] by force

lemma pCons-induct′ [case-names 0 pCons0 pCons]:
  P p
  if zero      : P 0
  and pCons0   : ⋀a. a ≠ 0 ⟹ P (pCons a 0)
  and pCons-n0 : ⋀a p. p ≠ 0 ⟹ P p ⟹ P (pCons a p)
proof (induct p, rule zero)
  case (pCons a p) thus ?case by (cases p = 0, simp-all add: pCons0 pCons-n0)
qed

lemma pCons-decomp: pCons a p = pCons a 0 + pCons 0 p
  by (induct p, simp-all)

lemma coeffs-smult-eq-smult-coeffs:
  coeffs (smult x p) = map ((∗) x) (strip-while (λc. x ∗ c = 0) (coeffs p))
proof−
  have coeffs (smult x p) = strip-while ((=) 0) (map ((∗) x) (coeffs p))
    by (induct p) auto
  thus ?thesis by (simp add: strip-while-map comp-def)
qed

lemma coeffs-smult-eq-smult-coeffs-no-zero-divisors:
```

coeffs (*smult x p*) = *map* ((∗) *x*) (*coeffs p*) **if** $x \neq 0$
for *x* :: *'a*::{*comm-semiring-0*,*semiring-no-zero-divisors*} **and** *p* :: *'a poly*
using *that cCons-def*[*of x* ∗ -] *cCons-def*[*of* - *coeffs* -]
by (*induct p*, *simp*, *fastforce*)

2.2 Derivatives

The type sort on *pderiv* is overly restrictive for algebraic purposes, so here is the relevant material with the type sort relaxed.

function *polyderiv* :: (*'a* :: *comm-monoid-add*) *poly* ⇒ *'a poly*
where *polyderiv* (*pCons a p*) = (*if p* = *0 then 0 else p* + *pCons 0* (*polyderiv p*))
by (*auto intro*: *pCons-cases*)

termination *polyderiv*
by (*relation measure degree*) *simp-all*

declare *polyderiv.simps*[*simp del*]

lemma *polyderiv-0* [*simp*]: *polyderiv 0* = *0*
using *polyderiv.simps* [*of 0 0*] **by** *simp*

lemma *polyderiv-pCons*: *polyderiv* (*pCons a p*) = *p* + *pCons 0* (*polyderiv p*)
by (*simp add*: *polyderiv.simps*)

lemma *polyderiv-1* [*simp*]: *polyderiv 1* = *0*
by (*simp add*: *one-pCons polyderiv-pCons*)

2.3 Additive polynomials

The Binomial Theorem implies that to every polynomial can be associated a finite list of polynomials that describe how the original polynomial evaluates on binomial arguments.

function *additive-poly-poly* :: *'a*::*comm-semiring-1 poly* ⇒ *'a poly poly*
where
additive-poly-poly (*pCons a p*) =
(*if p* = *0*
then [:[:*a*:]:]
else smult (*pCons 0 1*) (*additive-poly-poly p*) + *pCons* [:*a*:] (*additive-poly-poly p*)
)
by (*auto intro*: *pCons-cases*)

termination *additive-poly-poly*
by (*relation measure degree*) *simp-all*

declare *additive-poly-poly.simps*[*simp del*]

lemma *additive-poly-poly-0* [*simp*]: *additive-poly-poly 0* = *0* using *additive-poly-poly.simps*[*of 0 0*] **by** *simp*

lemma *additive-poly-poly-pCons0*:
$a \neq 0 \Longrightarrow$ *additive-poly-poly* [:*a*:] = [:[:*a*:]:]
using *additive-poly-poly.simps*[*of a 0*] **by** *argo*

lemma *additive-poly-poly-pCons*:
$p \neq 0 \Longrightarrow$
additive-poly-poly (*pCons a p*) =
smult (*pCons 0 1*) (*additive-poly-poly p*) + *pCons* [:*a*:] (*additive-poly-poly p*)
by (*simp add*: *additive-poly-poly.simps*)

lemma *additive-poly-poly*: *poly p* (*x* + *y*) = *poly* (*poly* (*additive-poly-poly p*) [:*x*:])
y
by (*induct p rule*: *pCons-induct′*)
(*simp-all add*: *algebra-simps additive-poly-poly-pCons0 additive-poly-poly-pCons*)

lemma *additive-poly-poly-coeff0*: *coeff* (*additive-poly-poly p*) *0* = *p*
by (*induct p rule*: *pCons-induct′*)
(*simp-all add*: *additive-poly-poly-pCons0 additive-poly-poly-pCons*)

lemma *additive-poly-poly-coeff1*: *coeff* (*additive-poly-poly p*) *1* = *polyderiv p*
by (*induct p rule*: *pCons-induct′*)
(*simp-all add*:
ac-simps additive-poly-poly-pCons0 additive-poly-poly-pCons additive-poly-poly-coeff0 polyderiv-pCons
)

lemma *additive-poly-poly-eq0-iff*: *additive-poly-poly p* = *0* ⟷ *p* = *0*
using *additive-poly-poly-coeff0*[*of p*] **by** *fastforce*

lemma *additive-poly-poly-degree*: *degree* (*additive-poly-poly p*) = *degree p*
proof (*induct p rule*: *pCons-induct′*)
case (*pCons a p*)
moreover have
degree (*smult* (*pCons 0 1*) (*additive-poly-poly p*)) <
degree (*pCons* [:*a*:] (*additive-poly-poly p*))
proof (*rule degree-lessI*, *safe*)
fix *k* **assume** $k \geq$ *degree* (*pCons* [:*a*:] (*additive-poly-poly p*))
with *pCons*(*1*) **show** *coeff* (*smult* (*pCons 0 1*) (*additive-poly-poly p*)) *k* = *0*
using *additive-poly-poly-eq0-iff coeff-eq-0*[*of - k*] **by** *force*
qed (*simp add*: *pCons*(*1*) *additive-poly-poly-eq0-iff*)
ultimately show *?case*
using *degree-add-eq-right additive-poly-poly-pCons degree-pCons-eq additive-poly-poly-eq0-iff*
by *metis*
qed (*simp*, *simp add*: *additive-poly-poly-pCons0*)

lemma *poly-additive-poly-poly-pCons*:
poly (*additive-poly-poly* (*pCons a p*)) [:*x*:] =

pCons a (*poly* (*additive-poly-poly p*) [:*x*:]) + *smult x* (*poly* (*additive-poly-poly p*) [:*x*:])
if $p \neq 0$
using *that additive-poly-poly-pCons pCons-decomp*
by (*fastforce simp add*: *add.assoc*[*symmetric*] *add.commute*)

lemma *poly-additive-poly-poly-eq0-iff*:
poly (*additive-poly-poly p*) [:*x*:] = *0* $\longleftrightarrow$ *p* = *0*
proof
show *p* = *0* $\Longrightarrow$ *poly* (*additive-poly-poly p*) [:*x*:] = *0*
using *additive-poly-poly-eq0-iff* **by** *simp*
next
show *poly* (*additive-poly-poly p*) [:*x*:] = *0* $\Longrightarrow$ *p* = *0*
proof (*induct p rule*: *pCons-induct′*)
next
case (*pCons a p*)
moreover define *poly-app-x* :: *′a poly* ⇒ *′a poly*
where *poly-app-x* ≡ λ*p*. *poly* (*additive-poly-poly p*) [:*x*:]
moreover from *this pCons*(*1*) **have**
coeff (*poly-app-x* (*pCons a p*)) (*Suc* (*degree* (*poly-app-x p*))) = *lead-coeff* (*poly-app-x p*)
using *poly-additive-poly-poly-pCons coeff-eq-0*[*of poly-app-x p*] **by** *fastforce*
ultimately show *?case* **using** *leading-coeff-neq-0* **by** *simp*
qed (*simp*, *simp add*: *additive-poly-poly-pCons0*)
qed

lemma *degree-poly-additive-poly-poly*:
degree (*poly* (*additive-poly-poly p*) [:*x*:]) = *degree p*
proof (*induct p rule*: *pCons-induct′*)
case (*pCons a p*)
define *poly-app-x* :: *′a poly* ⇒ *′a poly*
where *poly-app-x* ≡ λ*p*. *poly* (*additive-poly-poly p*) [:*x*:]
from *pCons*(*1*) *poly-app-x-def*
have *deg*: *degree* (*pCons a* (*poly-app-x p*)) = *Suc* (*degree* (*poly-app-x p*))
using *degree-pCons-eq*[*of poly-app-x p a*] *poly-additive-poly-poly-eq0-iff*
by *auto*
from *pCons*(*1*) **have** *degree* (*smult x* (*poly-app-x p*)) ≤ *degree* (*poly-app-x p*)
using *degree-smult-le*[*of x poly-app-x p*] **by** *blast*
also have ... < *Suc* (*degree* (*poly-app-x p*)) **by** *blast*
finally have *degree* (*smult x* (*poly-app-x p*)) < *degree* (*pCons a* (*poly-app-x p*)) **using** *deg* **by** *argo*
with *pCons poly-app-x-def* **show** *?case*
by (*metis deg poly-additive-poly-poly-pCons degree-add-eq-left degree-pCons-eq*)
qed (*simp*, *simp add*: *additive-poly-poly-pCons0*)

lemma *coeff-poly-additive-poly-poly*:
coeff (*poly* (*additive-poly-poly p*) [:*x*:]) *n* = *poly* (*coeff* (*additive-poly-poly p*) *n*) *x*
proof−
have

$\forall\, k \le n.$
coeff (*poly* (*additive-poly-poly* *p*) [:*x*:]) *k* = *poly* (*coeff* (*additive-poly-poly* *p*) *k*) *x*
proof (*induct p rule*: *pCons-induct′*, *safe*)
fix *k* **case** (*pCons0 a*)
moreover have *coeff* [:*a*:] *k* = *poly* (*coeff* [:[:*a*:]:] *k*) *x* **by** (*cases k*, *simp-all*)
ultimately show
coeff (*poly* (*additive-poly-poly* [:*a*:]) [:*x*:]) *k* =
poly (*coeff* (*additive-poly-poly* [:*a*:]) *k*) *x*
using *additive-poly-poly-pCons0* **by** *fastforce*
next
case (*pCons a p*) **fix** *k* **assume** $k \le n$ **with** *pCons* **show**
coeff (*poly* (*additive-poly-poly* (*pCons a p*)) [:*x*:]) *k* =
poly (*coeff* (*additive-poly-poly* (*pCons a p*)) *k*) *x*
using *poly-additive-poly-poly-pCons additive-poly-poly-pCons*
by (*cases k*) (*fastforce*, *force simp add*: *add.commute*)
qed *simp*
thus *?thesis* **by** *blast*
qed

lemma *additive-poly-poly-decomp*:
poly p $(x + y) = (\sum i \le degree\ p.\ (poly\ (coeff\ (additive\text{-}poly\text{-}poly\ p)\ i)\ x) * y\ \hat{}\ i)$
using *poly-altdef*[*of - y*] *coeff-poly-additive-poly-poly*[*of p x*] *additive-poly-poly*[*of p x y*]
degree-poly-additive-poly-poly[*of p x*]
by *presburger*

end

theory *Parametric-Equiv-Depth*

imports
HOL.Rat
HOL−Computational-Algebra.Fraction-Field
HOL−Computational-Algebra.Primes
HOL−Library.Function-Algebras
HOL−Library.Set-Algebras
HOL.Topological-Spaces
Polynomial-Extras

begin

3 Common structures for adelic types

3.1 Preliminaries

lemma *ex-least-nat-le'*:
 fixes $P :: nat \Rightarrow bool$
 assumes $P\ n$
 shows $\exists k{\le}n.\ (\forall i{<}k.\ \neg\ P\ i) \wedge P\ k$
 by (*metis assms exists-least-iff not-le-imp-less*)

lemma *ex-ex1-least-nat-le*:
 fixes $P :: nat \Rightarrow bool$
 assumes $\exists n.\ P\ n$
 shows $\exists !k.\ P\ k \wedge (\forall i.\ P\ i \longrightarrow k \le i)$
 by (*smt* (*verit*) *assms ex-least-nat-le' le-eq-less-or-eq nle-le*)

lemma *least-le-least*:
 $Least\ Q \le Least\ P$
 if $P : \exists !x.\ P\ x \wedge (\forall y.\ P\ y \longrightarrow x \le y)$
 and $Q : \exists !x.\ Q\ x \wedge (\forall y.\ Q\ y \longrightarrow x \le y)$
 and PQ: $\forall x.\ P\ x \longrightarrow Q\ x$
 for $P\ Q :: 'a{::}order \Rightarrow bool$
 using *PQ Least1-le*[*OF Q*] *Least1I*[*OF P*] **by** *blast*

lemma *linorder-wlog'* [*case-names le sym*]:
 $[\![$
  $\bigwedge a\ b.\ f\ a \le f\ b \Longrightarrow P\ a\ b;$
  $\bigwedge a\ b.\ P\ b\ a \Longrightarrow P\ a\ b$
 $]\!] \Longrightarrow P\ a\ b$
 for $f :: 'a \Rightarrow 'b{::}linorder$
 by (*cases rule*: *le-cases*[*of f a f b*]) *blast+*

lemma *log-powi-cancel* [*simp*]:
 $a > 0 \Longrightarrow a \ne 1 \Longrightarrow log\ a\ (a\ powi\ b) = b$
 by (*cases* $b \ge 0$, *simp-all add*: *power-int-def power-inverse log-inverse*)

3.2 Existence of primes

class *factorial-comm-ring* = *factorial-semiring* + *comm-ring*
begin subclass *comm-ring-1* **.. end**

class *factorial-idom* = *factorial-semiring* + *idom*
begin subclass *factorial-comm-ring* **.. end**

class *nontrivial-factorial-semiring* = *factorial-semiring* +
 assumes *nontrivial-elem-exists*: $\exists x.\ x \ne 0 \wedge normalize\ x \ne 1$
begin

lemma *primeE*:
 obtains p **where** $prime\ p$

proof −
  from *nontrivial-elem-exists* **obtain** x **where** x: $x \neq 0$ *normalize* $x \neq 1$ **by** *fast*
  from $x(1)$ **obtain** A
    where $\forall x.\ x \in\# A \longrightarrow$ *prime* x *normalize* (*prod-mset* A) = *normalize* x
    by (*auto elim*: *prime-factorization-exists'*)
  with $x(2)$ **obtain** p **where** *prime* p **by** *fastforce*
  with *that* **show** *thesis* **by** *fast*
qed

lemma *primes-exist*: $\exists p.$ *prime* p
proof −
  obtain p **where** *prime* p **by** (*elim primeE*)
  thus *?thesis* **by** *fast*
qed

end

class *nontrivial-factorial-comm-ring* = *nontrivial-factorial-semiring* + *comm-ring*
begin
  subclass *factorial-comm-ring* **..**
end

class *nontrivial-factorial-idom* = *nontrivial-factorial-semiring* + *idom*
begin
  subclass *nontrivial-factorial-comm-ring* **..**
  subclass *factorial-idom* **..**
end

instance *int* :: *nontrivial-factorial-idom*
proof
  have (*2*::*int*) $\neq 0$ *normalize* (*2*::*int*) $\neq 1$ **by** *auto*
  thus $\exists x$::*int*. $x \neq 0 \land$ *normalize* $x \neq 1$ **by** *fast*
qed

typedef (**overloaded**) *'a prime* =
  {*p*::*'a*::*nontrivial-factorial-semiring*. *prime p*}
  using *primes-exist* **by** *fast*

lemma *Rep-prime-n0*: *Rep-prime p* $\neq 0$
  using *Rep-prime*[*of p*] *prime-imp-nonzero* **by** *simp*

lemma *Rep-prime-not-unit*: $\neg$ *is-unit* (*Rep-prime p*)
  using *Rep-prime*[*of p*] **by** *auto*

abbreviation *pdvd* ::
  'a::*nontrivial-factorial-semiring prime* $\Rightarrow$ *'a* $\Rightarrow$ *bool* (**infix** *pdvd 50*)
  where *p pdvd a* $\equiv$ (*Rep-prime p*) *dvd a*
abbreviation *pmultiplicity p* $\equiv$ *multiplicity* (*Rep-prime p*)

lemma *multiplicity-distinct-primes*:
$p \neq q \Longrightarrow$ *pmultiplicity p* (*Rep-prime q*) $= 0$
using *Rep-prime-inject Rep-prime*
multiplicity-distinct-prime-power[*of Rep-prime p Rep-prime q 1*]
by *auto*

3.3 Generic algebraic equality

context
fixes *R*
assumes *sympR* : *symp R*
and *transpR*: *transp R*
begin

lemma *transp-left*: $R\ x\ z \Longrightarrow R\ y\ z \Longrightarrow R\ x\ y$
by (*metis sympR sympD transpR transpD*)

lemma *transp-right*: $R\ x\ y \Longrightarrow R\ x\ z \Longrightarrow R\ y\ z$
by (*metis sympR sympD transpR transpD*)

lemma *transp-iffs*:
assumes *R x y* **shows** $R\ x\ z \longleftrightarrow R\ y\ z$ **and** $R\ z\ x \longleftrightarrow R\ z\ y$
using *assms transpD*[*OF transpR*] *transp-left transp-right* **by** (*blast*, *blast*)

lemma *transp-cong*: *R w z* **if** *R x w* **and** *R y z* **and** *R x y*
using *that transpD*[*OF transpR*] *transp-right* **by** *blast*

lemma *transp-cong-sym*: *R x y* **if** *R x w* **and** *R y z* **and** *R w z*
using *that transp-cong sympD*[*OF sympR*] **by** *blast*

end

lemma *equivp-imp-symp*: *equivp R* $\Longrightarrow$ *symp R*
by (*simp add*: *equivp-reflp-symp-transp*)

locale *generic-alg-equality* =
fixes *alg-eq* :: $'a \Rightarrow 'a \Rightarrow$ *bool*
assumes *equivp*: *equivp alg-eq*
begin

lemmas *refl* = *equivp-reflp* [*OF equivp*]
and *sym* [*sym*] = *equivp-symp* [*OF equivp*]
and *trans* [*trans*] = *equivp-transp* [*OF equivp*]
and *trans-left* = *transp-left* [*OF equivp-imp-symp equivp-imp-transp*, *OF equivp equivp*]
and *trans-right* = *transp-right* [*OF equivp-imp-symp equivp-imp-transp*, *OF equivp equivp*]
and *trans-iffs* = *transp-iffs* [*OF equivp-imp-symp equivp-imp-transp*, *OF*

equivp equivp]
  and *cong* = *transp-cong* [*OF equivp-imp-symp equivp-imp-transp*, *OF equivp equivp*]
  and *cong-sym* = *transp-cong-sym* [*OF equivp-imp-symp equivp-imp-transp*, *OF equivp equivp*]

lemma *sym-iff*: *alg-eq x y* = *alg-eq y x*
  using *sym* **by** *blast*

end

locale *ab-group-alg-equality* = *generic-alg-equality alg-eq*
  for *alg-eq* :: *'a::ab-group-add* ⇒ *'a* ⇒ *bool*
+
  assumes *conv-0*: *alg-eq x y* ⟷ *alg-eq* (*x* − *y*) *0*
begin

lemmas *trans0-iff* = *trans-iffs*(*1*)[*of* - - *0*]

lemma *add-equiv0-left*: *alg-eq* (*x* + *y*) *y* **if** *alg-eq x 0*
  using *that conv-0* **by** *fastforce*

lemma *add-equiv0-right*: *alg-eq* (*x* + *y*) *x* **if** *alg-eq y 0*
  using *that conv-0* **by** *fastforce*

lemma *add*:
  alg-eq (*x1* + *x2*) (*y1* + *y2*) **if** *alg-eq x1 y1* **and** *alg-eq x2 y2*
proof (*rule iffD2*, *rule conv-0*)
  from *that* **have** *alg-eq* (*x1* − *y1*) *0* **and** *alg-eq* (*x2* − *y2*) *0*
    using *conv-0* **by** *auto*
  hence *alg-eq* ((*x1* − *y1*) + (*x2* − *y2*)) *0* **using** *add-equiv0-left trans* **by** *blast*
  moreover have (*x1* − *y1*) + (*x2* − *y2*) = (*x1* + *x2*) − (*y1* + *y2*) **by** *simp*
  ultimately show *alg-eq* ((*x1* + *x2*) − (*y1* + *y2*)) *0* **by** *simp*
qed

lemma *add-left*: *alg-eq* (*x* + *y*) (*x* + *y'*) **if** *alg-eq y y'*
  using *that refl add* **by** *simp*

lemma *add-right*: *alg-eq* (*x* + *y*) (*x'* + *y*) **if** *alg-eq x x'*
  using *that refl add* **by** *simp*

lemma *add-0-left*: *alg-eq* (*x* + *y*) *y* **if** *alg-eq x 0*
  using *that add-right*[*of x 0*] **by** *simp*

lemma *add-0-right*: *alg-eq* (*x* + *y*) *x* **if** *alg-eq y 0*
  using *that add-left*[*of y 0*] **by** *simp*

lemma *sum-zeros*:

finite A $\Longrightarrow \forall$ *a*$\in$*A*. *alg-eq* (*f a*) *0* $\Longrightarrow$ *alg-eq* (*sum f A*) *0*
by (*induct A rule*: *finite-induct*, *simp add*: *refl*, *use add* **in** *force*)

lemma *uminus*: *alg-eq x y* $\Longrightarrow$ *alg-eq* ($-x$) ($-y$)
using *conv-0*[*of x y*] *conv-0*[*of* $-y$ $-x$] *sym*[*of* $-y$ $-x$] **by** *force*

lemma *uminus-iff*:
alg-eq x y $\longleftrightarrow$ *alg-eq* ($-x$) ($-y$)
using *uminus*[*of x y*] *uminus*[*of* $-x$ $-y$] **by** *fastforce*

lemma *add-0*: *alg-eq* ($x + y$) *0* **if** *alg-eq x 0* **and** *alg-eq y 0*
using *that add*[*of x 0 y 0*] **by** *auto*

lemma *minus*:
alg-eq x1 y1 $\Longrightarrow$ *alg-eq x2 y2* $\Longrightarrow$
alg-eq ($x1 - x2$) ($y1 - y2$)
using *uminus*[*of x2 y2*] *add*[*of x1 y1* $-x2$ $-y2$] **by** *simp*

lemma *minus-0*: *alg-eq* ($x - y$) *0* **if** *alg-eq x 0* **and** *alg-eq y 0*
using *that minus*[*of x 0 y 0*] **by** *simp*

lemma *minus-left*: *alg-eq* ($x - y$) ($x - y'$) **if** *alg-eq y y'*
using *that refl minus* **by** *simp*

lemma *minus-right*: *alg-eq* ($x - y$) ($x' - y$) **if** *alg-eq x x'*
using *that refl minus* **by** *simp*

lemma *minus-0-left*: *alg-eq* ($x - y$) ($-y$) **if** *alg-eq x 0*
using *that refl minus-right*[*of x 0*] **by** *simp*

lemma *minus-0-right*: *alg-eq* ($x - y$) *x* **if** *alg-eq y 0*
using *that refl minus-left*[*of y 0*] **by** *simp*

end

locale *ring-alg-equality* = *ab-group-alg-equality alg-eq*
for *alg-eq* :: *'a::comm-ring* $\Rightarrow$ *'a* $\Rightarrow$ *bool*
+
assumes *mult-0-right*: *alg-eq y 0* $\Longrightarrow$ *alg-eq* ($x * y$) *0*
begin

lemma *mult-0-left*: *alg-eq x 0* $\Longrightarrow$ *alg-eq* ($x * y$) *0*
using *mult-0-right*[*of x y*] **by** (*simp add*: *mult.commute*)

lemma *mult*:
alg-eq x1 y1 $\Longrightarrow$ *alg-eq x2 y2* $\Longrightarrow$ *alg-eq* ($x1 * x2$) ($y1 * y2$)
using *mult-0-right*[*of* $x2 - y2$ *x1*] *mult-0-left*[*of* $x1 - y1$ *y2*] *conv-0 trans*
by (*force simp add*: *algebra-simps*)

lemma *mult-left*: *alg-eq y y′* $\Longrightarrow$ *alg-eq* $(x * y)$ $(x * y')$
  using *refl mult* **by** *force*

lemma *mult-right*: *alg-eq x x′* $\Longrightarrow$ *alg-eq* $(x * y)$ $(x' * y)$
  using *refl mult* **by** *force*

end

locale *ring-1-alg-equality* = *ring-alg-equality alg-eq*
  for *alg-eq* :: *′a::comm-ring-1* $\Rightarrow$ *′a* $\Rightarrow$ *bool*
begin

lemma *mult-one-left*: *alg-eq x 1* $\Longrightarrow$ *alg-eq* $(x * y)$ *y*
  using *mult-right* **by** *fastforce*

lemma *mult-one-right*: *alg-eq y 1* $\Longrightarrow$ *alg-eq* $(x * y)$ *x*
  using *mult-left* **by** *fastforce*

lemma *prod-ones*:
  finite A $\Longrightarrow$ $\forall\, a \in A$. *alg-eq* (*f a*) *1* $\Longrightarrow$
    alg-eq (*prod f A*) *1*
  by (*induct A rule*: *finite-induct*, *simp add*: *refl*, *use mult* **in** *force*)

lemma *prod-with-zero*:
  alg-eq (*prod f* (*insert a A*)) *0* **if** *finite A* **and** *alg-eq* (*f a*) *0*
  using *that prod.insert*[*of A* $-$ {*a*} *a f*] *mult-0-left* **by** *fastforce*

lemma *pow*: *alg-eq* $(x \,\hat{}\, n)$ $(y \,\hat{}\, n)$ **if** *alg-eq x y*
  using *that refl mult* **by** (*induct n*, *simp-all*)

lemma *pow-equiv0*: *alg-eq* $(x \,\hat{}\, n)$ *0* **if** $n > 0$ **and** *alg-eq x 0*
  by (*metis that zero-power pow*)

end

3.4 Indexed equality

3.4.1 General structure

locale *p-equality* =
  fixes *p-equal* :: *′a* $\Rightarrow$ *′b::comm-ring* $\Rightarrow$ *′b* $\Rightarrow$ *bool*
  assumes *p-alg-equality*: *ring-alg-equality* (*p-equal p*)
begin

lemmas *p-group-alg-equality* = *ring-alg-equality.axioms*(*1*)[*OF p-alg-equality*]
lemmas *p-generic-alg-equality* = *ab-group-alg-equality.axioms*(*1*)[*OF p-group-alg-equality*]

lemmas *equivp* = *generic-alg-equality.equivp* [*OF p-generic-alg-equality*]

and *refl*[*simp*] = *generic-alg-equality.refl* [*OF p-generic-alg-equality*]
and *sym* [*sym*] = *generic-alg-equality.sym* [*OF p-generic-alg-equality*]
and *trans* [*trans*] = *generic-alg-equality.trans* [*OF p-generic-alg-equality*]
and *sym-iff* = *generic-alg-equality.sym-iff* [*OF p-generic-alg-equality*]
and *trans-left* = *generic-alg-equality.trans-left* [*OF p-generic-alg-equality*]
and *trans-right* = *generic-alg-equality.trans-right* [*OF p-generic-alg-equality*]
and *trans-iffs* = *generic-alg-equality.trans-iffs* [*OF p-generic-alg-equality*]
and *cong* = *generic-alg-equality.cong* [*OF p-generic-alg-equality*]
and *cong-sym* = *generic-alg-equality.cong-sym* [*OF p-generic-alg-equality*]

lemmas *conv-0* = *ab-group-alg-equality.conv-0* [*OF p-group-alg-equality*]
and *trans0-iff* = *ab-group-alg-equality.trans0-iff* [*OF p-group-alg-equality*]
and *add-equiv0-left* = *ab-group-alg-equality.add-equiv0-left* [*OF p-group-alg-equality*]
and *add-equiv0-right* = *ab-group-alg-equality.add-equiv0-right* [*OF p-group-alg-equality*]
and *add* = *ab-group-alg-equality.add* [*OF p-group-alg-equality*]
and *add-left* = *ab-group-alg-equality.add-left* [*OF p-group-alg-equality*]
and *add-right* = *ab-group-alg-equality.add-right* [*OF p-group-alg-equality*]
and *add-0-left* = *ab-group-alg-equality.add-0-left* [*OF p-group-alg-equality*]
and *add-0-right* = *ab-group-alg-equality.add-0-right* [*OF p-group-alg-equality*]
and *sum-zeros* = *ab-group-alg-equality.sum-zeros* [*OF p-group-alg-equality*]
and *uminus* = *ab-group-alg-equality.uminus* [*OF p-group-alg-equality*]
and *uminus-iff* = *ab-group-alg-equality.uminus-iff* [*OF p-group-alg-equality*]
and *add-0* = *ab-group-alg-equality.add-0* [*OF p-group-alg-equality*]
and *minus* = *ab-group-alg-equality.minus* [*OF p-group-alg-equality*]
and *minus-0* = *ab-group-alg-equality.minus-0* [*OF p-group-alg-equality*]
and *minus-left* = *ab-group-alg-equality.minus-left* [*OF p-group-alg-equality*]
and *minus-right* = *ab-group-alg-equality.minus-right* [*OF p-group-alg-equality*]
and *minus-0-left* = *ab-group-alg-equality.minus-0-left* [*OF p-group-alg-equality*]
and *minus-0-right* = *ab-group-alg-equality.minus-0-right* [*OF p-group-alg-equality*]

lemmas *mult-0-left* = *ring-alg-equality.mult-0-left* [*OF p-alg-equality*]
and *mult-0-right* = *ring-alg-equality.mult-0-right* [*OF p-alg-equality*]
and *mult* = *ring-alg-equality.mult* [*OF p-alg-equality*]
and *mult-left* = *ring-alg-equality.mult-left* [*OF p-alg-equality*]
and *mult-right* = *ring-alg-equality.mult-right* [*OF p-alg-equality*]

definition *globally-p-equal* :: *'b* ⇒ *'b* ⇒ *bool*
where *globally-p-equal-def*[*simp*]: *globally-p-equal x y* = (∀ *p*::*'a*. *p-equal p x y*)

lemma *globally-p-equalI*[*intro*]: *globally-p-equal x y* **if** ⋀*p*::*'a*. *p-equal p x y*
using *that* **unfolding** *globally-p-equal-def* **by** *blast*

lemma *globally-p-equalD*: *globally-p-equal x y* ⟹ *p-equal p x y*
unfolding *globally-p-equal-def* **by** *fast*

lemma *globally-p-nequalE*:
assumes ¬ *globally-p-equal x y*
obtains *p* **where** ¬ *p-equal p x y*
using *assms*

unfolding *globally-p-equal-def*
by *fast*

sublocale *globally-p-equality*: *ring-alg-equality globally-p-equal*
proof (*standard*, *intro equivpI*)
show *reflp globally-p-equal*
using *refl* **unfolding** *globally-p-equal-def* **by** (*fastforce intro*: *reflpI*)
show *symp globally-p-equal*
using *sym* **unfolding** *globally-p-equal-def* **by** (*fastforce intro*: *sympI*)
show *transp globally-p-equal*
using *trans* **unfolding** *globally-p-equal-def* **by** (*fastforce intro*: *transpI*)
fix *x y* :: *'b*
show *globally-p-equal x y* = *globally-p-equal* (*x* − *y*) *0*
and *globally-p-equal y 0* ⟹ *globally-p-equal* (*x* ∗ *y*) *0*
using *conv-0 mult-0-right*
unfolding *globally-p-equal-def*
by *auto*
qed

lemmas *globally-p-equal-equivp* = *globally-p-equality.equivp*
and *globally-p-equal-refl*[*simp*] = *globally-p-equality.refl*
and *globally-p-equal-sym* = *globally-p-equality.sym*
and *globally-p-equal-trans* = *globally-p-equality.trans*
and *globally-p-equal-trans-left* = *globally-p-equality.trans-left*
and *globally-p-equal-trans-right* = *globally-p-equality.trans-right*
and *globally-p-equal-trans-iffs* = *globally-p-equality.trans-iffs*
and *globally-p-equal-cong* = *globally-p-equality.cong*
and *globally-p-equal-conv-0* = *globally-p-equality.conv-0*
and *globally-p-equal-trans0-iff* = *globally-p-equality.trans0-iff*
and *globally-p-equal-add-equiv0-left* = *globally-p-equality.add-equiv0-left*
and *globally-p-equal-add-equiv0-right* = *globally-p-equality.add-equiv0-right*
and *globally-p-equal-add* = *globally-p-equality.add*
and *globally-p-equal-uminus* = *globally-p-equality.uminus*
and *globally-p-equal-uminus-iff* = *globally-p-equality.uminus-iff*
and *globally-p-equal-add-0* = *globally-p-equality.add-0*
and *globally-p-equal-minus* = *globally-p-equality.minus*
and *globally-p-equal-mult-0-left* = *globally-p-equality.mult-0-left*
and *globally-p-equal-mult-0-right* = *globally-p-equality.mult-0-right*
and *globally-p-equal-mult* = *globally-p-equality.mult*
and *globally-p-equal-mult-left* = *globally-p-equality.mult-left*
and *globally-p-equal-mult-right* = *globally-p-equality.mult-right*

lemma *globally-p-equal-transfer-equals-rsp*:
p-equal p x1 y1 ⟷ *p-equal p x2 y2*
if *globally-p-equal x1 x2* **and** *globally-p-equal y1 y2*
using *that trans-iffs globally-p-equalD* **by** *blast*

end

locale *p-equality-1* = *p-equality p-equal*
  for *p-equal* :: *'a* ⇒ *'b::comm-ring-1* ⇒ *'b* ⇒ *bool*
begin

lemma *p-alg-equality-1*: *ring-1-alg-equality* (*p-equal p*)
  using *ring-1-alg-equality.intro p-alg-equality* **by** *blast*

lemmas *mult-one-left* = *ring-1-alg-equality.mult-one-left* [*OF p-alg-equality-1*]
  and *mult-one-right* = *ring-1-alg-equality.mult-one-right* [*OF p-alg-equality-1*]
  and *prod-ones* = *ring-1-alg-equality.prod-ones* [*OF p-alg-equality-1*]
  and *pow* = *ring-1-alg-equality.pow* [*OF p-alg-equality-1*]
  and *pow-equiv0* = *ring-1-alg-equality.pow-equiv0* [*OF p-alg-equality-1*]

end

locale *p-equality-no-zero-divisors* = *p-equality*
+
  assumes *no-zero-divisors*:
    ¬ *p-equal p x 0* ⟹ ¬ *p-equal p y 0* ⟹
      ¬ *p-equal p* (*x* ∗ *y*) *0*
begin

lemma *mult-equiv-0-iff*:
  p-equal p (*x* ∗ *y*) *0* ⟷
    p-equal p x 0 ∨ *p-equal p y 0*
  using *mult-0-right no-zero-divisors* **by** (*fastforce simp add*: *mult.commute*)

end

locale *p-equality-no-zero-divisors-1* = *p-equality-no-zero-divisors p-equal*
  for *p-equal* ::
    'a ⇒ *'b::comm-ring-1* ⇒ *'b* ⇒ *bool*
+
  assumes *nontrivial*: ∃ *x*. ¬ *p-equal p x 0*
begin

sublocale *p-equality-1* **..**

lemma *one-p-nequal-zero*: ¬ *p-equal p* (*1::'b*) (*0::'b*)
  using *nontrivial mult-equiv-0-iff*[*of p 1::'b*] **by** *fastforce*

lemma *zero-p-nequal-one*: ¬ *p-equal p* (*0::'b*) (*1::'b*)
  using *nontrivial one-p-nequal-zero sym* **by** *blast*

lemma *neg-one-p-nequal-zero*: ¬ *p-equal p* (−*1::'b*) (*0::'b*)
  using *one-p-nequal-zero uminus* **by** *fastforce*

lemma *pow-equiv-0-iff*: *p-equal p* $(x \,\widehat{}\, n)$ *0* $\longleftrightarrow$ *p-equal p x 0* $\wedge$ $n \neq 0$
proof (*cases n*)
 case (*Suc k*) **show** *?thesis* **by** (*simp only*: *Suc*, *induct k*, *simp-all add*: *mult-equiv-0-iff*)
qed (*simp add*: *one-p-nequal-zero*)

lemma *pow-equiv-base*: *p-equal p* $(x \,\widehat{}\, n)$ $(y \,\widehat{}\, n)$ **if** *p-equal p x y*
 using *that mult* **by** (*induct n*) *auto*

end

locale *p-equality-div-inv* = *p-equality-no-zero-divisors-1 p-equal*
 for *p-equal* ::
  $'a \Rightarrow 'b$::{*comm-ring-1*, *inverse*, *divide-trivial*} $\Rightarrow$
   $'b \Rightarrow$ *bool*
+
 assumes *divide-inverse* : $\bigwedge x\ y :: 'b.\ x\ /\ y = x * \mathit{inverse}\ y$
  and *inverse-inverse* : $\bigwedge x::'b.\ \mathit{inverse}\ (\mathit{inverse}\ x) = x$
  and *inverse-mult* : $\bigwedge x\ y :: 'b.\ \mathit{inverse}\ (x * y) = \mathit{inverse}\ x * \mathit{inverse}\ y$
  and *inverse-equiv0-iff*: *p-equal p* (*inverse x*) *0* $\longleftrightarrow$ *p-equal p x 0*
  and *divide-self-equiv* : $\neg$ *p-equal p x 0* $\Longrightarrow$ *p-equal p* $(x\ /\ x)$ *1*
begin

Global algebra facts

lemma *inverse-eq-divide*: *inverse x* = *1* / *x* **for** $x :: 'b$
 using *divide-inverse* **by** *auto*

lemma *inverse-div-inverse*: *inverse x* / *inverse y* = *y* / *x* **for** $x\ y :: 'b$
 by (*simp add*: *divide-inverse inverse-inverse*)

lemma *inverse-0*[*simp*]: *inverse* $(0::'b) = 0$
 using *inverse-eq-divide* **by** *simp*

lemma *inverse-1*[*simp*]: *inverse* $(1::'b) = 1$
 using *inverse-eq-divide* **by** *simp*

lemma *inverse-pow*: *inverse* $(x \,\widehat{}\, n)$ = *inverse* $x \,\widehat{}\, n$ **for** $x :: 'b$
 by (*induct n*) (*simp-all add*: *inverse-mult*)

lemma *power-int-minus*: *power-int x* $(-n)$ = *inverse* (*power-int x n*) **for** $x :: 'b$
 by (*cases* $n \geq 0$) (*simp-all add*: *power-int-def inverse-pow inverse-inverse*)

lemma *power-int-minus-divide*: *power-int x* $(-n)$ = *1* / (*power-int x n*) **for** $x :: 'b$
 using *power-int-minus inverse-eq-divide* **by** *presburger*

lemma *power-int-0-left-if*: *power-int* $(0 :: 'b)$ *m* = (*if m* = *0 then 1 else 0*)
 by (*auto simp add*: *power-int-def zero-power*)

lemma *power-int-0-left* [*simp*]: $m \neq 0 \Longrightarrow$ *power-int* (0 :: $'b$) $m = 0$
by (*simp add*: *power-int-0-left-if*)

lemma *power-int-1-left* [*simp*]: *power-int* (1::$'b$) $n = 1$
by (*auto simp*: *power-int-def*)

lemma *inverse-power-int*: *inverse* (*power-int* x n) = *power-int* x ($-n$) **for** x :: $'b$
using *inverse-pow inverse-inverse* **unfolding** *power-int-def* **by** (*cases* n) *auto*

lemma *power-int-inverse*:
power-int (*inverse* x) n = *inverse* (*power-int* x n) **for** x :: $'b$
proof (*cases* $n > 0$)
case *True* **thus** *?thesis* **using** *inverse-pow power-int-def* **by** *metis*
next
case *False*
moreover define m **where** $m \equiv - n$
ultimately have $m \geq 0$ **by** *fastforce*
hence *power-int* (*inverse* x) ($-m$) = *inverse* (*power-int* x ($-m$))
proof (*induct* m *rule*: *int-ge-induct*)
case (*step* m)
from *step*(1) **have** *inverse* x *powi* $-$ ($m + 1$) = x $*$ *inverse* x *powi* ($-$ m)
using *power-Suc*[*of* x *nat* m]
by (*fastforce simp add*: *Suc-nat-eq-nat-zadd1 inverse-inverse power-int-def*)
with *step* **show** *?case*
by (
simp add : *inverse-inverse inverse-mult add.commute power-int-def*
flip: *Suc-nat-eq-nat-zadd1*
)
qed *simp*
with *m-def* **show** *?thesis* **by** *force*
qed

lemma *power-int-mult-distrib*:
power-int ($x * y$) m = *power-int* x m $*$ *power-int* y m **for** x y :: $'b$
by (*cases* $m \geq 0$) (*simp-all add*: *power-int-def power-mult-distrib inverse-mult*)

lemma *inverse-divide*: *inverse* (a / b) = b / a **for** a b :: $'b$
by (*metis divide-inverse inverse-mult inverse-inverse mult.commute*)

lemma *minus-divide-left*: $-$ (a / b) = ($-$ a) / b **for** a b :: $'b$
by (*simp add*: *divide-inverse algebra-simps*)

lemma *times-divide-times-eq*: (a / b) $*$ (c / d) = ($a * c$) / ($b * d$) **for** a b :: $'b$
by (*simp add*: *divide-inverse inverse-mult algebra-simps*)

lemma *divide-pow*: (x ^ n) / (y ^ n) = (x / y) ^ n **for** x y :: $'b$
proof (*induct* n)
case (*Suc* n) **thus** *?case* **using** *times-divide-times-eq*[*of* x y x ^ n y ^ n] **by** *force*
qed *simp*

lemma *power-int-divide-distrib*:
power-int (*x* / *y*) *m* = *power-int x m* / *power-int y m* **for** *x y* :: *'b*
by (*cases m* ≥ *0*) (*simp-all add*: *power-int-def divide-pow inverse-divide inverse-div-inverse*)

lemma *power-int-one-over*: *power-int* (*1* / *x*) *n* = *1* / *power-int x n* **for** *x* :: *'b*
by (*auto simp add*: *power-int-inverse simp flip*: *inverse-eq-divide*)

lemma *add-divide-distrib*: (*a* + *b*) / *c* = (*a* / *c*) + (*b* / *c*) **for** *a b c* :: *'b*
by (*simp add*: *divide-inverse algebra-simps*)

lemma *diff-divide-distrib*: (*a* − *b*) / *c* = (*a* / *c*) − (*b* / *c*) **for** *a b c* :: *'b*
by (*simp add*: *divide-inverse algebra-simps*)

lemma *times-divide-eq-right*: *a* ∗ (*b* / *c*) = (*a* ∗ *b*) / *c* **for** *a b c* :: *'b*
using *times-divide-times-eq*[*of a 1*] **by** *force*

lemma *times-divide-eq-left*: (*b* / *c*) ∗ *a* = (*b* ∗ *a*) / *c* **for** *a b c* :: *'b*
using *times-divide-eq-right* **by** (*simp add*: *ac-simps*)

lemma *divide-divide-eq-left*: (*a* / *b*) / *c* = *a* / (*b* ∗ *c*) **for** *a b c* :: *'b*
using *divide-inverse mult.assoc inverse-mult*[*of b c*] **by** *metis*

lemma *swap-numerator*: *a* ∗ (*b* / *c*) = (*a* / *c*) ∗ *b* **for** *a b c* :: *'b*
by (*metis times-divide-eq-right mult.commute*)

lemma *divide-divide-times-eq*: (*a* / *b*) / (*c* / *d*) = (*a* ∗ *d*) / (*b* ∗ *c*) **for** *a b c d* :: *'b*
by (*metis divide-inverse inverse-divide times-divide-times-eq*)

Algebra facts by place

lemma *right-inverse-equiv*: *p-equal p* (*x* ∗ *inverse x*) *1* **if** ¬ *p-equal p x 0*
using *that divide-self-equiv divide-inverse* **by** *force*

lemma *left-inverse-equiv*: *p-equal p* (*inverse x* ∗ *x*) *1* **if** ¬ *p-equal p x 0*
using *that right-inverse-equiv* **by** (*simp add*: *mult.commute*)

lemma *inverse-p-unique*:
p-equal p (*inverse x*) *y* **if** *p-equal p* (*x* ∗ *y*) *1*
proof−
from *that* **have** *x-n0*: ¬ *p-equal p x 0*
using *mult-0-left trans-right zero-p-nequal-one* **by** *blast*
with *that* **have** *p-equal p* (*inverse x* ∗ *x* ∗ *y*) (*inverse x* ∗ *x* ∗ *inverse x*)
by (*metis right-inverse-equiv trans-left mult-left mult.assoc*)
moreover
have *p-equal p* (*inverse x* ∗ *x* ∗ *y*) *y*
and *p-equal p* (*inverse x* ∗ *x* ∗ *inverse x*) (*inverse x*)
using *x-n0 left-inverse-equiv mult-right*

by (*fastforce*, *fastforce*)
ultimately show *p-equal p* (*inverse x*) *y* **using** *cong sym* **by** *blast*
qed

lemma *inverse-equiv-imp-equiv*:
p-equal p (*inverse a*) (*inverse b*) $\Longrightarrow$ *p-equal p a b*
proof (*cases p-equal p a 0*, *metis inverse-equiv0-iff trans-right trans-left*)
case *False*
moreover assume *ab*: *p-equal p* (*inverse a*) (*inverse b*)
ultimately have $\neg$ *p-equal p b 0* **using** *inverse-equiv0-iff trans-right trans-left*
by *meson*
moreover from *False ab* **have**
p-equal p b (*a* $*$ (*inverse b* $*$ *b*))
by (*metis mult-left right-inverse-equiv trans-right mult-right mult-1-left mult.assoc*)
ultimately show *p-equal p a b*
by (*metis left-inverse-equiv mult-left trans sym mult-1-right*)
qed

lemma *inverse-pcong*: *p-equal p* (*inverse x*) (*inverse y*) **if** *p-equal p x y*
proof (
cases p-equal p x 0, *metis that trans-right inverse-equiv0-iff trans-left*,
intro inverse-p-unique
)
case *False* **with** *that* **show** *p-equal p* (*x* $*$ *inverse y*) *1*
using *mult-right right-inverse-equiv cong refl* **by** *blast*
qed

lemma *inverse-equiv-iff-equiv*:
p-equal p (*inverse a*) (*inverse b*) $\longleftrightarrow$ *p-equal p a b*
using *inverse-equiv-imp-equiv inverse-pcong* **by** *blast*

lemma *power-int-equiv-0-iff*:
p-equal p (*power-int x n*) *0* $\longleftrightarrow$ *p-equal p x 0* $\wedge$ $n \neq 0$
proof (*cases* $n \geq 0$)
case *False* **thus** *?thesis*
using *inverse-pow*[*of x nat* ($-n$)] *inverse-equiv0-iff*[*of p x* ^ *nat* ($-$ *n*)]
pow-equiv-0-iff[*of p x nat* ($-$ *n*)]
by (*force simp add*: *power-int-def*)
qed (*simp add*: *pow-equiv-0-iff power-int-def*)

lemma *power-diff-conv-inverse-equiv*:
$m \leq n \Longrightarrow$ *p-equal p* (*x* ^ ($n - m$)) (*x* ^ *n* $*$ *inverse x* ^ *m*)
if $\neg$ *p-equal p x 0*
proof (*induct n*)
case (*Suc n*) **show** *?case*
proof (*cases Suc n* = *m*)
case *True* **show** *?thesis*
proof (*cases m* = *0*)
case *False*

```
      moreover have p-equal p 1 (x ^ m * inverse x ^ m)
        using that pow-equiv-0-iff right-inverse-equiv inverse-pow sym by metis
      ultimately show ?thesis using Suc ‹Suc n = m› by simp
    qed simp
  next
    case False
    with Suc(2) have mn: m ≤ n by linarith
    with Suc(1) have p-equal p (x ^ Suc (n − m)) (x ^ Suc n * inverse x ^ m)
      using mult-left by (simp add: mult.assoc)
    moreover from mn have Suc (n − m) = Suc n − m by linarith
    ultimately show ?thesis by argo
  qed
qed simp

lemma power-int-add-1-equiv:
  p-equal p (power-int x (m + 1)) (power-int x m * x)
  if ¬ p-equal p x 0 ∨ m ≠ −1
proof (cases p-equal p x 0)
  case True
  with that have p-equal p (power-int x (m + 1)) 0 and p-equal p (power-int x
m * x) 0
    using power-int-equiv-0-iff mult-0-right by auto
  thus ?thesis using trans-left by blast
next
  case False show ?thesis
  proof (cases m + 1 rule: int-cases3)
    case zero
    hence m = −1 by auto
    with False show ?thesis using left-inverse-equiv sym power-int-def by auto
  next
    case (pos n)
    hence nat (m + 1) = Suc (nat m) by auto
    moreover from pos have power-int x (m + 1) = x ^ nat (m + 1) by auto
    ultimately have power-int x (m + 1) = power-int x m * x by (simp add:
power-int-def)
    thus ?thesis by fastforce
  next
    case (neg n)
    hence nat (− m) = Suc n by fastforce
    hence power-int x m * x = inverse x ^ n * (inverse x * x) unfolding
power-int-def by auto
    with False neg show ?thesis
      using mult-left left-inverse-equiv mult-1-right sym unfolding power-int-def
by fastforce
  qed
qed

lemma power-int-add-equiv:
  ¬ p-equal p x 0 ∨ m + n ≠ 0 ⟹
```

p-equal p (*power-int x* (*m* + *n*)) (*power-int x m* ∗ *power-int x n*)
proof (*induct m n rule*: *linorder-wlog*)
fix *m n* :: *int* **assume** *mn*: *m* ≤ *n* **and** *x*: ¬ *p-equal p x 0* ∨ *m* + *n* ≠ *0*
show *p-equal p* (*power-int x* (*m* + *n*)) (*power-int x m* ∗ *power-int x n*)
proof (*cases p-equal p x 0*)
case *True*
moreover from *this x* **have** *p-equal p* (*power-int x m*) *0* ∨ *p-equal p* (*power-int x n*) *0*
using *power-int-equiv-0-iff* **by** *presburger*
ultimately show *?thesis*
using *x mult-0-left mult-0-right power-int-equiv-0-iff trans-left* **by** *meson*
next
case *False*
have
m ≤ *n* ⟹
p-equal p (*power-int x* (*m* + *n*)) (*power-int x m* ∗ *power-int x n*)
proof (*induct n rule*: *int-ge-induct*)
case *base* **show** *?case*
proof (*cases m* ≥ *0*)
case *True* **thus** *?thesis*
using *nat-add-distrib*[*of m*] *power-add*[*of x*] **unfolding** *power-int-def* **by** *fastforce*
next
case *False* **thus** *?thesis*
using *nat-add-distrib*[*of* −*m* −*m*] *power-add*[*of inverse x*]
unfolding *power-int-def*
by *auto*
qed
next
case (*step n*)
from *False* **have** *p-equal p* (*x powi* (*m* + (*n* + *1*))) (*x powi* (*m* + *n*) ∗ *x*)
using *power-int-add-1-equiv*[*of p x m* + *n*] **by** (*simp add*: *add.assoc*)
with *step*(*2*) *False* **have** *p-equal p* (*x powi* (*m* + (*n* + *1*))) (*x powi m* ∗ (*x powi n* ∗ *x*))
using *mult-right trans* **by** (*force simp flip*: *mult.assoc*)
with *False* **show** *p-equal p* (*x powi* (*m* + (*n* + *1*))) (*x powi m* ∗ *x powi* (*n* + *1*))
using *trans power-int-add-1-equiv sym mult-left* **by** *meson*
qed
with *mn* **show** *?thesis* **by** *fast*
qed
qed (*simp add*: *ac-simps*)

lemma *power-int-diff-equiv*:
p-equal p (*power-int x* (*m* − *n*)) (*power-int x m* / *power-int x n*)
if ¬ *p-equal p x 0* ∨ *m* ≠ *n*
using *that power-int-add-equiv*[*of p x m* −*n*] *power-int-minus-divide times-divide-eq-right*
by *auto*

lemma *power-int-minus-mult-equiv*:
  p-equal p (*power-int x* (*n* − *1*) ∗ *x*) (*power-int x n*)
  if ¬ *p-equal p x 0* ∨ *n* ≠ *0*
  using *that sym power-int-add-1-equiv*[*of p x n* − *1*] **by** *auto*

lemma *power-int-not-equiv-zero*:
  ¬ *p-equal p x 0* ∨ *n* = *0* ⟹ ¬ *p-equal p* (*power-int x n*) *0*
  by (*subst power-int-equiv-0-iff*) *auto*

lemma *power-int-equiv-base*: *p-equal p* (*power-int x n*) (*power-int y n*) **if** *p-equal p x y*
  by (
    cases n ≥ *0*, *metis that power-int-def pow-equiv-base*,
    metis that inverse-pcong power-int-def pow-equiv-base
)

lemma *divide-equiv-0-iff*:
  p-equal p (*x* / *y*) *0* ⟷ *p-equal p x 0* ∨ *p-equal p y 0*
  using *divide-inverse mult-equiv-0-iff inverse-equiv0-iff* **by** *metis*

lemma *divide-pcong*: *p-equal p* (*w* / *x*) (*y* / *z*) **if** *p-equal p w y* **and** *p-equal p x z*
  by (*metis that divide-inverse inverse-pcong mult*)

lemma *divide-left-equiv*: *p-equal p x y* ⟹ *p-equal p* (*x* / *z*) (*y* / *z*)
  by (*metis mult-right divide-inverse*)

lemma *divide-right-equiv*: *p-equal p x y* ⟹ *p-equal p* (*z* / *x*) (*z* / *y*)
  by (*metis inverse-pcong mult-left divide-inverse*)

lemma *add-frac-equiv*:
  p-equal p ((*a* / *b*) + (*c* / *d*)) ((*a* ∗ *d* + *c* ∗ *b*) / (*b* ∗ *d*))
  if ¬ *p-equal p b 0* **and** ¬ *p-equal p d 0*
proof−
  from *that* **have**
    (*a* ∗ *d* + *c* ∗ *b*) / (*b* ∗ *d*) =
      a ∗ *d* ∗ (*inverse b* ∗ *inverse d*) + *c* ∗ *b* ∗ (*inverse b* ∗ *inverse d*)
    by (*metis add-divide-distrib divide-inverse inverse-mult*)
  with *that* **show** *?thesis*
    using *mult-left*[*of p d* ∗ *inverse d 1 a* / *b*] *mult-right*[*of p b* ∗ *inverse b 1 c* / *d*]
    by (*simp add*: *ac-simps divide-inverse right-inverse-equiv add sym*)
qed

lemma *mult-divide-mult-cancel-right*:
  p-equal p ((*a* ∗ *b*) / (*c* ∗ *b*)) (*a* / *c*) **if** ¬ *p-equal p b 0*
proof−
  have (*a* ∗ *b*) / (*c* ∗ *b*) = *a* ∗ *b* ∗ (*inverse c* ∗ *inverse b*)
    by (*metis divide-inverse inverse-mult*)
  with *that* **show** *?thesis*
    using *mult-left*[*of p b* ∗ *inverse b 1 a* / *c*]

by (*simp add*: *ac-simps divide-inverse right-inverse-equiv sym*)
qed

lemma *mult-divide-mult-cancel-right2*:
p-equal p ((*a* ∗ *c*) / (*c* ∗ *b*)) (*a* / *b*) **if** ¬ *p-equal p c 0*
by (*metis that mult-divide-mult-cancel-right mult.commute*)

lemma *mult-divide-mult-cancel-left*:
p-equal p ((*c* ∗ *a*) / (*c* ∗ *b*)) (*a* / *b*) **if** ¬ *p-equal p c 0*
by (*metis that mult-divide-mult-cancel-right mult.commute*)

lemma *mult-divide-mult-cancel-left2*:
p-equal p ((*c* ∗ *a*) / (*b* ∗ *c*)) (*a* / *b*) **if** ¬ *p-equal p c 0*
by (*metis that mult-divide-mult-cancel-left mult.commute*)

lemma *mult-divide-cancel-right*: *p-equal p* ((*a* ∗ *b*) / *b*) *a* **if** ¬ *p-equal p b 0*
using *that mult-divide-mult-cancel-right*[*of p b a 1*] **by** *auto*

lemma *mult-divide-cancel-left*: *p-equal p* ((*a* ∗ *b*) / *a*) *b* **if** ¬ *p-equal p a 0*
using *that mult-divide-cancel-right mult.commute* **by** *metis*

lemma *divide-equiv-equiv*: (*p-equal p* (*b* / *c*) *a*) = (*p-equal p b* (*a* ∗ *c*)) **if** ¬ *p-equal p c 0*
proof
assume *p-equal p* (*b* / *c*) *a*
hence *p-equal p* (*b* ∗ (*inverse c* ∗ *c*)) (*a* ∗ *c*) **by** (*metis divide-inverse mult-right mult.assoc*)
with *that* **show** *p-equal p b* (*a* ∗ *c*)
by (*metis left-inverse-equiv mult-left mult-1-right trans-right*)
next
assume *p-equal p b* (*a* ∗ *c*)
hence *p-equal p* (*b* / *c*) (*a* ∗ (*c* ∗ *inverse c*))
by (*metis mult-right mult.assoc divide-inverse*)
with *that* **show** *p-equal p* (*b* / *c*) *a*
using *mult-left right-inverse-equiv trans*
by *fastforce*
qed

lemma *neg-divide-equiv-equiv*:
p-equal p (− (*b* / *c*)) *a* ⟷ *p-equal p* (−*b*) (*a* ∗ *c*) **if** ¬ *p-equal p c 0*
using *that uminus-iff*[*of p* − (*b* / *c*) *a*] *divide-equiv-equiv uminus-iff*[*of p b*] **by** *force*

lemma *equiv-divide-imp*:
p-equal p (*a* ∗ *c*) *b* ⟹ *p-equal p a* (*b* / *c*) **if** ¬ *p-equal p c 0*
using *that divide-left-equiv mult-divide-cancel-right trans-right* **by** *blast*

lemma *add-divide-equiv-iff*:
p-equal p (*x* + *y* / *z*) ((*x* ∗ *z* + *y*) / *z*) **if** ¬ *p-equal p z 0*

using *that add sym mult-divide-cancel-right* **by** (*fastforce simp add*: *add-divide-distrib*)

lemma *divide-add-equiv-iff*: *p-equal p* (*x* / *z* + *y*) ((*x* + *y* ∗ *z*) / *z*) **if** ¬ *p-equal p z 0*
using *that add-divide-equiv-iff*[*of p z y x*] **by** (*simp add*: *ac-simps*)

lemma *diff-divide-equiv-iff*: *p-equal p* (*x* − *y* / *z*) ((*x* ∗ *z* − *y*) / *z*) **if** ¬ *p-equal p z 0*
using *that minus sym mult-divide-cancel-right* **by** (*fastforce simp add*: *diff-divide-distrib*)

lemma *minus-divide-add-equiv-iff*:
p-equal p (− (*x* / *z*) + *y*) ((− *x* + *y* ∗ *z*) / *z*) **if** ¬ *p-equal p z 0*
by (*metis that minus-divide-left divide-add-equiv-iff*)

lemma *divide-diff-equiv-iff*: *p-equal p* (*x* / *z* − *y*) ((*x* − *y* ∗ *z*) / *z*) **if** ¬ *p-equal p z 0*
by (*metis that diff-conv-add-uminus minus-mult-left divide-add-equiv-iff*)

lemma *minus-divide-diff-equiv-iff*:
p-equal p (− (*x* / *z*) − *y*) ((− *x* − *y* ∗ *z*) / *z*) **if** ¬ *p-equal p z 0*
by (*metis that divide-diff-equiv-iff minus-divide-left*)

lemma *divide-mult-cancel-left*: *p-equal p* (*a* / (*a* ∗ *b*)) (*1* / *b*) **if** ¬ *p-equal p a 0*
by (*metis that divide-divide-eq-left divide-left-equiv divide-self-equiv*)

lemma *divide-mult-cancel-right*: *p-equal p* (*b* / (*a* ∗ *b*)) (*1* / *a*) **if** ¬ *p-equal p b 0*
by (*metis that divide-mult-cancel-left mult.commute*)

lemma *diff-frac-equiv*:
p-equal p (*x* / *y* − *w* / *z*) ((*x* ∗ *z* − *w* ∗ *y*) / (*y* ∗ *z*))
if ¬ *p-equal p y 0* **and** ¬ *p-equal p z 0*
proof−
from *that* **have** *p-equal p* (*x* / *y* − *w* / *z*) ((*x* ∗ *z* + − (*w* ∗ *y*)) / (*y* ∗ *z*))
using *diff-conv-add-uminus minus-divide-left add-frac-equiv minus-mult-left* **by** *metis*
thus *?thesis* **by** *force*
qed

lemma *frac-equiv-equiv*:
p-equal p (*x* / *y*) (*w* / *z*) ⟷ *p-equal p* (*x* ∗ *z*) (*w* ∗ *y*)
if ¬ *p-equal p y 0* **and** ¬ *p-equal p z 0*
by (*metis that divide-equiv-equiv times-divide-eq-left sym*)

lemma *inverse-equiv-1-iff*:
p-equal p (*inverse x*) *1* ⟷ *p-equal p x 1*
using *one-p-nequal-zero trans-right inverse-equiv0-iff mult-left right-inverse-equiv mult-1-right*
mult-right sym mult-1-left
by *metis*

lemma *inverse-neg-1-equiv*: *p-equal p* (*inverse* (−*1*)) (−*1*)
  using *inverse-p-unique* **by** *auto*

lemma *globally-inverse-neg-1*: *globally-p-equal* (*inverse* (−*1*)) (−*1*)
  using *inverse-neg-1-equiv* **by** *blast*

lemma *inverse-uminus-equiv*: *p-equal p* (*inverse* (−*x*)) (− *inverse x*)
  using *inverse-mult*[*of* −*1 x*] *inverse-neg-1-equiv mult-right* **by** *fastforce*

lemma *minus-divide-right-equiv*: *p-equal p* (*a* / (− *b*)) (− (*a* / *b*))
  using *divide-inverse inverse-uminus-equiv mult-left trans* **by** *fastforce*

lemma *minus-divide-divide-equiv*: *p-equal p* ((− *a*) / (− *b*)) (*a* / *b*)
  using *refl minus-divide-left minus-divide-right-equiv trans* **by** *fastforce*

lemma *divide-minus1-equiv*: *p-equal p* (*x* / (−*1*)) (− *x*)
  using *minus-divide-right-equiv*[*of p x 1*] **by** *simp*

lemma *divide-cancel-right*:
  p-equal p (*a* / *c*) (*b* / *c*) ⟷ *p-equal p c 0* ∨ *p-equal p a b*
proof
  assume *p-equal p* (*a* / *c*) (*b* / *c*)
  hence *p-equal p* ((*c* ∗ *a*) / *c*) ((*c* ∗ *b*) / *c*)
    using *mult-left times-divide-eq-right* **by** *metis*
  thus *p-equal p c 0* ∨ *p-equal p a b* **using** *mult-divide-cancel-left cong* **by** *blast*
qed (*metis divide-left-equiv divide-equiv-0-iff trans-left*)

lemma *divide-cancel-left*:
  p-equal p (*c* / *a*) (*c* / *b*) ⟷ *p-equal p c 0* ∨ *p-equal p a b*
proof
  assume ∗: *p-equal p* (*c* / *a*) (*c* / *b*)
  show *p-equal p c 0* ∨ *p-equal p a b*
  proof (*cases p-equal p a 0 p-equal p c 0 rule*: *case-split*[*case-product case-split*])
    case *False-False*
    from ∗ *False-False*(*1*) **have** *p-equal p* ((*c* ∗ *a*) / *c*) ((*c* ∗ *b*) / *c*)
      using *divide-equiv-0-iff trans divide-equiv-equiv times-divide-eq-left sym divide-left-equiv*
     by *metis*
    with *False-False*(*2*) **show** *?thesis* **using** *mult-divide-cancel-left cong* **by** *blast*
  qed (*simp*, *metis* ∗ *divide-equiv-0-iff trans0-iff trans-left*, *simp*)
qed (*metis divide-right-equiv divide-equiv-0-iff trans-left*)

lemma *divide-equiv-1-iff*:
  p-equal p (*a* / *b*) *1* ⟷ ¬ *p-equal p b 0* ∧ *p-equal p a b*
proof (*standard*, *standard*)
  assume *p-equal p* (*a* / *b*) *1*
  moreover from *this* **show** ¬ *p-equal p b 0*
    using *divide-equiv-0-iff zero-p-nequal-one trans-right* **by** *metis*

ultimately show *p-equal p a b*
using *mult-left mult-1-right times-divide-eq-right mult-divide-cancel-left trans-right*
by *metis*
qed (*metis divide-left-equiv divide-self-equiv trans*)

lemma *divide-equiv-minus-1-iff*:
$p\text{-}equal\ p\ (a\ /\ b)\ (-\ 1) \longleftrightarrow \neg\ p\text{-}equal\ p\ b\ 0 \wedge p\text{-}equal\ p\ a\ (-\ b)$
proof−
have $p\text{-}equal\ p\ (a\ /\ b)\ (-\ 1) \longleftrightarrow p\text{-}equal\ p\ (-\ (a\ /\ b))\ 1$
using *uminus-iff* **by** *fastforce*
also have $\ldots \longleftrightarrow \neg\ p\text{-}equal\ p\ (-\ b)\ 0 \wedge p\text{-}equal\ p\ a\ (-\ b)$
using *minus-divide-right-equiv sym trans-right divide-equiv-1-iff* **by** *blast*
finally show *?thesis* **using** *uminus-iff* **by** *fastforce*
qed

end

3.5 Indexed depth functions

3.5.1 General structure

locale *p-equality-depth* = *p-equality* +
fixes $p\text{-}depth :: 'a \Rightarrow 'b{::}comm\text{-}ring \Rightarrow int$
assumes *depth-of-0*[*simp*]: $p\text{-}depth\ p\ 0 = 0$
and *depth-equiv*: $p\text{-}equal\ p\ x\ y \Longrightarrow p\text{-}depth\ p\ x = p\text{-}depth\ p\ y$
and *depth-uminus*: $p\text{-}depth\ p\ (-x) = p\text{-}depth\ p\ x$
and *depth-pre-nonarch*:
$\neg\ p\text{-}equal\ p\ x\ 0 \Longrightarrow p\text{-}depth\ p\ x < p\text{-}depth\ p\ y \Longrightarrow$
$p\text{-}depth\ p\ (x + y) = p\text{-}depth\ p\ x$
$[\![\ \neg\ p\text{-}equal\ p\ x\ 0;\ \neg\ p\text{-}equal\ p\ (x + y)\ 0;\ p\text{-}depth\ p\ x = p\text{-}depth\ p\ y\ ]\!]$
$\Longrightarrow p\text{-}depth\ p\ (x + y) \geq p\text{-}depth\ p\ x$
begin

lemma *depth-equiv-0*: $p\text{-}equal\ p\ x\ 0 \Longrightarrow p\text{-}depth\ p\ x = 0$
using *depth-of-0 depth-equiv* **by** *presburger*

lemma *depth-equiv-uminus*: $p\text{-}equal\ p\ y\ (-x) \Longrightarrow p\text{-}depth\ p\ y = p\text{-}depth\ p\ x$
using *depth-equiv depth-uminus* **by** *force*

lemma *depth-add-equiv0-left*: $p\text{-}equal\ p\ x\ 0 \Longrightarrow p\text{-}depth\ p\ (x + y) = p\text{-}depth\ p\ y$
using *add-equiv0-left depth-equiv* **by** *blast*

lemma *depth-add-equiv0-right*: $p\text{-}equal\ p\ y\ 0 \Longrightarrow p\text{-}depth\ p\ (x + y) = p\text{-}depth\ p\ x$
using *add-equiv0-right depth-equiv* **by** *blast*

lemma *depth-add-equiv*:
$p\text{-}depth\ p\ (x + y) = p\text{-}depth\ p\ (w + z)$ **if** *p-equal p x w* **and** *p-equal p y z*
using *that add depth-equiv* **by** *auto*

lemma *depth-diff*: $p\text{-}depth\ p\ (x - y) = p\text{-}depth\ p\ (y - x)$

using *depth-uminus*[*of p y* $-$ *x*] **by** *auto*

lemma *depth-diff-equiv0-left*: *p-equal p x 0* $\Longrightarrow$ *p-depth p* $(x - y)$ = *p-depth p y*
using *depth-add-equiv0-right*[*of p x* $-y$] *depth-uminus* **by** *fastforce*

lemma *depth-diff-equiv0-right*: *p-equal p y 0* $\Longrightarrow$ *p-depth p* $(x - y)$ = *p-depth p x*
using *depth-diff-equiv0-left*[*of p y x*] *depth-uminus*[*of p x* $-$ *y*] **by** *simp*

lemma *depth-diff-equiv*:
p-depth p $(x - y)$ = *p-depth p* $(w - z)$ **if** *p-equal p x w* **and** *p-equal p y z*
using *that minus depth-equiv* **by** *auto*

lemma *depth-pre-nonarch-diff-left*:
p-depth p $(x - y)$ = *p-depth p x* **if** ¬ *p-equal p x 0* **and** *p-depth p x* < *p-depth p y*
using *that depth-pre-nonarch*(*1*)[*of p x* $-y$] *depth-uminus* **by** *simp*

lemma *depth-pre-nonarch-diff-right*:
p-depth p $(x - y)$ = *p-depth p y* **if** ¬ *p-equal p y 0* **and** *p-depth p y* < *p-depth p x*
using *that depth-pre-nonarch-diff-left*[*of p y x*] *depth-uminus*[*of p x* $-$ *y*] **by** *simp*

lemma *depth-nonarch*:
⟦ ¬ *p-equal p x 0*; ¬ *p-equal p y 0*; ¬ *p-equal p* $(x + y)$ *0* ⟧
$\Longrightarrow$ *p-depth p* $(x + y) \geq$ *min* (*p-depth p x*) (*p-depth p y*)
using *depth-pre-nonarch*[*of p x y*] *depth-pre-nonarch*[*of p y x*]
by (*cases p-depth p x p-depth p y rule*: *linorder-cases*, *auto simp add*: *add.commute*)

lemma *depth-nonarch-diff*:
⟦ ¬ *p-equal p x 0*; ¬ *p-equal p y 0*; ¬ *p-equal p* $(x - y)$ *0* ⟧
$\Longrightarrow$ *p-depth p* $(x - y) \geq$ *min* (*p-depth p x*) (*p-depth p y*)
using *depth-nonarch*[*of p x* $-y$] *depth-uminus uminus-iff* **by** *fastforce*

lemma *depth-sum-nonarch*:
finite A $\Longrightarrow$ ¬ *p-equal p* (*sum f A*) *0* $\Longrightarrow$
p-depth p (*sum f A*) $\geq$ *Min* {*p-depth p* (*f a*) | *a*. *a* ∈ *A* ∧ ¬ *p-equal p* (*f a*) *0* }
proof (*induct A rule*: *finite-induct*)
case (*insert a A*)
define *D*
where *D* ≡ λ*A*. {*p-depth p* (*f a*) |*a*. *a* ∈ *A* ∧ ¬ *p-equal p* (*f a*) *0*}
consider (*fa0*) *p-equal p* (*f a*) *0* |
(*sum0*) *p-equal p* (*sum f A*) *0* |
(*neither*) ¬ *p-equal p* (*f a*) *0* ¬ *p-equal p* (*sum f A*) *0*
by *blast*
thus *p-depth p* (*sum f* (*insert a A*)) $\geq$ *Min* (*D* (*insert a A*))
proof *cases*
case *fa0*
with *insert D-def* **have** *p-depth p* (*sum f A*) $\geq$ *Min* (*D A*) **using** *add* **by** *fastforce*

moreover from *fa0 insert*(*1*,*2*) **have** *p-depth p* (*sum f* (*insert a A*)) = *p-depth p* (*sum f A*)
using *depth-add-equiv0-left* **by** *auto*
moreover from *fa0 D-def* **have** *D* (*insert a A*) = *D A* **by** *blast*
ultimately show *?thesis* **by** *argo*
next
case *sum0*
with *D-def insert*(*1*,*2*,*4*) **have** *p-depth p* (*f a*) ∈ *D* (*insert a A*) **using** *add* **by** *force*
moreover from *insert*(*1*,*2*) *sum0* **have** *p-depth p* (*sum f* (*insert a A*)) = *p-depth p* (*f a*)
using *depth-add-equiv0-right* **by** *fastforce*
ultimately show *?thesis* **using** *D-def insert*(*1*) **by** *force*
next
case *neither*
from *D-def neither*(*1*) **have** *D* (*insert a A*) = *insert* (*p-depth p* (*f a*)) (*D A*) **by** *blast*
hence *Min* (*D* (*insert a A*)) = *Min* (*insert* (*p-depth p* (*f a*)) (*D A*)) **by** *simp*
also have . . . = *min* (*p-depth p* (*f a*)) (*Min* (*D A*))
proof (*rule Min-insert*)
from *D-def insert*(*1*) **show** *finite* (*D A*) **by** *simp*
from *D-def insert*(*1*) *neither*(*2*) **show** *D A* ≠ {} **using** *sum-zeros* **by** *fast*
qed
finally show *?thesis* **using** *neither D-def insert depth-nonarch* **by** *fastforce*
qed
qed *simp*

lemma *obtains-depth-sum-nonarch-witness*:
assumes *finite A* **and** ¬ *p-equal p* (*sum f A*) *0*
obtains *a*
where *a* ∈ *A* **and** ¬ *p-equal p* (*f a*) *0* **and** *p-depth p* (*sum f A*) ≥ *p-depth p* (*f a*)
proof−
define *D* **where** *D* ≡ {*p-depth p* (*f a*) |*a*. *a* ∈ *A* ∧ ¬ *p-equal p* (*f a*) *0*}
from *assms D-def* **obtain** *a*
where *a*: *a* ∈ *A* ¬ *p-equal p* (*f a*) *0 Min D* = *p-depth p* (*f a*)
using *sum-zeros*[*of A p f*] *obtains-Min*[*of D*] **by** *auto*
from *assms D-def a*(*3*) **have** *p-depth p* (*sum f A*) ≥ *p-depth p* (*f a*)
using *depth-sum-nonarch* **by** *fastforce*
with *that a*(*1*,*2*) **show** *thesis* **by** *blast*
qed

lemma *depth-add-cancel-imp-eq-depth*:
p-depth p x = *p-depth p y* **if** ¬ *p-equal p x 0* **and** *p-depth p* (*x* + *y*) > *p-depth p x*
using *that depth-pre-nonarch*(*1*)[*of p x y*] *depth-pre-nonarch*(*1*)[*of p y x*] *depth-add-equiv0-right*
by (*fastforce simp add*: *ac-simps*)

lemma *depth-diff-cancel-imp-eq-depth-left*:

p-depth p x = *p-depth p y* **if** ¬ *p-equal p x 0* **and** *p-depth p* (*x* − *y*) > *p-depth p x*
using *that depth-add-cancel-imp-eq-depth*[*of p x* −*y*] *depth-uminus* **by** *auto*

lemma *depth-diff-cancel-imp-eq-depth-right*:
p-depth p x = *p-depth p y* **if** ¬ *p-equal p y 0* **and** *p-depth p* (*x* − *y*) > *p-depth p y*
by (*metis that depth-diff-cancel-imp-eq-depth-left depth-diff*)

lemma *level-closed-add*:
p-depth p (*x* + *y*) ≥ *n*
if ¬ *p-equal p* (*x* + *y*) *0* **and** *p-depth p x* ≥ *n* **and** *p-depth p y* ≥ *n*
proof−
consider
p-equal p x 0 | *p-equal p y 0* |
(*both-n0*) ¬ *p-equal p x 0* ¬ *p-equal p y 0*
by *blast*
thus *?thesis*
proof *cases*
case *both-n0* **with** *that* **show** *?thesis* **using** *depth-nonarch* **by** *fastforce*
qed (*simp add*: *that*(*3*) *depth-add-equiv0-left*, *simp add*: *that*(*2*) *depth-add-equiv0-right*)
qed

lemma *level-closed-diff*:
p-depth p (*x* − *y*) ≥ *n*
if ¬ *p-equal p* (*x* − *y*) *0* **and** *p-depth p x* ≥ *n* **and** *p-depth p y* ≥ *n*
using *that level-closed-add*[*of p x* −*y*] *depth-uminus* **by** *auto*

definition *p-depth-set* :: *'a* ⇒ *int* ⇒ *'b set*
where *p-depth-set p n* = {*x*. ¬ *p-equal p x 0* ⟶ *p-depth p x* ≥ *n*}

definition *global-depth-set* :: *int* ⇒ *'b set*
where
global-depth-set n =
{*x*. ∀ *p*. ¬ *p-equal p x 0* ⟶ *p-depth p x* ≥ *n*}

lemma *global-depth-set*: *global-depth-set n* = (⋂ *p*. *p-depth-set p n*)
unfolding *p-depth-set-def global-depth-set-def* **by** *auto*

lemma *p-depth-setD*:
¬ *p-equal p x 0* ⟹ *x* ∈ *p-depth-set p n* ⟹
p-depth p x ≥ *n*
unfolding *p-depth-set-def* **by** *blast*

lemma *nonpos-p-depth-setD*:
n ≤ *0* ⟹ *x* ∈ *p-depth-set p n* ⟹ *p-depth p x* ≥ *n*
by (*cases p-equal p x 0*, *simp-all add*: *depth-equiv-0 p-depth-setD*)

lemma *global-depth-setD*:

$\neg$ *p-equal p x 0* $\Longrightarrow$ $x \in$ *global-depth-set n* $\Longrightarrow$
p-depth p x $\geq n$
unfolding *global-depth-set-def* **by** *blast*

lemma *nonpos-global-depth-setD*:
$n \leq 0 \Longrightarrow x \in$ *global-depth-set n* $\Longrightarrow$ *p-depth p x* $\geq n$
by (*cases p-equal p x 0*, *simp-all add*: *depth-equiv-0 global-depth-setD*)

lemma *p-depth-set-0*: $0 \in$ *p-depth-set p n*
unfolding *p-depth-set-def* **by** *fastforce*

lemma *global-depth-set-0*: $0 \in$ *global-depth-set n*
unfolding *global-depth-set-def* **by** *fastforce*

lemma *p-depth-set-equiv0*: *p-equal p x 0* $\Longrightarrow$ $x \in$ *p-depth-set p n*
unfolding *p-depth-set-def* **by** *blast*

lemma *p-depth-setI*:
$x \in$ *p-depth-set p n* **if** $\neg$ *p-equal p x 0* $\longrightarrow$ *p-depth p x* $\geq n$
using *that* **unfolding** *p-depth-set-def* **by** *blast*

lemma *p-depth-set-by-equivI*:
$x \in$ *p-depth-set p n* **if** *p-equal p x y* **and** $y \in$ *p-depth-set p n*
by (*metis that trans p-depth-setD depth-equiv p-depth-setI*)

lemma *global-depth-setI*:
$x \in$ *global-depth-set n*
if $\bigwedge p$. $\neg$ *p-equal p x 0* $\Longrightarrow$ *p-depth p x* $\geq n$
using *that* **unfolding** *global-depth-set-def* **by** *blast*

lemma *global-depth-setI'*: $x \in$ *global-depth-set n* **if** $\bigwedge p$. *p-depth p x* $\geq n$
by (*intro global-depth-setI*, *rule that*)

lemma *global-depth-setI''*: $x \in$ *global-depth-set n* **if** $\bigwedge p$. $x \in$ *p-depth-set p n*
using *that global-depth-set* **by** *blast*

lemma *p-depth-set-antimono*:
$n \leq m \Longrightarrow$ *p-depth-set p n* $\supseteq$ *p-depth-set p m*
unfolding *p-depth-set-def* **by** *force*

lemma *global-depth-set-antimono*:
$n \leq m \Longrightarrow$ *global-depth-set n* $\supseteq$ *global-depth-set m*
unfolding *global-depth-set-def* **by** *force*

lemma *p-depth-set-add*:
$x + y \in$ *p-depth-set p n* **if** $x \in$ *p-depth-set p n* **and** $y \in$ *p-depth-set p n*
proof (*intro p-depth-setI*, *clarify*)
assume *nequiv0*: $\neg$ *p-equal p* $(x + y)$ *0*
consider *p-equal p x 0* | *p-equal p y 0* | $\neg$ *p-equal p x 0* $\neg$ *p-equal p y 0*

  by *blast*
 thus *p-depth p* $(x + y) \geq n$
  by (
   cases, *metis that(2) nequiv0 add-0-left depth-equiv trans p-depth-setD*,
   metis that(1) nequiv0 add-0-right depth-equiv trans p-depth-setD,
   metis that nequiv0 level-closed-add p-depth-setD
)
qed

lemma *global-depth-set-add*:
 $x + y \in$ *global-depth-set n* **if** $x \in$ *global-depth-set n* **and** $y \in$ *global-depth-set n*
 using *that p-depth-set-add global-depth-set* **by** *simp*

lemma *p-depth-set-uminus*: $-x \in$ *p-depth-set p n* **if** $x \in$ *p-depth-set p n*
 using *that uminus depth-uminus* **unfolding** *p-depth-set-def* **by** *fastforce*

lemma *global-depth-set-uminus*: $-x \in$ *global-depth-set n* **if** $x \in$ *global-depth-set n*
 using *that p-depth-set-uminus global-depth-set* **by** *simp*

lemma *p-depth-set-minus*:
 $x - y \in$ *p-depth-set p n* **if** $x \in$ *p-depth-set p n* **and** $y \in$ *p-depth-set p n*
 using *that p-depth-set-add p-depth-set-uminus* **by** *fastforce*

lemma *global-depth-set-minus*:
 $x - y \in$ *global-depth-set n* **if** $x \in$ *global-depth-set n* **and** $y \in$ *global-depth-set n*
 using *that p-depth-set-minus global-depth-set* **by** *simp*

lemma *p-depth-set-elt-set-plus*:
 x *+o p-depth-set p n* = *p-depth-set p n* **if** $x \in$ *p-depth-set p n*
 unfolding *elt-set-plus-def*
proof (*standard*, *safe*)
 fix *y* **assume** *y*: $y \in$ *p-depth-set p n*
 define *z* **where** $z \equiv y - x$
 with *that y* **have** $z \in$ *p-depth-set p n* **using** *p-depth-set-minus* **by** *simp*
 with *z-def* **show** $\exists z \in$*p-depth-set p n*. $y = x + z$ **by** *force*
qed (*simp add*: *that p-depth-set-add*)

lemma *global-depth-set-elt-set-plus*:
 x *+o global-depth-set n* = *global-depth-set n* **if** $x \in$ *global-depth-set n*
 unfolding *elt-set-plus-def*
proof (*standard*, *safe*)
 fix *y* **assume** *y*: $y \in$ *global-depth-set n*
 define *z* **where** $z \equiv y - x$
 with *that y* **have** $z \in$ *global-depth-set n* **using** *global-depth-set-minus* **by** *simp*
 with *z-def* **show** $\exists z \in$*global-depth-set n*. $y = x + z$ **by** *force*
qed (*simp add*: *that global-depth-set-add*)

lemma *p-depth-set-elt-set-plus-closed*:
 x *+o p-depth-set p m* $\subseteq$ *p-depth-set p n* **if** $m \geq n$ **and** $x \in$ *p-depth-set p n*

using *that set-plus-mono p-depth-set-antimono p-depth-set-elt-set-plus* **by** *metis*

lemma *global-depth-set-elt-set-plus-closed*:
x $+o$ *global-depth-set m* $\subseteq$ *global-depth-set n*
if $m \geq n$ **and** $x \in$ *global-depth-set n*
using *that set-plus-mono global-depth-set-antimono global-depth-set-elt-set-plus* **by** *metis*

lemma *p-depth-set-plus-coeffs*:
set xs $\subseteq$ *p-depth-set p n* $\Longrightarrow$
set ys $\subseteq$ *p-depth-set p n* $\Longrightarrow$
set (*plus-coeffs xs ys*) $\subseteq$ *p-depth-set p n*
proof (*induct xs ys rule*: *list-induct2'*)
case (*4 x xs y ys*) **thus** *?case*
using *cCons-def*[*of x* + *y plus-coeffs xs ys*] *p-depth-set-add* **by** *auto*
qed *simp-all*

lemma *global-depth-set-plus-coeffs*:
set (*plus-coeffs xs ys*) $\subseteq$ *global-depth-set n*
if *set xs* $\subseteq$ *global-depth-set n* **and** *set ys* $\subseteq$ *global-depth-set n*
using *that p-depth-set-plus-coeffs*[*of xs - n ys*] *global-depth-set*[*of n*] **by** *fastforce*

lemma *p-depth-set-poly-pCons*:
set (*coeffs* (*pCons a f*)) $\subseteq$ *p-depth-set p n*
if $a \in$ *p-depth-set p n* **and** *set* (*coeffs f*) $\subseteq$ *p-depth-set p n*
by (*cases* $a = 0 \wedge f = 0$) (*simp add*: *p-depth-set-0*, *use that* **in** *auto*)

lemma *p-depth-set-poly-add*:
set (*coeffs* ($f + g$)) $\subseteq$ *p-depth-set p n*
if *set* (*coeffs f*) $\subseteq$ *p-depth-set p n* **and** *set* (*coeffs g*) $\subseteq$ *p-depth-set p n*
using *that coeffs-plus-eq-plus-coeffs p-depth-set-plus-coeffs* **by** *metis*

lemma *global-depth-set-poly-add*:
set (*coeffs* ($f + g$)) $\subseteq$ *global-depth-set n*
if *set* (*coeffs f*) $\subseteq$ *global-depth-set n*
and *set* (*coeffs g*) $\subseteq$ *global-depth-set n*
using *that p-depth-set-poly-add*[*of f - n g*] *global-depth-set* **by** *fastforce*

end

locale *p-equality-depth-no-zero-divisors* =
p-equality-no-zero-divisors + *p-equality-depth*
+ **assumes** *depth-mult-additive*:
$\neg$ *p-equal p* ($x * y$) *0* $\Longrightarrow$ *p-depth p* ($x * y$) = *p-depth p x* + *p-depth p y*
begin

lemma *depth-mult3-additive*:
p-depth p ($x * y * z$) = *p-depth p x* + *p-depth p y* + *p-depth p z*

if ¬ *p-equal p* (*x* ∗ *y* ∗ *z*) *0*
using *that mult-0-left depth-mult-additive* **by** *fastforce*

lemma *p-depth-set-times*:
x ∗ *y* ∈ *p-depth-set p n*
if *n* ≥ *0* **and** *x* ∈ *p-depth-set p n* **and** *y* ∈ *p-depth-set p n*
proof (*intro p-depth-setI*, *clarify*)
assume ¬ *p-equal p* (*x* ∗ *y*) *0*
moreover from *this* **have** ¬ *p-equal p x 0* **and** ¬ *p-equal p y 0*
by (*metis mult-0-left*, *metis mult-0-right*)
ultimately show *p-depth p* (*x* ∗ *y*) ≥ *n*
using *that p-depth-setD*[*of p x*] *p-depth-setD*[*of p y*] *depth-mult-additive* **by** *fastforce*
qed

lemma *global-depth-set-times*:
x ∗ *y* ∈ *global-depth-set n*
if *n* ≥ *0* **and** *x* ∈ *global-depth-set n* **and** *y* ∈ *global-depth-set n*
using *that p-depth-set-times global-depth-set* **by** *simp*

lemma *poly-p-depth-set-poly*:
set (*coeffs f*) ⊆ *p-depth-set p n* ⟹ *poly f x* ∈ *p-depth-set p n*
if *n* ≥ *0* **and** *x* ∈ *p-depth-set p n*
by (*induct f*, *simp add*: *p-depth-set-0*, *use that p-depth-set-times p-depth-set-add* **in** *force*)

lemma *poly-global-depth-set-poly*:
poly f x ∈ *global-depth-set n*
if *n* ≥ *0*
and *x* ∈ *global-depth-set n*
and *set* (*coeffs f*) ⊆ *global-depth-set n*
using *that poly-p-depth-set-poly global-depth-set*
by *fastforce*

lemma *p-depth-set-ideal*:
x ∗ *y* ∈ *p-depth-set p n*
if *x* ∈ *p-depth-set p 0* **and** *y* ∈ *p-depth-set p n*
proof (*intro p-depth-setI*, *clarify*)
assume ¬ *p-equal p* (*x* ∗ *y*) *0*
moreover from *this* **have** ¬ *p-equal p x 0* **and** ¬ *p-equal p y 0*
by (*metis mult-0-left*, *metis mult-0-right*)
ultimately show *p-depth p* (*x* ∗ *y*) ≥ *n*
using *that p-depth-setD*[*of p x*] *p-depth-setD*[*of p y*] *depth-mult-additive* **by** *fastforce*
qed

lemma *global-depth-set-ideal*:
x ∗ *y* ∈ *global-depth-set n*
if *x* ∈ *global-depth-set 0* **and** *y* ∈ *global-depth-set n*

using *that p-depth-set-ideal global-depth-set* **by** *simp*

lemma *poly-p-depth-set-poly-ideal*:
set (*coeffs f*) $\subseteq$ *p-depth-set p n* $\Longrightarrow$ *poly f x* $\in$ *p-depth-set p n*
if $x \in$ *p-depth-set p 0*
using *that p-depth-set-0 p-depth-set-ideal p-depth-set-add* **by** (*induct f*) *auto*

lemma *poly-global-depth-set-poly-ideal*:
poly f x $\in$ *global-depth-set n*
if $x \in$ *global-depth-set 0* **and** *set* (*coeffs f*) $\subseteq$ *global-depth-set n*
using *that poly-p-depth-set-poly-ideal global-depth-set* **by** *fastforce*

lemma *p-depth-set-poly-smult*:
set (*coeffs* (*smult x f*)) $\subseteq$ *p-depth-set p n*
if $n \geq 0$ **and** $x \in$ *p-depth-set p n* **and** *set* (*coeffs f*) $\subseteq$ *p-depth-set p n*
proof−
have *set* (*coeffs* (*smult x f*)) = (∗) *x* ‘ *set* (*strip-while* ($\lambda c.\ x * c = 0$) (*coeffs f*))
using *coeffs-smult-eq-smult-coeffs set-map* **by** *metis*
also have ... $\subseteq$ *set* (*map* ((∗) *x*) (*coeffs f*))
using *set-strip-while image-mono set-map* **by** *fastforce*
finally show *?thesis* **using** *that p-depth-set-times* **by** *fastforce*
qed

lemma *global-depth-set-poly-smult*:
set (*coeffs* (*smult x f*)) $\subseteq$ *global-depth-set n*
if $n \geq 0$
and $x \in$ *global-depth-set n*
and *set* (*coeffs f*) $\subseteq$ *global-depth-set n*
using *that p-depth-set-poly-smult global-depth-set*
by *fastforce*

lemma *p-depth-set-poly-smult-ideal*:
set (*coeffs* (*smult x f*)) $\subseteq$ *p-depth-set p n*
if $x \in$ *p-depth-set p 0* **and** *set* (*coeffs f*) $\subseteq$ *p-depth-set p n*
proof−
have *set* (*coeffs* (*smult x f*)) = (∗) *x* ‘ *set* (*strip-while* ($\lambda c.\ x * c = 0$) (*coeffs f*))
using *coeffs-smult-eq-smult-coeffs set-map* **by** *metis*
also have ... $\subseteq$ *set* (*map* ((∗) *x*) (*coeffs f*))
using *set-strip-while image-mono set-map* **by** *fastforce*
finally show *?thesis* **using** *that p-depth-set-ideal* **by** *fastforce*
qed

lemma *global-depth-set-poly-smult-ideal*:
set (*coeffs* (*smult x f*)) $\subseteq$ *global-depth-set n*
if $x \in$ *global-depth-set 0* **and** *set* (*coeffs f*) $\subseteq$ *global-depth-set n*
using *that p-depth-set-poly-smult-ideal global-depth-set* **by** *fastforce*

lemma *p-depth-set-poly-times*:
set (*coeffs f*) $\subseteq$ *p-depth-set p n* $\Longrightarrow$

set (*coeffs* *g*) ⊆ *p-depth-set* *p* *n* ⟹
    set (*coeffs* (*f* ∗ *g*)) ⊆ *p-depth-set* *p* *n*
  if *n* ≥ *0*
proof (*induct f*)
  case (*pCons a f*)
  hence *set* (*coeffs* (*f* ∗ *g*)) ⊆ *p-depth-set* *p* *n* **by** *auto*
  moreover from *pCons*(*1*,*3*) **have** *a* ∈ *p-depth-set* *p* *n* **by** *fastforce*
  ultimately show *?case*
   using *that pCons*(*4*) *mult-pCons-left p-depth-set-poly-add p-depth-set-poly-smult*
        p-depth-set-poly-pCons p-depth-set-0
    by *auto*
qed *simp*

lemma *global-depth-set-poly-times*:
  set (*coeffs* (*f* ∗ *g*)) ⊆ *global-depth-set* *n*
  if *n* ≥ *0*
  and *set* (*coeffs* *g*) ⊆ *global-depth-set* *n*
  and *set* (*coeffs* *f*) ⊆ *global-depth-set* *n*
  using *that p-depth-set-poly-times*[*of n f - g*] *global-depth-set* **by** *fastforce*

lemma *p-depth-set-poly-times-ideal*:
  set (*coeffs* *f*) ⊆ *p-depth-set* *p* *0* ⟹
    set (*coeffs* *g*) ⊆ *p-depth-set* *p* *n* ⟹
    set (*coeffs* (*f* ∗ *g*)) ⊆ *p-depth-set* *p* *n*
proof (*induct f*)
  case (*pCons a f*)
  hence *set* (*coeffs* (*f* ∗ *g*)) ⊆ *p-depth-set* *p* *n* **by** *auto*
  moreover from *pCons*(*1*,*3*) **have** *a* ∈ *p-depth-set* *p* *0* **by** *fastforce*
  ultimately show *?case*
   using *pCons*(*4*) *mult-pCons-left p-depth-set-poly-add p-depth-set-poly-smult-ideal*
        p-depth-set-poly-pCons p-depth-set-0
    by *simp*
qed *simp*

lemma *global-depth-set-poly-times-ideal*:
  set (*coeffs* (*f* ∗ *g*)) ⊆ *global-depth-set* *n*
  if *set* (*coeffs* *f*) ⊆ *global-depth-set* *0*
  and *set* (*coeffs* *g*) ⊆ *global-depth-set* *n*
  using *that p-depth-set-poly-times-ideal*[*of f - g*] *global-depth-set* **by** *fastforce*

end

locale *p-equality-depth-no-zero-divisors-1* =
  p-equality-no-zero-divisors-1 + *p-equality-depth-no-zero-divisors*
begin

lemma *depth-of-1*[*simp*]: *p-depth* *p* *1* = *0*
  using *one-p-nequal-zero depth-mult-additive*[*of p 1*] **by** *auto*

lemma *depth-of-neg1*: *p-depth p* (-1) = *0*
  using *depth-uminus* **by** *auto*

lemma *depth-pow-additive*: *p-depth p* $(x \mathbin{\hat{}} n)$ = *int n* ∗ *p-depth p x*
proof (*cases p-equal p x 0*)
  case *True* **thus** *?thesis* **using** *pow-equiv-0-iff depth-equiv-0* **by** (*cases n* = *0*) *auto*
next
  case *False* **show** *?thesis*
  proof (*induct n*)
    case (*Suc n*)
    from *False* **have** *p-depth p* $(x \mathbin{\hat{}} Suc\ n)$ = *p-depth p x* + *p-depth p* $(x \mathbin{\hat{}} n)$
      using *pow-equiv-0-iff depth-mult-additive* **by** *force*
    also from *Suc* **have** ... = (*int n* + *1*) ∗ *p-depth p x* **by** *algebra*
    finally show *?case* **by** *auto*
  qed *simp*
qed

lemma *depth-prod-summative*:
  finite X $\Longrightarrow$ ¬ *p-equal p* ($\prod$ *X*) *0* $\Longrightarrow$
    p-depth p ($\prod$ *X*) = *sum* (*p-depth p*) *X*
  by (*induct X rule*: *finite-induct*, *simp*, *use depth-mult-additive mult-0-right* **in** *force*)

lemma *p-depth-set-1*: *1* ∈ *p-depth-set p 0*
  using *p-depth-setI* **by** *simp*

lemma *pos-p-depth-set-1*: *n* > *0* $\Longrightarrow$ *1* ∉ *p-depth-set p n*
  using *one-p-nequal-zero p-depth-setD* **by** *fastforce*

lemma *global-depth-set-1*: *1* ∈ *global-depth-set 0*
  using *p-depth-set-1 global-depth-set* **by** *blast*

lemma *pos-global-depth-set-1*: *n* > *0* $\Longrightarrow$ *1* ∉ *global-depth-set n*
  using *pos-p-depth-set-1 global-depth-set* **by** *blast*

lemma *p-depth-set-neg1*: −*1* ∈ *p-depth-set p 0*
  using *depth-of-neg1 p-depth-setI* **by** *simp*

lemma *global-depth-set-neg1*: −*1* ∈ *global-depth-set 0*
  using *p-depth-set-neg1 global-depth-set* **by** *blast*

lemma *p-depth-set-pow*:
  $x \mathbin{\hat{}} Suc\ k$ ∈ *p-depth-set p n* **if** $n \geq 0$ **and** *x* ∈ *p-depth-set p n*
  by (*induct k*, *simp-all add*: *that p-depth-set-times*)

lemma *global-depth-set-pow*:
  $x \mathbin{\hat{}} Suc\ k$ ∈ *global-depth-set n* **if** $n \geq 0$ **and** *x* ∈ *global-depth-set n*

using *that p-depth-set-pow global-depth-set* **by** *simp*

lemma *p-depth-set-0-pow*: $x \,\hat{}\, k \in$ *p-depth-set p 0* **if** $x \in$ *p-depth-set p 0*
by (*induct k*, *simp-all add*: *that p-depth-set-1 p-depth-set-times*)

lemma *global-depth-set-0-pow*: $x \,\hat{}\, k \in$ *global-depth-set 0* **if** $x \in$ *global-depth-set 0*
using *that p-depth-set-0-pow global-depth-set* **by** *simp*

lemma *p-depth-set-of-nat*: *of-nat n* $\in$ *p-depth-set p 0*
using *p-depth-set-0 p-depth-set-1 p-depth-set-add* **by** (*induct n*) *simp-all*

lemma *global-depth-set-of-nat*: *of-nat n* $\in$ *global-depth-set 0*
using *p-depth-set-of-nat global-depth-set* **by** *blast*

lemma *p-depth-set-of-int*: *of-int n* $\in$ *p-depth-set p 0*
using *p-depth-set-of-nat p-depth-set-neg1 p-depth-set-minus* **by** (*induct n*) *simp-all*

lemma *global-depth-set-of-int*: *of-int n* $\in$ *global-depth-set 0*
using *p-depth-set-of-int global-depth-set* **by** *blast*

lemma *p-depth-set-polyderiv*:
set (*coeffs f*) $\subseteq$ *p-depth-set p n* $\Longrightarrow$
set (*coeffs* (*polyderiv f*)) $\subseteq$ *p-depth-set p n*
proof (*induct f*)
case (*pCons a f*)
from *pCons*(*1*) **have**
set (*coeffs* (*polyderiv* (*pCons a f*))) =
set (*plus-coeffs* (*coeffs f*) (*coeffs* (*pCons 0* (*polyderiv f*))))
using *polyderiv-pCons coeffs-plus-eq-plus-coeffs* **by** *metis*
moreover from *pCons*(*1*,*3*) **have** *set* (*coeffs f*) $\subseteq$ *p-depth-set p n* **by** *fastforce*
ultimately show *?case*
using *pCons*(*2*) *p-depth-set-poly-pCons p-depth-set-0 p-depth-set-plus-coeffs* **by** *presburger*
qed *simp*

lemma *global-depth-set-polyderiv*:
set (*coeffs* (*polyderiv f*)) $\subseteq$ *global-depth-set n*
if *set* (*coeffs f*) $\subseteq$ *global-depth-set n*
using *that p-depth-set-polyderiv global-depth-set* **by** *fastforce*

lemma *poly-p-depth-set-polyderiv*:
poly (*polyderiv f*) $x \in$ *p-depth-set p n*
if $n \geq 0$ **and** *set* (*coeffs f*) $\subseteq$ *p-depth-set p n* **and** $x \in$ *p-depth-set p n*
using *that poly-p-depth-set-poly p-depth-set-polyderiv* **by** *blast*

lemma *poly-global-depth-set-polyderiv*:
poly (*polyderiv f*) $x \in$ *global-depth-set n*
if $n \geq 0$
and *set* (*coeffs f*) $\subseteq$ *global-depth-set n*

and $x \in$ *global-depth-set n*
using *that poly-p-depth-set-polyderiv global-depth-set* **by** *fastforce*

lemma *p-set-depth-additive-poly-poly*:
set (*coeffs f*) $\subseteq$ *p-depth-set p m* $\Longrightarrow$
set (*coeffs* (*coeff* (*additive-poly-poly f*) *n*)) $\subseteq$ *p-depth-set p m*
if $m \geq 0$
proof (*induct f arbitrary*: *n rule*: *pCons-induct'*)
case (*pCons0 a*) **thus** *?case* **using** *additive-poly-poly-pCons0* **by** (*cases n*, *fastforce*, *fastforce*)
next
case (*pCons a f*) **show** *?case*
proof (*cases n*)
case *0* **with** *pCons*(*1*,*3*) **show** *?thesis* **by** (*simp add*: *additive-poly-poly-coeff0 cCons-def*)
next
case (*Suc k*)
moreover have
set (*coeffs* (*pCons 0 1* $*$ *coeff* (*additive-poly-poly f*) *n* + *coeff* (*additive-poly-poly f*) *k*))
$\subseteq$ *p-depth-set p m*
by (
intro p-depth-set-poly-add p-depth-set-poly-times-ideal that pCons(*2*),
simp add: *p-depth-set-0 p-depth-set-1*, *use pCons*(*1*,*3*) **in** *auto*
)
ultimately show
set (*coeffs* (*coeff* (*additive-poly-poly* (*pCons a f*)) *n*)) $\subseteq$ *p-depth-set p m*
using *pCons*(*1*) *additive-poly-poly-pCons coeff-add coeff-smult coeff-pCons-Suc* **by** *fastforce*
qed
qed *simp*

lemma *global-depth-set-additive-poly-poly*:
set (*coeffs* (*coeff* (*additive-poly-poly f*) *n*)) $\subseteq$ *global-depth-set m*
if $m \geq 0$ **and** *set* (*coeffs f*) $\subseteq$ *global-depth-set m*
using *that p-set-depth-additive-poly-poly global-depth-set* **by** *fastforce*

lemma *poly-p-depth-set-additive-poly-poly*:
poly (*coeff* (*additive-poly-poly f*) *n*) $x \in$ *p-depth-set p m*
if $m \geq 0$ **and** *set* (*coeffs f*) $\subseteq$ *p-depth-set p m* **and** $x \in$ *p-depth-set p m*
using *that poly-p-depth-set-poly p-set-depth-additive-poly-poly* **by** *auto*

lemma *poly-global-depth-set-additive-poly-poly*:
poly (*coeff* (*additive-poly-poly f*) *n*) $x \in$ *global-depth-set m*
if $m \geq 0$
and *set* (*coeffs f*) $\subseteq$ *global-depth-set m*
and $x \in$ *global-depth-set m*
using *that poly-p-depth-set-additive-poly-poly global-depth-set* **by** *fastforce*

lemma *sum-poly-additive-poly-poly-depth-bound*:
fixes $p :: 'a$ **and** $f :: 'b$ *poly* **and** $x\ y :: 'b$ **and** $a\ b\ n :: nat$
defines $g \equiv \lambda i.$ *poly* (*coeff* (*additive-poly-poly f*) (*Suc i*)) $x * y$ ^ (*Suc i*)
defines $S \equiv$ *sum g* $\{a..b\}$
assumes *coeffs*: *set* (*coeffs f*) $\subseteq$ *p-depth-set p 0*
and *arg* : $x \in$ *p-depth-set p 0*
and *sum* : $\neg$ *p-equal p S 0*
obtains *j* **where** $j \in \{a..b\}$ **and** *p-depth p S* $\geq$ *int* (*Suc j*) $*$ *p-depth p y*
proof−
from *S-def sum* **obtain** *j* **where** *j*:
$j \in \{a..b\}$ $\neg$ *p-equal p* (*g j*) *0 p-depth p S* $\geq$ *p-depth p* (*g j*)
using *obtains-depth-sum-nonarch-witness* **by** *blast*
from *g-def j*(*2*)
have *y-coeff-n0*: $\neg$ *p-equal p* (*poly* (*coeff* (*additive-poly-poly f*) (*Suc j*)) *x*) *0*
using *mult-0-left*[*of p - y* ^ *Suc j*]
by *blast*
from *g-def j*(*2*,*3*) **have**
p-depth p (*g j*) =
p-depth p (*poly* (*coeff* (*additive-poly-poly f*) (*Suc j*)) *x*) +
int (*Suc j*) $*$ *p-depth p y*
using *depth-mult-additive*[*of p - y* ^ *Suc j*] *depth-pow-additive*[*of p y Suc j*] **by**
auto
also from *coeffs arg*
have ... $\geq$ *int* (*Suc j*) $*$ *p-depth p y*
using *y-coeff-n0 poly-p-depth-set-additive-poly-poly*[*of 0 f p x Suc j*]
p-depth-setD[*of p poly* (*coeff* (*additive-poly-poly f*) (*Suc j*)) *x 0*]
by *auto*
finally have *p-depth p* (*g j*) $\geq$ *int* (*Suc j*) $*$ *p-depth p y* **by** *blast*
with *j*(*3*) **have** *p-depth p S* $\geq$ *int* (*Suc j*) $*$ *p-depth p y* **by** *simp*
with *j*(*1*) *that* **show** *thesis* **by** *blast*
qed

end

3.5.2 Depth of inverses and division

locale *p-equality-depth-div-inv* =
p-equality-div-inv + *p-equality-depth-no-zero-divisors-1*
begin

lemma *inverse-depth*: *p-depth p* (*inverse x*) = − *p-depth p x*
proof (*cases p-equal p x 0*)
case *False*
from *False* **have** *p-depth p* ($x *$ *inverse x*) = *0*
using *depth-equiv right-inverse-equiv depth-of-1* **by** *metis*
moreover from *False* **have** $\neg$ *p-equal p* ($x *$ *inverse x*) *0*
using *right-inverse-equiv trans-right one-p-nequal-zero* **by** *blast*
ultimately show *?thesis* **using** *depth-mult-additive* **by** *simp*
qed (*simp add*: *inverse-equiv0-iff depth-equiv-0*)

lemma *depth-powi-additive*: *p-depth p* (*power-int x n*) = *n* ∗ *p-depth p x*
proof (*cases n* ≥ *0*)
  case *True* **thus** *?thesis* **using** *depth-pow-additive* **unfolding** *power-int-def* **by** *auto*
next
  case *False*
  moreover from *this* **have** (− *int* (*nat* (−*n*))) = *n* **by** *simp*
  ultimately show *?thesis*
   using *power-int-def inverse-pow inverse-depth depth-pow-additive minus-mult-left* **by** *metis*
qed

lemma *p-depth-set-inverse*: *inverse x* ∈ *p-depth-set p 0* **if** *p-depth p x* = *0*
  using *that inverse-depth p-depth-setI* **by** *force*

lemma *divide-depth*: *p-depth p* (*x* / *y*) = *p-depth p x* − *p-depth p y* **if** ¬ *p-equal p* (*x* / *y*) *0*
  using *that* **by** (*simp add*: *divide-inverse depth-mult-additive inverse-depth*)

lemma *p-depth-set-divide*:
  x / *y* ∈ *p-depth-set p n* **if** *x* ∈ *p-depth-set p n* **and** *p-depth p y* = *0*
  by (
    auto intro : *p-depth-setI*
      simp add: *that p-depth-set-equiv0 divide-equiv-0-iff p-depth-setD divide-depth*
)

lemma *p-depth-poly-deriv-quotient*:
  p-depth p ((*poly f a*) / (*poly* (*polyderiv f*) *a*)) ≥ *n*
  if *n* ≥ *0*
  and *set* (*coeffs f*) ⊆ *p-depth-set p n*
  and *a* ∈ *p-depth-set p n*
  and ¬ *p-equal p* (*poly f a*) *0*
  and ¬ *p-equal p* (*poly* (*polyderiv f*) *a*) *0*
  and *p-depth p* (*poly f a*) ≥ *2* ∗ *p-depth p* (*poly* (*polyderiv f*) *a*)
  using *that divide-equiv-0-iff divide-depth poly-p-depth-set-polyderiv poly-p-depth-set-poly*
      p-depth-setD[*of p poly* (*polyderiv f*) *a n*]
  by *force*

lemma *p-depth-set-poly-deriv-quotient*:
  (*poly f a*) / (*poly* (*polyderiv f*) *a*) ∈ *p-depth-set p n*
  if *n* ≥ *0*
  and *set* (*coeffs f*) ⊆ *p-depth-set p n*
  and *a* ∈ *p-depth-set p n*
  and ¬ *p-equal p* (*poly f a*) *0*
  and ¬ *p-equal p* (*poly* (*polyderiv f*) *a*) *0*
  and *p-depth p* (*poly f a*) ≥ *2* ∗ *p-depth p* (*poly* (*polyderiv f*) *a*)
  using *that divide-equiv-0-iff p-depth-poly-deriv-quotient p-depth-setI*
  by *presburger*

end

3.6 Global equality and restriction of places

3.6.1 Locales

locale *global-p-equality* = *p-equality*
+
  fixes *p-restrict* :: *'b::comm-ring* ⇒ (*'a* ⇒ *bool*) ⇒ *'b*
  assumes *p-restrict-equiv* : *P p* ⟹ *p-equal p* (*p-restrict x P*) *x*
  and *p-restrict-equiv0* : ¬ *P p* ⟹ *p-equal p* (*p-restrict x P*) *0*
begin

lemma *p-restrict-equiv-sym*: *P p* ⟹ *p-equal p x* (*p-restrict x P*)
  using *p-restrict-equiv sym* **by** *presburger*

lemma *p-restrict-equiv0-iff*:
  p-equal p (*p-restrict x P*) *0* ⟷ *p-equal p x 0* ∨ ¬ *P p*
  by (*metis p-restrict-equiv p-restrict-equiv0 trans trans-right*)

lemma *p-restrict-restrict-equiv-conj*:
  p-equal p (*p-restrict* (*p-restrict x P*) *Q*) (*p-restrict x* (λ*p*. *P p* ∧ *Q p*))
proof−
  consider (*both*) *Q p P p* | (*Q*) *Q p* ¬ *P p* | (*nQ*) ¬ *Q p* **by** *blast*
  thus *?thesis*
  proof *cases*
    case *both* **thus** *?thesis* **using** *p-restrict-equiv trans trans-left* **by** *metis*
  next
    case *Q* **thus** *?thesis* **using** *p-restrict-equiv p-restrict-equiv0 trans trans-left* **by** *metis*
  next
    case *nQ* **thus** *?thesis* **using** *p-restrict-equiv0 trans-left* **by** *metis*
  qed
qed

lemma *p-restrict-restrict-equiv*: *p-equal p* (*p-restrict* (*p-restrict x P*) *P*) (*p-restrict x P*)
  using *p-restrict-restrict-equiv-conj*[*of p x P P*] **by** *metis*

lemma *p-restrict-0-equiv0*: *p-equal p* (*p-restrict 0 P*) *0*
  by (*metis p-restrict-equiv p-restrict-equiv0*)

lemma *p-restrict-add-equiv*: *p-equal p* (*p-restrict* (*x* + *y*) *P*) (*p-restrict x P* + *p-restrict y P*)
  by (
    cases P p, *metis p-restrict-equiv sym add trans*,
    metis p-restrict-equiv0 trans-left add-0-left
)

lemma *p-restrict-add-mixed-equiv*:
p-equal p (*p-restrict x P* + *p-restrict y Q*) (*x* + *y*) **if** *P p* **and** *Q p*
using *that p-restrict-equiv add* **by** *simp*

lemma *p-restrict-add-mixed-equiv-drop-right*:
p-equal p (*p-restrict x P* + *p-restrict y Q*) *x* **if** *P p* **and** ¬ *Q p*
using *that p-restrict-equiv*[*of P p x*] *p-restrict-equiv0*[*of Q p y*] *add* **by** *fastforce*

lemma *p-restrict-add-mixed-equiv-drop-left*:
p-equal p (*p-restrict x P* + *p-restrict y Q*) *y* **if** ¬ *P p* **and** *Q p*
using *that p-restrict-equiv*[*of Q p y*] *p-restrict-equiv0*[*of P p x*] *add* **by** *fastforce*

lemma *p-restrict-add-mixed-equiv-drop-both*:
p-equal p (*p-restrict x P* + *p-restrict y Q*) *0* **if** ¬ *P p* **and** ¬ *Q p*
using *that p-restrict-equiv0*[*of P p x*] *p-restrict-equiv0*[*of Q p y*] *add* **by** *fastforce*

lemma *p-restrict-uminus-equiv*: *p-equal p* (*p-restrict* (−*x*) *P*) (− *p-restrict x P*)
by (
cases P p, *metis p-restrict-equiv sym uminus trans*,
metis p-restrict-equiv0 uminus trans-left minus-zero
)

lemma *p-restrict-minus-equiv*: *p-equal p* (*p-restrict* (*x* − *y*) *P*) (*p-restrict x P* − *p-restrict y P*)
by (
cases P p, *metis p-restrict-equiv sym minus trans*,
metis p-restrict-equiv0 minus trans-left diff-zero
)

lemma *p-restrict-minus-mixed-equiv*:
p-equal p (*p-restrict x P* − *p-restrict y Q*) (*x* − *y*) **if** *P p* **and** *Q p*
using *that p-restrict-equiv minus* **by** *simp*

lemma *p-restrict-minus-mixed-equiv-drop-right*:
p-equal p (*p-restrict x P* − *p-restrict y Q*) *x* **if** *P p* **and** ¬ *Q p*
using *that p-restrict-equiv*[*of P p x*] *p-restrict-equiv0*[*of Q p y*] *minus* **by** *fastforce*

lemma *p-restrict-minus-mixed-equiv-drop-left*:
p-equal p (*p-restrict x P* − *p-restrict y Q*) (−*y*) **if** ¬ *P p* **and** *Q p*
using *that p-restrict-equiv*[*of Q p y*] *p-restrict-equiv0*[*of P p x*] *minus* **by** *fastforce*

lemma *p-restrict-minus-mixed-equiv-drop-both*:
p-equal p (*p-restrict x P* − *p-restrict y Q*) *0* **if** ¬ *P p* **and** ¬ *Q p*
using *that p-restrict-equiv0*[*of P p x*] *p-restrict-equiv0*[*of Q p y*] *minus* **by** *fastforce*

lemma *p-restrict-times-equiv*:
p-equal p ((*p-restrict x P*) ∗ (*p-restrict y Q*))
(*p-restrict* (*x* ∗ *y*) (λ*p*. *P p* ∧ *Q p*))
proof−

consider (*both*) *P p Q p* | (*P*) *P p* ¬ *Q p* | (*nP*) ¬ *P p* **by** *blast*
thus *?thesis*
proof *cases*
case *both* **thus** *?thesis* **using** *p-restrict-equiv sym mult trans* **by** *metis*
next
case *P* **thus** *?thesis*
using *p-restrict-equiv p-restrict-equiv0 mult trans-left mult-zero-right* **by** *metis*
next
case *nP* **thus** *?thesis*
using *p-restrict-equiv p-restrict-equiv0 mult trans-left mult-zero-left* **by** *metis*
qed
qed

lemma *p-restrict-times-equiv′*:
p-equal p ((*p-restrict x P*) ∗ (*p-restrict y P*)) (*p-restrict* (*x* ∗ *y*) *P*)
using *p-restrict-times-equiv*[*of p x P y P*] **by** *simp*

lemma *p-restrict-p-equalI*:
p-equal p (*p-restrict x P*) *y*
if *P p* ⟶ *p-equal p x y*
and ¬ *P p* ⟶ *p-equal p y 0*
by (
cases P p, *metis that*(*1*) *p-restrict-equiv trans*, *metis that*(*2*) *p-restrict-equiv0 trans-left*
)

lemma *p-restrict-p-equal-p-restrictI*:
p-equal p (*p-restrict x P*) (*p-restrict y Q*)
if (*P p* ∧ *Q p*) ⟶ *p-equal p x y*
and (*P p* ∧ ¬ *Q p*) ⟶ *p-equal p x 0*
and (¬ *P p* ∧ *Q p*) ⟶ *p-equal p y 0*
proof (*cases P p Q p rule*: *case-split*[*case-product case-split*])
case *True-True* **thus** *?thesis* **using** *that*(*1*) *p-restrict-equiv sym cong* **by** *metis*
next
case *True-False* **thus** *?thesis* **using** *that*(*2*) *p-restrict-equiv p-restrict-equiv0 sym cong* **by** *metis*
next
case *False-True* **thus** *?thesis* **using** *that*(*3*) *p-restrict-equiv p-restrict-equiv0 sym cong* **by** *metis*
next
case *False-False* **thus** *?thesis* **using** *p-restrict-equiv0 refl cong* **by** *metis*
qed

lemma *p-restrict-p-equal-p-restrict-by-sameI*:
p-equal p (*p-restrict x P*) (*p-restrict y P*) **if** *P p* ⟶ *p-equal p x y*
using *that p-restrict-p-equal-p-restrictI* **by** *auto*

end

locale *global-p-equality-div-inv* =
p-equality-div-inv p-equal + *global-p-equality p-equal p-restrict*
for *p-equal* ::
$'a \Rightarrow 'b$::{*comm-ring-1*, *inverse*, *divide-trivial*} $\Rightarrow$
$'b \Rightarrow$ *bool*
and *p-restrict* :: $'b \Rightarrow ('a \Rightarrow bool) \Rightarrow 'b$
begin

lemma *inverse-p-restrict-equiv*: *p-equal p* (*inverse* (*p-restrict x P*)) (*p-restrict* (*inverse x*) *P*)
by (
cases p-equal p x 0 $\vee \neg$ *P p*,
metis p-restrict-equiv0-iff inverse-equiv0-iff trans-left,
intro inverse-p-unique, *metis p-restrict-times-equiv′ p-restrict-equiv right-inverse-equiv trans*
)

end

locale *global-p-equal* = *global-p-equality*
+ **assumes** *global-imp-eq*: *globally-p-equal x y* $\Longrightarrow$ *x* = *y*
begin

lemma *global-eq-iff*: *globally-p-equal x y* $\longleftrightarrow$ *x* = *y*
using *global-imp-eq* **by** *fastforce*

lemma *p-restrict-true*[*simp*] : *p-restrict x* (λ*x. True*) = *x*
using *p-restrict-equiv*[*of* λ*x. True*] *global-imp-eq* **by** *blast*

lemma *p-restrict-false*[*simp*]: *p-restrict x* (λ*x. False*) = *0*
using *p-restrict-equiv0*[*of* λ*x. False*] *global-imp-eq* **by** *blast*

lemma *p-restrict-restrict*:
p-restrict (*p-restrict x P*) *Q* = *p-restrict x* (λ*p. P p* $\wedge$ *Q p*)
using *p-restrict-restrict-equiv-conj global-imp-eq* **by** *blast*

lemma *p-restrict-restrict′*[*simp*]: *p-restrict* (*p-restrict x P*) *P* = *p-restrict x P*
using *p-restrict-restrict* **by** *fastforce*

lemma *p-restrict-image-restrict*: *p-restrict x P* = *x* **if** *x* $\in$ (λ*z. p-restrict z P*) ‘ *B*
proof−
from *that* **obtain** *z* **where** *x* = *p-restrict z P* **by** *blast*
thus *?thesis* **using** *p-restrict-restrict′* **by** *auto*
qed

lemma *p-restrict-zero*: *p-restrict 0 P* = *0*
using *p-restrict-0-equiv0 global-imp-eq* **by** *blast*

lemma *p-restrict-add*: *p-restrict* $(x + y)$ P = *p-restrict* x P + *p-restrict* y P
using *p-restrict-add-equiv global-imp-eq* **by** *blast*

lemma *p-restrict-uminus*: *p-restrict* $(-x)$ P = − *p-restrict* x P
using *p-restrict-uminus-equiv global-imp-eq* **by** *blast*

lemma *p-restrict-minus*: *p-restrict* $(x - y)$ P = *p-restrict* x P − *p-restrict* y P
using *p-restrict-minus-equiv global-imp-eq* **by** *blast*

lemma *p-restrict-times*:
(*p-restrict* x P) ∗ (*p-restrict* y Q) = *p-restrict* $(x * y)$ $(\lambda p.\ P\ p \wedge Q\ p)$
using *p-restrict-times-equiv global-imp-eq* **by** *blast*

lemma *times-p-restrict*: x ∗ (*p-restrict* y P) = *p-restrict* $(x * y)$ P
using *p-restrict-times*[*of* x $\lambda p.$ *True*] **by** *auto*

lemmas *p-restrict-pre-finite-adele-simps* =
p-restrict-zero p-restrict-add p-restrict-uminus p-restrict-minus p-restrict-times

lemma *p-equal-0-iff-p-restrict-eq-0*: *p-equal* p x *0* ⟷ *p-restrict* x $((=)\ p)$ = *0*
proof (*standard*, *rule global-imp-eq*, *standard*)
fix q **show** *p-equal* p x *0* ⟹ *p-equal* q (*p-restrict* x $((=)\ p)$) *0*
using *p-restrict-equiv*[*of* $(=)$ p] *trans p-restrict-equiv0*[*of* $(=)$ p]
by (*cases* $p = q$, *blast*, *blast*)
qed (*metis* (*full-types*) *p-restrict-equiv-sym*)

lemma *p-equal-iff-p-restrict-eq*:
p-equal p x y ⟷ *p-restrict* x $((=)\ p)$ = *p-restrict* y $((=)\ p)$
using *conv-0 p-equal-0-iff-p-restrict-eq-0 p-restrict-minus* **by** *auto*

lemma *p-restrict-eqI*:
y = *p-restrict* x P
if P: $\bigwedge p.\ P\ p \Longrightarrow$ *p-equal* p y x
and *not-P*: $\bigwedge p.\ \neg\ P\ p \Longrightarrow$ *p-equal* p y *0*
proof (*intro global-imp-eq*, *standard*)
fix $p :: 'a$ **show** *p-equal* p y (*p-restrict* x P)
by (*cases* P p, *metis* P *p-restrict-equiv trans-left*, *metis not-P p-restrict-equiv0 trans-left*)
qed

lemma *p-restrict-eq-p-restrict-mixedI*:
p-restrict x P = *p-restrict* y Q
if *both*: $\bigwedge p{::}'a.\ P\ p \Longrightarrow Q\ p \Longrightarrow$ *p-equal* p x y
and P : $\bigwedge p{::}'a.\ P\ p \Longrightarrow \neg\ Q\ p \Longrightarrow$ *p-equal* p x *0*
and Q : $\bigwedge p{::}'a.\ \neg\ P\ p \Longrightarrow Q\ p \Longrightarrow$ *p-equal* p y *0*
by (
intro p-restrict-eqI, *metis both Q p-restrict-equiv trans p-restrict-equiv0 trans-left*,
metis P p-restrict-equiv trans p-restrict-equiv0

)

lemma *p-restrict-eq-p-restrictI*:
p-restrict x P = *p-restrict y P* **if** $\bigwedge p::'a.$ *P p* $\Longrightarrow$ *p-equal p x y*
by (*intro p-restrict-eqI*, *metis that trans p-restrict-equiv*, *simp add*: *p-restrict-equiv0*)

lemma *p-restrict-eq-p-restrict-iff*:
p-restrict x P = *p-restrict y P* $\longleftrightarrow$
($\forall$ *p*::'*a*. *P p* $\longrightarrow$ *p-equal p x y*)
by (*standard*, *safe*, *metis p-restrict-equiv trans-right*, *simp add*: *p-restrict-eq-p-restrictI*)

lemma *p-restrict-eq-zero-iff*:
p-restrict x P = *0* $\longleftrightarrow$ ($\forall$ *p*. *P p* $\longrightarrow$ *p-equal p x 0*)
proof−
have (*p-restrict x P* = *0*) = ($\forall$ *p*. *p-equal p* (*p-restrict x P*) *0*)
using *global-imp-eq* **by** *auto*
thus *?thesis* **by** (*metis p-restrict-equiv p-restrict-equiv0 trans trans-right*)
qed

lemma *p-restrict-decomp*:
x = *p-restrict x P* + *p-restrict x* (λp. $\neg$ *P p*)
proof (*intro global-imp-eq*, *standard*)
fix *p* **show** *p-equal p x* (*p-restrict x P* + *p-restrict x* (λp. $\neg$ *P p*))
using *p-restrict-add-mixed-equiv-drop-left p-restrict-add-mixed-equiv-drop-right*
sym **by** *metis*
qed

lemma *restrict-complement*:
range (λx. *p-restrict x P*) + *range* (λx. *p-restrict x* (λp. $\neg$ *P p*))
= *UNIV*
using *p-restrict-decomp*[*of - P*]
set-plus-intro[*of*
p-restrict - P range (λx. *p-restrict x P*)
p-restrict - (λp. $\neg$ *P p*)
range (λx. *p-restrict x* (λp. $\neg$ *P p*))
]
by *fastforce*

end

locale *global-p-equal-no-zero-divisors* =
global-p-equal p-equal p-restrict + *p-equality-no-zero-divisors p-equal*
for *p-equal* :: '*a* $\Rightarrow$ '*b*::*comm-ring* $\Rightarrow$ '*b* $\Rightarrow$ *bool*
and *p-restrict* :: '*b* $\Rightarrow$ ('*a* $\Rightarrow$ *bool*) $\Rightarrow$ '*b*

locale *global-p-equal-no-zero-divisors-1* =
global-p-equal-no-zero-divisors p-equal p-restrict + *p-equality-no-zero-divisors-1*

p-equal
  for *p-equal* :: *'a* ⇒ *'b*::*comm-ring-1* ⇒ *'b* ⇒ *bool*
  and *p-restrict* :: *'b* ⇒ (*'a* ⇒ *bool*) ⇒ *'b*
begin

lemma *pow-eq-0-iff*: *x* ^ *n* = *0* ⟷ *x* = *0* ∧ *n* ≠ *0* **for** *x* :: *'b*
  using *global-eq-iff pow-equiv-0-iff*[*of - x n*] **by** *force*

end

locale *global-p-equal-div-inv* =
  global-p-equality-div-inv p-equal p-restrict
+ *global-p-equal p-equal p-restrict*
  for *p-equal* ::
    'a ⇒ *'b*::{*comm-ring-1*, *inverse*, *divide-trivial*} ⇒
      'b ⇒ *bool*
  and *p-restrict* :: *'b* ⇒ (*'a* ⇒ *bool*) ⇒ *'b*
begin

lemma *power-int-eq-0-iff*:
  power-int x n = *0* ⟷ *x* = *0* ∧ *n* ≠ *0* **for** *x* :: *'b*
  using *global-eq-iff power-int-equiv-0-iff*[*of - x n*] **by** *fastforce*

lemma *inverse-eq0-iff*: *inverse x* = *0* ⟷ *x* = *0* **for** *x* :: *'b*
  using *inverse-equiv0-iff global-eq-iff* **by** *fastforce*

lemma *inverse-neg-1* [*simp*]: *inverse* (−*1*::*'b*) = −*1*
  using *global-imp-eq globally-inverse-neg-1* **by** *force*

lemma *inverse-uminus*: *inverse* (−*x*) = − *inverse x* **for** *x* :: *'b*
  using *global-imp-eq inverse-uminus-equiv* **by** *blast*

lemma *global-right-inverse*: *x* ∗ *inverse x* = *1* **if** ∀ *p*. ¬ *p-equal p x 0*
  using *that global-imp-eq right-inverse-equiv* **by** *fast*

lemma *right-inverse*: *x* ∗ *inverse x* = *p-restrict 1* (λ*p*. ¬ *p-equal p x 0*)
proof (*intro global-imp-eq*, *standard*)
  fix *p* :: *'a* **show** *p-equal p* (*x* ∗ *inverse x*) (*p-restrict 1* (λ*p*. ¬ *p-equal p x 0*))
  by (
   cases p-equal p x 0, *metis inverse-equiv0-iff mult-0-right p-restrict-equiv0 trans-left*,
    metis right-inverse-equiv p-restrict-equiv trans-left
)
qed

lemma *global-left-inverse*: *inverse x* ∗ *x* = *1* **if** ∀ *p*. ¬ *p-equal p x 0*
  using *that global-imp-eq left-inverse-equiv* **by** *fast*

lemma *left-inverse*: *inverse x* ∗ *x* = *p-restrict 1* (λ*p*. ¬ *p-equal p x 0*) by (*metis right-inverse mult.commute*)

lemma *inverse-unique*: *inverse* $x = y$ **if** $x * y = 1$ **for** $x\ y :: 'b$
using *that inverse-p-unique global-imp-eq*[*of inverse x y*] **by** *fastforce*

lemma *inverse-p-restrict*: *inverse* (*p-restrict x P*) = *p-restrict* (*inverse x*) *P*
using *global-imp-eq inverse-p-restrict-equiv* **by** *blast*

lemma *power-diff-conv-inverse*:
$x \mathbin{\hat{}} (n - m) = x \mathbin{\hat{}} n * inverse\ x \mathbin{\hat{}} m$ **if** $m < n$
for $x :: 'b$
proof (*intro global-imp-eq*, *standard*)
fix *p* **show** *p-equal p* $(x \mathbin{\hat{}} (n - m))$ $(x \mathbin{\hat{}} n * inverse\ x \mathbin{\hat{}} m)$
proof (*cases p-equal p x 0*)
case *True*
from *that True* **have** *p-equal p* $(x \mathbin{\hat{}} n * inverse\ x \mathbin{\hat{}} m)$ *0*
using *pow-equiv-0-iff inverse-equiv0-iff mult-0-left mult-0-right* **by** *force*
moreover from *that True* **have** *p-equal p* $(x \mathbin{\hat{}} (n - m))$ *0* **using** *pow-equiv-0-iff*
by *fastforce*
ultimately show *?thesis* **using** *trans-left* **by** *fast*
next
case *False* **with** *that* **show** *?thesis* **using** *power-diff-conv-inverse-equiv* **by** *simp*
qed
qed

lemma *power-int-add-1*: *power-int* $x\ (m + 1) =$ *power-int* $x\ m * x$ **if** $m \neq -1$ **for**
$x :: 'b$
using *that power-int-add-1-equiv* **by** (*auto intro*: *global-imp-eq*)

lemma *power-int-add*:
power-int $x\ (m + n) =$ *power-int* $x\ m *$ *power-int* $x\ n$
if $(\forall p.\ \neg$ *p-equal p x 0*$) \lor m + n \neq 0$ **for** $x :: 'b$
using *that power-int-add-equiv* **by** (*auto intro*: *global-imp-eq*)

lemma *power-int-diff*:
power-int $x\ (m - n) =$ *power-int* $x\ m$ / *power-int* $x\ n$ **if** $m \neq n$ **for** $x :: 'b$
using *that power-int-diff-equiv* **by** (*auto intro*: *global-imp-eq*)

lemma *power-int-minus-mult*: *power-int* $x\ (n - 1) * x =$ *power-int* $x\ n$ **if** $n \neq 0$
for $x :: 'b$
using *that power-int-minus-mult-equiv* **by** (*auto intro*: *global-imp-eq*)

lemma *power-int-not-eq-zero*:
$x \neq 0 \lor n = 0 \Longrightarrow$ *power-int* $x\ n \neq 0$ **for** $x :: 'b$
by (*subst power-int-eq-0-iff*) *auto*

lemma *minus-divide-right*: $a\ /\ (-\ b) = -\ (a\ /\ b)$ **for** $a\ b :: 'b$
using *global-imp-eq minus-divide-right-equiv* **by** *blast*

lemma *minus-divide-divide*: $(- a) / (- b) = a / b$ **for** $a\ b :: 'b$
using *global-imp-eq minus-divide-divide-equiv* **by** *blast*

lemma *divide-minus1* [*simp*]: $x / (-1) = - x$ **for** $x :: 'b$
using *global-imp-eq divide-minus1-equiv* **by** *blast*

lemma *divide-self*: $x / x =$ *p-restrict 1* ($\lambda p.\ \neg$ *p-equal p x 0*)
by (*metis divide-inverse right-inverse*)

lemma *global-divide-self*: $x / x = 1$ **if** $\forall p.\ \neg$ *p-equal p x 0*
by (*metis that global-right-inverse divide-inverse*)

lemma *global-mult-divide-mult-cancel-right*:
$(a * b) / (c * b) = a / c$ **if** $\forall p.\ \neg$ *p-equal p b 0*
using *that global-imp-eq mult-divide-mult-cancel-right*[*of - b a c*] **by** *fastforce*

lemma *global-mult-divide-mult-cancel-left*:
$(c * a) / (c * b) = a / b$ **if** $\forall p.\ \neg$ *p-equal p c 0*
by (*metis that global-mult-divide-mult-cancel-right mult.commute*)

lemma *divide-p-restrict-left*: (*p-restrict x P*) / y = *p-restrict* (x / y) P
using *p-restrict-eqI*[*of P* (*p-restrict x P*) / *y*] *p-restrict-equiv*[*of P - x*]
p-restrict-equiv0[*of P - x*] *divide-left-equiv*
by *fastforce*

lemma *divide-p-restrict-right*: x / (*p-restrict y P*) = *p-restrict* (x / y) P
using *p-restrict-eqI*[*of P x* / (*p-restrict y P*)] *p-restrict-equiv*[*of P - y*]
p-restrict-equiv0[*of P - y*] *divide-right-equiv*
by *fastforce*

lemma *inj-divide-right*:
inj ($\lambda b.\ b / a$) $\longleftrightarrow$ ($\forall p.\ \neg$ *p-equal p a 0*)
unfolding *inj-def*
proof (*standard*, *safe*)
fix p
assume *inj*: $\forall x\ y :: 'b.\ x / a = y / a \longrightarrow x = y$
and a : *p-equal p a 0*
from a **have** (*p-restrict 1* ((=) *p*)) / a = (*p-restrict 0* ((=) *p*)) / a
by (*simp add*: *p-restrict-eq-p-restrictI divide-equiv-0-iff divide-p-restrict-left*)
with *inj* **show** *False* **using** *p-restrict-eq-p-restrict-iff one-p-nequal-zero* **by** *blast*
next
fix $x\ y :: 'b$
assume a: $\forall p.\ \neg$ *p-equal p a 0* **and** *xy*: $x / a = y / a$
thus $x = y$
using *times-divide-eq-right*[*of a x a*] *times-divide-eq-right*[*of a y a*]
mult-divide-cancel-left[*of - a x*] *mult-divide-cancel-left*[*of - a y*]
cong global-imp-eq[*of x y*]
by *fastforce*
qed

end

locale *global-p-equality-depth* = *global-p-equality* + *p-equality-depth*
begin

lemma *globally-p-equal-p-depth*: *globally-p-equal x y* $\Longrightarrow$ *p-depth p x* = *p-depth p y*
  using *depth-equiv globally-p-equalD* **by** *simp*

lemma *p-restrict-depth*:
  P p $\Longrightarrow$ *p-depth p* (*p-restrict x P*) = *p-depth p x*
  ¬ *P p* $\Longrightarrow$ *p-depth p* (*p-restrict x P*) = *0*
  by (*metis p-restrict-equiv depth-equiv*, *metis p-restrict-equiv0 depth-equiv-0*)

lemma *p-restrict-add-mixed-depth-drop-right*:
  p-depth p (*p-restrict x P* + *p-restrict y Q*) = *p-depth p x* **if** *P p* **and** ¬ *Q p*
  using *that p-restrict-add-mixed-equiv-drop-right depth-equiv* **by** *auto*

lemma *p-restrict-add-mixed-depth-drop-left*:
  p-depth p (*p-restrict x P* + *p-restrict y Q*) = *p-depth p y* **if** ¬ *P p* **and** *Q p*
  using *that p-restrict-add-mixed-equiv-drop-left depth-equiv* **by** *auto*

lemma *p-depth-set-p-restrict*:
  p-restrict x P ∈ *p-depth-set p n* **if** *P p* ⟶ *x* ∈ *p-depth-set p n*
proof (*rule p-depth-set-by-equivI*)
  show *p-equal p* (*p-restrict x P*) (*if P p then x else 0*)
    using *p-restrict-equiv p-restrict-equiv0* **by** *simp*
  from *that* **show** (*if P p then x else 0*) ∈ *p-depth-set p n*
    using *p-depth-set-0* **by** *auto*
qed

lemma *global-depth-set-p-restrict*:
  p-restrict x P ∈ *global-depth-set n* **if** *x* ∈ *global-depth-set n*
  using *that p-depth-set-p-restrict global-depth-set* **by** *auto*

lemma *p-depth-set-closed-under-p-restrict-image*:
  (λ*x*. *p-restrict x P*) ‘ *p-depth-set p n* ⊆ *p-depth-set p n*
  using *p-depth-set-p-restrict* **by** *blast*

lemma *global-depth-set-closed-under-p-restrict-image*:
  (λ*x*. *p-restrict x P*) ‘ *global-depth-set n* ⊆ *global-depth-set n*
  using *global-depth-set-p-restrict* **by** *blast*

lemma *global-depth-set-p-restrictI*:
  p-restrict x P ∈ *global-depth-set n*
  if ⋀*p*. *P p* $\Longrightarrow$ *x* ∈ *p-depth-set p n*
  by (

intro global-depth-setI,
metis that p-restrict-equiv p-restrict-equiv0 trans p-depth-setD p-restrict-depth(*1*)
)

lemma *p-depth-set-image-under-one-p-restrict-image*:
(λx. *p-restrict x* ((=) *p*)) ‘ *p-depth-set p n* ⊆ *global-depth-set n*
using *global-depth-set-p-restrictI* **by** *force*

end

locale *global-p-equality-depth-no-zero-divisors-1* =
p-equality-depth-no-zero-divisors-1 p-equal p-depth
+ *global-p-equality-depth p-equal p-restrict p-depth*
for *p-equal* ::
′*a* ⇒ ′*b*::*comm-ring-1* ⇒ ′*b* ⇒ *bool*
and *p-restrict* :: ′*b* ⇒ (′*a* ⇒ *bool*) ⇒ ′*b*
and *p-depth* :: ′*a* ⇒ ′*b* ⇒ *int*

locale *global-p-equality-depth-div-inv* =
p-equality-depth-div-inv + *global-p-equality-depth*
begin

sublocale *global-p-equality-div-inv* **..**

lemma *global-depth-set-inverse*:
p-restrict (*inverse x*) (λp. *p-depth p x* = *0*) ∈ *global-depth-set 0*
using *global-depth-set-p-restrictI p-depth-set-inverse* **by** *simp*

lemma *global-depth-set-divide*:
p-restrict (*x* / *y*) (λp. *p-depth p y* = *0*) ∈ *global-depth-set n*
if *x* ∈ *global-depth-set n*
using *that global-depth-set-p-restrictI global-depth-set p-depth-set-divide* **by** *auto*

lemma *global-depth-set-divideI*:
p-restrict (*x* / *y*) (λp. *p-depth p y* = *0*) ∈ *global-depth-set n*
if ⋀*p*. *p-depth p y* = *0* ⟹ *x* ∈ *p-depth-set p n*
using *that global-depth-set-p-restrictI global-depth-set p-depth-set-divide* **by** *auto*

end

locale *global-p-equal-depth* = *global-p-equal* + *global-p-equality-depth*
begin

lemma *p-depth-set-vs-global-depth-set-image-under-one-p-restrict-image*:
(λx. *p-restrict x* ((=) *p*)) ‘ *p-depth-set p n* =
(λx. *p-restrict x* ((=) *p*)) ‘*global-depth-set n*

proof (*standard*, *standard*, *clarify*)
  fix x **assume** $x \in$ *p-depth-set p n*
  hence *p-restrict x* $((=)\ p) \in$ *global-depth-set n*
    using *p-depth-set-image-under-one-p-restrict-image* **by** *fast*
  thus *p-restrict x* $((=)\ p) \in (\lambda x.$ *p-restrict x* $((=)\ p))$ '*global-depth-set n*
    using *p-restrict-restrict'*[*of x* $(=)$ *p*, *symmetric*] **by** *blast*
qed (*use global-depth-set* **in** *fast*)

end

locale *global-p-equal-depth-no-zero-divisors* =
  global-p-equal-no-zero-divisors + *p-equality-depth-no-zero-divisors*
begin
sublocale *global-p-equal-depth* **..**
end

locale *global-p-equal-depth-div-inv* =
  global-p-equality-depth-div-inv + *global-p-equal-depth-no-zero-divisors*
begin

sublocale *global-p-equal-div-inv* **..**

lemma *p-depth-set-eq-coset*:
  p-depth-set p n = (*w powi n*) $*o$ (*p-depth-set p 0*)
  if *p-depth p w* = *1* **and** $\forall p.\ \neg$ *p-equal p w 0*
  unfolding *elt-set-times-def*
proof *safe*
  fix x **assume** $x \in$ *p-depth-set p n*
  moreover define y **where** $y \equiv w$ *powi* $(-n) * x$
  ultimately have $y \in$ *p-depth-set p 0*
   using *that*(*1*) *mult-0-right p-depth-setD depth-mult-additive depth-powi-additive*
    by (*intro p-depth-setI*, *clarify*, *fastforce*)
  moreover from *y-def* **have** $x = w$ *powi* $n * y$
    using *that*(*2*) *power-int-equiv-0-iff global-right-inverse power-int-minus*
    by (*force simp flip*: *mult.assoc*)
  ultimately show $\exists y \in$*p-depth-set p 0*. $x = w$ *powi* $n * y$ **by** *metis*
next
  fix x **assume** $x \in$ *p-depth-set p 0*
  thus *w powi* $n * x \in$ *p-depth-set p n*
   using *that*(*1*) *mult-0-right p-depth-setD depth-mult-additive depth-powi-additive*
    by (*intro p-depth-setI*, *clarify*, *fastforce*)
qed

lemma *global-depth-set-eq-coset*:
  global-depth-set n = (*w powi n*) $*o$ (*global-depth-set 0*) **if** $\forall p.$ *p-depth p w* = *1*
  unfolding *elt-set-times-def*
proof *safe*
  fix x **assume** x: $x \in$ *global-depth-set n*

define *y* **where** $y \equiv w\ powi\ (-n) * x$
from *that* **have** $\forall\, p.\ \neg\ p\text{-}equal\ p\ w\ 0$ **using** *depth-equiv-0* **by** *fastforce*
with *y-def* **have** $x = w\ powi\ n * y$
using *power-int-equiv-0-iff global-right-inverse power-int-minus mult.assoc mult-1-left*
by *metis*
moreover from *x y-def* **have** $y \in global\text{-}depth\text{-}set\ 0$
using *that mult-0-right global-depth-setD depth-mult-additive depth-powi-additive*
by (*intro global-depth-setI*, *fastforce*)
ultimately show $\exists\, y{\in}global\text{-}depth\text{-}set\ 0.\ x = w\ powi\ n * y$ **by** *metis*
next
fix *x* **assume** $x \in global\text{-}depth\text{-}set\ 0$
thus $w\ powi\ n * x \in global\text{-}depth\text{-}set\ n$
using *that mult-0-right global-depth-setD depth-mult-additive depth-powi-additive*
by (*intro global-depth-setI*, *fastforce*)
qed

lemma *p-depth-set-eq-coset′*:
$p\text{-}depth\text{-}set\ p\ 0 = (w\ powi\ (-n)) *o\ (p\text{-}depth\text{-}set\ p\ n)$
if $p\text{-}depth\ p\ w = 1$ **and** $\forall\, p.\ \neg\ p\text{-}equal\ p\ w\ 0$
proof−
from *that*(2) **have** $w\ powi\ (-n) * w\ powi\ n = 1$
using *power-int-minus power-int-equiv-0-iff global-left-inverse* **by** *simp*
with *that* **show** *?thesis*
using *p-depth-set-eq-coset set-times-rearrange2 power-int-minus set-one-times*
by *metis*
qed

lemma *global-depth-set-eq-coset′*:
$global\text{-}depth\text{-}set\ 0 = (w\ powi\ (-n)) *o\ (global\text{-}depth\text{-}set\ n)$ **if** $\forall\, p.\ p\text{-}depth\ p\ w = 1$
proof−
from *that* **have** $\forall\, p.\ \neg\ p\text{-}equal\ p\ w\ 0$ **using** *depth-equiv-0* **by** *fastforce*
hence $w\ powi\ (-n) * w\ powi\ n = 1$
using *power-int-minus power-int-equiv-0-iff global-left-inverse* **by** *presburger*
with *that* **show** *?thesis*
using *global-depth-set-eq-coset set-times-rearrange2 power-int-minus set-one-times*
by *metis*
qed

end

3.6.2 Embedding of base type constants

locale *global-p-equality-w-consts* = *global-p-equality p-equal p-restrict*
for *p-equal* ::
$'a \Rightarrow\ 'b{::}comm\text{-}ring\text{-}1 \Rightarrow\ 'b \Rightarrow bool$
and *p-restrict* :: $'b \Rightarrow ('a \Rightarrow bool) \Rightarrow\ 'b$
+
fixes *func-of-consts* :: $('a \Rightarrow\ 'c{::}ring\text{-}1) \Rightarrow\ 'b$

assumes *consts-1* [*simp*]: *func-of-consts 1* = *1*
and *consts-diff* [*simp*]: *func-of-consts* (*f* − *g*) = *func-of-consts f* − *func-of-consts g*
and *consts-mult* [*simp*]:
func-of-consts (*f* ∗ *g*) = *func-of-consts f* ∗ *func-of-consts g*
begin

abbreviation *consts* ≡ *func-of-consts*
abbreviation *const a* ≡ *consts* (*λp. a*)

lemma *consts-0*[*simp*]: *consts 0* = *0*
using *consts-diff*[*of 0 0*] **by** *force*

lemma *const-0*[*simp*]: *const 0* = *0*
using *consts-0* **unfolding** *zero-fun-def* **by** *simp*

lemma *const-1*[*simp*]: *const 1* = *1*
using *consts-1* **unfolding** *one-fun-def* **by** *simp*

lemma *consts-uminus* [*simp*]: *consts* (− *f*) = − *consts f*
using *consts-diff*[*of 0 f*] **by** *fastforce*

lemma *const-uminus* [*simp*]: *const* (−*a*) = − *const a*
using *consts-uminus* **unfolding** *fun-Compl-def* **by** *auto*

lemma *const-diff* [*simp*]: *const* (*a* − *b*) = *const a* − *const b*
using *consts-diff* **unfolding** *fun-diff-def* **by** *auto*

lemma *consts-add* [*simp*]: *consts* (*f* + *g*) = *consts f* + *consts g*
proof−
have *consts f* + *consts g* = *consts* (*f* − (−*g*)) **by** (*simp flip*: *diff-minus-eq-add*)
thus *?thesis* **by** *simp*
qed

lemma *const-add* [*simp*]: *const* (*a* + *b*) = *const a* + *const b*
using *consts-add* **unfolding** *plus-fun-def* **by** *auto*

lemma *const-mult* [*simp*]: *const* (*a* ∗ *b*) = *const a* ∗ *const b*
using *consts-mult* **unfolding** *times-fun-def* **by** *auto*

lemma *const-of-nat*: *const* (*of-nat n*) = *of-nat n*
by (*induct n*) *simp-all*

lemma *const-of-int*: *const* (*of-int z*) = *of-int z*
by (*cases z rule*: *int-cases2*, *simp-all add*: *const-of-nat*)

end

locale *global-p-equal-w-consts* =
global-p-equal p-equal p-restrict
\+ *global-p-equality-w-consts p-equal p-restrict func-of-consts*
for *p-equal* ::
$'a \Rightarrow 'b$::*comm-ring-1* $\Rightarrow 'b \Rightarrow$ *bool*
and *p-restrict* :: $'b \Rightarrow ('a \Rightarrow$ *bool*$) \Rightarrow 'b$
and *func-of-consts* :: $('a \Rightarrow 'c$::*ring-1*$) \Rightarrow 'b$

locale *global-p-equality-w-inj-consts* = *global-p-equality-w-consts*
\+
assumes *p-equal-func-of-consts-0*: *p-equal p* (*consts f*) *0* $\longleftrightarrow$ *f p* = *0*
begin

lemmas *p-equal-func-of-const-0* = *p-equal-func-of-consts-0*[*of - λp. -*]

lemma *p-equal-func-of-consts*:
p-equal p (*consts f*) (*consts g*) $\longleftrightarrow$ *f p* = *g p*
by (*metis conv-0 consts-diff p-equal-func-of-consts-0 fun-diff-def right-minus-eq*)

lemma *globally-p-equal-func-of-consts*:
globally-p-equal (*func-of-consts f*) (*func-of-consts g*) $\longleftrightarrow$
($\forall$ *p. f p* = *g p*)
using *p-equal-func-of-consts* **unfolding** *globally-p-equal-def* **by** *auto*

lemma *const-p-equal*: *p-equal p* (*const a*) (*const b*) $\Longrightarrow$ *a* = *b*
using *p-equal-func-of-consts* **by** *force*

end

locale *global-p-equality-w-inj-consts-char-0* =
global-p-equality-w-inj-consts p-equal p-restrict func-of-consts
for *p-equal* ::
$'a \Rightarrow 'b$::*comm-ring-1* $\Rightarrow 'b \Rightarrow$ *bool*
and *p-restrict* :: $'b \Rightarrow ('a \Rightarrow$ *bool*$) \Rightarrow 'b$
and *func-of-consts* :: $('a \Rightarrow 'c$::*ring-char-0*$) \Rightarrow 'b$
begin

lemma *const-of-nat-p-equal-0-iff*: *p-equal p* (*of-nat n*) *0* $\longleftrightarrow$ *n* = *0*
by (*metis const-of-nat p-equal-func-of-const-0 of-nat-eq-0-iff*)

lemma *const-of-int-p-equal-0-iff*: *p-equal p* (*of-int z*) *0* $\longleftrightarrow$ *z* = *0*
by (*metis const-of-int of-int-eq-0-iff p-equal-func-of-const-0*)

end

locale *global-p-equality-div-inv-w-inj-consts* =

p-equality-div-inv + *global-p-equality-w-inj-consts*
begin

lemma *consts-divide-self-equiv*: *p-equal p* ((*consts f*) / (*consts f*)) *1* **if** *f p* ≠ *0*
using *that*
by (*simp add*: *p-equal-func-of-consts divide-self-equiv flip*: *consts-0*)

lemma *const-divide-self-equiv*: *p-equal p* ((*const c*) / (*const c*)) *1* **if** *c* ≠ *0*
using *that* **by** (*simp add*: *consts-divide-self-equiv*)

end

locale *global-p-equal-w-inj-consts* =
global-p-equal p-equal p-restrict
\+ *global-p-equality-w-inj-consts p-equal p-restrict*
for *p-equal* ::
′*a* ⇒ ′*b*::*comm-ring-1* ⇒ ′*b* ⇒ *bool*
and *p-restrict* :: ′*b* ⇒ (′*a* ⇒ *bool*) ⇒ ′*b*
begin

lemma *consts-eq-iff*: *consts f* = *consts g* ⟷ *f* = *g*
using *global-eq-iff globally-p-equal-func-of-consts* **by** *blast*

lemma *consts-eq-0-iff*: *consts f* = *0* ⟷ *f* = *0*
using *consts-eq-iff* **by** *force*

lemma *inj-consts*: *inj consts*
using *consts-eq-iff injI* **by** *meson*

lemma *const-eq-iff*: *const a* = *const b* ⟷ *a* = *b*
by (*metis consts-eq-iff*)

lemma *const-eq-0-iff*: *const a* = *0* ⟷ *a* = *0*
using *consts-eq-0-iff* **unfolding** *zero-fun-def* **by** *metis*

lemma *inj-const*: *inj const*
using *const-eq-iff injI* **by** *meson*

end

locale *global-p-equal-div-inv-w-inj-consts* =
global-p-equal-div-inv + *global-p-equal-w-inj-consts*
begin

sublocale *global-p-equality-div-inv-w-inj-consts* **..**

lemma *consts-divide-self*:

(*consts f*) / (*consts f*) = *p-restrict 1* (λ*p*::′*a*. *f p* ≠ *0*)
by (*simp add*: *p-equal-func-of-consts-0 divide-self*)

lemma *const-right-inverse* [*simp*]: *const a* ∗ *inverse* (*const a*) = *1* **if** *a* ≠ *0*
using *that global-right-inverse p-equal-func-of-const-0* **by** *blast*

lemma *const-left-inverse* [*simp*]: *inverse* (*const a*) ∗ *const a* = *1* **if** *a* ≠ *0*
by (*metis that const-right-inverse mult.commute*)

lemma *const-divide-self*: (*const c*) / (*const c*) = *1* **if** *c* ≠ *0*
using *that* **by** (*simp add*: *divide-inverse*)

lemma *mult-const-divide-mult-const-cancel-right*:
(*y* ∗ *const a*) / (*z* ∗ *const a*) = *y* / *z* **if** *a* ≠ *0*
using *that global-mult-divide-mult-cancel-right p-equal-func-of-const-0* **by** *blast*

lemma *mult-const-divide-mult-const-cancel-left*:
(*const a* ∗ *y*) / (*const a* ∗ *z*) = *y* / *z* **if** *a* ≠ *0*
by (*metis that mult-const-divide-mult-const-cancel-right mult.commute*)

lemma *mult-const-divide-mult-const-cancel-left-right*:
(*const a* ∗ *y*) / (*z* ∗ *const a*) = *y* / *z* **if** *a* ≠ *0*
by (*metis that mult.commute mult-const-divide-mult-const-cancel-left*)

lemma *mult-const-divide-mult-const-cancel-right-left*:
(*y* ∗ *const a*) / (*const a* ∗ *z*) = *y* / *z* **if** *a* ≠ *0*
by (*metis that mult.commute mult-const-divide-mult-const-cancel-left*)

end

locale *global-p-equal-div-inv-w-inj-consts-char-0* =
global-p-equality-w-inj-consts-char-0 p-equal p-restrict func-of-consts
\+ *global-p-equal-div-inv-w-inj-consts p-equal p-restrict func-of-consts*
for *p-equal* ::
′*a* ⇒ ′*b*::{*comm-ring-1*, *ring-char-0*, *inverse*, *divide-trivial*} ⇒
′*b* ⇒ *bool*
and *p-restrict* :: ′*b* ⇒ (′*a* ⇒ *bool*) ⇒ ′*b*
and *func-of-consts* :: (′*a* ⇒ ′*c*::*ring-char-0*) ⇒ ′*b*
begin

context
fixes *n* :: *nat*
assumes *n* ≠ *0*
begin

lemma *left-inverse-const-of-nat*[*simp*]: *inverse* (*of-nat n* :: ′*b*) ∗ *of-nat n* = *1*
by (*metis* ‹*n* ≠ *0*› *const-of-nat const-left-inverse of-nat-eq-0-iff*)

lemma *right-inverse-const-of-nat*[*simp*]: *of-nat n* ∗ *inverse* (*of-nat n* :: ′*b*) = *1*
by (*metis left-inverse-const-of-nat mult.commute*)

lemma *divide-self-const-of-nat*[*simp*]: (*of-nat n* :: ′*b*) / *of-nat n* = *1*
by (*metis* ‹*n* ≠ *0*› *of-nat-eq-0-iff const-of-nat const-divide-self*)

lemma *mult-const-of-nat-divide-const-mult-of-nat-cancel-right*:
(*a* ∗ *of-nat n*) / (*b* ∗ *of-nat n*) = *a* / *b* **for** *a b* :: ′*b*
using ‹*n* ≠ *0*› *const-of-nat of-nat-eq-0-iff*
mult-const-divide-mult-const-cancel-right
by *metis*

lemma *mult-const-of-nat-divide-const-mult-of-nat-cancel-left*:
(*of-nat n* ∗ *a*) / (*of-nat n* ∗ *b*) = *a* / *b* **for** *a b* :: ′*b*
by (*metis mult-const-of-nat-divide-const-mult-of-nat-cancel-right mult.commute*)

end

context
fixes *z* :: *int*
assumes *z* ≠ *0*
begin

lemma *left-inverse-const-of-int*[*simp*]: *inverse* (*of-int z* :: ′*b*) ∗ *of-int z* = *1*
by (*metis* ‹*z* ≠ *0*› *const-of-int const-left-inverse of-int-eq-0-iff*)

lemma *right-inverse-const-of-int*[*simp*]: *of-int z* ∗ *inverse* (*of-int z* :: ′*b*) = *1*
by (*metis left-inverse-const-of-int mult.commute*)

lemma *divide-self-const-of-int*[*simp*]: (*of-int z* :: ′*b*) / *of-int z* = *1*
by (*metis* ‹*z* ≠ *0*› *of-int-eq-0-iff const-of-int const-divide-self*)

lemma *mult-const-of-int-divide-const-mult-of-int-cancel-right*:
(*a* ∗ *of-int z*) / (*b* ∗ *of-int z*) = *a* / *b* **for** *a b* :: ′*b*
by (
metis ‹*z* ≠ *0*› *const-of-int of-int-eq-0-iff*
mult-const-divide-mult-const-cancel-right
)

lemma *mult-const-of-int-divide-const-mult-of-int-cancel-left*:
(*of-int z* ∗ *a*) / (*of-int z* ∗ *b*) = *a* / *b* **for** *a b* :: ′*b*
by (*metis mult-const-of-int-divide-const-mult-of-int-cancel-right mult.commute*)

end

context
includes *rat.lifting*
begin

lift-definition *const-of-rat* :: *rat* ⇒ *'b*
  is λx. *of-int* (*fst x*) / *of-int* (*snd x*)
  unfolding *ratrel-def*
proof (*clarify*, *unfold fst-conv snd-conv*)
  fix *a b c d* :: *int*
  assume *ratrel*: $b \neq 0$ $d \neq 0$ $a * d = c * b$
  define *oi* :: *int* ⇒ *'b* **where** *oi* ≡ *of-int*
  moreover from *this ratrel*(*1*,*3*) **have** (*oi a* / *oi b*) ∗ (*oi d* / *oi d*) = *oi c* / *oi d*
    by (*metis of-int-mult times-divide-eq-right divide-self-const-of-int mult-1-right mult.commute*)
  ultimately show *oi a* / *oi b* = *oi c* / *oi d* **using** *ratrel*(*2*) **by** *fastforce*
qed

lemma *inj-const-of-rat*: *inj const-of-rat*
proof
  fix *x y* **show** *const-of-rat x* = *const-of-rat y* ⟹ *x* = *y*
  proof (*transfer*, *clarify*, *unfold ratrel-iff fst-conv snd-conv*, *clarify*)
    fix *a b c d* :: *int*
    define *oi* :: *int* ⇒ *'b* **where** *oi* ≡ *of-int*
    assume *ratrel*: $b \neq 0$ $d \neq 0$ *oi a* / *oi b* = *oi c* / *oi d*
    from *oi-def ratrel*(*1*,*3*) **have** *oi a* ∗ *oi d* = (*oi d* / *oi d*) ∗ *oi c* ∗ *oi b*
      by (*metis swap-numerator divide-self-const-of-int mult-1-right mult.assoc mult.commute*)
    with *oi-def ratrel*(*2*) **show** $a * d = c * b$ **using** *of-int-eq-iff* **by** *fastforce*
  qed
qed

lemma *const-of-rat-rat*: *const-of-rat* (*Rat.Fract a b*) = (*of-int a* :: *'b*) / *of-int b*
  by *transfer simp*

lemma *const-of-rat-0* [*simp*]: *const-of-rat 0* = *0*
  by *transfer simp*

lemma *const-of-rat-1* [*simp*]: *const-of-rat 1* = *1*
  by *transfer simp*

lemma *const-of-rat-minus*: *const-of-rat* (− *a*) = − *const-of-rat a*
  by (*transfer*, *simp add*: *minus-divide-left*)

lemma *const-of-rat-add*: *const-of-rat* (*a* + *b*) = *const-of-rat a* + *const-of-rat b*
proof (*transfer*, *clarify*, *unfold fst-conv snd-conv of-int-add of-int-mult*)
  fix *a b c d* :: *int*
  assume *b*: $b \neq 0$ **and** *d*: $d \neq 0$
  define *w x y z* :: *'b*
    where *w* ≡ *of-int a* **and** *x* ≡ *of-int b*
    and *y* ≡ *of-int c* **and** *z* ≡ *of-int d*
  from *d z-def* **have** $(w * z + y * x) / (x * z) = w / x + (y * x) / (x * z)$
    by (
      metis add-divide-distrib mult-const-of-int-divide-const-mult-of-int-cancel-right

)
with *b x-def* **show** $(w * z + y * x) / (x * z) = w / x + y / z$
by (*metis mult.commute mult-const-of-int-divide-const-mult-of-int-cancel-right*)
qed

lemma *const-of-rat-mult*: *const-of-rat* $(a * b)$ = *const-of-rat* a $*$ *const-of-rat* b
by (*transfer*, *simp*, *metis times-divide-times-eq*)

lemma *const-of-rat-p-equal-0-iff*: *p-equal p* (*const-of-rat a*) *0* $\longleftrightarrow$ $a = 0$
by (*transfer fixing*: *p-equal p*, *simp*, *metis divide-equiv-0-iff const-of-int-p-equal-0-iff*)

end

lemma *const-of-rat-eq-0-iff* [*simp*]: *const-of-rat a* = *0* $\longleftrightarrow$ $a = 0$
using *const-of-rat-p-equal-0-iff global-eq-iff* **by** *fastforce*

lemma *const-of-rat-neg-one* [*simp*]: *const-of-rat* $(- 1) = -1$
by (*simp add*: *const-of-rat-minus*)

lemma *const-of-rat-diff*: *const-of-rat* $(a - b)$ = *const-of-rat* a $-$ *const-of-rat* b
using *const-of-rat-add*[*of a* $-b$] **by** (*simp add*: *const-of-rat-minus*)

lemma *const-of-rat-sum*: *const-of-rat* $(\sum a \in A.\ f\ a)$ = $(\sum a \in A.$ *const-of-rat* $(f\ a))$
by (*induct rule*: *infinite-finite-induct*) (*auto simp*: *const-of-rat-add*)

lemma *const-of-rat-prod*: *const-of-rat* $(\prod a \in A.\ f\ a)$ = $(\prod a \in A.$ *const-of-rat* $(f\ a))$
by (*induct rule*: *infinite-finite-induct*) (*auto simp*: *const-of-rat-mult*)

lemma *const-of-rat-inverse*: *const-of-rat* (*inverse a*) = *inverse* (*const-of-rat a*)
by (*cases* $a = 0$, *simp*, *rule inverse-unique*[*symmetric*], *simp flip*: *const-of-rat-mult*)

lemma *left-inverse-const-of-rat*[*simp*]:
inverse (*const-of-rat a*) $*$ *const-of-rat a* = *1* **if** $a \neq 0$
by (*metis that const-of-rat-inverse const-of-rat-mult Fields.left-inverse const-of-rat-1*)

lemma *right-inverse-const-of-rat*[*simp*]:
const-of-rat a $*$ *inverse* (*const-of-rat a*) = *1* **if** $a \neq 0$
by (*metis that left-inverse-const-of-rat mult.commute*)

lemma *const-of-rat-divide*: *const-of-rat* (a / b) = *const-of-rat a* / *const-of-rat b*
by (*metis Fields.divide-inverse const-of-rat-mult const-of-rat-inverse divide-inverse*)

lemma *divide-self-const-of-rat*[*simp*]: (*const-of-rat a*) / (*const-of-rat a*) = *1* **if** $a \neq$
0
by (*metis that divide-inverse right-inverse-const-of-rat*)

lemma *const-of-rat-power*: *const-of-rat* $(a\ \hat{}\ n)$ = (*const-of-rat a*) $\hat{}\ n$
by (*induct n*) (*simp-all add*: *const-of-rat-mult*)

lemma *const-of-rat-eq-iff* [*simp*]: *const-of-rat a* = *const-of-rat b* ⟷ *a* = *b*
  by (*metis inj-const-of-rat inj-def*)

lemma *zero-eq-const-of-rat-iff* [*simp*]: *0* = *const-of-rat a* ⟷ *0* = *a*
  by *simp*

lemma *const-of-rat-eq-1-iff* [*simp*]: *const-of-rat a* = *1* ⟷ *a* = *1*
  using *const-of-rat-eq-iff* [*of - 1*] **by** *simp*

lemma *one-eq-const-of-rat-iff* [*simp*]: *1* = *const-of-rat a* ⟷ *1* = *a*
  by *simp*

Collapse nested embeddings.

lemma *const-of-rat-of-nat-eq* [*simp*]: *const-of-rat* (*of-nat n*) = *of-nat n*
  by (*induct n*) (*simp-all add*: *const-of-rat-add*)

lemma *const-of-rat-of-int-eq* [*simp*]: *const-of-rat* (*of-int z*) = *of-int z*
  by (*cases z rule*: *int-diff-cases*) (*simp add*: *const-of-rat-diff*)

Product of all p-adic embeddings of the field of rationals.

definition *range-const-of-rat* :: *'b set* ($\mathbb{Q}_{\forall\, p}$)
  where $\mathbb{Q}_{\forall\, p}$ = *range const-of-rat*

lemma *range-const-of-rat-cases* [*cases set*: *range-const-of-rat*]:
  assumes *q* ∈ $\mathbb{Q}_{\forall\, p}$
  obtains (*const-of-rat*) *r* **where** *q* = *const-of-rat r*
proof −
  from *assms* **have** *q* ∈ *range const-of-rat* **by** (*simp only*: *range-const-of-rat-def*)
  then obtain *r* **where** *q* = *const-of-rat r* **..**
  then show *thesis* **..**
qed

lemma *range-const-of-rat-cases'*:
  assumes *x* ∈ $\mathbb{Q}_{\forall\, p}$
  obtains *a b* **where** *b* > *0 coprime a b x* = *of-int a* / *of-int b*
proof −
  from *assms* **obtain** *r* **where** *x* = *const-of-rat r*
    by (*auto simp*: *range-const-of-rat-def*)
  obtain *a b* **where** *quot*: *quotient-of r* = (*a*,*b*) **by** *force*
  have *b* > *0* **using** *quotient-of-denom-pos*[*OF quot*] **.**
  moreover have *coprime a b* **using** *quotient-of-coprime*[*OF quot*] **.**
  moreover have *x* = *of-int a* / *of-int b* **unfolding** ‹*x* = *const-of-rat r*›
      quotient-of-div[*OF quot*] **by** (*simp add*: *const-of-rat-divide*)
  ultimately show *?thesis* **using** *that* **by** *blast*
qed

lemma *range-const-of-rat-of-rat* [*simp*]: *const-of-rat r* ∈ $\mathbb{Q}_{\forall\, p}$
  by (*simp add*: *range-const-of-rat-def*)

lemma *range-const-of-rat-of-int* [*simp*]: *of-int* $z \in \mathbb{Q}_{\forall\, p}$
by (*subst const-of-rat-of-int-eq* [*symmetric*]) (*rule range-const-of-rat-of-rat*)

lemma *Ints-subset-range-const-of-rat*: $\mathbb{Z} \subseteq \mathbb{Q}_{\forall\, p}$
using *Ints-cases range-const-of-rat-of-int* **by** *blast*

lemma *range-const-of-rat-of-nat* [*simp*]: *of-nat* $n \in \mathbb{Q}_{\forall\, p}$
by (*subst const-of-rat-of-nat-eq* [*symmetric*]) (*rule range-const-of-rat-of-rat*)

lemma *Nats-subset-range-const-of-rat*: $\mathbb{N} \subseteq \mathbb{Q}_{\forall\, p}$
using *Ints-subset-range-const-of-rat Nats-subset-Ints* **by** *blast*

lemma *range-const-of-rat-0* [*simp*]: $0 \in \mathbb{Q}_{\forall\, p}$
unfolding *range-const-of-rat-def* **by** (*rule range-eqI*, *rule const-of-rat-0* [*symmetric*])

lemma *range-const-of-rat-1* [*simp*]: $1 \in \mathbb{Q}_{\forall\, p}$
unfolding *range-const-of-rat-def* **by** (*rule range-eqI*, *rule const-of-rat-1* [*symmetric*])

lemma *range-const-of-rat-add* [*simp*]:
$a \in \mathbb{Q}_{\forall\, p} \Longrightarrow$
$b \in \mathbb{Q}_{\forall\, p} \Longrightarrow$
$a + b \in \mathbb{Q}_{\forall\, p}$
by (*metis range-const-of-rat-cases range-const-of-rat-of-rat const-of-rat-add*)

lemma *range-const-of-rat-minus-iff* [*simp*]:
$- a \in \mathbb{Q}_{\forall\, p} \longleftrightarrow$
$a \in \mathbb{Q}_{\forall\, p}$
by (
metis range-const-of-rat-cases range-const-of-rat-of-rat add.inverse-inverse
const-of-rat-minus
)

lemma *range-const-of-rat-diff* [*simp*]:
$a \in \mathbb{Q}_{\forall\, p} \Longrightarrow$
$b \in \mathbb{Q}_{\forall\, p} \Longrightarrow$
$a - b \in \mathbb{Q}_{\forall\, p}$
by (*metis range-const-of-rat-add range-const-of-rat-minus-iff diff-conv-add-uminus*)

lemma *range-const-of-rat-mult* [*simp*]:
$a \in \mathbb{Q}_{\forall\, p} \Longrightarrow$
$b \in \mathbb{Q}_{\forall\, p} \Longrightarrow$
$a * b \in \mathbb{Q}_{\forall\, p}$
by (*metis range-const-of-rat-cases range-const-of-rat-of-rat const-of-rat-mult*)

lemma *range-const-of-rat-inverse* [*simp*]:
$a \in \mathbb{Q}_{\forall\, p} \Longrightarrow$
inverse $a \in \mathbb{Q}_{\forall\, p}$
by (*metis range-const-of-rat-cases range-const-of-rat-of-rat const-of-rat-inverse*)

lemma *range-const-of-rat-divide* [*simp*]:
$a \in \mathbb{Q}_{\forall\, p} \Longrightarrow$
$b \in \mathbb{Q}_{\forall\, p} \Longrightarrow$
$a\ /\ b \in \mathbb{Q}_{\forall\, p}$
by (*simp add*: *divide-inverse*)

lemma *range-const-of-rat-power* [*simp*]:
$a \in \mathbb{Q}_{\forall\, p} \Longrightarrow$
$a\ \widehat{}\ n \in \mathbb{Q}_{\forall\, p}$
by (*metis range-const-of-rat-cases range-const-of-rat-of-rat const-of-rat-power*)

lemma *range-const-of-rat-sum* [*intro*]:
$(\bigwedge x.\ x \in A \Longrightarrow f\ x \in \mathbb{Q}_{\forall\, p}) \Longrightarrow$
$sum\ f\ A \in \mathbb{Q}_{\forall\, p}$
by (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *range-const-of-rat-prod* [*intro*]:
$(\bigwedge x.\ x \in A \Longrightarrow f\ x \in \mathbb{Q}_{\forall\, p})$
$\Longrightarrow prod\ f\ A \in \mathbb{Q}_{\forall\, p}$
by (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *range-const-of-rat-induct* [*case-names const-of-rat*, *induct set*: *range-const-of-rat*]:
$q \in \mathbb{Q}_{\forall\, p} \Longrightarrow$
$(\bigwedge r.\ P\ (const\text{-}of\text{-}rat\ r)) \Longrightarrow P\ q$
by (*rule range-const-of-rat-cases*) *auto*

lemma *p-adic-Rats-p-equal-0-iff*:
$a \in \mathbb{Q}_{\forall\, p} \Longrightarrow$
$p\text{-}equal\ p\ a\ 0 \longleftrightarrow a = 0$
by (*induct a rule*: *range-const-of-rat-induct*, *simp*, *rule const-of-rat-p-equal-0-iff*)

lemma *range-const-of-rat-infinite*: $\neg$ *finite* $\mathbb{Q}_{\forall\, p}$
by (*auto dest*!: *finite-imageD simp*: *inj-on-def infinite-UNIV-char-0 range-const-of-rat-def*)

lemma *left-inverse-range-const-of-rat*:
$a \in \mathbb{Q}_{\forall\, p} \Longrightarrow a \neq 0 \Longrightarrow$
$inverse\ a * a = 1$
by (
induct a rule: *range-const-of-rat-induct*, *rule left-inverse-const-of-rat*,
simp add: *const-of-rat-p-equal-0-iff*
)

lemma *right-inverse-range-const-of-rat*:
$a \in \mathbb{Q}_{\forall\, p} \Longrightarrow a \neq 0 \Longrightarrow$
$a * inverse\ a = 1$
by (*metis left-inverse-range-const-of-rat mult.commute*)

lemma *divide-self-range-const-of-rat*:
$a \in \mathbb{Q}_{\forall\, p} \Longrightarrow a \neq 0 \Longrightarrow$

a / a = *1*
by (*metis divide-inverse right-inverse-range-const-of-rat*)

lemma *range-const-of-rat-add-iff*:
$a \in \mathbb{Q}_{\forall\, p} \vee$
$b \in \mathbb{Q}_{\forall\, p} \Longrightarrow$
$a + b \in \mathbb{Q}_{\forall\, p} \longleftrightarrow$
$a \in \mathbb{Q}_{\forall\, p} \wedge b \in \mathbb{Q}_{\forall\, p}$
by (*metis range-const-of-rat-add range-const-of-rat-diff add-diff-cancel add-diff-cancel-left'*)

lemma *range-const-of-rat-diff-iff*:
$a \in \mathbb{Q}_{\forall\, p} \vee$
$b \in \mathbb{Q}_{\forall\, p} \Longrightarrow$
$a - b \in \mathbb{Q}_{\forall\, p} \longleftrightarrow$
$a \in \mathbb{Q}_{\forall\, p} \wedge b \in \mathbb{Q}_{\forall\, p}$
by (*metis range-const-of-rat-add-iff diff-add-cancel*)

lemma *range-const-of-rat-mult-iff*:
$a * b \in \mathbb{Q}_{\forall\, p} \longleftrightarrow$
$a \in \mathbb{Q}_{\forall\, p} \wedge b \in \mathbb{Q}_{\forall\, p}$
if
$a \in \mathbb{Q}_{\forall\, p} - \{0\} \vee$
$b \in \mathbb{Q}_{\forall\, p} - \{0\}$
proof−
have ∗:
$\bigwedge a\ b :: 'b.$
$a \in \mathbb{Q}_{\forall\, p} - \{0\} \Longrightarrow$
$a * b \in \mathbb{Q}_{\forall\, p} \longleftrightarrow$
$b \in \mathbb{Q}_{\forall\, p}$
proof
fix $a\ b :: 'b$
assume $a \in \mathbb{Q}_{\forall\, p} - \{0\}$
and $a * b \in \mathbb{Q}_{\forall\, p}$
thus $b \in \mathbb{Q}_{\forall\, p}$
using *range-const-of-rat-mult*[*of inverse a*] *left-inverse-range-const-of-rat*[*of a*]
by (*fastforce simp flip*: *mult.assoc*)
qed *simp*
show *?thesis*
proof (*cases* $a \in \mathbb{Q}_{\forall\, p} - \{0\}$)
case *True* **thus** *?thesis* **using** ∗[*of a b*] **by** *fast*
next
case *False*
with *that* **have** $b \in \mathbb{Q}_{\forall\, p} - \{0\}$ **by** *blast*
moreover from *this* **have**
$a * b \in \mathbb{Q}_{\forall\, p} \longleftrightarrow$
$a \in \mathbb{Q}_{\forall\, p}$
by (*metis* ∗ *mult.commute*)
ultimately show *?thesis* **by** *blast*
qed

qed

lemma *range-const-of-rat-inverse-iff* [*simp*]:
  inverse $a \in \mathbb{Q}_{\forall\, p} \longleftrightarrow$
    $a \in \mathbb{Q}_{\forall\, p}$
  by (*metis range-const-of-rat-inverse inverse-inverse*)

lemma *range-const-of-rat-divide-iff*:
  $a \,/\, b \in \mathbb{Q}_{\forall\, p} \longleftrightarrow$
    $a \in \mathbb{Q}_{\forall\, p} \wedge b \in \mathbb{Q}_{\forall\, p}$
  if
    $a \in \mathbb{Q}_{\forall\, p} - \{0\} \vee$
      $b \in \mathbb{Q}_{\forall\, p} - \{0\}$
  using *that range-const-of-rat-inverse inverse-inverse*[*of b*] *range-const-of-rat-inverse-iff*
    divide-inverse[*of a b*] *range-const-of-rat-mult-iff*[*of a inverse b*]
  by *force*

end

lemma *Fract-eq0-iff*: *Fraction-Field.Fract a b* $= 0 \longleftrightarrow a = 0 \vee b = 0$
  by *transfer simp*

locale *global-p-equal-div-inv-w-inj-idom-consts* =
  global-p-equal-div-inv-w-inj-consts p-equal p-restrict func-of-consts
+ *global-p-equality-w-inj-consts p-equal p-restrict func-of-consts*
  for *p-equal* ::
    $'a \Rightarrow 'b$::{*comm-ring-1*, *inverse*, *divide-trivial*} $\Rightarrow$
      $'b \Rightarrow$ *bool*
  and *p-restrict* :: $'b \Rightarrow ('a \Rightarrow bool) \Rightarrow 'b$
  and *func-of-consts* :: $('a \Rightarrow 'c$::*idom*$) \Rightarrow 'b$
begin

lift-definition *const-of-fract* :: $'c$ *fract* $\Rightarrow 'b$
  is λx::$'c \times 'c$. *const* (*fst x*) / *const* (*snd x*)
proof (*clarify*, *unfold fractrel-iff fst-conv snd-conv*, *clarify*)
  fix *a b c d* :: $'c$
  assume *fractrel*: $b \neq 0$ $d \neq 0$ $a * d = c * b$
  from *fractrel*(*1*,*3*) **have** (*const a* / *const b*) $*$ (*const d* / *const d*) = *const c* /
const d
    by (*metis const-mult times-divide-eq-right const-divide-self mult-1-right swap-numerator*)
  with *fractrel*(*2*) **show** *const a* / *const b* = *const c* / *const d*
    using *const-divide-self* **by** *fastforce*
qed

lemma *inj-const-of-fract*: *inj const-of-fract*
proof
  fix *x y* **show** *const-of-fract x* = *const-of-fract y* $\Longrightarrow x = y$
  proof (*transfer*, *clarify*, *unfold fractrel-iff fst-conv snd-conv*, *clarify*)
    fix *a b c d* :: $'c$

assume *fractrel*: $b \neq 0$ $d \neq 0$ *const a* / *const b* = *const c* / *const d*
from *fractrel*(*1*,*3*) **have** *const d* ∗ *const a* = *const d* ∗ (*const c* / *const d* ∗ *const b*)
by (*metis swap-numerator const-divide-self mult-1-right*)
with *fractrel*(*2*) **have** $d * a = c * b$
by (*metis mult.assoc swap-numerator const-divide-self mult-1-left const-mult const-eq-iff*)
thus $a * d = c * b$ **by** *algebra*
qed
qed

lemma *const-of-fract-0* [*simp*]: *const-of-fract 0* = *0*
by *transfer simp*

lemma *const-of-fract-1* [*simp*]: *const-of-fract 1* = *1*
by *transfer simp*

lemma *const-of-fract-Fract*: *const-of-fract* (*Fraction-Field.Fract a b*) = *const a* / *const b*
by *transfer simp*

lemma *const-of-fract-p-eq-0-iff* [*simp*]: *p-equal p* (*const-of-fract x*) *0* ⟷ *x* = *0*
by (*cases x*)
(*simp add*: *const-of-fract-Fract divide-equiv-0-iff p-equal-func-of-const-0 Fract-eq0-iff*)

lemma *const-of-fract-equal-0-iff* [*simp*]: *const-of-fract x* = *0* ⟷ *x* = *0*
by (*simp flip*: *global-eq-iff*)

lemma *const-of-fract-minus*: *const-of-fract* (− *a*) = − *const-of-fract a*
by (*transfer*, *simp add*: *minus-divide-left fun-Compl-def*)

lemma *const-of-fract-add*: *const-of-fract* (*a* + *b*) = *const-of-fract a* + *const-of-fract b*
by (
transfer, *clarify*,
simp add: *add-divide-distrib mult-const-divide-mult-const-cancel-right*
mult-const-divide-mult-const-cancel-right-left
)

lemma *const-of-fract-mult*: *const-of-fract* (*a* ∗ *b*) = *const-of-fract a* ∗ *const-of-fract b*
by (*transfer*, *simp add*: *times-divide-times-eq*)

lemma *const-of-fract-neg-one* [*simp*]: *const-of-fract* (− *1*) = −*1*
by (*simp add*: *const-of-fract-minus*)

lemma *const-of-fract-diff*: *const-of-fract* (*a* − *b*) = *const-of-fract a* − *const-of-fract b*
using *const-of-fract-add*[*of a* −*b*] **by** (*simp add*: *const-of-fract-minus*)

lemma *const-of-fract-sum*:
  const-of-fract $(\sum a \in A.\ f\ a) = (\sum a \in A.\ const\text{-}of\text{-}fract\ (f\ a))$
  by (*induct rule*: *infinite-finite-induct*) (*auto simp*: *const-of-fract-add*)

lemma *const-of-fract-prod*:
  const-of-fract $(\prod a \in A.\ f\ a) = (\prod a \in A.\ const\text{-}of\text{-}fract\ (f\ a))$
  by (*induct rule*: *infinite-finite-induct*) (*auto simp*: *const-of-fract-mult*)

lemma *const-of-fract-inverse*: *const-of-fract* (*inverse a*) = *inverse* (*const-of-fract*
a)
 by (*cases a* = *0*, *simp*, *rule inverse-unique*[*symmetric*], *simp flip*: *const-of-fract-mult*)

lemma *left-inverse-const-of-fract*[*simp*]:
  inverse (*const-of-fract a*) ∗ *const-of-fract a* = *1* **if** $a \neq 0$
 by (*metis that const-of-fract-inverse const-of-fract-mult Fields.left-inverse const-of-fract-1*)

lemma *right-inverse-const-of-fract*[*simp*]:
  (*const-of-fract a*) ∗ *inverse* (*const-of-fract a*) = *1* **if** $a \neq 0$
  by (*metis that left-inverse-const-of-fract mult.commute*)

lemma *const-of-fract-divide*: *const-of-fract* (*a* / *b*) = *const-of-fract a* / *const-of-fract*
b
 by (*metis divide-inverse const-of-fract-mult const-of-fract-inverse Fields.divide-inverse*)

lemma *p-adic-prod-divide-self-of-fract*[*simp*]:
  (*const-of-fract a*) / (*const-of-fract a*) = *1* **if** $a \neq 0$
  by (*metis that divide-inverse right-inverse-const-of-fract*)

lemma *const-of-fract-power*: *const-of-fract* (*a* ^ *n*) = (*const-of-fract a*) ^ *n*
  by (*induct n*) (*simp-all add*: *const-of-fract-mult*)

lemma *const-of-fract-eq-iff* [*simp*]:
  const-of-fract a = *const-of-fract b* ⟷ *a* = *b*
  by (*metis inj-const-of-fract inj-def*)

lemma *zero-eq-const-of-fract-iff* [*simp*]: *0* = *const-of-fract a* ⟷ *0* = *a*
  by *simp*

lemma *const-of-fract-eq-1-iff* [*simp*]: *const-of-fract a* = *1* ⟷ *a* = *1*
  using *const-of-fract-eq-iff* [*of - 1*] **by** *simp*

lemma *one-eq-const-of-fract-iff* [*simp*]: *1* = *const-of-fract a* ⟷ *1* = *a*
  by *simp*

Collapse nested embeddings.

lemma *const-of-fract-numer-eq-const* [*simp*]: *const-of-fract* (*Fraction-Field.Fract a*
1) = *const a*
  by (*metis const-of-fract-Fract const-1 div-by-1*)

lemma *const-of-fract-of-nat-eq* [*simp*]: *const-of-fract* (*of-nat n*) = *of-nat n*
  by (*induct n*) (*simp-all add*: *const-of-fract-add*)

lemma *const-of-fract-of-int-eq* [*simp*]: *const-of-fract* (*of-int z*) = *of-int z*
  by (*cases z rule*: *int-diff-cases*) (*simp add*: *const-of-fract-diff*)

Product of all p-adic embeddings of the field of fractions of the base ring.

definition *range-const-of-fract* :: *'b set* ($\mathcal{Q}_{\forall\, p}$)
  where $\mathcal{Q}_{\forall\, p}$ = *range const-of-fract*

lemma *range-const-of-fract-cases* [*cases set*: *range-const-of-fract*]:
  assumes $q \in \mathcal{Q}_{\forall\, p}$
  obtains (*const-of-fract*) *r* **where** *q* = *const-of-fract r*
proof −
 from *assms* **have** *q* ∈ *range const-of-fract* **by** (*simp only*: *range-const-of-fract-def*)
  then obtain *r* **where** *q* = *const-of-fract r* **..**
  then show *thesis* **..**
qed

lemma *range-const-of-fract-cases'*:
  assumes $x \in \mathcal{Q}_{\forall\, p}$
  obtains *a b* **where** *b* ≠ *0 x* = *const a* / *const b*
proof −
 from *assms* **obtain** *r* **where** *x* = *const-of-fract r* **by** (*auto simp*: *range-const-of-fract-def*)
  moreover obtain *a b* **where** *Fract*: *b* ≠ *0 r* = *Fraction-Field.Fract a b*
    using *Fract-cases* **by** *meson*
  ultimately have *x* = *const a* / *const b* **using** *const-of-fract-Fract* **by** *blast*
  with *that Fract*(*1*) **show** *thesis* **by** *fast*
qed

lemma *range-const-of-fract-of-fract* [*simp*]:
  const-of-fract r $\in \mathcal{Q}_{\forall\, p}$
  by (*simp add*: *range-const-of-fract-def*)

lemma *range-const-of-fract-const* [*simp*]: *const a* $\in \mathcal{Q}_{\forall\, p}$
  unfolding *range-const-of-fract-def*
  by   (*standard*, *rule const-of-fract-numer-eq-const*[*symmetric*], *simp*)

lemma *range-const-of-fract-of-int* [*simp*]: *of-int z* $\in \mathcal{Q}_{\forall\, p}$
  by (*subst const-of-fract-of-int-eq* [*symmetric*]) (*rule range-const-of-fract-of-fract*)

lemma *Ints-subset-range-const-of-fract*: $\mathbb{Z} \subseteq \mathcal{Q}_{\forall\, p}$
  using *Ints-cases range-const-of-fract-of-int* **by** *blast*

lemma *range-const-of-fract-of-nat* [*simp*]: *of-nat n* $\in \mathcal{Q}_{\forall\, p}$
  by (*subst const-of-fract-of-nat-eq* [*symmetric*]) (*rule range-const-of-fract-of-fract*)

lemma *Nats-subset-range-const-of-fract*: $\mathbb{N} \subseteq \mathcal{Q}_{\forall\, p}$

using *Ints-subset-range-const-of-fract Nats-subset-Ints* **by** *blast*

lemma *range-const-of-fract-0* [*simp*]: $0 \in \mathcal{Q}_{\forall\, p}$
unfolding *range-const-of-fract-def* **by** (*rule range-eqI*, *rule const-of-fract-0* [*symmetric*])

lemma *range-const-of-fract-1* [*simp*]: $1 \in \mathcal{Q}_{\forall\, p}$
unfolding *range-const-of-fract-def* **by** (*rule range-eqI*, *rule const-of-fract-1* [*symmetric*])

lemma *range-const-of-fract-add* [*simp*]:
$a \in \mathcal{Q}_{\forall\, p} \Longrightarrow$
$b \in \mathcal{Q}_{\forall\, p} \Longrightarrow$
$a + b \in \mathcal{Q}_{\forall\, p}$
by (*metis range-const-of-fract-cases range-const-of-fract-of-fract const-of-fract-add*)

lemma *range-const-of-fract-minus-iff* [*simp*]:
$- a \in \mathcal{Q}_{\forall\, p} \longleftrightarrow$
$a \in \mathcal{Q}_{\forall\, p}$
by (
metis range-const-of-fract-cases range-const-of-fract-of-fract add.inverse-inverse const-of-fract-minus
)

lemma *range-const-of-fract-diff* [*simp*]:
$a \in \mathcal{Q}_{\forall\, p} \Longrightarrow$
$b \in \mathcal{Q}_{\forall\, p} \Longrightarrow$
$a - b \in \mathcal{Q}_{\forall\, p}$
by (*metis range-const-of-fract-add range-const-of-fract-minus-iff diff-conv-add-uminus*)

lemma *range-const-of-fract-mult* [*simp*]:
$a \in \mathcal{Q}_{\forall\, p} \Longrightarrow$
$b \in \mathcal{Q}_{\forall\, p} \Longrightarrow$
$a * b \in \mathcal{Q}_{\forall\, p}$
by (*metis range-const-of-fract-cases range-const-of-fract-of-fract const-of-fract-mult*)

lemma *range-const-of-fract-inverse* [*simp*]:
$a \in \mathcal{Q}_{\forall\, p} \Longrightarrow$
inverse $a \in \mathcal{Q}_{\forall\, p}$
by (*metis range-const-of-fract-cases range-const-of-fract-of-fract const-of-fract-inverse*)

lemma *range-const-of-fract-divide* [*simp*]:
$a \in \mathcal{Q}_{\forall\, p} \Longrightarrow$
$b \in \mathcal{Q}_{\forall\, p} \Longrightarrow$
$a\ /\ b \in \mathcal{Q}_{\forall\, p}$
by (*simp add*: *divide-inverse*)

lemma *range-const-of-fract-power* [*simp*]:
$a \in \mathcal{Q}_{\forall\, p} \Longrightarrow$
$a\ \widehat{}\ n \in \mathcal{Q}_{\forall\, p}$
by (*metis range-const-of-fract-cases range-const-of-fract-of-fract const-of-fract-power*)

lemma *range-const-of-fract-sum* [*intro*]:
$(\bigwedge x.\ x \in A \Longrightarrow$
$f\ x \in \mathcal{Q}_{\forall\ p}) \Longrightarrow$
$sum\ f\ A \in \mathcal{Q}_{\forall\ p}$
by (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *range-const-of-fract* [*intro*]:
$(\bigwedge x.\ x \in A \Longrightarrow f\ x \in \mathcal{Q}_{\forall\ p})$
$\Longrightarrow prod\ f\ A \in \mathcal{Q}_{\forall\ p}$
by (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *range-const-of-fract-induct* [*case-names const-of-fract*, *induct set*: *range-const-of-fract*]:
$q \in \mathcal{Q}_{\forall\ p} \Longrightarrow$
$(\bigwedge r.\ P\ (const\text{-}of\text{-}fract\ r)) \Longrightarrow P\ q$
by (*rule range-const-of-fract-cases*) *auto*

lemma *p-adic-Fracts-p-equal-0-iff*:
$a \in \mathcal{Q}_{\forall\ p} \Longrightarrow$
$p\text{-}equal\ p\ a\ 0 \longleftrightarrow a = 0$
by (*induct a rule*: *range-const-of-fract-induct*, *simp*)

lemma *range-const-of-fract-infinite*:
infinite $\mathcal{Q}_{\forall\ p}$ **if** *infinite* $(UNIV :: 'c\ set)$
using *that finite-imageD*[*OF - inj-const*] *range-const-of-fract-const*
finite-subset[*of range const*]
by *blast*

lemma *left-inverse-range-const-of-fract*:
$a \in \mathcal{Q}_{\forall\ p} \Longrightarrow a \neq 0 \Longrightarrow$
$inverse\ a * a = 1$
by (*induct a rule*: *range-const-of-fract-induct*, *rule left-inverse-const-of-fract*, *simp*)

lemma *right-inverse-range-const-of-fract*:
$a \in \mathcal{Q}_{\forall\ p} \Longrightarrow a \neq 0 \Longrightarrow$
$a * inverse\ a = 1$
by (*metis left-inverse-range-const-of-fract mult.commute*)

lemma *divide-self-range-const-of-fract*:
$a \in \mathcal{Q}_{\forall\ p} \Longrightarrow a \neq 0 \Longrightarrow$
$a\ /\ a = 1$
by (*metis divide-inverse right-inverse-range-const-of-fract*)

lemma *range-const-of-fract-add-iff*:
$a \in \mathcal{Q}_{\forall\ p} \vee$
$b \in \mathcal{Q}_{\forall\ p} \Longrightarrow$
$a + b \in \mathcal{Q}_{\forall\ p} \longleftrightarrow$
$a \in \mathcal{Q}_{\forall\ p} \wedge b \in \mathcal{Q}_{\forall\ p}$

using *range-const-of-fract-add range-const-of-fract-diff add-diff-cancel add-diff-cancel-left'*
by *metis*

lemma *range-const-of-fract-diff-iff*:
$a \in \mathcal{Q}_{\forall\, p} \vee$
$b \in \mathcal{Q}_{\forall\, p} \Longrightarrow$
$a - b \in \mathcal{Q}_{\forall\, p} \longleftrightarrow$
$a \in \mathcal{Q}_{\forall\, p} \wedge b \in \mathcal{Q}_{\forall\, p}$
by (*metis range-const-of-fract-add-iff diff-add-cancel*)

lemma *range-const-of-fract-mult-iff*:
$a * b \in \mathcal{Q}_{\forall\, p} \longleftrightarrow$
$a \in \mathcal{Q}_{\forall\, p} \wedge b \in \mathcal{Q}_{\forall\, p}$
if
$a \in \mathcal{Q}_{\forall\, p} - \{0\} \vee$
$b \in \mathcal{Q}_{\forall\, p} - \{0\}$
proof−
have $*$:
$\bigwedge a\ b :: 'b.$
$a \in \mathcal{Q}_{\forall\, p} - \{0\} \Longrightarrow$
$a * b \in \mathcal{Q}_{\forall\, p} \longleftrightarrow$
$b \in \mathcal{Q}_{\forall\, p}$
proof
fix $a\ b :: 'b$
assume $a \in \mathcal{Q}_{\forall\, p} - \{0\}$
and $a * b \in \mathcal{Q}_{\forall\, p}$
thus $b \in \mathcal{Q}_{\forall\, p}$
using *range-const-of-fract-mult*[*of inverse a*] *left-inverse-range-const-of-fract*[*of a*]
by (*fastforce simp flip*: *mult.assoc*)
qed *simp*
show *?thesis*
proof (*cases* $a \in \mathcal{Q}_{\forall\, p} - \{0\}$)
case *True* **thus** *?thesis* **using** $*$[*of a b*] **by** *fast*
next
case *False*
with *that* **have** $b \in \mathcal{Q}_{\forall\, p} - \{0\}$ **by** *blast*
moreover from *this* **have**
$a * b \in \mathcal{Q}_{\forall\, p} \longleftrightarrow$
$a \in \mathcal{Q}_{\forall\, p}$
by (*metis* $*$ *mult.commute*)
ultimately show *?thesis* **by** *blast*
qed
qed

lemma *range-const-of-fract-inverse-iff* [*simp*]:
inverse $a \in \mathcal{Q}_{\forall\, p} \longleftrightarrow$
$a \in \mathcal{Q}_{\forall\, p}$
by (*metis range-const-of-fract-inverse inverse-inverse*)

lemma *range-const-of-fract-divide-iff*:
  a / $b \in \mathcal{Q}_{\forall\, p} \longleftrightarrow$
    $a \in \mathcal{Q}_{\forall\, p} \wedge b \in \mathcal{Q}_{\forall\, p}$
  if
   $a \in \mathcal{Q}_{\forall\, p} - \{0\} \vee$
    $b \in \mathcal{Q}_{\forall\, p} - \{0\}$
  using *that range-const-of-fract-inverse inverse-inverse*[*of b*]
    range-const-of-fract-inverse-iff divide-inverse[*of a b*]
    range-const-of-fract-mult-iff[*of a inverse b*]
  by *force*

end

locale *global-p-equality-w-inj-index-consts* = *global-p-equality-w-inj-consts*
+
  fixes *index-inj* :: $'a \Rightarrow 'c$
  assumes *index-inj* : *inj index-inj*
  and *index-inj-nzero*: *index-inj p* $\neq$ *0*
begin

abbreviation *global-uniformizer* $\equiv$ *consts index-inj*
abbreviation *index-const p* $\equiv$ *const* (*index-inj p*)

lemma *global-uniformizer-locally-uniformizes*: *p-equal p global-uniformizer* (*index-const p*)
  using *p-equal-func-of-consts* **by** *auto*

lemma *index-const-globally-nequiv-0*: $\neg$ *p-equal q* (*index-const p*) *0*
  using *p-equal-func-of-consts-0 index-inj-nzero* **by** *fast*

lemma *global-uniformizer-globally-nequiv-0*: $\neg$ *p-equal p global-uniformizer 0*
  using *p-equal-func-of-consts-0 index-inj-nzero* **by** *simp*

end

locale *global-p-equality-no-zero-divisors-1-w-inj-index-consts* =
  global-p-equality-w-inj-index-consts + *p-equality-no-zero-divisors-1*
begin

lemma *index-const-pow-globally-nequiv-0*: $\neg$ *p-equal q* (*index-const p* ^ *n*) *0*
  using *index-const-globally-nequiv-0 pow-equiv-0-iff* **by** *fast*

lemma *global-uniformizer-pow-globally-nequiv-0*: $\neg$ *p-equal q* (*global-uniformizer* ^ *n*) *0*
  using *global-uniformizer-globally-nequiv-0 pow-equiv-0-iff* **by** *blast*

end

locale *global-p-equality-depth-w-inj-index-consts* =
p-equality-depth p-equal p-depth
\+ *global-p-equality-w-inj-index-consts p-equal p-restrict func-of-consts index-inj*
for *p-equal* :: $'a \Rightarrow 'b$::*comm-ring-1* $\Rightarrow 'b \Rightarrow$ *bool*
and *p-restrict* :: $'b \Rightarrow ('a \Rightarrow$ *bool*$) \Rightarrow 'b$
and *p-depth* :: $'a \Rightarrow 'b \Rightarrow$ *int*
and *func-of-consts* :: $('a \Rightarrow 'c$::{*factorial-semiring*,*ring-1*}$) \Rightarrow 'b$
and *index-inj* :: $'a \Rightarrow 'c$
\+
assumes *p-depth-func-of-consts*:
p-depth p (*func-of-consts f*) = *int* (*multiplicity* (*index-inj p*) (*f p*))
and *index-inj-nunit* : ¬ *is-unit* (*index-inj p*)
and *index-inj-coprime*:
p ≠ *q* ⟹ *multiplicity* (*index-inj p*) (*index-inj q*) = *0*
begin

lemma *p-depth-set-consts*: *func-of-consts f* ∈ *p-depth-set p 0*
using *p-depth-func-of-consts p-depth-setI* **by** *simp*

lemma *global-depth-set-consts*: *func-of-consts f* ∈ *global-depth-set 0*
using *p-depth-set-consts global-depth-set* **by** *blast*

lemma *p-depth-1*[*simp*]: *p-depth p* (*1*::$'b$) = *0*
using *p-depth-func-of-consts*[*of p 1*] **by** *simp*

lemma *p-depth-index-const*: *p-depth p* (*index-const p*) = *1*
using *p-depth-func-of-consts index-inj-nzero index-inj-nunit multiplicity-self* **by** *fastforce*

lemma *p-depth-index-const'*: *p-depth q* (*index-const p*) = *of-bool* (*q* = *p*)
by (*cases q* = *p*, *simp-all add*: *p-depth-index-const p-depth-func-of-consts index-inj-coprime*)

lemma *p-depth-global-uniformizer*: *p-depth p* (*global-uniformizer*::$'b$) = *1*
using *p-depth-func-of-consts index-inj-nzero index-inj-nunit multiplicity-self* **by** *fastforce*

lemma *p-depth-consts-func-ge0*: *p-depth p* (*consts f*) ≥ *0*
using *p-depth-func-of-consts* **by** *fastforce*

end

locale *global-p-equal-depth-no-zero-divisors-w-inj-index-consts* =
global-p-equal p-equal p-restrict
\+ *global-p-equality-depth-no-zero-divisors-1 p-equal p-restrict p-depth*
\+ *global-p-equality-depth-w-inj-index-consts p-equal p-restrict p-depth func-of-consts index-inj*
for *p-equal* :: $'a \Rightarrow 'b$::*comm-ring-1* $\Rightarrow 'b \Rightarrow$ *bool*

and *p-restrict* :: $'b \Rightarrow ('a \Rightarrow bool) \Rightarrow 'b$
and *p-depth* :: $'a \Rightarrow 'b \Rightarrow int$
and *func-of-consts* :: $('a \Rightarrow 'c::\{factorial\text{-}semiring,ring\text{-}1\}) \Rightarrow 'b$
and *index-inj* :: $'a \Rightarrow 'c$
begin

sublocale *global-p-equal-w-inj-consts* **..**
sublocale *global-p-equal-depth-no-zero-divisors* **..**

lemma *p-depth-index-const-pow*: *p-depth p* ((*index-const p*) ^ *n*) = *int n*
using *p-depth-index-const depth-pow-additive* **by** *auto*

lemma *p-depth-index-const-pow'*: *p-depth q* ((*index-const p*) ^ *n*) = *int n* ∗ *of-bool* (*q* = *p*)
using *p-depth-index-const' depth-pow-additive* **by** *fastforce*

lemma *p-depth-global-uniformizer-pow*: *p-depth p* (*global-uniformizer* ^ *n*) = *int n*
using *p-depth-global-uniformizer depth-pow-additive* **by** *auto*

end

locale *global-p-equal-depth-div-inv-w-inj-index-consts* =
global-p-equal-depth-div-inv p-equal p-depth p-restrict
\+ *global-p-equal-depth-no-zero-divisors-w-inj-index-consts*
p-equal p-restrict p-depth func-of-consts index-inj
for *p-equal* ::
$'a \Rightarrow 'b::\{comm\text{-}ring\text{-}1,\ inverse,\ divide\text{-}trivial\} \Rightarrow$
$'b \Rightarrow bool$
and *p-depth* :: $'a \Rightarrow 'b \Rightarrow int$
and *p-restrict* :: $'b \Rightarrow ('a \Rightarrow bool) \Rightarrow 'b$
and *func-of-consts* :: $('a \Rightarrow 'c::\{factorial\text{-}semiring,ring\text{-}1\}) \Rightarrow 'b$
and *index-inj* :: $'a \Rightarrow 'c$
begin

lemma *index-const-powi-globally-nequiv-0*: ¬ *p-equal q* (*index-const p powi n*) *0*
using *index-const-globally-nequiv-0 power-int-equiv-0-iff* **by** *simp*

lemma *global-uniformizer-powi-globally-nequiv-0*: ¬ *p-equal q* (*global-uniformizer powi n*) *0*
using *global-uniformizer-globally-nequiv-0 power-int-equiv-0-iff* **by** *blast*

lemma *index-const-powi-add*:
(*index-const p*) *powi* (*m* + *n*) = (*index-const p*) *powi m* ∗ (*index-const p*) *powi n*
using *power-int-add index-const-globally-nequiv-0* **by** *blast*

lemma *global-uniformizer-powi-add*:
global-uniformizer powi (*m* + *n*) = *global-uniformizer powi m* ∗ *global-uniformizer powi n*
using *power-int-add global-uniformizer-globally-nequiv-0* **by** *blast*

lemma *p-depth-index-const-powi*: *p-depth p* (*index-const p powi n*) = *n*
  by (*cases n* ≥ *0*)
    (*simp-all add*: *power-int-def p-depth-index-const-pow inverse-depth flip*: *inverse-pow*)

lemma *p-depth-index-const-powi′*: *p-depth q* (*index-const p powi n*) = *n* ∗ *of-bool* (*q* = *p*)
  by (
    cases q = *p*, *simp add*: *p-depth-index-const-powi*, *cases n* ≥ *0*,
      simp-all add: *power-int-def p-depth-index-const-pow′ inverse-depth flip*: *inverse-pow*
)

lemma *p-depth-global-uniformizer-powi*: *p-depth p* (*global-uniformizer powi n*) = *n*
  by (
    cases n ≥ *0*,
     simp-all add: *power-int-def p-depth-global-uniformizer-pow inverse-depth flip*: *inverse-pow*
)

lemma *local-uniformizer-locally-uniformizes*:
  p-depth p ((*index-const p*) *powi* (−*n*) ∗ *x*) = *0* **if** *p-depth p x* = *n*
  using *that depth-equiv-0 mult-0-right index-const-powi-globally-nequiv-0 no-zero-divisors*
    depth-mult-additive p-depth-index-const-powi
  by (*cases p-equal p x 0*) *auto*

lemma *global-uniformizer-locally-uniformizes*:
  p-depth p (*global-uniformizer powi* (−*n*) ∗ *x*) = *0* **if** *p-depth p x* = *n*
  using *that depth-equiv-0 mult-0-right global-uniformizer-powi-globally-nequiv-0 no-zero-divisors*
    depth-mult-additive p-depth-global-uniformizer-powi
  by (*cases p-equal p x 0*) *auto*

lemma *p-depth-set-eq-index-const-coset*:
  p-depth-set p n = ((*index-const p*) *powi n*) ∗*o* (*p-depth-set p 0*)
  using *p-depth-set-eq-coset p-depth-index-const index-const-globally-nequiv-0* **by** *blast*

lemma *p-depth-set-eq-index-const-coset′*:
  p-depth-set p 0 = ((*index-const p*) *powi* (−*n*)) ∗*o* (*p-depth-set p n*)
  using *p-depth-set-eq-coset′ p-depth-index-const index-const-globally-nequiv-0* **by** *blast*

lemma *p-depth-set-eq-global-uniformizer-coset*:
  p-depth-set p n = (*global-uniformizer powi n*) ∗*o* (*p-depth-set p 0*)
  using *p-depth-set-eq-coset p-depth-global-uniformizer global-uniformizer-globally-nequiv-0*
  by *blast*

lemma *p-depth-set-eq-global-uniformizer-coset'*:
p-depth-set p 0 = (*global-uniformizer powi* (−*n*)) ∗*o* (*p-depth-set p n*)
using *p-depth-set-eq-coset' p-depth-global-uniformizer global-uniformizer-globally-nequiv-0*
by *blast*

lemma *global-depth-set-eq-global-uniformizer-coset*:
global-depth-set n = (*global-uniformizer powi n*) ∗*o* (*global-depth-set 0*)
using *global-depth-set-eq-coset p-depth-global-uniformizer* **by** *simp*

lemma *global-depth-set-eq-global-uniformizer-coset'*:
global-depth-set 0 = (*global-uniformizer powi* (−*n*)) ∗*o* (*global-depth-set n*)
using *global-depth-set-eq-coset' p-depth-global-uniformizer* **by** *simp*

definition *shift-p-depth* :: *'a* ⇒ *int* ⇒ *'b* ⇒ *'b*
where *shift-p-depth p n x* = *x* ∗ (*consts* (λ*q*. *if q* = *p then index-inj p else 1*)) *powi n*

lemma *shift-p-depth-0*[*simp*]: *shift-p-depth p n 0* = *0*
by (*simp add*: *shift-p-depth-def*)

lemma *shift-p-depth-by-0*[*simp*]: *shift-p-depth p 0 x* = *x*
unfolding *shift-p-depth-def* **by** *fastforce*

lemma *shift-p-depth-add*[*simp*]:
shift-p-depth p n (*x* + *y*) = *shift-p-depth p n x* + *shift-p-depth p n y*
using *shift-p-depth-def distrib-right*[*of x y*] **by** *presburger*

lemma *shift-p-depth-uminus*[*simp*]:
shift-p-depth p n (− *x*) = − *shift-p-depth p n x*
using *shift-p-depth-def mult-minus-left*[*of x*] **by** *presburger*

lemma *shift-p-depth-minus*[*simp*]:
shift-p-depth p n (*x* − *y*) = *shift-p-depth p n x* − *shift-p-depth p n y*
using *shift-p-depth-add*[*of p n x* −*y*] **by** *auto*

lemma *shift-p-depth-equiv-at-place*: *p-equal p* (*shift-p-depth p n x*) (*x* ∗ (*index-const p powi n*))
using *shift-p-depth-def p-equal-func-of-consts power-int-equiv-base mult-left* **by** *simp*

lemma *shift-p-depth-equiv-at-other-places*:
p-equal q (*shift-p-depth p n x*) *x* **if** *q* ≠ *p*
using *that p-equal-func-of-consts*[*of q - 1*] *power-int-equiv-base power-int-1-left shift-p-depth-def mult-left*
by *fastforce*

lemma *shift-p-depth-equiv0-iff*:
p-equal q (*shift-p-depth p n x*) *0* ⟷ *p-equal q x 0*

using *shift-p-depth-equiv-at-place trans0-iff mult-equiv-0-iff power-int-equiv-0-iff index-const-globally-nequiv-0 shift-p-depth-equiv-at-other-places*
by (*metis* (*full-types*))

lemma *shift-p-depth-equiv-iff*:
p-equal q (*shift-p-depth p n x*) (*shift-p-depth p n y*) = *p-equal q x y*
using *conv-0 shift-p-depth-minus shift-p-depth-equiv0-iff* **by** *metis*

lemma *shift-p-depth-trivial-iff*:
shift-p-depth p n x = *x* ⟷ *n* = *0* ∨ *p-equal p x 0*
proof
assume *shift-p-depth p n x* = *x*
hence *p-equal p* (*x* ∗ (*1* − *index-const p powi n*)) *0*
using *shift-p-depth-equiv-at-place conv-0 mult-1-right right-diff-distrib* **by** *metis*
thus *n* = *0* ∨ *p-equal p x 0*
using *mult-equiv-0-iff conv-0 sym depth-equiv p-depth-1 p-depth-index-const-powi* **by** *metis*
next
assume *n* = *0* ∨ *p-equal p x 0*
moreover have *n* = *0* ⟹ *shift-p-depth p n x* = *x* **by** *simp*
moreover have *p-equal p x 0* ⟹ *p-equal p* (*shift-p-depth p n x*) *x*
using *shift-p-depth-equiv-at-place mult-0-left trans trans-left* **by** *blast*
ultimately have *p-equal p* (*shift-p-depth p n x*) *x* **by** *fastforce*
thus *shift-p-depth p n x* = *x* **using** *global-imp-eq shift-p-depth-equiv-at-other-places* **by** *force*
qed

lemma *shift-p-depth-times-right*: *shift-p-depth p n* (*x* ∗ *y*) = *x* ∗ *shift-p-depth p n y*
proof (*intro global-imp-eq*, *standard*)
fix *q* **show** *p-equal q* (*shift-p-depth p n* (*x* ∗ *y*)) (*x* ∗ *shift-p-depth p n y*)
proof (*cases q* = *p*)
case *True* **thus** *p-equal q* (*shift-p-depth p n* (*x* ∗ *y*)) (*x* ∗ *shift-p-depth p n y*)
using *shift-p-depth-equiv-at-place*[*of p n x* ∗ *y*] *mult.assoc*[*of x y*] *shift-p-depth-equiv-at-place mult-left sym trans*
by *presburger*
qed (*use shift-p-depth-equiv-at-other-places mult-left sym trans* **in** *fast*)
qed

lemma *shift-p-depth-times-left*: *shift-p-depth p n* (*x* ∗ *y*) = *shift-p-depth p n x* ∗ *y*
using *shift-p-depth-times-right mult.commute* **by** *metis*

lemma *shift-shift-p-depth*: *shift-p-depth p n* (*shift-p-depth p m x*) = *shift-p-depth p* (*m* + *n*) *x*
proof (*intro global-imp-eq*, *standard*)
fix *q* **show** *p-equal q* (*shift-p-depth p n* (*shift-p-depth p m x*)) (*shift-p-depth p* (*m* + *n*) *x*)
proof (*cases q* = *p*)
case *True*

thus *?thesis*
using *shift-p-depth-equiv-at-place*[*of p n shift-p-depth p m x*]
shift-p-depth-equiv-at-place[*of p m x*]
mult-right trans mult.assoc[*of x*, *symmetric*]
index-const-globally-nequiv-0
power-int-add[*of index-const p m n*]
shift-p-depth-equiv-at-place[*of p m + n x*]
trans-left[*of*
p shift-p-depth p n (*shift-p-depth p m x*) *x* ∗ *index-const p powi* (*m* + *n*)
shift-p-depth p (*m* + *n*) *x*
]
by *presburger*
next
case *False*
hence *p-equal q* (*shift-p-depth p n* (*shift-p-depth p m x*)) *x*
and *p-equal q* ((*shift-p-depth p* (*m* + *n*) *x*)) *x*
using *shift-p-depth-equiv-at-other-places trans*
by (*blast*, *blast*)
thus *?thesis* **using** *trans-left* **by** *blast*
qed
qed

lemma *shift-shift-p-depth-image*:
shift-p-depth p n ‘ *shift-p-depth p m* ‘ *A* = *shift-p-depth p* (*m* + *n*) ‘ *A*
proof *safe*
fix *a* **assume** *a*: *a* ∈ *A*
have ∗: *shift-p-depth p n* (*shift-p-depth p m a*) = *shift-p-depth p* (*m* + *n*) *a*
using *shift-shift-p-depth* **by** *auto*
from *a* **show** *shift-p-depth p n* (*shift-p-depth p m a*) ∈ *shift-p-depth p* (*m* + *n*) ‘ *A*
using ∗ **by** *fast*
from *a* **show** *shift-p-depth p* (*m* + *n*) *a* ∈ *shift-p-depth p n* ‘ *shift-p-depth p m* ‘ *A*
using ∗[*symmetric*] **by** *fast*
qed

lemma *shift-p-depth-at-place*:
p-depth p (*shift-p-depth p n x*) = *p-depth p x* + *n* **if** *n* = *0* ∨ ¬ *p-equal p x 0*
proof (*cases n* = *0*)
case *True*
moreover from *this* **have** *p-equal p* (*shift-p-depth p n x*) *x*
using *shift-p-depth-equiv-at-place power-int-0-right mult-1-right* **by** *metis*
ultimately show *?thesis* **using** *depth-equiv* **by** *fastforce*
next
case *False*
with *that* **have** ¬ *p-equal p* (*x* ∗ (*index-const p powi n*)) *0*
using *no-zero-divisors power-int-equiv-0-iff index-const-globally-nequiv-0* **by** *auto*
hence *p-depth p* (*shift-p-depth p n x*) = *p-depth p x* + *p-depth p* (*index-const p*

powi n)
  using *depth-equiv shift-p-depth-equiv-at-place depth-mult-additive* **by** *metis*
 thus *?thesis* **using** *p-depth-index-const-powi* **by** *simp*
qed

lemma *shift-p-depth-at-other-places*:
 p-depth q (*shift-p-depth p n x*) = *p-depth q x* **if** $q \neq p$
 using *that depth-equiv shift-p-depth-equiv-at-other-places* **by** *blast*

lemma *p-depth-set-shift-p-depth-closed*:
 shift-p-depth p n x ∈ *p-depth-set p m* **if** *p-depth p x* ≥ *m* − *n*
 using *that shift-p-depth-equiv0-iff shift-p-depth-at-place p-depth-setI* **by** *auto*

lemma *global-depth-set-shift-p-depth-closed*:
 shift-p-depth p n x ∈ *global-depth-set m*
 if *x* ∈ *global-depth-set m*
 and ¬ *p-equal p x 0* ⟶ *p-depth p x* ≥ *m* − *n*
 using *that p-depth-set-shift-p-depth-closed p-depth-setD shift-p-depth-at-other-places*
   shift-p-depth-equiv0-iff global-depth-setD global-depth-setI
 by *metis*

lemma *shift-p-depth-mem-p-depth-set*:
 p-depth p x ≥ *m* − *n*
 if ¬ *p-equal p x 0* **and** *shift-p-depth p n x* ∈ *p-depth-set p m*
 using *that shift-p-depth-equiv0-iff shift-p-depth-at-place*
   p-depth-setD[*of p shift-p-depth p n x*]
 by *fastforce*

lemma *shift-p-depth-mem-global-depth-set*:
 p-depth p x ≥ *m* − *n*
 if ¬ *p-equal p x 0* **and** *shift-p-depth p n x* ∈ *global-depth-set m*
 using *that shift-p-depth-equiv0-iff shift-p-depth-at-place*
   global-depth-setD[*of p shift-p-depth p n x*]
 by *fastforce*

lemma *shift-p-depth-p-depth-set*: *shift-p-depth p n* ‘ *p-depth-set p m* = *p-depth-set p* (*m* + *n*)
proof *safe*
 fix *x* **assume** *x* ∈ *p-depth-set p m*
 thus *shift-p-depth p n x* ∈ *p-depth-set p* (*m* + *n*)
  using *shift-p-depth-at-place shift-p-depth-equiv0-iff*
  unfolding *p-depth-set-def*
  by *simp*
next
 fix *x* **assume** *x* ∈ *p-depth-set p* (*m* + *n*)
 hence ¬ *p-equal p x 0* ⟶ *p-depth p x* ≥ *m* + *n*
  using *p-depth-setD* **by** *blast*
 hence
  ¬ *p-equal p* (*shift-p-depth p* (−*n*) *x*) *0* ⟶

p-depth p (*shift-p-depth p* (−*n*) *x*) ≥ *m*
  using *shift-p-depth-at-place shift-p-depth-equiv0-iff* **by** *force*
hence *shift-p-depth p* (−*n*) *x* ∈ *p-depth-set p m* **using** *p-depth-setI* **by** *simp*
moreover have *x* = *shift-p-depth p n* (*shift-p-depth p* (−*n*) *x*)
  using *shift-shift-p-depth* **by** *force*
ultimately show *x* ∈ *shift-p-depth p n* ' *p-depth-set p m* **by** *blast*
qed

lemma *shift-p-depth-cong*:
p-equal q (*shift-p-depth p n x*) (*shift-p-depth p n y*) **if** *p-equal q x y*
using *that cong-sym shift-p-depth-equiv-at-place mult-right*
shift-p-depth-equiv-at-other-places
by (*cases q* = *p*, *blast*, *blast*)

lemma *shift-p-depth-p-restrict*:
shift-p-depth p n (*p-restrict x P*) = *p-restrict* (*shift-p-depth p n x*) *P*
proof (*intro global-imp-eq*, *standard*)
fix *q*
show *p-equal q* (*shift-p-depth p n* (*p-restrict x P*)) (*p-restrict* (*shift-p-depth p n x*) *P*)
  by (
    cases P q,
    metis p-restrict-equiv shift-p-depth-cong trans-right,
    metis p-restrict-equiv0 shift-p-depth-cong shift-p-depth-0 trans-left
)
qed

lemma *shift-p-depth-p-restrict-global-depth-set-memI*:
shift-p-depth p n (*p-restrict x* ((=) *p*)) ∈ *global-depth-set m*
if ¬ *p-equal p x 0* ⟶ *p-depth p x* ≥ *m* − *n*
using *that shift-p-depth-p-restrict shift-p-depth-equiv0-iff shift-p-depth-at-place*
global-depth-set-p-restrictI p-depth-setI
by *fastforce*

lemma *shift-p-depth-p-restrict-global-depth-set-memI′*:
shift-p-depth p n (*p-restrict x* ((=) *p*)) ∈ *global-depth-set* (*m* + *n*)
if *x* ∈ *p-depth-set p m*
proof−
from *that* **have** *p-restrict x* ((=) *p*) ∈ *p-depth-set p m*
using *p-depth-set-p-restrict* **by** *simp*
hence *p-restrict* (*shift-p-depth p n x*) ((=) *p*) ∈ *p-depth-set p* (*m* + *n*)
  using *shift-p-depth-p-depth-set shift-p-depth-p-restrict*[*of p n x* (=) *p*] **by** *force*
hence *p-restrict* (*shift-p-depth p n x*) ((=) *p*) ∈ *global-depth-set* (*m* + *n*)
  using *global-depth-set-p-restrictI*[*of* (=) *p*] **by** *force*
thus *?thesis* **using** *shift-p-depth-p-restrict*[*of p n x* (=) *p*] **by** *auto*
qed

lemma *shift-p-depth-p-restrict-global-depth-set-image*:
shift-p-depth p n ' (λ*x*. *p-restrict x* ((=) *p*)) ' *global-depth-set m*

$= (\lambda x.\ \textit{p-restrict}\ x\ ((=)\ p))$ ‘ $\textit{global-depth-set}\ (m + n)$
proof *safe*
fix x **assume** $x \in \textit{global-depth-set}\ m$
hence
$\textit{shift-p-depth}\ p\ n\ (\textit{p-restrict}\ x\ ((=)\ p)) \in \textit{global-depth-set}\ (m + n)$
using *shift-p-depth-p-restrict-global-depth-set-memI′ global-depth-set* **by** *blast*
moreover have
$\textit{shift-p-depth}\ p\ n\ (\textit{p-restrict}\ x\ ((=)\ p)) =$
$\textit{p-restrict}\ (\textit{shift-p-depth}\ p\ n\ (\textit{p-restrict}\ x\ ((=)\ p)))\ ((=)\ p)$
using *shift-p-depth-p-restrict p-restrict-restrict′* **by** *presburger*
ultimately show
$\textit{shift-p-depth}\ p\ n\ (\textit{p-restrict}\ x\ ((=)\ p)) \in$
$(\lambda x.\ \textit{p-restrict}\ x\ ((=)\ p))$ ‘ $\textit{global-depth-set}\ (m + n)$
by *fast*
next
fix x **assume** $x \in \textit{global-depth-set}\ (m + n)$
moreover define y **where** $y \equiv \textit{shift-p-depth}\ p\ (-n)\ (\textit{p-restrict}\ x\ ((=)\ p))$
ultimately have $y \in \textit{global-depth-set}\ m$
using *shift-p-depth-p-restrict-global-depth-set-memI′*[*of* $x\ p\ m + n\ -n$]
global-depth-set[*of* $m + n$]
by *simp*
moreover from *y-def* **have**
$\textit{p-restrict}\ x\ ((=)\ p) = \textit{shift-p-depth}\ p\ n\ (\textit{p-restrict}\ y\ ((=)\ p))$
using *shift-p-depth-p-restrict*[*of* $p\ n\ y\ (=)\ p$] *p-restrict-restrict′*[*of* $x\ (=)\ p$]
shift-shift-p-depth[*of* $p\ n\ -n$ *p-restrict* $x\ ((=)\ p)$]
by *force*
ultimately show
$\textit{p-restrict}\ x\ ((=)\ p) \in$
$\textit{shift-p-depth}\ p\ n$ ‘ $(\lambda x.\ \textit{p-restrict}\ x\ ((=)\ p))$ ‘ $\textit{global-depth-set}\ m$
using *y-def* **by** *fast*
qed

end

3.7 Topological patterns

3.7.1 By place

Convergence of sequences as measured by depth

context *p-equality-depth*
begin

definition *p-limseq-condition* ::
$'a \Rightarrow (nat \Rightarrow 'b) \Rightarrow 'b \Rightarrow int \Rightarrow$
$nat \Rightarrow bool$
where
$\textit{p-limseq-condition}\ p\ X\ x\ n\ K =$
$(\forall k \geq K.\ \neg\ \textit{p-equal}\ p\ (X\ k)\ x \longrightarrow \textit{p-depth}\ p\ (X\ k - x) > n)$

lemma *p-limseq-conditionI* [*intro*]:
  p-limseq-condition p X x n K
  if
    $\bigwedge k.\ k \geq K \Longrightarrow \neg$ *p-equal p* (*X k*) *x* $\Longrightarrow$
      p-depth p $(X\ k - x) > n$
  using *that* **unfolding** *p-limseq-condition-def* **by** *blast*

lemma *p-limseq-conditionD*:
  p-depth p $(X\ k - x) > n$
  if *p-limseq-condition p X x n K* **and** $k \geq K$ **and** $\neg$ *p-equal p* (*X k*) *x*
  using *that* **unfolding** *p-limseq-condition-def* **by** *blast*

abbreviation *p-limseq* ::
  $'a \Rightarrow (nat \Rightarrow 'b) \Rightarrow 'b \Rightarrow bool$
  where *p-limseq p X x* $\equiv$ ($\forall\, n.\ \exists\, K.$ *p-limseq-condition p X x n K*)

lemma *p-limseq-p-cong*:
  p-limseq p X x
  if $\forall\, n{\geq}N.$ *p-equal p* (*X n*) (*Y n*) **and** *p-equal p x y* **and** *p-limseq p Y y*
proof
  fix *n*
  from *that*(*3*) **obtain** *M* **where** *M*: *p-limseq-condition p Y y n M* **by** *fastforce*
  define *K* **where** $K \equiv$ *max M N*
  with *M that*(*1*,*2*) **have** *p-limseq-condition p X x n K*
    using *trans*[*of p X - Y - y*] *trans-left*[*of p X - y x*] *p-limseq-conditionD*[*of p Y y n M*]
          depth-diff-equiv
    by   *fastforce*
  thus $\exists\, K.$ *p-limseq-condition p X x n K* **by** *auto*
qed

lemma *p-limseq-p-constant*: *p-limseq p X x* **if** $\forall\, n.$ *p-equal p* (*X n*) *x*
  using *that* **by** *fast*

lemma *p-limseq-constant*: *p-limseq p* ($\lambda n.\ x$) *x*
  using *p-limseq-p-constant* **by** *auto*

lemma *p-limseq-p-eventually-constant*:
  p-limseq p X x **if** $\forall_F$ *k in sequentially. p-equal p* (*X k*) *x*
proof
  fix *n*
  from *that* **obtain** *K* **where** *K*: $\forall\, k{\geq}K.$ *p-equal p* (*X k*) *x*
    using *eventually-sequentially* **by** *auto*
  hence *p-limseq-condition p X x n K* **by** *blast*
  thus $\exists\, K.$ *p-limseq-condition p X x n K* **by** *blast*
qed

lemma *p-limseq-0* [*simp*]: *p-limseq p 0 0*
  using *p-limseq-p-constant* **by** *simp*

lemma *p-limseq-0-iff*: *p-limseq p 0 x* $\longleftrightarrow$ *p-equal p x 0*
proof
  have $\neg$ *p-equal p x 0* $\Longrightarrow$ $\neg$ *p-limseq p 0 x*
  proof
    assume *x*: $\neg$ *p-equal p x 0 p-limseq p 0 x*
    have $\forall$ *n. p-depth p x* > *n*
    proof
      fix *n*
      from *x* **obtain** *K* :: *nat* **where** $\forall$ *k*$\geq$*K*. *p-depth p x* > *n*
        using *p-limseq-conditionD zero-fun-apply diff-0 depth-uminus sym* **by** *metis*
      thus *p-depth p x* > *n* **by** *fast*
    qed
    thus *False* **by** *fast*
  qed
  thus *p-limseq p 0 x* $\Longrightarrow$ *p-equal p x 0* **by** *blast*
qed (*use sym* **in** *force*)

lemma *p-limseq-add*: *p-limseq p* (*X* + *Y*) (*x* + *y*) **if** *p-limseq p X x* **and** *p-limseq p Y y*
proof
  fix *n*
  from *that* **obtain** *K-x K-y*
    where *K-x*: *p-limseq-condition p X x n K-x*
    and *K-y*: *p-limseq-condition p Y y n K-y*
    by *blast*
  define *K* **where** *K* $\equiv$ *max K-x K-y*
  have *p-limseq-condition p* (*X* + *Y*) (*x* + *y*) *n K*
  proof (*standard*, *unfold plus-fun-apply*)
    fix *k* **assume** *k*: *k* $\geq$ *K* $\neg$ *p-equal p* (*X k* + *Y k*) (*x* + *y*)
    from *K-def k*(*1*) **have** *k-K-x-y*: *k* $\geq$ *K-x k* $\geq$ *K-y* **by** *auto*
    have $*$:
      $\bigwedge$*X Y x y*.
        p-equal p (*X k*) *x* $\Longrightarrow$ *p-depth p* (*Y k* − *y*) > *n* $\Longrightarrow$
          p-depth p (*X k* + *Y k* − (*x* + *y*)) > *n*
    proof−
      fix *X Y x y*
      assume *X*: *p-equal p* (*X k*) *x* **and** *Y*: *p-depth p* (*Y k* − *y*) > *n*
      have *p-depth p* (*X k* + *Y k* − (*x* + *y*)) = *p-depth p* (*X k* + (*Y k* − (*x* + *y*)))
        by (*simp add*: *algebra-simps*)
      also from *X* **have** ... = *p-depth p* (*x* + (*Y k* − (*x* + *y*)))
        using *add-right depth-equiv* **by** *blast*
      also have ... = *p-depth p* (*Y k* − *y*) **by** *fastforce*
      finally show *p-depth p* (*X k* + *Y k* − (*x* + *y*)) > *n* **using** *Y* **by** *metis*
    qed
    from *k*(*2*) **consider**
      (*x-not-y*) *p-equal p* (*X k*) *x* $\neg$ *p-equal p* (*Y k*) *y* |
      (*y-not-x*) $\neg$ *p-equal p* (*X k*) *x* *p-equal p* (*Y k*) *y* |
      (*neither*) $\neg$ *p-equal p* (*X k*) *x* $\neg$ *p-equal p* (*Y k*) *y*

```
      using add by blast
    thus p-depth p (X k + Y k − (x + y)) > n
    proof cases
      case x-not-y
      from K-y k-K-x-y(2) x-not-y(2) have p-depth p (Y k − y) > n
        using p-limseq-conditionD by auto
      with x-not-y(1) show ?thesis using ∗ by blast
    next
      case y-not-x
      from K-x k-K-x-y(1) y-not-x(1) have p-depth p (X k − x) > n
        using p-limseq-conditionD by auto
     with y-not-x(2) have p-depth p (Y k + X k − (y + x)) > n using ∗ by force
      thus ?thesis by (simp add: algebra-simps)
    next
      case neither
        moreover from K-x K-y k-K-x-y this have min (p-depth p (X k − x))
(p-depth p (Y k − y)) > n
        using p-limseq-conditionD by simp
      moreover from k(2) have ¬ p-equal p (X k − x + (Y k − y)) 0
        using conv-0 by (simp add: algebra-simps)
      ultimately have p-depth p (X k − x + (Y k − y)) > n
        using conv-0 depth-nonarch[of p X k − x Y k − y] by auto
      thus ?thesis by (simp add: algebra-simps)
    qed
  qed
  thus ∃ K. p-limseq-condition p (X + Y) (x + y) n K by blast
qed

lemma p-limseq-uminus: p-limseq p (−X) (−x) if p-limseq p X x
proof
  fix n from that obtain K where p-limseq-condition p X x n K by metis
  hence p-limseq-condition p (−X) (−x) n K
    using uminus depth-diff unfolding p-limseq-condition-def by auto
  thus ∃ K. p-limseq-condition p (−X) (−x) n K by blast
qed

lemma p-limseq-diff: p-limseq p (X − Y) (x − y) if p-limseq p X x and p-limseq
p Y y
  using that p-limseq-uminus[of p Y y] p-limseq-add[of p X x −Y −y] by force

lemma p-limseq-conv-0: p-limseq p X x ⟷ p-limseq p (λn. X n − x) 0
proof
  assume p-limseq p X x
  moreover have X − (λn. x) = (λn. X n − x) by fastforce
  ultimately show p-limseq p (λn. X n − x) 0
    using p-limseq-diff[of p X x λn. x] p-limseq-constant by force
next
  assume p-limseq p (λn. X n − x) 0
  moreover have (λn. X n − x) + (λn. x) = X by force
```

ultimately show *p-limseq p X x*
using *p-limseq-add*[*of p* $\lambda n.$ *X n* $-$ *x 0* $\lambda n.$ *x*] *p-limseq-constant* **by** *simp*
qed

lemma *p-limseq-unique*: *p-equal p x y* **if** *p-limseq p X x* **and** *p-limseq p X y*
using *that p-limseq-diff*[*of p X x X y*] *p-limseq-0-iff*[*of p x* $-$ *y*] *conv-0*[*of p*] **by** *auto*

lemma *p-limseq-eventually-nequiv-0*:
$\forall_F$ *k in sequentially.* $\neg$ *p-equal p* (*X k*) *0*
if $\neg$ *p-equal p x 0* **and** *p-limseq p X x*
proof$-$
have $\neg$ ($\forall$*K*. $\exists$*k*$\geq$*K*. *p-equal p* (*X k*) *0*)
proof
assume $\forall$*K*. $\exists$*k*$\geq$*K*. *p-equal p* (*X k*) *0*
with *that* **have** $\forall$*n*. *p-depth p x* $>$ *n*
by (*metis p-limseq-conditionD trans-right depth-diff-equiv0-left*)
thus *False* **by** *blast*
qed
thus *?thesis* **using** *eventually-sequentially* **by** *force*
qed

lemma *p-limseq-bdd-depth*:
assumes $\neg$ *p-equal p x 0* **and** *p-limseq p X x*
obtains *d* **where** $\forall$*k*. $\neg$ *p-equal p* (*X k*) *0* $\longrightarrow$ *p-depth p* (*X k*) $\leq$ *d*
proof$-$
from *assms*(*2*) **obtain** *K* **where** *K*: *p-limseq-condition p X x* (*p-depth p x*) *K* **by** *auto*
define *D* **where** *D* $\equiv$ {*p-depth p x*} $\cup$ {*p-depth p* (*X k*) | *k*. *k* $<$ *K*}
define *d* **where** *d* $\equiv$ *Max D*
have $\forall$*k*. $\neg$ *p-equal p* (*X k*) *0* $\longrightarrow$ *p-depth p* (*X k*) $\leq$ *d*
proof *clarify*
fix *k* **show** *p-depth p* (*X k*) $\leq$ *d*
proof (*cases k* $\geq$ *K*)
case *True*
with *assms*(*1*) *K* **have** *p-depth p* (*X k*) $\leq$ *p-depth p x*
using *depth-equiv depth-pre-nonarch-diff-right*[*of p x X k*] *p-limseq-conditionD*
by *fastforce*
also from *D-def d-def* **have** ... $\leq$ *d* **using** *Max-ge*[*of D p-depth p x*] **by** *simp*
finally show *?thesis* **by** *blast*
next
case *False* **with** *D-def d-def* **show** *?thesis* **using** *Max-ge*[*of D p-depth p* (*X k*)] **by** *fastforce*
qed
qed
with *that* **show** *thesis* **by** *blast*
qed

lemma *p-limseq-eventually-bdd-depth*:

$\forall_F$ *k in sequentially. p-depth p* $(X\ k) \leq$ *p-depth p x*
if $\neg$ *p-equal p x 0* **and** *p-limseq p X x*
proof−
from *that*(*2*) **obtain** *K* **where** *K*: *p-limseq-condition p X x* (*p-depth p x*) *K* **by** *auto*
have $\forall k \geq K$. *p-depth p* $(X\ k) \leq$ *p-depth p x*
proof *clarify*
fix *k* **assume** $k \geq K$
with *that*(*1*) *K* **show** *p-depth p* $(X\ k) \leq$ *p-depth p x*
using *depth-equiv depth-pre-nonarch-diff-right*[*of p x X k*] *p-limseq-conditionD* **by** *fastforce*
qed
thus *?thesis* **using** *eventually-sequentially* **by** *blast*
qed

lemma *p-limseq-delayed-iff*:
p-limseq p X x $\longleftrightarrow$ *p-limseq p* $(\lambda n.\ X\ (N + n))$ *x*
proof (*standard*, *safe*)
fix *n*
assume *p-limseq p X x*
from *this* **obtain** *K* **where** *p-limseq-condition p X x n K* **by** *fastforce*
hence *p-limseq-condition p* $(\lambda n.\ X\ (N + n))$ *x n K*
unfolding *p-limseq-condition-def* **by** *simp*
thus $\exists K$. *p-limseq-condition p* $(\lambda n.\ X\ (N + n))$ *x n K* **by** *blast*
next
fix *n*
assume *p-limseq p* $(\lambda n.\ X\ (N + n))$ *x*
from *this* **obtain** *M* **where** *M*: *p-limseq-condition p* $(\lambda n.\ X\ (N + n))$ *x n M* **by** *auto*
define *K* **where** $K \equiv M + N$
have *p-limseq-condition p X x n K*
proof
fix *k* **assume** *k*: $k \geq K$ **and** *X*: $\neg$ *p-equal p* $(X\ k)$ *x*
from *k K-def* **have** $N + (k - N) = k$ **by** *simp*
moreover from *k K-def* **have** $k - N \geq M$ **by** *auto*
moreover from *k K-def X* **have** $\neg$ *p-equal p* $(X\ (N + (k - N)))$ *x*
by (*simp add*: *algebra-simps*)
ultimately show *p-depth p* $(X\ k - x) > n$ **using** *M p-limseq-conditionD* **by** *metis*
qed
thus $\exists K$. *p-limseq-condition p X x n K* **by** *blast*
qed

lemma *p-depth-set-p-limseq-closed*:
$x \in$ *p-depth-set p n*
if *p-limseq p X x* **and** $\forall_F$ *k in sequentially.* $X\ k \in$ *p-depth-set p n*
proof (*intro p-depth-setI*, *clarify*)
assume *x-n0*: $\neg$ *p-equal p x 0*
from *that*(*1*) **obtain** *K* **where** *K*: *p-limseq-condition p X x n K* **by** *blast*

```
  from that(2) obtain N where N: ∀k≥N. X k ∈ p-depth-set p n
    using eventually-sequentially by fastforce
  define k where k ≡ max N K
  show p-depth p x ≥ n
  proof (cases p-equal p (X k) x)
    case True with N x-n0 k-def show ?thesis
      using trans-right[of p - x] p-depth-setD[of p X k] depth-equiv by fastforce
  next
    case False
    with K k-def have X-x: p-depth p (X k − x) > n
      using p-limseq-conditionD by force
    show ?thesis
    proof (cases p-equal p (X k) 0)
      case True thus ?thesis using X-x depth-diff-equiv0-left by simp
    next
      case False
      with N x-n0 k-def show p-depth p x ≥ n
        using X-x depth-pre-nonarch-diff-right[of p x X k] p-depth-setD[of p - n] by
fastforce
    qed
  qed
qed
```

Cauchy sequences by place

```
definition p-cauchy-condition ::
  'a ⇒ (nat ⇒ 'b) ⇒ int ⇒ nat ⇒ bool
  where
    p-cauchy-condition p X n K =
      (∀k1≥K. ∀k2≥K.
        ¬ p-equal p (X k1) (X k2) ⟶ p-depth p (X k1 − X k2) > n
      )

lemma p-cauchy-conditionI:
  p-cauchy-condition p X n K
  if
    ⋀k k'. k ≥ K ⟹ k' ≥ K ⟹
      ¬ p-equal p (X k) (X k') ⟹ p-depth p (X k − X k') > n
  using that unfolding p-cauchy-condition-def by blast

lemma p-cauchy-conditionD:
  p-depth p (X k − X k') > n
  if  p-cauchy-condition p X n K and k ≥ K and k' ≥ K
  and ¬ p-equal p (X k) (X k')
  using that unfolding p-cauchy-condition-def by blast

lemma p-cauchy-condition-mono-seq-tail:
  p-cauchy-condition p X n K ⟹ p-cauchy-condition p X n K'
  if K' ≥ K
  using that unfolding p-cauchy-condition-def by auto
```

lemma *p-cauchy-condition-mono-depth*:
  p-cauchy-condition p X n K $\Longrightarrow$ *p-cauchy-condition p X m K*
  if $m \leq n$
  using *that* **unfolding** *p-cauchy-condition-def* **by** *fastforce*

abbreviation *p-cauchy p X* $\equiv$ ($\forall$ *n*. $\exists$ *K*. *p-cauchy-condition p X n K*)

lemma *p-cauchy-condition-LEAST*:
  p-cauchy-condition p X n (*LEAST K*. *p-cauchy-condition p X n K*)
  if *p-cauchy p X*
  using *that Least1I*[*OF ex-ex1-least-nat-le*] **by** *blast*

lemma *p-cauchy-condition-LEAST-mono*:
  (*LEAST K*. *p-cauchy-condition p X m K*) $\leq$ (*LEAST K*. *p-cauchy-condition p X n K*)
  if *p-cauchy p X* **and** $m \leq n$
  using *that least-le-least*[*OF ex-ex1-least-nat-le ex-ex1-least-nat-le*]
      p-cauchy-condition-mono-depth
  by *force*

definition *p-consec-cauchy-condition* ::
  'a $\Rightarrow$ (*nat* $\Rightarrow$ *'b*) $\Rightarrow$ *int* $\Rightarrow$ *nat* $\Rightarrow$ *bool*
  where
    p-consec-cauchy-condition p X n K =
      ($\forall$ *k*$\geq$*K*.
        $\neg$ *p-equal p* (*X* (*Suc k*)) (*X k*) $\longrightarrow$ *p-depth p* (*X* (*Suc k*) $-$ *X k*) $>$ *n*
)

lemma *p-consec-cauchy-conditionI*:
  p-consec-cauchy-condition p X n K
  if
    $\bigwedge$*k*. $k \geq K \Longrightarrow \neg$ *p-equal p* (*X* (*Suc k*)) (*X k*) $\Longrightarrow$
      p-depth p (*X* (*Suc k*) $-$ *X k*) $>$ *n*
  using *that* **unfolding** *p-consec-cauchy-condition-def* **by** *blast*

lemma *p-consec-cauchy*:
  p-cauchy p X = ($\forall$ *n*. $\exists$ *K*. *p-consec-cauchy-condition p X n K*)
proof (*standard*, *safe*)
  fix *n*
  assume *p-cauchy p X*
  from *this* **obtain** *K* **where** *K*: *p-cauchy-condition p X n K* **by** *auto*
  hence *p-consec-cauchy-condition p X n K*
    using *p-cauchy-conditionD* **by** (*force intro*: *p-consec-cauchy-conditionI*)
  thus $\exists$ *K*. *p-consec-cauchy-condition p X n K* **by** *blast*
next
  fix *n*
  assume $\forall$ *n*. $\exists$ *K*. *p-consec-cauchy-condition p X n K*
  from *this* **obtain** *K* **where** *K*: *p-consec-cauchy-condition p X n K* **by** *blast*

have *p-cauchy-condition p X n K*
proof (*intro p-cauchy-conditionI*)
fix *k k'* **show**
$k \geq K \Longrightarrow k' \geq K \Longrightarrow$
$\neg$ *p-equal p* (*X k*) (*X k'*) $\Longrightarrow$ *p-depth p* (*X k* $-$ *X k'*) $> n$
proof (*induct k' k rule*: *linorder-wlog*)
case (*le a b*)
define *c* **where** $c \equiv b - a$
have
$a \geq K \Longrightarrow \neg$ *p-equal p* (*X* (*a* + *c*)) (*X a*) $\Longrightarrow$
p-depth p (*X* (*a* + *c*) $-$ *X a*) $> n$
proof (*induct c*)
case (*Suc c*)
have *decomp*: *X* (*a* + *Suc c*) $-$ *X a* = (*X* (*a* + *Suc c*) $-$ *X* (*a* + *c*)) + (*X* (*a* + *c*) $-$ *X a*)
by *fastforce*
consider *p-equal p* (*X* (*a* + *Suc c*)) (*X* (*a* + *c*)) |
p-equal p (*X* (*a* + *c*)) (*X a*) |
$\neg$ *p-equal p* (*X* (*a* + *Suc c*)) (*X* (*a* + *c*))
$\neg$ *p-equal p* (*X* (*a* + *c*)) (*X a*)
by *blast*
thus *?case*
proof (*cases*, *metis Suc trans decomp conv-0 depth-add-equiv0-left*)
case *2*
with *Suc*(*3*) **have** $\neg$ *p-equal p* (*X* (*a* + *Suc c*)) (*X* (*a* + *c*)) **using** *trans* **by** *blast*
with *K Suc*(*2*) **have** *p-depth p* (*X* (*a* + *Suc c*) $-$ *X* (*a* + *c*)) $> n$
using *p-consec-cauchy-condition-def*[*of p X n K*] **by** *force*
with *2* **show** *?thesis* **using** *decomp conv-0 depth-add-equiv0-right* **by** *presburger*
next
case *3*
moreover from *K Suc*(*2*) *3*(*1*) **have** *p-depth p* (*X* (*a* + *Suc c*) $-$ *X* (*a* + *c*)) $> n$
using *p-consec-cauchy-condition-def*[*of p X n K*] **by** *force*
moreover from *3*(*2*) *Suc*(*1*,*2*) **have** *p-depth p* (*X* (*a* + *c*) $-$ *X a*) $> n$ **by** *fast*
ultimately show *?thesis* **using** *Suc*(*3*) *decomp conv-0 depth-nonarch*[*of p*] **by** *fastforce*
qed
qed *simp*
with *le c-def* **show** *?case* **by** *force*
qed (*metis sym depth-diff*)
qed
thus $\exists K$. *p-cauchy-condition p X n K* **by** *blast*
qed

end

context *p-equality-depth-no-zero-divisors*
begin

lemma *p-limseq-mult-both-0*: *p-limseq p* $(X * Y)$ *0* **if** *p-limseq p X 0* **and** *p-limseq p Y 0*
proof
fix *n*
from *that* **obtain** *K-X K-Y*
where *p-limseq-condition p X 0* $|n|$ *K-X*
and *p-limseq-condition p Y 0* $|n|$ *K-Y*
by *metis*
moreover define *K* **where** $K \equiv$ *max K-X K-Y*
ultimately have *p-limseq-condition p* $(X * Y)$ *0 n K*
using *K-def mult-0-left mult-0-right depth-mult-additive*
unfolding *p-limseq-condition-def*
by *fastforce*
thus $\exists K.$ *p-limseq-condition p* $(X * Y)$ *0 n K* **by** *blast*
qed

lemma *p-limseq-constant-mult-0*: *p-limseq p* $(\lambda n.\ x * Y\ n)$ *0* **if** *p-limseq p Y 0*
proof
fix *n*
from *that* **obtain** *K* **where** *p-limseq-condition p Y 0* $(n -$ *p-depth p x*$)$ *K* **by** *auto*
hence *p-limseq-condition p* $(\lambda n.\ x * Y\ n)$ *0 n K*
using *mult-0-right depth-mult-additive* **unfolding** *p-limseq-condition-def* **by** *auto*
thus $\exists K.$ *p-limseq-condition p* $(\lambda n.\ x * Y\ n)$ *0 n K* **by** *auto*
qed

lemma *p-limseq-mult-0-right*: *p-limseq p* $(X * Y)$ *0* **if** *p-limseq p X x* **and** *p-limseq p Y 0*
proof−
from *that*(*1*) **have** *p-limseq p* $(\lambda n.\ X\ n - x)$ *0* **using** *p-limseq-conv-0* **by** *blast*
with *that*(*2*) **have** *p-limseq p* $((\lambda n.\ X\ n - x) * Y)$ *0* **using** *p-limseq-mult-both-0* **by** *blast*
moreover have $(\lambda n.\ X\ n - x) * Y = X * Y - (\lambda n.\ x * Y\ n)$
unfolding *times-fun-def fun-diff-def* **by** (*auto simp add*: *algebra-simps*)
ultimately have *p-limseq p* $(X * Y - (\lambda n.\ x * Y\ n))$ *0* **by** *auto*
with *that*(*2*) **have** *p-limseq p* $(X * Y - (\lambda n.\ x * Y\ n) + (\lambda n.\ x * Y\ n))$ *0*
using *p-limseq-constant-mult-0*[*of p Y x*]
p-limseq-add[*of p* $X * Y - (\lambda n.\ x * Y\ n)$ *0* $\lambda n.\ x * Y\ n$]
by *simp*
thus *?thesis* **by** (*simp add*: *algebra-simps*)
qed

lemma *p-limseq-mult*: *p-limseq p* $(X * Y)$ $(x * y)$ **if** *p-limseq p X x* **and** *p-limseq p Y y*
proof−

from *that* **have** *p-limseq p* $(X * (\lambda n.\ Y\ n - y))$ *0*
using *p-limseq-conv-0 p-limseq-mult-0-right* **by** *simp*
moreover from *that* **have** *p-limseq p* $(\lambda n.\ y * (X\ n - x))$ *0*
using *p-limseq-conv-0 p-limseq-constant-mult-0* **by** *blast*
ultimately have *p-limseq p* $(X * (\lambda n.\ Y\ n - y) + (\lambda n.\ y * (X\ n - x)))$ *0*
using *p-limseq-add*[*of p* $X * (\lambda n.\ Y\ n - y)$ *0*] **by** *force*
moreover have
$X * (\lambda n.\ Y\ n - y) + (\lambda n.\ y * (X\ n - x)) = (\lambda n.\ (X * Y)\ n - x * y)$
by (*auto simp add*: *algebra-simps*)
ultimately show *p-limseq p* $(X * Y)$ $(x * y)$ **using** *p-limseq-conv-0* **by** *metis*
qed

lemma *poly-continuous-p-open-nbhds*:
p-limseq p $(\lambda n.$ *poly f* $(X\ n))$ (*poly f x*) **if** *p-limseq p X x*
proof (*induct f*)
case (*pCons a f*)
have $(\lambda n.$ *poly* (*pCons a f*) $(X\ n)) = (\lambda n.\ a) + X * (\lambda n.$ *poly f* $(X\ n))$
by *auto*
with *that pCons*(*2*) **show** *?case* **using** *p-limseq-constant p-limseq-add p-limseq-mult*
by *simp*
qed (*simp flip*: *zero-fun-def*)

end

context *p-equality-depth-no-zero-divisors-1*
begin

lemma *p-limseq-pow*: *p-limseq p* $(\lambda k.\ (X\ k)\hat{}n)$ $(x\ \hat{}\ n)$ **if** *p-limseq p X x*
proof (*induct n*)
case (*Suc n*)
moreover have $(\lambda k.\ X\ k\ \hat{}\ Suc\ n) = X * (\lambda k.\ X\ k\ \hat{}\ n)$ **by** *auto*
ultimately show *?case* **using** *that p-limseq-mult* **by** *simp*
qed (*simp add*: *p-limseq-constant*)

lemma *p-limseq-poly*: *p-limseq p* $(\lambda n.$ *poly f* $(X\ n))$ (*poly f x*) **if** *p-limseq p X x*
proof (*induct f*)
case (*pCons a f*)
have $(\lambda n.$ *poly* (*pCons a f*) $(X\ n)) = (\lambda n.\ a) + X * (\lambda n.$ *poly f* $(X\ n))$
by *auto*
with *that pCons*(*2*) **show** *?case* **using** *p-limseq-mult p-limseq-add p-limseq-constant*
by *auto*
qed (*simp flip*: *zero-fun-def*)

end

context *p-equality-depth-div-inv*
begin

lemma *p-limseq-inverse*:

p-limseq p ($\lambda n.$ *inverse* (*X n*)) (*inverse x*)
if $\neg$ *p-equal p x 0* **and** *p-limseq p X x*
proof
fix *n*
from *that* **obtain** *K1* **where** *K1*: $\forall k \geq K1.$ *p-depth p* (*X k*) $\leq$ *p-depth p x*
using *p-limseq-eventually-bdd-depth eventually-sequentially* **by** *fastforce*
from *that* **obtain** *K2* **where** *K2*: $\forall k \geq K2.$ $\neg$ *p-equal p* (*X k*) *0*
using *p-limseq-eventually-nequiv-0 eventually-sequentially* **by** *force*
from *that*(*2*) **obtain** *K3*
where *K3*: *p-limseq-condition p X x* ($|n|$ + *2* $*$ *p-depth p x*) *K3*
by *fast*
define *K* **where** $K \equiv$ *Max* {*K1*, *K2*, *K3*}
have *p-limseq-condition p* ($\lambda n.$ *inverse* (*X n*)) (*inverse x*) *n K*
proof
fix *k* **assume** *k*: $k \geq K$ **and** *X-k*: $\neg$ *p-equal p* (*inverse* (*X k*)) (*inverse x*)
from *k*(*1*) *K-def K2* **have** *p-equal p* (*inverse* (*X k*) $*$ *X k*) *1* **using** *left-inverse-equiv* **by** *simp*
moreover from *that*(*1*) **have** *p-equal p* (*x* $*$ *inverse x*) *1* **using** *right-inverse-equiv* **by** *simp*
ultimately have *p-equal p* (*inverse* (*X k*) $*$ (*x* $-$ *X k*) $*$ *inverse x*) (*inverse* (*X k*) $-$ *inverse x*)
using *minus mult-left*[*of p x* $*$ *inverse x 1 inverse* (*X k*)]
mult-right[*of p inverse* (*X k*) $*$ *X k 1 inverse x*]
by (*simp add*: *algebra-simps*)
with *X-k* **have**
p-depth p (*inverse* (*X k*) $-$ *inverse x*) =
p-depth p (*inverse* (*X k*)) + *p-depth p* (*X k* $-$ *x*) + *p-depth p* (*inverse x*)
using *depth-equiv trans-right conv-0 depth-mult3-additive depth-diff* **by** *metis*
also from *k X-k K-def K1 K2 K3* **have** ... $>$ *n*
using *inverse-depth inverse-equiv-iff-equiv p-limseq-conditionD*
by (*fastforce simp add*: *algebra-simps*)
finally show *p-depth p* (*inverse* (*X k*) $-$ *inverse x*) $>$ *n* **by** *force*
qed
thus $\exists K.$ *p-limseq-condition p* ($\lambda n.$ *inverse* (*X n*)) (*inverse x*) *n K* **by** *blast*
qed

lemma *p-limseq-divide*:
p-limseq p ($\lambda n.$ *X n* / *Y n*) (*x* / *y*)
if $\neg$ *p-equal p y 0* **and** *p-limseq p X x* **and** *p-limseq p Y y*
proof−
have ($\lambda n.$ *X n* / *Y n*) = *X* $*$ ($\lambda n.$ *inverse* (*Y n*)) **using** *divide-inverse* **by** *force*
with *that* **show** *?thesis* **using** *p-limseq-mult p-limseq-inverse divide-inverse* **by** *simp*
qed

end

context *global-p-equality-depth*
begin

lemma *p-limseq-p-restrict-iff*:
p-limseq p (λn. *p-restrict* (X n) P) $x \longleftrightarrow$ *p-limseq p X x* **if** P p
using *that sym p-restrict-equiv p-limseq-p-cong*[*of 0 p* λn. *p-restrict* (X n) P X x x]
p-limseq-p-cong[*of 0 p X* λn. *p-restrict* (X n) P x x]
by *auto*

lemma *p-limseq-p-restricted*: *p-limseq p* (λn. *p-restrict* (X n) P) *0* **if** $\neg$ P p
using *that p-restrict-equiv0 p-limseq-p-cong* **by** *blast*

end

Base of open sets at local places

context *global-p-equal-depth*
begin

abbreviation *p-open-nbhd* :: *'a* $\Rightarrow$ *int* $\Rightarrow$ *'b* $\Rightarrow$ *'b set*
where *p-open-nbhd p n x* $\equiv$ x +*o p-depth-set p n*

abbreviation *local-p-open-nbhds p* $\equiv$ { *p-open-nbhd p n x* | *n x*. *True* }
abbreviation *p-open-nbhds* $\equiv$ ($\bigcup p$. *local-p-open-nbhds p*)

lemma *p-open-nbhd*: $x \in$ *p-open-nbhd p n x*
unfolding *elt-set-plus-def* **using** *p-depth-set-0* **by** *force*

lemma *local-p-open-nbhd-is-open*: *generate-topology* (*local-p-open-nbhds p*) (*p-open-nbhd p n x*)
using *generate-topology.intros*(*4*)[*of - local-p-open-nbhds p*] **by** *blast*

lemma *p-open-nbhd-is-open*: *generate-topology p-open-nbhds* (*p-open-nbhd p n x*)
using *generate-topology.intros*(*4*)[*of - p-open-nbhds*] **by** *blast*

lemma *local-p-open-nbhds-are-coarser*:
generate-topology (*local-p-open-nbhds p*) $S \Longrightarrow$ *generate-topology p-open-nbhds S*
proof (
induct S rule: *generate-topology.induct*, *rule generate-topology.UNIV*,
use generate-topology.Int **in** *blast*, *use generate-topology.UN* **in** *blast*
)
case (*Basis S*)
hence $S \in$ *p-open-nbhds* **by** *blast*
thus *?case* **using** *generate-topology.Basis* **by** *meson*
qed

lemma *p-open-nbhd-diff-depth*:
p-depth p ($y - x$) $\geq n$ **if** $\neg$ *p-equal p y x* **and** $y \in$ *p-open-nbhd p n x*
proof−
from *that*(*2*) **have** $y - x \in$ *p-depth-set p n*
unfolding *elt-set-plus-def* **by** *fastforce*

```
  with that(1) show ?thesis using conv-0 p-depth-setD by blast
qed

lemma p-open-nbhd-eq-circle:
  p-open-nbhd p n x = {y. ¬ p-equal p y x ⟶ p-depth p (y − x) ≥ n}
proof safe

  fix y assume y ∉ p-open-nbhd p n x p-equal p y x
  thus False using conv-0 p-depth-set-equiv0 unfolding elt-set-plus-def by force
next
  fix y assume n ≤ p-depth p (y − x)
  thus y ∈ p-open-nbhd p n x
    unfolding elt-set-plus-def using p-depth-setI by force
qed (simp add: p-open-nbhd-diff-depth)

lemma p-open-nbhd-circle-multicentre:
  p-open-nbhd p n y = p-open-nbhd p n x if y ∈ p-open-nbhd p n x
proof−
  have ∗:
    ⋀x y. y ∈ p-open-nbhd p n x ⟹
      p-open-nbhd p n y ⊆ p-open-nbhd p n x
  proof clarify
    fix x y z assume yx: y ∈ p-open-nbhd p n x and zy: z ∈ p-open-nbhd p n y
    have ¬ p-equal p z x ⟶ p-depth p (z − x) ≥ n
    proof
      assume zx: ¬ p-equal p z x
      have zyx: z − x = (z − y) + (y − x) by auto
      consider
        p-equal p z y | p-equal p y x | (neq) ¬ p-equal p z y ¬ p-equal p y x
        by fast
      thus p-depth p (z − x) ≥ n
      proof (
        cases,
        metis zyx yx zx yx trans conv-0 depth-add-equiv0-left p-open-nbhd-diff-depth,
        metis zyx zy zx trans conv-0 depth-add-equiv0-right p-open-nbhd-diff-depth
      )
        case neq
        with yx zy have p-depth p (z − y) ≥ n and p-depth p (y − x) ≥ n
          using p-open-nbhd-diff-depth by simp-all
        hence min (p-depth p (z − y)) (p-depth p (y − x)) ≥ n by presburger
        moreover from neq zyx zx
          have  p-depth p (z − x) ≥ min (p-depth p (z − y)) (p-depth p (y − x))
          by    (metis conv-0 depth-nonarch)
        ultimately show ?thesis by linarith
      qed
    qed
    thus z ∈ p-open-nbhd p n x using p-open-nbhd-eq-circle by auto
  qed
  moreover from that have x ∈ p-open-nbhd p n y
```

using *p-open-nbhd-eq-circle sym depth-diff* **by** *force*
ultimately show *?thesis* **using** *that* **by** *auto*
qed

lemma *p-open-nbhd-circle-multicentre'*:
$x \in$ *p-open-nbhd p n y* $\longleftrightarrow$ $y \in$ *p-open-nbhd p n x*
using *p-open-nbhd p-open-nbhd-circle-multicentre* **by** *blast*

lemma *p-open-nbhd-p-restrict*:
p-open-nbhd p n (*p-restrict x P*) = *p-open-nbhd p n x* **if** *P p*
proof (*intro set-eqI*)
fix *y* **from** *that* **show**
$y \in$ *p-open-nbhd p n* (*p-restrict x P*) $\longleftrightarrow$
$y \in$ *p-open-nbhd p n x*
using *p-open-nbhd-eq-circle*[*of p-restrict x P p n*]
p-open-nbhd-eq-circle[*of x p n*] *p-restrict-equiv*[*of P p x*]
depth-diff-equiv[*of p y y p-restrict x P x*]
trans[*of p y p-restrict x P x*] *trans-left*[*of p y x p-restrict x P*]
by *force*
qed

lemma *p-restrict-p-open-nbhd-mem-iff*:
$y \in$ *p-open-nbhd p n x* $\longleftrightarrow$ *p-restrict y P* $\in$ *p-open-nbhd p n x* **if** *P p*
using *that p-open-nbhd-circle-multicentre' p-open-nbhd-p-restrict* **by** *auto*

lemma *p-open-nbhd-p-restrict-add-mixed-drop-right*:
p-open-nbhd p n (*p-restrict x P* + *p-restrict y Q*) = *p-open-nbhd p n x*
if *P p* **and** $\neg$ *Q p*
proof (*intro set-eqI*)
fix *z*
from *that* **have** $*$: *p-equal p* (*p-restrict x P* + *p-restrict y Q*) *x*
using *p-restrict-add-mixed-equiv-drop-right* **by** *simp*
hence
$z \in$ *p-open-nbhd p n* (*p-restrict x P* + *p-restrict y Q*) $\longleftrightarrow$
$\neg$ *p-equal p z x* $\longrightarrow$
p-depth p (z − (*p-restrict x P* + *p-restrict y Q*)) $\geq n$
using *p-open-nbhd-eq-circle trans trans-left* **by** *blast*
moreover have *p-depth p* (z − (*p-restrict x P* + *p-restrict y Q*)) = *p-depth p* (z
− x)
using $*$ *depth-diff-equiv* **by** *auto*
ultimately show
$z \in$ *p-open-nbhd p n* (*p-restrict x P* + *p-restrict y Q*) $\longleftrightarrow$
$z \in$ *p-open-nbhd p n x*
using *p-open-nbhd-eq-circle* **by** *auto*
qed

lemma *p-open-nbhd-p-restrict-add-mixed-drop-left*:
p-open-nbhd p n (*p-restrict x P* + *p-restrict y Q*) = *p-open-nbhd p n y*
if $\neg$ *P p* **and** *Q p*

using *that p-open-nbhd-p-restrict-add-mixed-drop-right*[*of Q p P y x*] **by** (*simp add*: *add.commute*)

lemma *p-open-nbhd-antimono*:
p-open-nbhd p n x ⊆ *p-open-nbhd p m x* **if** $m \leq n$
using *that p-open-nbhd-eq-circle* **by** *auto*

lemma *p-open-nbhds-open-subopen*:
generate-topology (*local-p-open-nbhds p*) *A* =
(∀ *a*∈*A*. ∃ *n*. *p-open-nbhd p n a* ⊆ *A*)
proof
show
generate-topology (*local-p-open-nbhds p*) *A* ⟹
∀ *a*∈*A*. ∃ *n*. *p-open-nbhd p n a* ⊆ *A*
proof (*induct A rule*: *generate-topology.induct*)
case (*Int A B*) **show** *?case*
proof
fix *x* **assume** *x*: *x* ∈ *A* ∩ *B*
from *x Int*(*2*) **obtain** *n-A* **where** *p-open-nbhd p n-A x* ⊆ *A* **by** *blast*
moreover from *x Int*(*4*) **obtain** *n-B* **where** *p-open-nbhd p n-B x* ⊆ *B* **by** *blast*
moreover define *n* **where** *n* = *max n-A n-B*
ultimately have *p-open-nbhd p n x* ⊆ *A* ∩ *B*
using *p-open-nbhd-antimono*[*of n-A n x p*] *p-open-nbhd-antimono*[*of n-B n x p*] **by** *auto*
thus ∃ *n*. *p-open-nbhd p n x* ⊆ *A* ∩ *B* **by** *blast*
qed
qed (*simp*, *fast*, *use p-open-nbhd-circle-multicentre* **in** *blast*)
next
assume ∗: ∀ *a*∈*A*. ∃ *n*. *p-open-nbhd p n a* ⊆ *A*
define *f* **where** *f* ≡ λ*a*. (*SOME n*. *p-open-nbhd p n a* ⊆ *A*)
have *A* = (⋃ *a*∈*A*. *p-open-nbhd p* (*f a*) *a*)
proof *safe*
fix *a* **show** *a* ∈ *A* ⟹ *a* ∈ (⋃ *a*∈*A*. *p-open-nbhd p* (*f a*) *a*)
using *p-open-nbhd* **by** *blast*
next
fix *x a* **assume** *a*: *a* ∈ *A* **and** *x*: *x* ∈ *p-open-nbhd p* (*f a*) *a*
from *a* ∗ **have** *ex-n*: ∃ *n*. *p-open-nbhd p n a* ⊆ *A* **by** *fastforce*
from *x f-def* **show** *x* ∈ *A* **using** *someI-ex*[*OF ex-n*] **by** *blast*
qed
moreover have ∀ *a*∈*A*. *generate-topology* (*local-p-open-nbhds p*) (*p-open-nbhd p* (*f a*) *a*)
using *generate-topology.Basis*[*of - local-p-open-nbhds p*] **by** *blast*
ultimately show *generate-topology* (*local-p-open-nbhds p*) *A*
using *generate-topology.UN*[*of* (λ*a*. *p-open-nbhd p* (*f a*) *a*) ‘ *A local-p-open-nbhds p*]
by *fastforce*
qed

```
lemma p-restrict-p-open-set-mem-iff:
  a ∈ A ⟷ p-restrict a P ∈ A
  if generate-topology (local-p-open-nbhds p) A and P p
 using that p-open-nbhds-open-subopen p-open-nbhd p-restrict-p-open-nbhd-mem-iff
by fast

lemma hausdorff:
  ∃ p n. (p-open-nbhd p n x) ∩ (p-open-nbhd p n y) = {} if x ≠ y
proof−
  from that obtain p :: 'a where p: ¬ p-equal p x y using global-imp-eq by blast
  define n where n ≡ p-depth p (x − y) + 1
  have (p-open-nbhd p n x) ∩ (p-open-nbhd p n y) = {}
    unfolding elt-set-plus-def
  proof (intro equals0I, clarify)
    fix a b
    assume  a : a ∈ p-depth-set p n
      and   b : b ∈ p-depth-set p n
      and   ab: x + a = y + b
    from ab have x − y = b − a by (simp add: algebra-simps)
    with p a b have p-depth p (x − y) ≥ n
      using p-depth-set-minus conv-0 p-depth-setD by metis
    with n-def show False by linarith
  qed
  thus ?thesis by blast
qed

sublocale t2-p-open-nbhds: t2-space generate-topology p-open-nbhds
proof (intro class.t2-space.intro, rule topological-space-generate-topology, unfold-locales)
  fix x y :: 'b assume x ≠ y
  from this obtain p and n where pn: (p-open-nbhd p n x) ∩ (p-open-nbhd p n
y) = {}
    using hausdorff by presburger
  thus
    ∃ V V'. generate-topology p-open-nbhds V ∧ generate-topology p-open-nbhds V'
∧
      x ∈ V ∧ y ∈ V' ∧ V ∩ V' = {}
    using p-open-nbhd-is-open p-open-nbhd by meson
qed

abbreviation p-open-nbhds-tendsto    ≡ t2-p-open-nbhds.tendsto
abbreviation p-open-nbhds-convergent ≡ t2-p-open-nbhds.convergent
abbreviation p-open-nbhds-LIMSEQ     ≡ t2-p-open-nbhds.LIMSEQ

lemma locally-limD: ∃ K. p-limseq-condition p X x n K if p-open-nbhds-LIMSEQ
X x
proof−
  from that have ∃ K. ∀ k≥K. X k ∈ p-open-nbhd p (n + 1) x
    using     p-open-nbhd-is-open p-open-nbhd
    unfolding t2-p-open-nbhds.tendsto-def eventually-sequentially
```

by *presburger*
thus $\exists K.$ *p-limseq-condition p X x n K* **using** *p-open-nbhd-eq-circle* **by** *force*
qed

lemma *globally-limseq-imp-locally-limseq*:
p-limseq p X x **if** *p-open-nbhds-LIMSEQ X x*
using *that locally-limD* **by** *blast*

lemma *globally-limseq-iff-locally-limseq*: *p-open-nbhds-LIMSEQ X x* = ($\forall$ *p. p-limseq p X x*)
proof (*standard*, *standard*, *use globally-limseq-imp-locally-limseq* **in** *fast*)
assume *X*: $\forall$ *p. p-limseq p X x*
show *p-open-nbhds-LIMSEQ X x*
unfolding *t2-p-open-nbhds.tendsto-def eventually-sequentially*
proof *clarify*
fix *S* :: *'b set*
have
generate-topology p-open-nbhds S $\Longrightarrow$ *x* $\in$ *S* $\Longrightarrow$
($\forall$ *p. p-limseq p X x*) $\Longrightarrow$ $\exists K. \forall k \geq K.$ *X k* $\in$ *S*
proof (*induct S rule*: *generate-topology.induct*)
case (*Int A B*)
from *Int*(*2*,*4*−*6*) **obtain** *Ka* **and** *Kb*
where $\forall k \geq Ka.$ *X k* $\in$ *A*
and $\forall k \geq Kb.$ *X k* $\in$ *B*
by *blast*
from *this* **have** $\forall k \geq$ *max Ka Kb. X k* $\in$ *A* $\cap$ *B* **by** *fastforce*
thus *?case* **by** *blast*
next
case (*Basis S*)
from *Basis*(*1*,*2*) **obtain** *p n* **where** *S* = *p-open-nbhd p n x*
using *p-open-nbhd-circle-multicentre* **by** *blast*
moreover from *Basis*(*3*) **have** $\exists K.$ *p-limseq-condition p X x* (*n* − *1*) *K* **by** *blast*
ultimately show *?case* **using** *p-limseq-conditionD p-open-nbhd-eq-circle* **by** *fastforce*
qed (*simp*, *blast*)
with *X* **show**
generate-topology p-open-nbhds S $\Longrightarrow$
x $\in$ *S* $\Longrightarrow$ $\exists K. \forall k \geq K.$ *X k* $\in$ *S*
by *fast*
qed
qed

lemma *p-open-nbhds-LIMSEQ-eventually-p-constant*:
p-equal p x c
if *p-open-nbhds-LIMSEQ X x* **and** $\forall_F$ *k in sequentially. p-equal p* (*X k*) *c*
proof−
have ¬ (¬ *p-equal p x c*)
proof

assume *contra*: ¬ *p-equal p x c*
define *d* **where** *d* ≡ *max 1* (*p-depth p* (*x* − *c*))
from *that*(*1*) **obtain** *K1* **where** *K1*: *p-limseq-condition p X x d K1*
using *locally-limD* **by** *blast*
from *that*(*2*) **obtain** *K2* **where** *K2*: ∀ *k*≥*K2*. *p-equal p* (*X k*) *c*
using *eventually-sequentially* **by** *auto*
define *K* **where** *K* ≡ *max K1 K2*
show *False*
proof (*cases p-equal p* (*X K*) *x*)
case *True* **with** *contra K2 K-def* **show** *False* **using** *trans-right*[*of p X K*] **by** *simp*
next
case *False*
with *K1 K-def* **have** *p-depth p* (*X K* − *x*) > *d* **using** *p-limseq-conditionD* **by** *simp*
with *d-def K2 K-def* **show** *False* **using** *depth-diff-equiv*[*of p X K c x x*] *depth-diff* **by** *simp*
qed
qed
thus *?thesis* **by** *blast*
qed

lemma *p-open-nbhds-LIMSEQ-p-restrict*:
p-equal p x 0 **if** *p-open-nbhds-LIMSEQ* (*λn. p-restrict* (*X n*) *P*) *x* **and** ¬ *P p*
using *that globally-limseq-iff-locally-limseq sym p-restrict-equiv0 p-limseq-0-iff p-limseq-p-cong*[*of 0 p 0 λn. p-restrict* (*X n*) *P x x*]
by *auto*

lemma *global-depth-set-p-open-nbhds-LIMSEQ-closed*:
x ∈ *global-depth-set n*
if *p-open-nbhds-LIMSEQ X x*
and ∀$_F$ *k in sequentially. X k* ∈ *global-depth-set n*
proof (*intro global-depth-setI''*)
fix *p*
from *that*(*2*) **have** ∀$_F$ *k in sequentially. X k* ∈ *p-depth-set p n*
using *eventually-sequentially global-depth-set* **by** *force*
with *that*(*1*) **show** *x* ∈ *p-depth-set p n*
using *globally-limseq-imp-locally-limseq p-depth-set-p-limseq-closed* **by** *blast*
qed

end

context *global-p-equal-depth-div-inv-w-inj-index-consts*
begin

lemma *shift-p-depth-p-open-nbhd*:
shift-p-depth p n ‘ *p-open-nbhd p m x* = *p-open-nbhd p* (*m* + *n*) (*shift-p-depth p n x*)
proof *safe*

fix *y* **assume** *y*: $y \in$ *p-open-nbhd p m x*
show *shift-p-depth p n y* $\in$ *p-open-nbhd p* $(m + n)$ (*shift-p-depth p n x*)
unfolding *p-open-nbhd-eq-circle*
proof *clarify*
assume $*$: $\neg$ *p-equal p* (*shift-p-depth p n y*) (*shift-p-depth p n x*)
hence $\neg$ *p-equal p y x* **using** *shift-p-depth-equiv-iff* **by** *force*
moreover from *this y* **have** *p-depth p* $(y - x) + n \geq m + n$
using *p-open-nbhd-eq-circle* **by** *auto*
ultimately show *p-depth p* (*shift-p-depth p n y* $-$ *shift-p-depth p n x*) $\geq m +$
n
using *conv-0 shift-p-depth-at-place shift-p-depth-minus* **by** *metis*
qed
next
fix *y* **assume** *y*: $y \in$ *p-open-nbhd p* $(m + n)$ (*shift-p-depth p n x*)
have *shift-p-depth p* $(-n)$ $y \in$ *p-open-nbhd p m x*
proof (*cases* $n = 0$, *use y* **in** *auto*)
case *n*: *False* **show** *?thesis*
proof (*cases p-equal p y* (*shift-p-depth p n x*))
case *True*
hence *p-equal p* (*shift-p-depth p* $(-n)$ *y*) *x*
using *shift-p-depth-equiv-iff*[*of p p* $-n$ *y*] *shift-shift-p-depth*[*of p* $-n$ *n x*] **by**
force
thus *shift-p-depth p* $(-n)$ $y \in$ *p-open-nbhd p m x* **using** *p-open-nbhd-eq-circle*
by *simp*
next
case *False*
moreover from *this y* **have** *p-depth p* ($y -$ *shift-p-depth p n x*) $+ (-n) \geq m$
using *p-open-nbhd-diff-depth* **by** *fastforce*
moreover have *shift-p-depth p* $(-n)$ (*shift-p-depth p n x*) $= x$
using *shift-shift-p-depth* **by** *simp*
ultimately have *p-depth p* (*shift-p-depth p* $(-n)$ $y - x$) $\geq m$
using *n conv-0 shift-p-depth-at-place shift-p-depth-minus* **by** *metis*
with *False* **show** *shift-p-depth p* $(-n)$ $y \in$ *p-open-nbhd p m x*
using *p-open-nbhd-eq-circle shift-p-depth-equiv-iff*[*of p p* $-n$ *y shift-p-depth*
p n x]
shift-shift-p-depth
by *blast*
qed
qed
moreover have $y =$ *shift-p-depth p n* (*shift-p-depth p* $(-n)$ *y*)
using *shift-shift-p-depth* **by** *auto*
ultimately show $y \in$ *shift-p-depth p n* ' *p-open-nbhd p m x* **by** *blast*
qed

lemma *shift-p-depth-p-open-set*:
generate-topology (*local-p-open-nbhds p*) (*shift-p-depth p n* ' *A*)
if *generate-topology* (*local-p-open-nbhds p*) *A*
proof (*rule iffD2*, *rule p-open-nbhds-open-subopen*, *clarify*)
fix *a* **assume** $a \in A$

with *that* **obtain** *m* **where** *p-open-nbhd p m a* ⊆ *A*
using *p-open-nbhds-open-subopen* **by** *fastforce*
hence *shift-p-depth p n ‘ p-open-nbhd p m a* ⊆ *shift-p-depth p n ‘ A* **by** *fast*
moreover have
shift-p-depth p n ‘ p-open-nbhd p m a = *p-open-nbhd p* (*m* + *n*) (*shift-p-depth p n a*)
using *shift-p-depth-p-open-nbhd* **by** *blast*
ultimately
show ∃ *m*. *p-open-nbhd p m* (*shift-p-depth p n a*) ⊆ *shift-p-depth p n ‘ A*
by *fast*
qed

end

locale *p-complete-global-p-equal-depth* = *global-p-equal-depth* +
assumes *complete*:
p-cauchy p X ⟹ *p-open-nbhds-convergent* (λ*n*. *p-restrict* (*X n*) ((=) *p*))
begin

lemma *p-cauchyE*:
assumes *p-cauchy p X*
obtains *x* **where** *p-open-nbhds-LIMSEQ* (λ*n*. *p-restrict* (*X n*) ((=) *p*)) *x*
using *assms complete t2-p-open-nbhds.convergent-def*
by *blast*

end

Hensel’s Lemma.

context *p-equality-div-inv*
begin

— A sequence to approach a root from within the ring of integers.
primrec *hensel-seq* :: *'a* ⇒ *'b poly* ⇒ *'b* ⇒ *nat* ⇒ *'b*
where *hensel-seq p f x 0* = *x* |
hensel-seq p f x (*Suc n*) = (
if p-equal p (*poly f* (*hensel-seq p f x n*)) *0 then hensel-seq p f x n*
else
hensel-seq p f x n − (*poly f* (*hensel-seq p f x n*)) / (*poly* (*polyderiv f*) (*hensel-seq p f x n*))
)

lemma *constant-hensel-seq-case*:
k ≥ *n* ⟹ *hensel-seq p f x k* = *hensel-seq p f x n*
if *p-equal p* (*poly f* (*hensel-seq p f x n*)) *0*
proof (*induct k*)
case (*Suc k*) **thus** *?case* **by** (*cases n* = *Suc k*, *simp*, *use that* **in** *force*)
qed *simp*

end

context *p-equality-depth-div-inv*
begin

— Context for inductive steps leading to Hensel's Lemma.
context
  fixes *p* :: *'a* **and** *f* :: *'b poly* **and** *x* :: *'b* **and** *n* :: *nat*
  assumes *f-deg* : *degree f* > *0*
    and *f-depth* : *set* (*coeffs f*) ⊆ *p-depth-set p 0*
    and *d-hensel*: *hensel-seq p f x n* ∈ *p-depth-set p 0*
    and *f* : ¬ *p-equal p* (*poly f* (*hensel-seq p f x n*)) *0*
    and *deriv-f* : ¬ *p-equal p* (*poly* (*polyderiv f*) (*hensel-seq p f x n*)) *0*
    and *df* :
      p-depth p (*poly f* (*hensel-seq p f x n*)) >
        2 ∗ *p-depth p* (*poly* (*polyderiv f*) (*hensel-seq p f x n*))
begin

lemma *hensel-seq-step-1*:
  p-depth p (*poly f* (*hensel-seq p f x* (*Suc n*))) >
    p-depth p (*poly f* ((*hensel-seq p f x n*)))
  if *next-f*: ¬ *p-equal p* (*poly f* (*hensel-seq p f x* (*Suc n*))) *0*
proof (*cases f*)
  case (*pCons c q*)
  define *a a'* **where** *a* ≡ *hensel-seq p f x n* **and** *a'* ≡ *hensel-seq p f x* (*Suc n*)
  define *b* **where** *b* ≡ − ((*poly f a*) / (*poly* (*polyderiv f*) *a*))
  define *c-add-poly* :: *nat* ⇒ *'b poly*
    where *c-add-poly* ≡ *coeff* (*additive-poly-poly f*)
  define *sum-fun Suc-sum-fun* :: *nat* ⇒ *'b*
    where *sum-fun* ≡ λ*i*. *poly* (*c-add-poly i*) *a* ∗ *b* ^ *i*
    and *Suc-sum-fun* ≡ λ*i*. *poly* (*c-add-poly* (*Suc i*)) *a* ∗ *b* ^ (*Suc i*)
  define *S* :: *'b* **where** *S* ≡ *sum Suc-sum-fun* {*1..degree q*}
  moreover from *sum-fun-def Suc-sum-fun-def* **have** *Suc-sum-fun*: *Suc-sum-fun* = *sum-fun* ∘ *Suc*
    by *auto*
  moreover have {*0..degree q*} = {*..degree q*} **by** *auto*
  moreover from *f-deg pCons* **have** *degree f* = *Suc* (*degree q*) **by** *auto*
  ultimately have *poly f a'* = *poly f a* + *poly* (*polyderiv f*) *a* ∗ *b* + *S*
   using *f a-def a'-def b-def c-add-poly-def sum-fun-def additive-poly-poly-decomp*[*of f a b*]
      sum-up-index-split[*of sum-fun 0*] *sum.atLeast-Suc-atMost-Suc-shift*[*of sum-fun 0*]
      sum-up-index-split[*of sum-fun* ∘ *Suc 0*] *additive-poly-poly-coeff0* [*of f*]
      additive-poly-poly-coeff1 [*of f*]
    by (*simp add*: *ac-simps*)
  with *deriv-f a-def b-def* **have** *equiv-sum*: *p-equal p* (*poly f a'*) *S*
    using *minus-divide-left times-divide-eq-right mult-divide-cancel-left add-left*
      add.right-inverse add-equiv0-left
    by *metis*
  with *next-f a'-def* **have** ¬ *p-equal p S 0* **using** *trans* **by** *fast*

from *this f-depth d-hensel a-def c-add-poly-def Suc-sum-fun-def S-def* **obtain** *j*
where *j*:
j ∈ {*1..degree q*} *p-depth p S* ≥ *int* (*Suc j*) ∗ *p-depth p b*
using *sum-poly-additive-poly-poly-depth-bound*[*of f p*] **by** *force*
from *f-depth f deriv-f d-hensel df a-def b-def* **have** *p-depth p b* ≥ *0*
using *p-depth-poly-deriv-quotient depth-uminus*[*of p b*] **by** *fastforce*
moreover from *j*(*1*) **have** *2* ≤ *int* (*Suc j*) **by** *force*
ultimately have *int* (*Suc j*) ∗ *p-depth p b* ≥ *2* ∗ *p-depth p b* **using** *mult-right-mono*
by *blast*
moreover from *f deriv-f a-def b-def*
have *p-depth p b* = *p-depth p* (*poly f a*) − *p-depth p* (*poly* (*polyderiv f*) *a*)
using *divide-equiv-0-iff depth-uminus divide-depth* **by** *metis*
ultimately show *p-depth p* (*poly f a′*) > *p-depth p* (*poly f a*)
using *j*(*2*) *df a-def equiv-sum depth-equiv* **by** *force*
qed

lemma *hensel-seq-step-2*:
¬ *p-equal p* (*poly* (*polyderiv f*) (*hensel-seq p f x* (*Suc n*))) *0*
p-depth p (*poly* (*polyderiv f*) (*hensel-seq p f x* (*Suc n*))) =
p-depth p (*poly* (*polyderiv f*) (*hensel-seq p f x n*))
proof−

define *a a′* **where** *a* ≡ *hensel-seq p f x n* **and** *a′* ≡ *hensel-seq p f x* (*Suc n*)
define *b* **where** *b* ≡ − ((*poly f a*) / (*poly* (*polyderiv f*) *a*))
from *f deriv-f a-def b-def*
have *db*: *p-depth p b* = *p-depth p* (*poly f a*) − *p-depth p* (*poly* (*polyderiv f*) *a*)
using *divide-equiv-0-iff depth-uminus divide-depth* **by** *metis*

have
p-depth p (*poly* (*polyderiv f*) (*a′*)) = *p-depth p* (*poly* (*polyderiv f*) *a*) ∧
¬ *p-equal p* (*poly* (*polyderiv f*) *a′*) *0*
proof (*cases degree* (*polyderiv f*))
case (*Suc m*)
define *c-add-poly* :: *nat* ⇒ *′b poly*
where *c-add-poly* ≡ *coeff* (*additive-poly-poly* (*polyderiv f*))
define *sum-fun* :: *nat* ⇒ *′b*
where *sum-fun* ≡ λ*i*. *poly* (*c-add-poly i*) *a* ∗ *b* ^ *i*
define *S* :: *′b* **where** *S* ≡ *sum* (*sum-fun* ∘ *Suc*) {*0..m*}
from *f a-def a′-def b-def Suc c-add-poly-def sum-fun-def S-def* **have** *diff-eq-sum*:
poly (*polyderiv f*) *a′* − *poly* (*polyderiv f*) *a* = *S*
using *additive-poly-poly-decomp*[*of polyderiv f a b*] *sum-up-index-split*[*of*
sum-fun 0]
sum.atLeast-Suc-atMost-Suc-shift[*of sum-fun 0*]
by (*simp add*: *algebra-simps additive-poly-poly-coeff0*)
show *?thesis*
proof (*cases p-equal p S 0*, *safe*)
case *True*
from *True* **show**
p-depth p (*poly* (*polyderiv f*) *a′*) = *p-depth p* (*poly* (*polyderiv f*) *a*)

using *diff-eq-sum conv-0 depth-equiv* **by** *metis*
from *deriv-f a-def True* **show** *p-equal p (poly (polyderiv f) a′) 0* $\Longrightarrow$ *False*
using *diff-eq-sum conv-0 trans-right* **by** *blast*
next
case *False*
from *this f-depth d-hensel a-def S-def sum-fun-def c-add-poly-def* **obtain** *j*
where *j*: *p-depth p S* $\geq$ *int (Suc j)* $*$ *p-depth p b*
using *p-depth-set-polyderiv[of f p 0]*
sum-poly-additive-poly-poly-depth-bound[of polyderiv f p a b 0 m]
by *force*
from *f-depth d-hensel deriv-f a-def* **have** *p-depth p (poly (polyderiv f) a)* $\geq$ *0*
using *poly-p-depth-set-polyderiv[of 0 f p a] p-depth-setD[of p poly (polyderiv f) a 0]*
by *simp*
hence
p-depth p (poly (polyderiv f) a) $\leq$
int (Suc j) $*$ *(2* $*$ *p-depth p (poly (polyderiv f) a)* $-$ *p-depth p (poly (polyderiv f) a))*
using *mult-right-mono[of 1 int (Suc j)]* **by** *auto*
also from *df a-def* **have** ... $<$ *int (Suc j)* $*$ *p-depth p b* **using** *db* **by** *simp*
finally have $*$:
p-depth p (poly (polyderiv f) a) $<$
p-depth p (poly (polyderiv f) a′ $-$ *poly (polyderiv f) a)*
using *j diff-eq-sum* **by** *fastforce*
with *deriv-f a-def*
show *p-depth p (poly (polyderiv f) a′) = p-depth p (poly (polyderiv f) a)*
using *depth-diff-cancel-imp-eq-depth-right* **by** *blast*
show *p-equal p (poly (polyderiv f) a′) 0* $\Longrightarrow$ *False*
using $*$ *depth-diff-equiv0-left* **by** *fastforce*
qed
qed (*elim degree-eq-zeroE, simp, use deriv-f* **in** *fastforce*)

thus *p-depth p (poly (polyderiv f) (a′)) = p-depth p (poly (polyderiv f) a)*
and ¬ *p-equal p (poly (polyderiv f) a′) 0*
by *auto*

qed

end

— Context for Hensel's Lemma.
context
fixes *p* :: *′a* **and** *f* :: *′b poly* **and** *x* :: *′b*
assumes *f-deg* : *degree f* $>$ *0*
and *f-depth* : *set (coeffs f)* $\subseteq$ *p-depth-set p 0*
and *d-init* : *x* $\in$ *p-depth-set p 0*
and *init-deriv* :
¬ *p-equal p (poly f x) 0* $\longrightarrow$ ¬ *p-equal p (poly (polyderiv f) x) 0*
and *d-init-deriv*:

$\neg$ *p-equal p* (*poly f x*) *0* $\longrightarrow$
p-depth p (*poly f x*) > *2* $*$ *p-depth p* (*poly* (*polyderiv f*) *x*)
begin

lemma
hensel-seq-adelic-int : *hensel-seq p f x n* $\in$ *p-depth-set p 0* (**is** *?A*)
and *hensel-seq-poly-depth* :
$\neg$ *p-equal p* (*poly f* (*hensel-seq p f x n*)) *0* $\longrightarrow$
p-depth p (*poly f* (*hensel-seq p f x n*)) $\geq$ *p-depth p* (*poly f x*) + *int n*
(**is** *?B*)
and *hensel-seq-deriv* :
$\neg$ *p-equal p* (*poly f* (*hensel-seq p f x n*)) *0* $\longrightarrow$
$\neg$ *p-equal p* (*poly* (*polyderiv f*) (*hensel-seq p f x n*)) *0*
(**is** *?C*)
and *hensel-seq-deriv-depth*:
$\neg$ *p-equal p* (*poly f* (*hensel-seq p f x n*)) *0* $\longrightarrow$
p-depth p (*poly* (*polyderiv f*) (*hensel-seq p f x n*)) = *p-depth p* (*poly* (*polyderiv f*) *x*)
(**is** *?D*)
and *hensel-seq-depth-ineq*:
$\neg$ *p-equal p* (*poly f* (*hensel-seq p f x n*)) *0* $\longrightarrow$
p-depth p (*poly f* (*hensel-seq p f x n*)) >
2 $*$ *p-depth p* (*poly* (*polyderiv f*) (*hensel-seq p f x n*))
(**is** *?E*)
proof−

have *?A* $\wedge$ *?B* $\wedge$ *?C* $\wedge$ *?D* $\wedge$ *?E*
proof (*induct n*, *use d-init init-deriv d-init-deriv* **in** *force*)
case (*Suc n*)
define *a a'* **where** *a* $\equiv$ *hensel-seq p f x n* **and** *a'* $\equiv$ *hensel-seq p f x* (*Suc n*)
show *?case*
proof (*cases p-equal p* (*poly f a*) *0*)
case *False* **show** *?thesis*
proof (*fold a-def a'-def*, *safe*)

from *False Suc a-def a'-def*
have *prev-int* : *a* $\in$ *p-depth-set p 0*
and *prev-poly-depth* : *p-depth p* (*poly f a*) $\geq$ *p-depth p* (*poly f x*) + *int n*
and *prev-deriv* : $\neg$ *p-equal p* (*poly* (*polyderiv f*) *a*) *0*
and *prev-deriv-depth*:
p-depth p (*poly* (*polyderiv f*) *a*) = *p-depth p* (*poly* (*polyderiv f*) *x*)
and *prev-depth-ineq* : *p-depth p* (*poly f a*) > *2* $*$ *p-depth p* (*poly* (*polyderiv f*) *a*)
by *auto*

define *b* **where** *b* $\equiv$ (*poly f a*) / (*poly* (*polyderiv f*) *a*)
with *f-depth False a-def a'-def b-def* **show** *a'* $\in$ *p-depth-set p 0*
using *prev-int prev-deriv prev-depth-ineq p-depth-set-minus*
p-depth-set-poly-deriv-quotient[*of 0 f p*]

by *force*

from *f-deg f-depth a-def a′-def False*
show *next-deriv*: *p-equal p* (*poly* (*polyderiv f*) *a′*) *0* ⟹ *False*
using *hensel-seq-step-2*(*1*) *prev-int prev-deriv prev-depth-ineq* **by** *blast*

from *f-deg f-depth a-def a′-def False* **show** *next-deriv-depth*:
p-depth p (*poly* (*polyderiv f*) *a′*) = *p-depth p* (*poly* (*polyderiv f*) *x*)
by (*metis hensel-seq-step-2*(*2*) *prev-int prev-deriv prev-depth-ineq prev-deriv-depth*)

moreover assume ∗: ¬ *p-equal p* (*poly f a′*) *0*

ultimately
show *p-depth p* (*poly f a′*) > *2* ∗ *p-depth p* (*poly* (*polyderiv f*) *a′*)
using *f-deg f-depth a-def a′-def False hensel-seq-step-1 prev-int prev-deriv*
prev-depth-ineq next-deriv prev-depth-ineq prev-deriv-depth next-deriv-depth
by *force*

from *f-deg f-depth a-def a′-def False*
show *p-depth p* (*poly f a′*) ≥ *p-depth p* (*poly f x*) + *int* (*Suc n*)
using ∗ *hensel-seq-step-1 prev-int prev-deriv prev-depth-ineq next-deriv*
prev-poly-depth
by *force*

qed
qed (*simp add*: *a-def Suc*)
qed

thus *?A ?B ?C ?D ?E* **by** *auto*

qed

lemma *hensel-seq-cauchy*: *p-cauchy p* (*hensel-seq p f x*)
proof (*rule iffD2*, *rule p-consec-cauchy*, *clarify*)
fix *n*
show ∃ *K*. *p-consec-cauchy-condition p* (*hensel-seq p f x*) *n K*
proof (*cases* ∃ *N*. *p-equal p* (*poly f* (*hensel-seq p f x N*)) *0*)
case *True*
from *this* **obtain** *N* **where** *N*: *p-equal p* (*poly f* (*hensel-seq p f x N*)) *0* **by** *fast*
have *p-consec-cauchy-condition p* (*hensel-seq p f x*) *n N*
proof (*intro p-consec-cauchy-conditionI*)
fix *k*
assume *k* ≥ *N*
and ¬ *p-equal p* (*hensel-seq p f x* (*Suc k*)) (*hensel-seq p f x k*)
with *N* **show** *p-depth p* (*hensel-seq p f x* (*Suc k*) − *hensel-seq p f x k*) > *n*
using *constant-hensel-seq-case*[*of p f x N k*] **by** *auto*
qed
thus *?thesis* **by** *blast*
next

case *False*
hence $*$: $\forall$ *n*. $\neg$ *p-equal p* (*poly f* (*hensel-seq p f x n*)) *0* **by** *fast*
define *d* **where** *d* $\equiv$ *p-depth p* (*poly f x*) $-$ *p-depth p* (*poly* (*polyderiv f*) *x*)
define *K* **where** *K* $\equiv$ *nat* (*n* $-$ *d* $+$ *1*)
have *p-consec-cauchy-condition p* (*hensel-seq p f x*) *n K*
proof (*intro p-consec-cauchy-conditionI*)
fix *k* **assume** *k* $\geq$ *K*
moreover define *a a'*
where *a* $\equiv$ *hensel-seq p f x k* **and** *a'* $\equiv$ *hensel-seq p f x* (*Suc k*)
moreover define *b* **where** *b* $\equiv$ $-$ ((*poly f a*) / (*poly* (*polyderiv f*) *a*))
moreover from *this a-def*
have *p-depth p b* $=$ *p-depth p* (*poly f a*) $-$ *p-depth p* (*poly* (*polyderiv f*) *a*)
using $*$ *hensel-seq-deriv divide-equiv-0-iff depth-uminus divide-depth* **by** *metis*
ultimately show *p-depth p* (*a'* $-$ *a*) $>$ *n*
using *d-def K-def* $*$ *hensel-seq-deriv-depth hensel-seq-poly-depth* **by** *fastforce*
qed
thus *?thesis* **by** *blast*
qed
qed

lemma *p-limseq-poly-hensel-seq*: *p-limseq p* (λ*n*. *poly f* (*hensel-seq p f x n*)) *0*
proof
fix *n*
define *K* **where** *K* $\equiv$ *nat* (*n* $-$ *p-depth p* (*poly f x*) $+$ *1*)
have *p-limseq-condition p* (λ*n*. *poly f* (*hensel-seq p f x n*)) *0 n K*
proof (*intro p-limseq-conditionI*)
fix *k* **assume** *k* $\geq$ *K* **and** $\neg$ *p-equal p* (*poly f* (*hensel-seq p f x k*)) *0*
with *K-def* **show** *p-depth p* (*poly f* (*hensel-seq p f x k*) $-$ *0*) $>$ *n*
using *hensel-seq-poly-depth* **by** *fastforce*
qed
thus $\exists$ *K*. *p-limseq-condition p* (λ*n*. *poly f* (*hensel-seq p f x n*)) *0 n K* **by** *fast*
qed

end

end

locale *p-complete-global-p-equal-depth-div-inv* $=$
global-p-equal-depth-div-inv $+$ *p-complete-global-p-equal-depth*
begin

lemma *hensel*:
fixes *p* :: *'a* **and** *f* :: *'b poly* **and** *x* :: *'b*
assumes *degree f* $>$ *0*
and *set* (*coeffs f*) $\subseteq$ *p-depth-set p 0*
and *x* $\in$ *p-depth-set p 0*
and

$\neg$ *p-equal p* (*poly f x*) *0* $\longrightarrow$ $\neg$ *p-equal p* (*poly* (*polyderiv f*) *x*) *0*
and
$\neg$ *p-equal p* (*poly f x*) *0* $\longrightarrow$
p-depth p (*poly f x*) > *2* $*$ *p-depth p* (*poly* (*polyderiv f*) *x*)
obtains *z* **where** *z* ∈ *p-depth-set p 0* **and** *p-equal p* (*poly f z*) *0*
proof−
define *X* **where** *X* ≡ *hensel-seq p f x*
define *res-X* **where** *res-X* ≡ λ*n*. *p-restrict* (*X n*) ((=) *p*)
from *assms X-def res-X-def* **obtain** *z* **where** *p-open-nbhds-LIMSEQ res-X z*
using *hensel-seq-cauchy p-cauchyE* **by** *blast*
hence *p-limseq p res-X z* **using** *globally-limseq-iff-locally-limseq* **by** *blast*
with *res-X-def* **have** *X-to-z*: *p-limseq p X z* **using** *p-limseq-p-restrict-iff* **by** *blast*
with *assms X-def* **have** *p-equal p* (*poly f z*) *0*
using *poly-continuous-p-open-nbhds p-limseq-poly-hensel-seq p-limseq-unique* **by** *metis*
moreover from *assms X-def* **have** *z* ∈ *p-depth-set p 0*
using *X-to-z hensel-seq-adelic-int p-depth-set-p-limseq-closed* **by** *fastforce*
ultimately show *thesis* **using** *that* **by** *blast*
qed

end

3.7.2 Globally

We use bounded depth to create a global metric.

context *p-equality-depth*
begin

definition *bdd-global-depth* :: *'b* ⇒ *nat*
where
bdd-global-depth x =
(*if globally-p-equal x 0 then 0 else Inf* {*nat* (*p-depth p x* + *1*) | *p*. $\neg$ *p-equal p x 0*})
— The plus-one is to distinguish depth-0 from negative depth.

lemma *bdd-global-depth-0*[*simp*]: *globally-p-equal x 0* ⟹ *bdd-global-depth x* = *0*
unfolding *bdd-global-depth-def* **by** *simp*

lemma *bdd-global-depthD*:
bdd-global-depth x = *Inf* {*nat* (*p-depth p x* + *1*) | *p*. $\neg$ *p-equal p x 0*}
if $\neg$ *globally-p-equal x 0*
using *that* **unfolding** *bdd-global-depth-def* **by** *argo*

lemma *bdd-global-depthD-as-image*:
bdd-global-depth x = (*INF p*∈{*p*. $\neg$ *p-equal p x 0*}. *nat* (*p-depth p x* + *1*))
if $\neg$ *globally-p-equal x 0*
using *that image-Collect* **unfolding** *bdd-global-depth-def* **by** *metis*

lemma *bdd-global-depth-cong*:

bdd-global-depth x = *bdd-global-depth y* **if** *globally-p-equal x y*
proof (*cases globally-p-equal x 0*)
case *True* **with** *that* **show** *?thesis* **using** *globally-p-equal-trans-right*[*of x y 0*] **by** *fastforce*
next
case *False*
moreover from *that* **have** {*p*. ¬ *p-equal p x 0*} = {*p*. ¬ *p-equal p y 0*}
using *trans*[*of - x y 0*] *trans-right*[*of - x y 0*] **by** *auto*
moreover from *that*
have (*λp*. *nat* (*p-depth p x* + *1*)) = (*λp*. *nat* (*p-depth p y* + *1*))
using *depth-equiv*[*of - x y*]
by *simp*
ultimately show *?thesis* **using** *that globally-p-equal-trans bdd-global-depthD-as-image* **by** *metis*
qed

lemma *bdd-global-depth-le*:
bdd-global-depth x ≤ *nat* (*p-depth p x* + *1*) **if** ¬ *p-equal p x 0*
using *that bdd-global-depthD-as-image cINF-lower*[*of λp*. *nat* (*p-depth p x* + *1*)]
by *fastforce*

lemma *bdd-global-depth-greatest*:
bdd-global-depth x ≥ *B*
if ¬ *p-equal p x 0*
and ∀ *q*. ¬ *p-equal q x 0* ⟶ *nat* (*p-depth q x* + *1*) ≥ *B*
using *that bdd-global-depthD-as-image*
cINF-greatest[*of*
{*p*. ¬ *p-equal p x 0*} *B λp*. *nat* (*p-depth p x* + *1*)
]
by *fastforce*

lemma *bdd-global-depth-witness*:
assumes ¬ *globally-p-equal x 0*
obtains *p* **where** ¬ *p-equal p x 0* **and** *bdd-global-depth x* = *nat* (*p-depth p x* + *1*)
proof−
define *D* **where** *D* ≡ {*nat* (*p-depth p x* + *1*) | *p*. ¬ *p-equal p x 0*}
from *assms* **obtain** *q* **where** ¬ *p-equal q x 0* **by** *blast*
with *D-def* **have** *nat* (*p-depth q x* + *1*) ∈ *D* **by** *force*
with *assms D-def* **obtain** *p*
where ¬ *p-equal p x 0*
and *bdd-global-depth x* = *nat* (*p-depth p x* + *1*)
using *wellorder-InfI*[*of - D*] *bdd-global-depthD*
by *auto*
with *that* **show** *thesis* **by** *meson*
qed

lemma *bdd-global-depth-eq-0*:
bdd-global-depth x = *0* **if** *p-depth p x* < *0*

using *that depth-equiv-0 bdd-global-depth-le*[*of p x*] **by** *fastforce*

lemma *bdd-global-depth-eq-0-iff*:
bdd-global-depth x = *0* $\longleftrightarrow$
($\exists$ *p*. *p-depth p x* < *0*) $\vee$ (*globally-p-equal x 0*)
(**is** *?L* = *?R*)
proof
assume *L*: *?L* **show** *?R*
proof *clarify*
assume $\neg$ *globally-p-equal x 0*
from *this* **obtain** *p* **where** $\neg$ *p-equal p x 0* **and** *bdd-global-depth x* = *nat* (*p-depth p x* + *1*)
using *bdd-global-depth-witness*[*of x*] **by** *auto*
with *L* **have** *p-depth p x* < *0* **by** *force*
thus $\exists$ *p*. *p-depth p x* < *0* **by** *blast*
qed
next
assume *?R* **thus** *?L* **by** (*cases globally-p-equal x 0*, *use bdd-global-depth-eq-0* **in** *auto*)
qed

lemma *bdd-global-depth-diff*: *bdd-global-depth* (*x* − *y*) = *bdd-global-depth* (*y* − *x*)
using *depth-diff conv-0 sym-iff*[*of - x y*] **unfolding** *bdd-global-depth-def* **by** *simp*

lemma *bdd-global-depth-uminus*: *bdd-global-depth* (− *x*) = *bdd-global-depth x*
using *depth-uminus uminus-iff*[*of - x 0*] **unfolding** *bdd-global-depth-def* **by** *auto*

lemma *bdd-global-depth-nonarch*:
bdd-global-depth (*x* + *y*) $\geq$ *min* (*bdd-global-depth x*) (*bdd-global-depth y*)
if $\neg$ *globally-p-equal* (*x* + *y*) *0*
proof−
consider *globally-p-equal x 0* | *globally-p-equal y 0* |
(*n0*) $\neg$ *globally-p-equal x 0* $\neg$ *globally-p-equal y 0*
by *blast*
thus *?thesis*
proof *cases*
case *n0*
define *N* :: '*b* $\Rightarrow$ '*a set* **where** *N* $\equiv$ λ*x*. {*p*. $\neg$ *p-equal p x 0*}
define *dp1* **where** *dp1* $\equiv$ λ*x p*. *p-depth p x* + *1*
define *trunc-dp1* **where** *trunc-dp1* $\equiv$ λ*x p*. *nat* (*dp1 x p*)
define *inf-trunc-dp1* :: '*b* $\Rightarrow$ *nat*
where *inf-trunc-dp1* $\equiv$ λ*x*. (*INF p*$\in$*N x*. *trunc-dp1 x p*)
from *inf-trunc-dp1-def* **have** *inf-trunc-dp1*:
$\bigwedge$*p x*. *p* $\in$ *N x* $\Longrightarrow$ *inf-trunc-dp1 x* $\leq$ *trunc-dp1 x p*
using *cINF-lower* **by** *fast*
define *choose-dpth* :: '*a* $\Rightarrow$ *int* **where**
choose-dpth $\equiv$ λ*p*.
if p-equal p x 0 then dp1 y p
else if p-equal p y 0 then dp1 x p

else min (*dp1 x p*) (*dp1 y p*)
from *that n0 N-def*
have *nempty*: $N\ x \neq \{\}\ N\ y \neq \{\}\ N\ (x + y) \neq \{\}$
using *globally-p-equalI*
by (*blast*, *blast*, *blast*)
have
Inf ((*nat* ∘ *choose-dpth*) ‘ ($N\ x \cup N\ y$)) =
min (*inf-trunc-dp1 x*) (*inf-trunc-dp1 y*)
proof (*intro cInf-eq-non-empty*)
fix *n* **assume** $n \in$ (*nat* ∘ *choose-dpth*) ‘ ($N\ x \cup N\ y$)
from *this* **obtain** *p* **where** $p \in N\ x \cup N\ y$ n = *nat* (*choose-dpth p*) **by** *auto*
thus *min* (*inf-trunc-dp1 x*) (*inf-trunc-dp1 y*) $\leq n$
using *N-def trunc-dp1-def choose-dpth-def inf-trunc-dp1*[*of p x*] *inf-trunc-dp1*[*of p y*]
by *auto*
next
fix *n*
assume ∗:
$\bigwedge m.\ m \in$ (*nat* ∘ *choose-dpth*) ‘ ($N\ x \cup N\ y$) $\Longrightarrow$
$n \leq m$
have $n \leq$ *inf-trunc-dp1 x*
unfolding *inf-trunc-dp1-def*
proof (*intro cInf-greatest*)
fix *m* **assume** $m \in$ *trunc-dp1 x* ‘ *N x*
from *this* **obtain** *p* **where** $p \in N\ x$ m = *trunc-dp1 x p* **by** *blast*
with *N-def trunc-dp1-def choose-dpth-def* ∗ **show** $n \leq m$
by (*cases* $p \in N\ y$, *fastforce*, *fastforce*)
qed (*simp add*: *nempty*(*1*))
moreover have $n \leq$ *inf-trunc-dp1 y*
unfolding *inf-trunc-dp1-def*
proof (*intro cInf-greatest*)
fix *m* **assume** $m \in$ *trunc-dp1 y* ‘ *N y*
from *this* **obtain** *p* **where** $p \in N\ y$ m = *trunc-dp1 y p* **by** *blast*
with *N-def trunc-dp1-def choose-dpth-def* ∗ **show** $n \leq m$
by (*cases* $p \in N\ x$, *fastforce*, *fastforce*)
qed (*simp add*: *nempty*(*2*))
ultimately
show $n \leq$ *min* (*inf-trunc-dp1 x*) (*inf-trunc-dp1 y*)
by *auto*
qed (*simp add*: *nempty*(*1*))
moreover from *n0*(*1*) **have** *inf-trunc-dp1 x* = *bdd-global-depth x*
using *bdd-global-depthD-as-image*
unfolding *N-def inf-trunc-dp1-def trunc-dp1-def dp1-def*
by *auto*
moreover from *n0*(*2*) **have** *inf-trunc-dp1 y* = *bdd-global-depth y*
using *bdd-global-depthD-as-image*
unfolding *N-def inf-trunc-dp1-def trunc-dp1-def dp1-def*
by *auto*
ultimately have

```
      min (bdd-global-depth x) (bdd-global-depth y) =
        Inf ((nat ∘ choose-dpth) ‘ (N x ∪ N y))
      by presburger
    also have . . . ≤ inf-trunc-dp1 (x + y)
      unfolding inf-trunc-dp1-def
    proof (intro cINF-mono nempty(3))
      fix p assume p: p ∈ N (x + y)
      with N-def have nequiv0:
        ¬ p-equal p (x + y) 0 ¬ p-equal p x 0 ∨ ¬ p-equal p y 0
        using add[of p x 0 y 0] by auto
      from nequiv0(2) consider
        p-equal p x 0 | p-equal p y 0 ¬ p-equal p x 0 |
        (neither) ¬ p-equal p x 0 ¬ p-equal p y 0
        by blast
      hence (nat ∘ choose-dpth) p ≤ trunc-dp1 (x + y) p
      proof cases
        case neither with nequiv0(1) show (nat ∘ choose-dpth) p ≤ trunc-dp1 (x
+ y) p
          using depth-nonarch
          by  (fastforce simp add: choose-dpth-def trunc-dp1-def dp1-def)
      qed (
        simp-all add: choose-dpth-def trunc-dp1-def dp1-def depth-add-equiv0-left
                    depth-add-equiv0-right
      )
      moreover from N-def nequiv0(2) have p ∈ N x ∪ N y by force
      ultimately show
        ∃ q∈N x ∪ N y. (nat ∘ choose-dpth) q ≤ trunc-dp1 (x + y) p
        by blast
    qed simp
    finally show ?thesis
    using that N-def inf-trunc-dp1-def trunc-dp1-def dp1-def bdd-global-depthD-as-image
by simp
  qed simp-all
qed

definition bdd-global-dist :: 'b ⇒ 'b ⇒ real where
  bdd-global-dist x y =
    (if globally-p-equal x y then 0 else inverse (2 ^ bdd-global-depth (x − y)))

lemma bdd-global-dist-eqD [simp]: bdd-global-dist x y = 0 if globally-p-equal x y
  using that unfolding bdd-global-dist-def by simp

lemma bdd-global-distD:
  bdd-global-dist x y = inverse (2 ^ bdd-global-depth (x − y)) if ¬ globally-p-equal
x y
  using that unfolding bdd-global-dist-def by fastforce

lemma bdd-global-dist-cong:
  bdd-global-dist w x = bdd-global-dist y z
```

if *globally-p-equal w y* **and** *globally-p-equal x z*
proof (*cases globally-p-equal w x*)
case *True* **with** *that* **show** *?thesis* **using** *globally-p-equal-cong* **using** *bdd-global-dist-eqD*
by *metis*
next
case *False* **with** *that* **show** *?thesis*
using *globally-p-equal-sym globally-p-equal-cong bdd-global-distD globally-p-equal-minus bdd-global-depth-cong*[*of* $w - x$ $y - z$]
by *metis*
qed

lemma *bdd-global-dist-nonneg*: *bdd-global-dist x y* $\geq$ *0*
unfolding *bdd-global-dist-def* **by** *auto*

lemma *bdd-global-dist-bdd*: *bdd-global-dist x y* $\leq$ *1*
using *le-imp-inverse-le*[*of 1 2* ^ *bdd-global-depth* $(x - y)$]
unfolding *bdd-global-dist-def*
by *auto*

lemma *bdd-global-dist-lessI*:
bdd-global-dist x y $<$ *e*
if *pos*: $e > 0$
and *depth*:
$\bigwedge p$. $\neg$ *p-equal p x y* $\Longrightarrow$
p-depth p $(x - y) \geq \lfloor$*log 2* (*inverse e*)$\rfloor$
proof (*cases* $e > 1$)
case *True*
moreover have *bdd-global-dist x y* $\leq$ *1* **by** (*rule bdd-global-dist-bdd*)
ultimately show *?thesis* **by** *linarith*
next
case *False*
hence *small*: $e \leq 1$ **by** *linarith*
show *?thesis*
proof (*cases globally-p-equal x y*)
case *False*
define *d* **where** $d \equiv \lfloor$*log 2* (*inverse e*)$\rfloor$
have
$\forall p$. $\neg$ *p-equal p x y* $\longrightarrow$
nat (*p-depth p* $(x - y) + 1$) $\geq$ *nat* $(d + 1)$
proof *clarify*
fix *p* **assume** $\neg$ *p-equal p x y*
with *depth d-def* **have** $*$: *p-depth p* $(x - y) \geq d$ **by** *simp*
moreover from *pos d-def* **have** $d \geq 0$
using *small le-imp-inverse-le zero-le-log-cancel-iff*[*of 2 inverse e*] **by** *force*
ultimately have *p-depth p* $(x - y) + 1 > 0$ **by** *linarith*
with $*$ **show** *nat* (*p-depth p* $(x - y) + 1$) $\geq$ *nat* $(d + 1)$
using *le-nat-iff*[*of p-depth p* $(x - y) + 1$ *nat d*] **by** *simp*
qed
with *False* **have** *bdd-global-depth* $(x - y) \geq$ *nat* $(d + 1)$

using *conv-0 bdd-global-depth-greatest* **by** *auto*
with *d-def* **have** *bdd-global-depth* $(x - y) \geq d + 1$ **by** *auto*
with *pos d-def* **have** $e >$ *inverse* $(2$ ^ *bdd-global-depth* $(x - y))$
using *real-of-int-floor-add-one-gt*[*of log 2* (*inverse e*)]
log-pow-cancel[*of 2 bdd-global-depth* $(x - y)$]
log-less-cancel-iff [*of 2 inverse e 2* ^ *bdd-global-depth* $(x - y)$]
less-imp-inverse-less
by *fastforce*
moreover from *False* **have** *bdd-global-dist x y* = *inverse* (2 ^ *bdd-global-depth* $(x - y))$
using *bdd-global-distD* **by** *force*
ultimately show *?thesis* **by** *simp*
qed (*simp add*: *pos*)
qed

lemma *bdd-global-dist-less-imp*:
p-depth p $(x - y) \geq \lfloor log\ 2\ (inverse\ e)\rfloor$
if $e > 0$ **and** $e \leq 1$ **and** ¬ *p-equal p x y* **and** *bdd-global-dist x y* $< e$
proof−
from *that* **have** *log 2* (*inverse e*) < *bdd-global-depth* $(x - y)$
using *bdd-global-distD inverse-less-imp-less*[*of - inverse e*] *log-less*[*of 2 inverse e*]
by *fastforce*
hence $\lfloor log\ 2\ (inverse\ e)\rfloor <$ *bdd-global-depth* $(x - y)$ **by** *linarith*
with *that*(*3*) **have** $\lfloor log\ 2\ (inverse\ e)\rfloor <$ *int* (*nat* (*p-depth p* $(x - y) + 1$))
using *conv-0 bdd-global-depth-le* **by** *fastforce*
moreover from *that*(*1*,*2*) **have** $\lfloor log\ 2\ (inverse\ e)\rfloor \geq 0$
using *le-imp-inverse-le zero-le-log-cancel-iff* [*of 2 inverse e*] **by** *force*
ultimately show *?thesis* **by** *linarith*
qed

lemma *bdd-global-dist-less-pow2-iff*:
bdd-global-dist x y < *inverse* (2 ^ *n*) ⟷
(∀ *p*. ¬ *p-equal p x y* ⟶ *p-depth p* $(x - y) \geq$ *int n*)
proof (*cases globally-p-equal x y*)
case *False* **show** *?thesis*
proof (*standard*, *standard*, *clarify*)
fix *p* **assume** ¬ *p-equal p x y*
moreover assume *bdd-global-dist x y* < *inverse* (2 ^ *n*)
ultimately show *p-depth p* $(x - y) \geq$ *int n*
using *le-imp-inverse-le*[*of 1 2* ^ *n*] *bdd-global-dist-less-imp*[*of inverse* (2 ^ *n*) *p x y*]
by *force*
next
assume ∀ *p*. ¬ *p-equal p x y* ⟶ *p-depth p* $(x - y) \geq$ *int n*
thus *bdd-global-dist x y* < *inverse* (2 ^ *n*) **by** (*auto intro*: *bdd-global-dist-lessI*)
qed
qed *simp*

lemma *bdd-global-dist-sym*: *bdd-global-dist x y* = *bdd-global-dist y x*
using *globally-p-equal-sym bdd-global-depth-diff* **unfolding** *bdd-global-dist-def* **by** *force*

lemma *bdd-global-dist-conv0*: *bdd-global-dist x y* = *bdd-global-dist* (*x* − *y*) *0*
using *globally-p-equal-conv-0* **unfolding** *bdd-global-dist-def* **by** *simp*

lemma *bdd-global-dist-conv0'*: *bdd-global-dist x y* = *bdd-global-dist 0* (*x* − *y*)
using *bdd-global-dist-conv0 bdd-global-dist-sym* **by** *simp*

lemma *bdd-global-dist-nonarch*:
bdd-global-dist x y ≤ *max* (*bdd-global-dist x z*) (*bdd-global-dist y z*)
proof (*cases globally-p-equal x y*)
case *True* **thus** *?thesis*
using *bdd-global-dist-nonneg*[*of x z*] *bdd-global-dist-nonneg*[*of y z*] **by** *simp*
next
case *xy*: *False* **show** *?thesis*
proof (*cases globally-p-equal x z*)
case *True* **thus** *?thesis*
using *bdd-global-dist-cong*[*of x z y y*] *bdd-global-dist-sym* **by** *auto*
next
case *xz*: *False* **show** *?thesis*
proof (*cases globally-p-equal y z*)
case *True* **thus** *?thesis*
using *bdd-global-dist-cong*[*of y z x x*] *bdd-global-dist-sym* **by** *auto*
next
case *False* **with** *xy xz* **show** *?thesis*
proof (*induct x y rule*: *linorder-wlog'*[*of* λ*x*. *bdd-global-dist x z*])
case (*le x y*)
from *le*(*2*) **have** ¬ *globally-p-equal* ((*x* − *z*) + (*z* − *y*)) *0*
using *globally-p-equal-conv-0*[*of x y*] **by** *fastforce*
moreover from *le*(*1*,*3*,*4*) **have** *bdd-global-depth* (*x* − *z*) ≥ *bdd-global-depth* (*y* − *z*)
using *bdd-global-distD inverse-le-imp-le* **by** *auto*
ultimately have *bdd-global-depth* (*x* − *y*) ≥ *bdd-global-depth* (*y* − *z*)
using *bdd-global-depth-nonarch*[*of x* − *z z* − *y*] *bdd-global-depth-diff*[*of y z*] **by** *auto*
with *le*(*1*,*2*,*4*)
show *bdd-global-dist x y* ≤ *max* (*bdd-global-dist x z*) (*bdd-global-dist y z*)
using *le-imp-inverse-le bdd-global-distD*
by *auto*
next
case (*sym x y*)
hence *bdd-global-dist y x* ≤ *max* (*bdd-global-dist y z*) (*bdd-global-dist x z*)
using *globally-p-equal-sym*[*of y x*] **by** *blast*
thus *?case* **using** *bdd-global-dist-sym max.commute* **by** *metis*
qed
qed
qed

qed

lemma *bdd-global-dist-ball-multicentre*:
$\{z.\ bdd\text{-}global\text{-}dist\ y\ z < e\} = \{z.\ bdd\text{-}global\text{-}dist\ x\ z < e\}$
if *bdd-global-dist x y < e*
proof−
define B **where** $B \equiv \lambda x.\ \{z.\ bdd\text{-}global\text{-}dist\ x\ z < e\}$
moreover have $\bigwedge x\ y.\ bdd\text{-}global\text{-}dist\ x\ y < e \Longrightarrow B\ y \subseteq B\ x$
unfolding *B-def*
proof *clarify*
fix *x y z*
assume *bdd-global-dist x y < e* **and** *bdd-global-dist y z < e*
thus *bdd-global-dist x z < e*
using *bdd-global-dist-nonarch*[*of x z y*] *bdd-global-dist-sym* **by** *simp*
qed
ultimately show *?thesis* **using** *that bdd-global-dist-sym* **by** *fastforce*
qed

lemma *bdd-global-dist-ball-at-0*:
$\{z.\ bdd\text{-}global\text{-}dist\ 0\ z < inverse\ (2\ \hat{}\ n)\} = global\text{-}depth\text{-}set\ (int\ n)$
using *bdd-global-dist-less-pow2-iff bdd-global-dist-sym* **unfolding** *global-depth-set-def*
by *auto*

lemma *bdd-global-dist-ball-UNIV*:
$\{z.\ bdd\text{-}global\text{-}dist\ x\ z < e\} = UNIV$ **if** $e > 1$
using *that bdd-global-dist-bdd order-le-less-subst1*[*of - id 1 e*] **by** *fastforce*

lemma *bdd-global-dist-ball-at-0-normalize*:
$\{z.\ bdd\text{-}global\text{-}dist\ 0\ z < e\} = global\text{-}depth\text{-}set\ (\lfloor log\ 2\ (inverse\ e)\rfloor)$
if $e > 0$ **and** $e \leq 1$
using *that bdd-global-dist-less-imp bdd-global-dist-sym bdd-global-dist-lessI*
unfolding *global-depth-set-def*
by *fastforce*

lemma *bdd-global-dist-ball-translate*:
$\{z.\ bdd\text{-}global\text{-}dist\ x\ z < e\} = x +o\ \{z.\ bdd\text{-}global\text{-}dist\ 0\ z < e\}$
unfolding *elt-set-plus-def*
proof *safe*
fix *y* **assume** *bdd-global-dist x y < e*
hence $bdd\text{-}global\text{-}dist\ 0\ (y - x) < e$
using *bdd-global-dist-sym bdd-global-dist-conv0'* **by** *presburger*
moreover have $y = x + (y - x)$ **by** *auto*
ultimately show $\exists\, b\in\{z.\ bdd\text{-}global\text{-}dist\ 0\ z < e\}.\ y = x + b$ **by** *fast*
next
fix *y* **assume** *bdd-global-dist 0 y < e*
thus $bdd\text{-}global\text{-}dist\ x\ (x + y) < e$
using *bdd-global-dist-conv0'*[*of x + y x*] *bdd-global-dist-sym* **by** *auto*
qed

lemma *bdd-global-dist-left-translate-continuous*:
bdd-global-dist $(x + y)$ $(x + z) < e$ **if** *bdd-global-dist* y $z < e$
using *that bdd-global-dist-conv0*[*of* y z] *bdd-global-dist-conv0*[*of* $x + y$ $x + z$] **by** *simp*

lemma *bdd-global-dist-right-translate-continuous*:
bdd-global-dist $(y + x)$ $(z + x) < e$ **if** *bdd-global-dist* y $z < e$
using *that bdd-global-dist-conv0*[*of* y z] *bdd-global-dist-conv0*[*of* $y + x$ $z + x$] **by** *simp*

definition *global-cauchy-condition* ::
$(nat \Rightarrow 'b) \Rightarrow nat \Rightarrow nat \Rightarrow bool$
where
global-cauchy-condition X n K =
$(\forall\, k1 \geq K.\ \forall\, k2 \geq K.\ \forall\, p.$
$\neg$ *p-equal* p $(X\ k1)$ $(X\ k2) \longrightarrow nat$ $(p\text{-}depth\ p\ (X\ k1 - X\ k2)) > n$
)

lemma *global-cauchy-conditionI*:
global-cauchy-condition X n K
if
$\bigwedge p\ k\ k'.\ k \geq K \Longrightarrow k' \geq K \Longrightarrow$
$\neg$ *p-equal* p $(X\ k)$ $(X\ k') \Longrightarrow nat$ $(p\text{-}depth\ p\ (X\ k - X\ k')) > n$
using *that* **unfolding** *global-cauchy-condition-def* **by** *blast*

lemma *global-cauchy-conditionD*:
nat $(p\text{-}depth\ p\ (X\ k - X\ k')) > n$
if *global-cauchy-condition* X n K **and** $k \geq K$ **and** $k' \geq K$
and $\neg$ *p-equal* p $(X\ k)$ $(X\ k')$
using *that* **unfolding** *global-cauchy-condition-def* **by** *blast*

lemma *global-cauchy-condition-mono-seq-tail*:
global-cauchy-condition X n $K \Longrightarrow$ *global-cauchy-condition* X n K'
if $K' \geq K$
using *that* **unfolding** *global-cauchy-condition-def* **by** *auto*

lemma *global-cauchy-condition-mono-depth*:
global-cauchy-condition X n $K \Longrightarrow$ *global-cauchy-condition* X m K
if $m \leq n$
using *that* **unfolding** *global-cauchy-condition-def* **by** *fastforce*

abbreviation *globally-cauchy* $X \equiv (\forall\, n.\ \exists\, K.$ *global-cauchy-condition* X n $K)$

lemma *global-cauchy-condition-LEAST*:
global-cauchy-condition X n (*LEAST* K. *global-cauchy-condition* X n K) **if** *globally-cauchy* X
using *that Least1I*[*OF ex-ex1-least-nat-le*] **by** *blast*

lemma *global-cauchy-condition-LEAST-mono*:

```
 (LEAST K. global-cauchy-condition X m K) ≤ (LEAST K. global-cauchy-condition
X n K)
  if globally-cauchy X and m ≤ n
  using that least-le-least[OF ex-ex1-least-nat-le ex-ex1-least-nat-le]
       global-cauchy-condition-mono-depth
  by    force

definition bdd-global-uniformity :: ('b × 'b) filter
  where bdd-global-uniformity = (INF e∈{0 <..}. principal {(x, y). bdd-global-dist
x y < e})

end

context global-p-equal-depth
begin

sublocale metric-space-by-bdd-depth:
  metric-space bdd-global-dist bdd-global-uniformity
    λU. ∀ x∈U.
      eventually (λ(x', y). x' = x ⟶ y ∈ U) bdd-global-uniformity
proof
  fix x y z
  define dxy dxz dyz
    where dxy ≡ bdd-global-dist x y
    and   dxz ≡ bdd-global-dist x z
    and   dyz ≡ bdd-global-dist y z
  hence dxy ≤ max dxz dyz using bdd-global-dist-nonarch by metis
  also from dxz-def dyz-def have ... ≤ max dxz dyz + min dxz dyz
    using bdd-global-dist-def by auto
  finally show dxy ≤ dxz + dyz by auto
 from dxy-def show (dxy = 0) = (x = y) using global-eq-iff unfolding bdd-global-dist-def
by simp
qed (simp only: bdd-global-uniformity-def, simp)

lemma nonneg-global-depth-set-LIMSEQ-closed:
  x ∈ global-depth-set (int n)
  if  lim: metric-space-by-bdd-depth.LIMSEQ X x
  and seq: ∀F k in sequentially. X k ∈ global-depth-set (int n)
proof (intro global-depth-setI)
  fix p assume p: ¬ p-equal p x 0
  from seq obtain K where K: ∀ k≥K. X k ∈ global-depth-set (int n)
    using eventually-sequentially by fastforce
  from lim have ∀F k in sequentially. bdd-global-dist (X k) x < inverse (2 ^ n)
    using metric-space-by-bdd-depth.tendstoD[of X x sequentially inverse (2 ^ n)]
by fastforce
  from this obtain K' :: nat
    where K': ∀ k≥K'. bdd-global-dist (X k) x < inverse (2 ^ n)
    using eventually-sequentially
    by    meson
```

define *m* **where** $m \equiv max\ K\ K'$
consider
(eq) *p-equal p* $(X\ m)$ x |
(0) $\neg$ *p-equal p* $(X\ m)$ x *p-equal p* $(X\ m)$ 0 |
($n0$) $\neg$ *p-equal p* $(X\ m)$ x $\neg$ *p-equal p* $(X\ m)$ 0
by *blast*
thus *p-depth p x* $\geq$ *int n*
proof *cases*
case *eq*
moreover from *this K m-def* **have** *X m* $\in$ *global-depth-set* (*int n*) **by** *auto*
ultimately show *?thesis* **using** *p trans-right global-depth-setD depth-equiv* **by** *metis*
next
case *0* **with** *K' m-def* **show** *?thesis*
using *bdd-global-dist-less-pow2-iff* [*of X m x n*] *depth-diff-equiv0-left* **by** *auto*
next
case *n0*
with *p K K' m-def* **show** *?thesis*
using *bdd-global-dist-less-pow2-iff* [*of X m x n*] *global-depth-setD*[*of p X m int n*]
depth-pre-nonarch-diff-right[*of p x X m*]
by *force*
qed
qed

lemma *globally-cauchy-vs-bdd-metric-Cauchy*:
globally-cauchy X = *metric-space-by-bdd-depth.Cauchy X* **for** *X* :: *nat* $\Rightarrow$ *'b*
proof
assume *X*: *globally-cauchy X* **show** *metric-space-by-bdd-depth.Cauchy X*
proof (*intro metric-space-by-bdd-depth.metric-CauchyI*)
fix *e* :: *real* **assume** *pos*: $e > 0$
show $\exists M.\ \forall m \geq M.\ \forall n \geq M.$ *bdd-global-dist* $(X\ m)$ $(X\ n) < e$
proof (*cases* $e > 1$)
case *True*
hence $\forall m \geq 0.\ \forall n \geq 0.$ *bdd-global-dist* $(X\ m)$ $(X\ n) < e$
using *bdd-global-dist-bdd order-le-less-subst1* [*of - id 1 e*] **by** *fastforce*
thus *?thesis* **by** *blast*
next
case *False*
hence *small*: $e \leq 1$ **by** *simp*
define *d* **where** $d \equiv \lfloor log\ 2\ (inverse\ e) \rfloor$
from *X* **obtain** *M* **where** *M*: *global-cauchy-condition X* (*nat d*) *M* **by** *fast*
have $\forall m \geq M.\ \forall n \geq M.$ *bdd-global-dist* $(X\ m)$ $(X\ n) < e$
proof (*clarify*, *intro bdd-global-dist-lessI pos*)
fix *m n* :: *nat* **and** *p* :: *'a*
assume $m \geq M$ **and** $n \geq M$ **and** $\neg$ *p-equal p* $(X\ m)$ $(X\ n)$
with *M* **have** *nat* (*p-depth p* $(X\ m - X\ n)$) $>$ *nat d* **using** *global-cauchy-conditionD* **by** *blast*
moreover from *d-def pos small* **have** $d \geq 0$

```
        using le-imp-inverse-le zero-le-log-cancel-iff[of 2 inverse e] by force
      ultimately show p-depth p (X m − X n) ≥ ⌊log 2 (inverse e)⌋
        using d-def by simp
    qed
    thus ?thesis by blast
  qed
 qed
next
 assume X: metric-space-by-bdd-depth.Cauchy X show globally-cauchy X
 proof
  fix n
  from X obtain M
    where M:
      ∀ m≥M. ∀ m'≥M. bdd-global-dist (X m) (X m') < inverse (2 ^ Suc n)
    using metric-space-by-bdd-depth.metric-CauchyD[of X inverse (2 ^ Suc n)]
    by  auto
  hence global-cauchy-condition X n M
    using bdd-global-dist-less-pow2-iff[of - - Suc n]
    by  (fastforce intro: global-cauchy-conditionI)
  thus ∃ M. global-cauchy-condition X n M by fast
 qed
qed

end

context global-p-equal-depth-no-zero-divisors
begin

lemma bdd-global-dist-bdd-mult-continuous:
 bdd-global-dist (w ∗ x) (w ∗ y) < e
 if pos : e > 0
 and bdd : ∀ p. ¬ p-equal p w 0 ⟶ p-depth p w ≥ n
 and input: bdd-global-dist x y < min (e ∗ 2 powi (n − 1)) 1
proof (cases e > 1, use bdd-global-dist-ball-UNIV in blast, intro bdd-global-dist-lessI
pos)
 case False
 fix p assume p: ¬ p-equal p (w ∗ x) (w ∗ y)
 from p have w: ¬ p-equal p w 0
  using mult-0-left cong[of p 0 w ∗ x 0 w ∗ y] sym by force
 from p have xy: ¬ p-equal p x y using mult-left by force
 from p have wxy: ¬ p-equal p (w ∗ (x − y)) 0
  using conv-0 right-diff-distrib[of w x y] by auto
 define d' where d' ≡ e ∗ 2 powi (n − 1)
 define d  where d  ≡ min d' 1
 define m  where m  ≡ ⌊log 2 (inverse d)⌋
 from pos d-def d'-def have d > 0 d ≤ 1 by auto
 with input m-def d-def d'-def have m ≤ p-depth p (x − y)
  using xy bdd-global-dist-less-imp[of d p x y] by fast
 with bdd have ∗: n + m ≤ p-depth p (w ∗ x − w ∗ y)
```

using *w wxy right-diff-distrib*[*of w x y*] *depth-mult-additive*[*of p w x* − *y*]
bdd-global-depth-le[*of p w*]
by *force*
moreover have *n* + *m* ≥ *log 2* (*inverse e*)
proof (*cases e* ∗ *2 powi* (*n* − *1*) < *1*)
case *True*
moreover from *m-def* **have** *real-of-int* (*n* + *m*) ≥ *real-of-int n* + (− *log 2 d* − *1*)
using *real-of-int-floor-ge-diff-one log-inverse* **by** *fastforce*
ultimately show *?thesis* **using** *d-def d′-def pos* **by** (*simp add*: *log-mult log-inverse algebra-simps*)
next
case *False*
moreover from *this pos* **have** *log 2* (*e* ∗ *2 powi* (*n* − *1*)) ≥ *0*
by (*simp add*: *le-imp-inverse-le log-mono log-inverse*)
ultimately show *?thesis* **using** *pos m-def d-def d′-def*
by (*fastforce simp add*: *log-mult log-inverse*)
qed
ultimately show *p-depth p* (*w* ∗ *x* − *w* ∗ *y*) ≥ ⌊*log 2* (*inverse e*)⌋ **by** *linarith*
qed

lemma *bdd-global-dist-limseq-bdd-mult*:
metric-space-by-bdd-depth.LIMSEQ (λ*k*. *w* ∗ *X k*) (*w* ∗ *x*)
if *bdd*: ∀ *p*. ¬ *p-equal p w 0* ⟶ *p-depth p w* ≥ *n*
and *seq*: *metric-space-by-bdd-depth.LIMSEQ X x*
proof
fix *e* :: *real* **assume** *e*: *e* > *0*
define *d* **where** *d* ≡ *min* (*e* ∗ *2 powi* (*n* − *1*)) *1*
with *e* **have** *d*: *d* > *0* **by** *auto*
with *seq* **obtain** *K* :: *nat* **where** ∀ *k*≥*K*. *bdd-global-dist* (*X k*) *x* < *d*
using *metric-space-by-bdd-depth.tendstoD*[*of X x*] *eventually-sequentially* **by** *meson*
with *bdd e d-def* **have** ∀ *k*≥*K*. *bdd-global-dist* (*w* ∗ *X k*) (*w* ∗ *x*) < *e*
using *bdd-global-dist-bdd-mult-continuous* **by** *blast*
thus $\forall_F$ *k in sequentially*. *bdd-global-dist* (*w* ∗ *X k*) (*w* ∗ *x*) < *e*
using *eventually-sequentially* **by** *blast*
qed

end

context *global-p-equal-depth-div-inv-w-inj-index-consts*
begin

lemma *bdd-global-dist-limseq-global-uniformizer-powi-mult*:
metric-space-by-bdd-depth.LIMSEQ
(λ*k*. *global-uniformizer powi n* ∗ *X k*) (*global-uniformizer powi n* ∗ *x*)
if *metric-space-by-bdd-depth.LIMSEQ X x*
using *that bdd-global-dist-limseq-bdd-mult*[*of - n*] *p-depth-global-uniformizer-powi* **by** *auto*

lemma *global-depth-set-LIMSEQ-closed*:
  $x \in$ *global-depth-set n*
  if *lim* : *metric-space-by-bdd-depth.LIMSEQ X x*
  and *depth*: $\forall_F$ *k in sequentially. X k* $\in$ *global-depth-set n*
proof−
  define *Y* **and** *y*
    where $Y \equiv \lambda k.$ *global-uniformizer powi* $(-n) * X\ k$
    and $y \equiv$ *global-uniformizer powi* $(-n) * x$
  from *depth Y-def* **have** $\forall_F$ *k in sequentially. Y k* $\in$ *global-depth-set 0*
    using *global-depth-set-eq-global-uniformizer-coset′*[*of n*] *eventually-sequentially*
by *auto*
  with *Y-def y-def lim* **have** $y \in$ *global-depth-set 0*
    using *bdd-global-dist-limseq-global-uniformizer-powi-mult*
        nonneg-global-depth-set-LIMSEQ-closed[*of Y y 0*]
    by *simp*
  moreover from *y-def* **have** *global-uniformizer powi n* $* y = x$
    using *global-uniformizer-powi-add*[*of n* $-n$] **by** (*simp add*: *ac-simps*)
  ultimately show *?thesis*
    using *global-depth-set-eq-global-uniformizer-coset*[*of n*] **by** *blast*
qed

end

3.8 Notation

consts
  p-equal :: $'a \Rightarrow 'b \Rightarrow 'b \Rightarrow bool$
  globally-p-equal ::
    $'b \Rightarrow 'b \Rightarrow bool$
    ((-/ $\simeq_{\forall\, p}$/ -) [*51*, *51*] *50*)
  p-restrict :: $'b \Rightarrow ('a \Rightarrow bool) \Rightarrow 'b$ (**infixl** *prestrict 70*)
  p-depth :: $'a \Rightarrow 'b \Rightarrow int$
  p-depth-set :: $'a \Rightarrow int \Rightarrow 'b\ set$ ($\mathcal{P}_{@\text{-}}$⁻)
  global-depth-set :: $int \Rightarrow 'b\ set$ ($\mathcal{P}_{\forall\, p}$⁻)
  global-unfrmzr-pows :: $('a \Rightarrow 'c) \Rightarrow 'b$

abbreviation *p-equal-notation* :: $'b \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$
  ((-/ $\simeq_{\text{-}}$/ -) [*51*, *51*, *51*] *50*)
  where *p-equal-notation x p y* $\equiv$ *p-equal p x y*

abbreviation *p-nequal-notation* :: $'b \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$
  ((-/ $\neg\simeq_{\text{-}}$/ -) [*51*, *51*, *51*] *50*)
  where *p-nequal-notation x p y* $\equiv \neg$ *p-equal p x y*

abbreviation *p-depth-notation* :: $'b \Rightarrow 'a \Rightarrow int$
  ((-°⁻) [*51*, *51*] *50*)
  where *p-depth-notation x p* $\equiv$ *p-depth p x*

abbreviation *p-depth-set-0-notation* :: $'a \Rightarrow 'b$ *set* ($\mathcal{O}_{@-}$)
  where *p-depth-set-0-notation p* $\equiv$ *p-depth-set p 0*

abbreviation *p-depth-set-1-notation* :: $'a \Rightarrow 'b$ *set* ($\mathcal{P}_{@-}$)
  where *p-depth-set-1-notation p* $\equiv$ *p-depth-set p 1*

abbreviation *global-depth-set-0-notation* :: $'b$ *set* ($\mathcal{O}_{\forall\, p}$)
  where *global-depth-set-0-notation* $\equiv$ *global-depth-set 0*

abbreviation *global-depth-set-1-notation* :: $'b$ *set* ($\mathcal{P}_{\forall\, p}$)
  where *global-depth-set-1-notation* $\equiv$ *global-depth-set 1*

abbreviation $\mathfrak{p} \equiv$ *global-unfrmzr-pows*

end

theory *FLS-Prime-Equiv-Depth*

imports
  Parametric-Equiv-Depth
  HOL.Equiv-Relations
  HOL−Computational-Algebra.Formal-Laurent-Series

begin

unbundle *fps-syntax*

4 Equivalence of sequences of formal Laurent series relative to divisibility of partial sums by a fixed prime

4.1 Preliminaries

4.1.1 Lift/transfer of equivalence relations.

lemma *relfp-transfer*:
  reflp r $\Longrightarrow$ *reflp* ($\lambda x\ y.\ r\ (f\ x)\ (f\ y)$)
  unfolding *reflp-def* **by** *fast*

lemma *symp-transfer*:
  symp r $\Longrightarrow$ *symp* ($\lambda x\ y.\ r\ (f\ x)\ (f\ y)$)
  unfolding *symp-def* **by** *fast*

lemma *transp-transfer*:
  transp r $\Longrightarrow$ *transp* ($\lambda x\ y.\ r\ (f\ x)\ (f\ y)$)
  unfolding *transp-def* **by** *fast*

lemma *equivp-transfer*:
equivp $r \Longrightarrow$ *equivp* $(\lambda x\ y.\ r\ (f\ x)\ (f\ y))$
using *relfp-transfer symp-transfer transp-transfer*
equivp-reflp-symp-transp[*of* r] *equivp-reflp-symp-transp*[*of* $\lambda x\ y.\ r\ (f\ x)\ (f\ y)$]
by *fast*

4.1.2 An additional fact about products/sums

lemma (**in** *comm-monoid-set*) *atLeastAtMost-int-shift-bounds*:
$F\ g\ \{m..n\} = F\ (g \circ plus\ m \circ int)\ \{..nat\ (n - m)\}$
if $m \leq n$
for $m\ n$ :: *int*
proof (*rule reindex-bij-witness*)
define i :: *nat* $\Rightarrow$ *int* **and** j :: *int* $\Rightarrow$ *nat*
where $i \equiv plus\ m \circ int$
and $j \equiv (\lambda x.\ nat\ (x - m))$
fix $a\ b$ **from** *i-def j-def that*
show $a \in \{m..n\} \Longrightarrow i\ (j\ a) = a$
and $a \in \{m..n\} \Longrightarrow j\ a \in \{..nat\ (n - m)\}$
and $b \in \{..nat\ (n - m)\} \Longrightarrow j\ (i\ b) = b$
and $b \in \{..nat\ (n - m)\} \Longrightarrow i\ b \in \{m..n\}$
and $a \in \{m..n\} \Longrightarrow (g \circ (+)\ m \circ int)\ (j\ a) = g\ a$
by *auto*
qed

4.1.3 An additional fact about algebraic powers of functions

lemma *pow-fun-apply*: $(f\ \hat{}\ n)\ x = (f\ x)\ \hat{}\ n$
by (*induct n*, *simp-all*)

4.1.4 Some additional facts about formal power and Laurent series

lemma *fps-shift-fps-mult-by-fps-const*:
fixes c :: $'a$::{*comm-monoid-add*,*mult-zero*}
shows *fps-shift n* (*fps-const c* $*$ f) = *fps-const c* $*$ *fps-shift n f*
and *fps-shift n* (f $*$ *fps-const c*) = *fps-shift n f* $*$ *fps-const c*
by (*auto intro*: *fps-ext*)

lemma *fps-shift-mult-Suc*:
fixes $f\ g$ :: $'a$::{*comm-monoid-add*,*mult-zero*} *fps*
shows *fps-shift* (*Suc n*) ($f*g$)
= *fps-const* ($f\$0$) $*$ *fps-shift* (*Suc n*) g + *fps-shift n* (*fps-shift 1 f* $*$ g)
and *fps-shift* (*Suc n*) ($f*g$)
= *fps-shift n* (f $*$ *fps-shift 1 g*) + *fps-shift* (*Suc n*) f $*$ *fps-const* ($g\$0$)
proof−
show *fps-shift* (*Suc n*) ($f*g$)
= *fps-const* ($f\$0$) $*$ *fps-shift* (*Suc n*) g + *fps-shift n* (*fps-shift 1 f* $*$ g)
proof (*intro fps-ext*)
fix k

```
    have {0<..Suc (k + n)} = {Suc 0..Suc (k + n)} by auto
    thus
      fps-shift (Suc n) (f∗g) $ k
        = (fps-const (f$0) ∗ fps-shift (Suc n) g + fps-shift n (fps-shift 1 f ∗ g)) $ k
      using fps-mult-nth[of - g]
            sum.head[of 0 k + Suc n λi. f$i ∗ g$(k + Suc n − i)]
            sum.atLeast-Suc-atMost-Suc-shift[of λi. f$i ∗ g$(k + Suc n − i) 0 k+n]
      by   simp
  qed
  show fps-shift (Suc n) (f∗g)
        = fps-shift n (f ∗ fps-shift 1 g) + fps-shift (Suc n) f ∗ fps-const (g$0)
  proof (intro fps-ext)
    fix k
    have ∀ i∈{0..k + n}. Suc (k + n) − i = Suc (k + n −i) by auto
    thus fps-shift (Suc n) (f∗g) $ k
        = (fps-shift n (f ∗ fps-shift 1 g) + fps-shift (Suc n) f ∗ fps-const (g$0)) $
k
      using fps-mult-nth[of f] by simp
  qed
qed

lemma fls-subdegree-minus':
  fls-subdegree (f − g) = fls-subdegree f
  if f ≠ 0 and fls-subdegree g > fls-subdegree f
  using that by (auto intro: fls-subdegree-eqI)

lemma fls-regpart-times-fps-X:
  fixes   f :: 'a::{comm-monoid-add,mult-zero,monoid-mult} fls
  assumes fls-subdegree f ≥ 0
  shows   fls-regpart f ∗ fps-X = fls-regpart (fls-shift (−1) f)
  and     fps-X ∗ fls-regpart f = fls-regpart (fls-shift (−1) f)
proof−
  show fls-regpart f ∗ fps-X  = fls-regpart (fls-shift (−1) f)
  proof (intro fps-ext)
    fix n show (fls-regpart f ∗ fps-X) $ n = fls-regpart (fls-shift (−1) f) $ n
      using assms by (cases n) auto
  qed
  thus fps-X ∗ fls-regpart f = fls-regpart (fls-shift (−1) f)
    by (simp add: fps-mult-fps-X-commute)
qed

lemma fls-regpart-times-fps-X-power:
  fixes   f :: 'a::semiring-1 fls
  assumes fls-subdegree f ≥ 0
  shows   fls-regpart f ∗ fps-X ^ n = fls-regpart (fls-shift (−n) f)
  and     fps-X ^ n ∗ fls-regpart f = fls-regpart (fls-shift (−n) f)
proof−
  show fps-X ^ n ∗ fls-regpart f = fls-regpart (fls-shift (−n) f)
  proof (induct n)
```

case (*Suc n*)
from *assms Suc*
have *A*: *fps-X* ^ *Suc n* ∗ *fls-regpart f* = *fls-regpart* (*fls-shift* (−(*int n* + *1*)) *f*)
using *fls-shift-nonneg-subdegree*[*of* −*int n f*]
by (*simp add*: *mult.assoc fls-regpart-times-fps-X*(*2*))
have *B*: − (*int n* + *1*) = − *int* (*Suc n*) **by** *simp*
have *fls-regpart* (*fls-shift* (−(*int n* + *1*)) *f*) = *fls-regpart* (*fls-shift* (− *int* (*Suc n*)) *f*)
using *arg-cong*[*OF B*] **by** *simp*
with *A* **show** *?case* **by** *simp*
qed *simp*
thus *fls-regpart f* ∗ *fps-X* ^ *n* = *fls-regpart* (*fls-shift* (−*n*) *f*)
by (*simp flip*: *fps-mult-fps-X-power-commute*)
qed

lemma *fls-base-factor-uminus*: *fls-base-factor* (−*f*) = − *fls-base-factor f*
unfolding *fls-base-factor-def* **by** *simp*

4.2 Partial evaluation of formal power series

Make a degree-n polynomial out of a formal power series.

definition *poly-of-fps*
where *poly-of-fps f n* = *Abs-poly* (*λi. if i* ≤ *n then f*$*i else 0*)

lemma *if-then-MOST-zero*:
∀∞*x*. (*if P x then f x else 0*) = *0*
if ∀∞*x*. ¬ *P x*
proof (*rule iffD2*, *rule MOST-iff-cofinite*)
have {*x*. (*if P x then f x else 0*) ≠ *0*} ⊆ {*x*. ¬ ¬ *P x*} **by** *auto*
with *that* **show** *finite* {*x*. (*if P x then f x else 0*) ≠ *0*}
using *iffD1*[*OF MOST-iff-cofinite*] *finite-subset* **by** *blast*
qed

lemma *truncated-fps-eventually-zero*:
(*λi. if i* ≤ *n then f*$*i else 0*) ∈ {*g*. ∀∞*n*. *g n* = *0*}
by (*auto intro*: *if-then-MOST-zero iffD2*[*OF eventually-cofinite*])

lemma *coeff-poly-of-fps*[*simp*]:
coeff (*poly-of-fps f n*) *i* = (*if i* ≤ *n then f*$*i else 0*)
unfolding *poly-of-fps-def*
using *truncated-fps-eventually-zero*[*of n f*]
Abs-poly-inverse[*of λi. if i* ≤ *n then f*$*i else 0*]
by *simp*

lemma *poly-of-fps-0*[*simp*]: *poly-of-fps 0 n* = *0*
by (*auto intro*: *poly-eqI*)

lemma *poly-of-fps-0th-conv-pCons*[*simp*]: *poly-of-fps f 0* = [:*f*$*0*:]

proof (*intro poly-eqI*)
  fix *i*
  show *coeff* (*poly-of-fps f 0*) *i* = *coeff* [:*f*$*0*:] *i*
    by (*cases i*) *auto*
qed

lemma *poly-of-fps-degree*: *degree* (*poly-of-fps f n*) ≤ *n*
  by (*auto intro*: *degree-le*)

lemma *poly-of-fps-Suc*:
  poly-of-fps f (*Suc n*) = *poly-of-fps f n* + *monom* (*f* $ *Suc n*) (*Suc n*)
proof (*intro poly-eqI*)
  fix *m*
  show
    coeff (*poly-of-fps f* (*Suc n*)) *m* =
      coeff (*poly-of-fps f n* + *monom* (*f* $ *Suc n*) (*Suc n*)) *m*
    by (*cases m* ≤ *Suc n*) *auto*
qed

lemma *poly-of-fps-Suc-nth-eq0*:
  poly-of-fps f (*Suc n*) = *poly-of-fps f n* **if** *f* $ *Suc n* = *0*
proof (*intro poly-eqI*)
  fix *i* **from** *that* **show** *coeff* (*poly-of-fps f* (*Suc n*)) *i* = *coeff* (*poly-of-fps f n*) *i*
    by (*cases i Suc n rule*: *linorder-cases*) *auto*
qed

lemma *poly-of-fps-Suc-conv-pCons-shift*:
  poly-of-fps f (*Suc n*) = *pCons* (*f*$*0*) (*poly-of-fps* (*fps-shift 1 f*) *n*)
proof (*intro poly-eqI*)
  fix *i*
  show *coeff* (*poly-of-fps f* (*Suc n*)) *i*
        = *coeff* (*pCons* (*f*$*0*) (*poly-of-fps* (*fps-shift 1 f*) *n*)) *i*
    by (*cases i*) *auto*
qed

abbreviation *fps-parteval f x n* ≡ *poly* (*poly-of-fps f n*) *x*
abbreviation *fls-base-parteval f* ≡ *fps-parteval* (*fls-base-factor-to-fps f*)

lemma *fps-parteval-0*[*simp*]: *fps-parteval 0 x n* = *0*
  by (*induct n*) *auto*

lemma *fps-parteval-0th*[*simp*]: *fps-parteval f x 0* = *f* $ *0*
  by *simp*

lemma *fps-parteval-Suc*:
  fps-parteval f x (*Suc n*) = *fps-parteval f x n* + (*f* $ *Suc n*) ∗ *x* ^ *Suc n*
  for *f* :: *'a*::*comm-semiring-1 fps*
  by (*simp add*: *poly-of-fps-Suc poly-monom*)

lemma *fps-parteval-Suc-cons*:
fps-parteval f x (*Suc n*) = *f*\$*0* + *x* ∗ *fps-parteval* (*fps-shift 1 f*) *x n*
by (*simp add*: *poly-of-fps-Suc-conv-pCons-shift*)

lemma *fps-parteval-at-0*[*simp*]: *fps-parteval f 0 n* = *f* \$ *0*
by (*cases n*) (*auto simp add*: *fps-parteval-Suc-cons*)

lemma *fps-parteval-conv-sum*:
fps-parteval f x n = ($\sum$ *k*≤*n*. *f*\$*k* ∗ *x*^*k*) **for** *x* :: ′*a*::*comm-semiring-1*
by (*induct n*) (*auto simp add*: *fps-parteval-Suc*)

lemma *fls-base-parteval-conv-sum*:
fls-base-parteval f x n = ($\sum$ *k*≤*n*. *f*\$\$(*fls-subdegree f* + *int k*) ∗ *x*^*k*)
for *f* :: ′*a*::*comm-semiring-1 fls* **and** *x* :: ′*a* **and** *n* :: *nat*
using *fps-parteval-conv-sum*[*of fls-base-factor-to-fps f*] *fls-base-factor-to-fps-nth*[*of f*]
by *presburger*

lemma *fps-parteval-fps-const*[*simp*]:
fps-parteval (*fps-const c*) *x n* = *c*
by (*induct n*) (*auto simp add*: *fps-parteval-Suc-cons fps-shift-fps-const*)

lemma *fps-parteval-1*[*simp*]: *fps-parteval 1 x n* = *1*
using *fps-parteval-fps-const*[*of 1*] **by** *simp*

lemma *fps-parteval-mult-by-fps-const*:
fps-parteval (*fps-const c* ∗ *f*) *x n* = *c* ∗ *fps-parteval f x n*
using *mult.commute*[*of c x*] *mult.assoc*[*of x c*] *mult.assoc*[*of c x*]
by (*induct n arbitrary*: *f*)
(*auto simp add*: *fps-parteval-Suc-cons fps-shift-fps-mult-by-fps-const*(*1*) *distrib-left*)

lemma *fps-parteval-plus*[*simp*]:
fps-parteval (*f* + *g*) *x n* = *fps-parteval f x n* + *fps-parteval g x n*
by (*induct n arbitrary*: *f g*)
(*auto simp add*: *fps-parteval-Suc-cons fps-shift-add algebra-simps*)

lemma *fps-parteval-uminus*[*simp*]:
fps-parteval (−*f*) *x n* = − *fps-parteval f x n* **for** *f* :: ′*a*::*comm-ring fps*
by (*induct n arbitrary*: *f*) (*auto simp add*: *fps-parteval-Suc-cons fps-shift-uminus*)

lemma *fps-parteval-minus*[*simp*]:
fps-parteval (*f* − *g*) *x n* = *fps-parteval f x n* − *fps-parteval g x n*
for *f* :: ′*a*::*comm-ring fps*
using *fps-parteval-plus*[*of f* −*g*] **by** *simp*

lemma *fps-parteval-mult-fps-X*:
fixes *f* :: ′*a*::*comm-semiring-1 fps*
shows *fps-parteval* (*f* ∗ *fps-X*) *x* (*Suc n*) = *x* ∗ *fps-parteval f x n*

and *fps-parteval* (*fps-X* ∗ *f*) *x* (*Suc n*) = *x* ∗ *fps-parteval f x n*
using *fps-shift-times-fps-X′*[*of f*]
by (*auto simp add*: *fps-parteval-Suc-cons simp flip*: *fps-mult-fps-X-commute*)

lemma *fps-parteval-mult-fps-X-power*:
fixes *f* :: *′a::comm-semiring-1 fps*
assumes *n* ≥ *m*
shows *fps-parteval* (*f* ∗ *fps-X* ^ *m*) *x n* = *x*^*m* ∗ *fps-parteval f x* (*n* − *m*)
and *fps-parteval* (*fps-X* ^ *m* ∗ *f*) *x n* = *x*^*m* ∗ *fps-parteval f x* (*n* − *m*)
proof−
have
n ≥ *m* ⟹ *fps-parteval* (*f* ∗ *fps-X* ^ *m*) *x n* = *x*^*m* ∗ *fps-parteval f x* (*n* − *m*)
proof (*induct m arbitrary*: *n*)
case (*Suc m*) **thus** *?case*
using *fps-shift-times-fps-X′*[*of f* ∗ *fps-X*^*m*]
by (*cases n*)
(*auto simp add*: *fps-parteval-Suc-cons mult.assoc simp flip*: *power-Suc2*)
qed *simp*
with *assms* **show** *fps-parteval* (*f* ∗ *fps-X* ^ *m*) *x n* = *x*^*m* ∗ *fps-parteval f x* (*n* − *m*)
by *fast*
thus *fps-parteval* (*fps-X* ^ *m* ∗ *f*) *x n* = *x*^*m* ∗ *fps-parteval f x* (*n* − *m*)
by (*simp add*: *fps-mult-fps-X-power-commute*)
qed

lemma *fps-parteval-mult-right*:
fps-parteval (*f* ∗ *g*) *x n* =
fps-parteval (*Abs-fps* (λ*i*. *f*$*i* ∗ *fps-parteval g x* (*n* − *i*))) *x n*
proof (*induct n arbitrary*: *f*)
case (*Suc n*)
have *1*:
poly-of-fps (*Abs-fps* (λ*i*. *f* $ *Suc i* ∗ *fps-parteval g x* (*n* − *i*))) *n*
= *poly-of-fps* (*Abs-fps* (λ*i*. *f* $ *Suc i* ∗ *fps-parteval g x* (*Suc n* − *Suc i*))) *n*
by (*intro poly-eqI*) *auto*
from *Suc* **have**
fps-parteval (*fps-shift 1 f* ∗ *g*) *x n*
= *fps-parteval* (*Abs-fps* (λ*i*. *f* $ *Suc i* ∗ *fps-parteval g x* (*n* − *i*))) *x n*
by *simp*
with *1* **have**
fps-parteval (*f* ∗ *g*) *x* (*Suc n*)
= *f*$*0* ∗ *fps-parteval g x* (*Suc n*)
+ *x* ∗ *fps-parteval* (*Abs-fps* (λ*i*. *f* $ *Suc i* ∗ *fps-parteval g x* (*n* − *i*))) *x n*

using *fps-shift-mult-Suc*(*1*)[*of 0 f g*]
by (*simp add*: *fps-parteval-Suc-cons fps-parteval-mult-by-fps-const algebra-simps*)
thus *?case* **by** (*simp add*: *fps-parteval-Suc-cons fps-shift-def*)
qed *simp*

lemma *fps-parteval-mult-left*:

fps-parteval $(f * g)$ x n =
  fps-parteval (*Abs-fps* (λi. *fps-parteval* f x $(n - i) * g\$i$)) x n
  using *fps-parteval-mult-right*[*of* g f]
  by (*simp add*: *mult.commute*)

lemma *fps-parteval-mult-conv-sum*:
  fixes f g :: *'a::comm-semiring-1 fps*
  shows *fps-parteval* $(f * g)$ x n = ($\sum k \le n$. $f\$k$ * *fps-parteval* g x $(n - k) * x\hat{}k$)
  and *fps-parteval* $(f * g)$ x n = ($\sum k \le n$. *fps-parteval* f x $(n - k) * g\$k * x\hat{}k$)
  using *fps-parteval-mult-right*[*of* f g n x]
    fps-parteval-conv-sum[*of Abs-fps* (λi. $f\$i$ * *fps-parteval* g x $(n - i)$)]
    fps-parteval-mult-left[*of* f g n x]
    fps-parteval-conv-sum[*of Abs-fps* (λi. *fps-parteval* f x $(n - i) * g\$i$)]
  by *auto*

lemma *fps-parteval-fls-base-factor-to-fps-diff*:
  fls-base-parteval $(f - g)$ x n = *fls-base-parteval* f x n
  if $f \neq 0$ **and** *fls-subdegree* g > *fls-subdegree* f + (*int* n)
  for f g :: *'a::comm-ring-1 fls*
  using *that*
  by (*induct* n) (*auto simp add*: *fps-parteval-Suc fls-subdegree-minus'*)

4.3 Vanishing of formal power series relative to divisibility of all partial sums

definition *fps-pvanishes* :: *'a::comm-semiring-1* ⇒ *'a fps* ⇒ *bool*
  where *fps-pvanishes* p f ≡ ($\forall n$::*nat*. p ^ *Suc* n *dvd* (*fps-parteval* f p n))

lemma *fps-pvanishesD*: *fps-pvanishes* p f ⟹ p ^ *Suc* n *dvd* (*fps-parteval* f p n)
  unfolding *fps-pvanishes-def* **by** *simp*

lemma *fps-pvanishesI*:
  fps-pvanishes p f **if** $\bigwedge n$::*nat*. p ^ *Suc* n *dvd* (*fps-parteval* f p n)
  using *that* **unfolding** *fps-pvanishes-def* **by** *simp*

lemma *fps-pvanishes-0*[*simp*]: *fps-pvanishes* p *0*
  by (*auto intro*: *fps-pvanishesI*)

lemma *fps-pvanishes-at-0*[*simp*]: *fps-pvanishes* *0* f = ($f\$0 = 0$)
  unfolding *fps-pvanishes-def* **by** *auto*

lemma *fps-pvanishes-at-unit*: *fps-pvanishes* p f **if** *is-unit* p
  by (*metis that is-unit-power-iff unit-imp-dvd fps-pvanishesI*)

lemma *fps-pvanishes-one-iff*:
  fps-pvanishes p *1* = *is-unit* p
  using *fps-pvanishes-def is-unit-power-iff*[*of* p *Suc* -] **by** *auto*

lemma *fps-pvanishes-uminus*[*simp*]:

fps-pvanishes p (−*f*) **if** *fps-pvanishes p f* **for** *f* :: *'a*::*comm-ring-1 fps*
using *fps-pvanishesD*[*OF that*] *iffD2*[*OF dvd-minus-iff*] **by** (*fastforce intro*: *fps-pvanishesI*)

lemma *fps-pvanishes-add*[*simp*]:
fps-pvanishes p (*f* + *g*) **if** *fps-pvanishes p f* **and** *fps-pvanishes p g*
for *f* :: *'a*::*comm-semiring-1 fps*
proof (*intro fps-pvanishesI*)
fix *n* **from** *that* **show** *p* ^ *Suc n dvd fps-parteval* (*f*+*g*) *p n*
using *fps-pvanishesD*[*of p - n*]
dvd-add[*of p*^*Suc n fps-parteval f p n fps-parteval g p n*]
by *auto*
qed

lemma *fps-pvanishes-minus*[*simp*]:
fps-pvanishes p (*f* − *g*) **if** *fps-pvanishes p f* **and** *fps-pvanishes p g*
for *f* :: *'a*::*comm-ring-1 fps*
using *that fps-pvanishes-uminus*[*of p g*] *fps-pvanishes-add*[*of p f* −*g*] **by** *simp*

lemma *fps-pvanishes-mult*: *fps-pvanishes p* (*f* ∗ *g*) **if** *fps-pvanishes p g*
proof (*intro fps-pvanishesI*)
fix *n*
have *p* ^ *Suc n dvd* (∑ *i*≤*n*. *f*$*i* ∗ *fps-parteval g p* (*n* − *i*) ∗ *p*^*i*)
proof (*intro dvd-sum*)
fix *k* **assume** *k* ∈ {..*n*}
with *that* **show** *p* ^ *Suc n dvd f* $ *k* ∗ *fps-parteval g p* (*n* − *k*) ∗ *p* ^ *k*
using *fps-pvanishesD*[*of p g*] *power-add*[*of p Suc* (*n* − *k*) *k*] **by** (*simp add*: *mult-dvd-mono*)
qed
thus *p* ^ *Suc n dvd fps-parteval* (*f* ∗ *g*) *p n* **by** (*simp add*: *fps-parteval-mult-conv-sum*(*1*))
qed

lemma *fps-pvanishes-mult-fps-X-iff*:
assumes (*p*::*'a*::*idom*) ≠ *0*
shows *fps-pvanishes p f* ⟷ *fps-pvanishes p* (*f* ∗ *fps-X*)
and *fps-pvanishes p f* ⟷ *fps-pvanishes p* (*fps-X* ∗ *f*)
proof−
show *fps-pvanishes p f* ⟷ *fps-pvanishes p* (*f* ∗ *fps-X*)
proof
assume *b*: *fps-pvanishes p* (*f* ∗ *fps-X*)
show *fps-pvanishes p f*
proof (*intro fps-pvanishesI*)
fix *n* **from** *b assms* **show** *p* ^ *Suc n dvd fps-parteval f p n*
using *fps-pvanishesD*[*of p f* ∗ *fps-X Suc n*] *fps-parteval-mult-fps-X*(*1*)[*of f n p*]
by *simp*
qed
qed (*auto intro*: *fps-pvanishes-mult simp add*: *mult.commute*)
thus *fps-pvanishes p f* ⟷ *fps-pvanishes p* (*fps-X* ∗ *f*)
by (*simp add*: *fps-mult-fps-X-commute*)

qed

lemma *fps-pvanishes-mult-fps-X-power-iff*:
  assumes $(p::'a::idom) \neq 0$
  shows *fps-pvanishes p f* $\longleftrightarrow$ *fps-pvanishes p* $(f * \textit{fps-X} \,\widehat{}\, n)$
  and *fps-pvanishes p f* $\longleftrightarrow$ *fps-pvanishes p* $(\textit{fps-X} \,\widehat{}\, n * f)$
proof −
  show *fps-pvanishes p f* $\longleftrightarrow$ *fps-pvanishes p* $(\textit{fps-X} \,\widehat{}\, n * f)$
  proof (*induct n*)
    case (*Suc n*) **with** *assms* **show** *?case*
      using *fps-pvanishes-mult-fps-X-iff*(*2*)[*of p fps-X* $\widehat{}$ *n* ∗ *f*]
      by (*simp add*: *mult.assoc*)
  qed *simp*
  thus *fps-pvanishes p f* $\longleftrightarrow$ *fps-pvanishes p* $(f * \textit{fps-X} \,\widehat{}\, n)$
    by (*simp flip*: *fps-mult-fps-X-power-commute*)
qed

4.4 Equivalence and depth of formal Laurent series relative to a prime

4.4.1 Definition of equivalence and basic facts

abbreviation *fps-primevanishes p* ≡ *fps-pvanishes* (*Rep-prime p*)

overloading
  p-equal-fls ≡
    p-equal :: *'a::nontrivial-factorial-comm-ring prime* ⇒
      'a fls ⇒ *'a fls* ⇒ *bool*
begin

definition *p-equal-fls* ::
  'a::nontrivial-factorial-comm-ring prime ⇒
    'a fls ⇒ *'a fls* ⇒ *bool*
  where
    p-equal-fls p f g ≡ *fps-primevanishes p* (*fls-base-factor-to-fps* $(f - g)$)

end

context
  fixes *p* :: *'a* :: *nontrivial-factorial-comm-ring prime*
begin

lemma *p-equal-flsD*:
  $f \simeq_p g \Longrightarrow$ *fps-primevanishes p* (*fls-base-factor-to-fps* $(f - g)$)
  for *f g* :: *'a fls*
  unfolding *p-equal-fls-def* **by** *simp*

lemma *p-equal-fls-imp-p-dvd*: *p pdvd* $((f - g)$ $$ (*fls-subdegree* $(f - g)$))
  if $f \simeq_p g$
  for *f g* :: *'a fls*

proof −
  have *int* (*0*::*nat*) = *0* **by** *simp*
  thus *?thesis*
    by (*metis*
        that p-equal-flsD fps-pvanishesD power-Suc0-right fps-parteval-0th
        fls-base-factor-to-fps-nth add-0-right
)
qed

lemma *p-equal-flsI*:
  fps-primevanishes p (*fls-base-factor-to-fps* (*f* − *g*)) $\Longrightarrow$ *f* $\simeq_p$ *g*
  for *f g* :: *'a fls*
  unfolding *p-equal-fls-def* **by** *simp*

lemma *p-equal-fls-conv-p-equal-0*:
  f $\simeq_p$ *g* $\longleftrightarrow$ (*f* − *g*) $\simeq_p$ *0*
  for *f g* :: *'a fls*
  unfolding *p-equal-fls-def* **by** *auto*

lemma *p-equal-fls-refl*: *f* $\simeq_p$ *f* **for** *f*:: *'a fls*
  by (*auto intro*: *p-equal-flsI*)

lemma *p-equal-fls-sym*:
  f $\simeq_p$ *g* $\Longrightarrow$ *g* $\simeq_p$ *f* **for** *f g* :: *'a fls*
  using *p-equal-flsD fps-pvanishes-uminus fls-uminus-subdegree*[*of f*−*g*]
  unfolding *p-equal-fls-def*
  by *fastforce*

lemma *p-equal-fls-shift-iff*:
  f $\simeq_p$ *g* $\longleftrightarrow$ (*fls-shift n f*) $\simeq_p$ (*fls-shift n g*)
  for *f g* :: *'a fls*
  using *fls-base-factor-shift*[*of n f* − *g*] **unfolding** *p-equal-fls-def* **by** *auto*

lemma *p-equal-fls-extended-sum-iff*:
  f $\simeq_p$ *g* $\longleftrightarrow$
    ($\forall$ *n*. (*Rep-prime p*) ^ *Suc n dvd*
      ($\sum$ *k*≤*n*. (*f* − *g*)$$(*N* + *int k*) ∗ (*Rep-prime p*)^*k*))
  if *N* ≤ *min* (*fls-subdegree f*) (*fls-subdegree g*)
proof −
  define *pp* **where** *pp* ≡ *Rep-prime p*
  define *dfg* **where** *dfg* ≡ *fls-subdegree* (*f* − *g*)
  define *D* **where** *D* ≡ *nat* (*dfg* − *N* − *1*)
  from *that dfg-def D-def*
    have *D*: *f* ≠ *g* $\Longrightarrow$ *N* ≠ *dfg* $\Longrightarrow$ *dfg* − *N* − *1* ≥ *0*
    using *fls-subdegree-minus*
    by *force*
  have
    (*f* $\simeq_p$ *g*) =
      ($\forall$ *n*. *pp* ^ *Suc n dvd* ($\sum$ *k*≤*n*. (*f* − *g*) $$ (*N* + *int k*) ∗ *pp* ^ *k*))

proof (*standard*, *standard*)
  fix *n* :: *nat*
  assume *fg*: $f \simeq_p g$
  consider
    $f = g$ |
    $N = dfg$ |
    (*n-le-D*) $f \neq g$ $N \neq dfg$ $n \leq D$ |
    (*n-gt-D*) $f \neq g$ $N \neq dfg$ $n > D$
    by *fastforce*
  thus pp ^ $Suc\ n\ dvd\ (\sum k \leq n.\ (f - g)\ \$\$\ (N + int\ k) * pp$ ^ $k)$
  proof *cases*
    case *n-le-D*
    with *D-def* **have** $\forall k \leq n.\ N + int\ k < dfg$ **using** *D* **by** *linarith*
    with *dfg-def* **show** *?thesis* **using** *fls-eq0-below-subdegree*[*of* - $f - g$] **by** *simp*
  next
    case *n-gt-D*
    with *D-def that*
      have *N-k* : $\forall k \leq D.\ N + int\ k < dfg$
      and *N-D-k*: $\forall k.\ N + int\ (Suc\ D + k) = dfg + int\ k$
      using *D*
      by (*force*, *linarith*)
    from *dfg-def* **have** $\forall k \leq D.\ (f - g)\$\$(N + int\ k) = 0$
      using *N-k fls-eq0-below-subdegree*[*of* - $f - g$] **by** *blast*
    hence *partsum*: $(\sum k \leq D.\ (f - g)\$\$(N + int\ k) * pp$^$k) = 0$ **by** *force*
    from *pp-def* ‹$f \simeq_p g$› **obtain** *k*
      where *k*: *fls-base-parteval* $(f - g)\ pp\ (n - Suc\ D) = pp$ ^ $Suc\ (n - Suc\ D) * k$
      using *p-equal-flsD fps-pvanishesD* **by** *blast*
    from ‹$n > D$› **have** *exp*: $Suc\ D + Suc\ (n - Suc\ D) = Suc\ n$ **by** *auto*
    from ‹$n > D$› **have**
      $(\sum k \leq n.\ (f - g)\$\$(N + int\ k) * pp$^$k) =$
        $(\sum k \leq D.\ (f - g)\$\$(N + int\ k) * pp$^$k) +$
        $(\sum k = Suc\ D..n.\ (f - g)\$\$(N + int\ k) * pp$^$k)$
      using *sum-up-index-split*[*of* - $D\ n - D$] **by** *auto*
    also from ‹$n > D$› **have**
      $\ldots = (\sum k \leq n - Suc\ D.$
        $(f - g)\ \$\$\ (N + int\ (Suc\ D + k)) * pp$ ^ $(Suc\ D + k))$
      using *partsum atLeast0AtMost*[*of* $n - Suc\ D$]
          sum.atLeastAtMost-shift-bounds[
            of $\lambda k.\ (f-g)\$\$(N + int\ k) * pp$^$k\ 0\ Suc\ D\ n - Suc\ D$
          ]
      by *auto*
    also from *D-def* **have**
      $\ldots = (\sum k \leq n - Suc\ D.\ (f - g)\ \$\$\ (dfg + int\ k) * (pp$ ^ $Suc\ D * pp$ ^ $k))$
      using *N-D-k power-add*[*of pp Suc D*] **by** *force*
    also have
      $\ldots = pp$ ^ $Suc\ D * (\sum k \leq n - Suc\ D.\ (f - g)\ \$\$\ (dfg + int\ k) * pp$ ^ $k)$
      by (*simp add*: *algebra-simps sum-distrib-left*)
    also have $\ldots = pp$ ^ $Suc\ D * (pp$ ^ $Suc\ (n - Suc\ D) * k)$

```
          using k dfg-def fls-base-parteval-conv-sum[of f − g n − Suc D pp] by
presburger
      also have ... = pp ^ (Suc D + Suc (n − Suc D)) ∗ k
        by (simp add: algebra-simps power-add)
      finally have (∑ k≤n. (f − g)$$(N + int k) ∗ pp^k) = pp ^ Suc n ∗ k
        using ‹n > D› by auto
      thus ?thesis by auto
    qed (simp, metis fg dfg-def pp-def fls-base-parteval-conv-sum p-equal-flsD fps-pvanishesD)
  next
    assume ∗: ∀ n. pp ^ Suc n dvd (∑ k≤n. (f − g)$$(N + int k) ∗ pp^k)
    consider f = g | N = dfg |  f ≠ g N ≠ dfg by blast
    thus f ≃p g
    proof cases
      case 1 thus ?thesis by (simp add: p-equal-fls-refl)
    next
      case 2 with pp-def dfg-def ∗ show ?thesis
        using p-equal-flsI fps-pvanishesI fls-base-parteval-conv-sum by metis
    next
      case 3 show ?thesis
      proof (intro p-equal-flsI fps-pvanishesI)
        fix n
        from 3 dfg-def D-def
          have ND     : ∀ k. N + int (Suc D + k) = dfg + int k
          and  dfg-N-n: nat (dfg − N) + n = Suc D + n
          and  sum-0  : (∑ k≤D. (f − g) $$ (N + int k) ∗ pp ^ k) = 0
          using D fls-eq0-below-subdegree[of - f − g]
          by    (fastforce, simp, simp)
        have
          (∑ k≤nat (dfg − N) + n. (f − g)$$(N + int k) ∗ pp^k) =
            (∑ k = Suc D..Suc D + n. (f − g) $$ (N + int k) ∗ pp ^ k)
          using dfg-N-n sum-0
                sum-up-index-split[of λk. (f − g)$$(N + int k) ∗ pp^k D Suc n]
          by    auto
        also have
          ... = sum ((λk. (f − g) $$ (N + int k) ∗ pp ^ k) ∘ (+) (Suc D))
            {0..n}
          using add.commute[of Suc D n] add-0-left[of Suc D]
                sum.atLeastAtMost-shift-bounds[
                  of λk. (f−g)$$(N + int k) ∗ pp^k 0 Suc D n
                ]
          by    argo
        also have ... = pp ^ Suc D ∗ (∑ k≤n. (f − g) $$ (dfg + int k) ∗ pp ^ k)
          using ND by (force simp add: algebra-simps atLeast0AtMost power-add
sum-distrib-left)
        finally have
          (∑ k≤nat (dfg − N) + n. (f − g)$$(N + int k) ∗ pp^k) =
            pp ^ Suc D ∗ fls-base-parteval (f − g) pp n
          using dfg-def fls-base-parteval-conv-sum[of f − g n pp] by force
        moreover have Suc (nat (dfg − N) + n) = Suc D + Suc n using dfg-N-n
```

by *presburger*
moreover from *pp-def* **have** pp ^ *Suc D* $\neq$ *0* **using** *Rep-prime-n0* **by** *auto*
ultimately have pp ^ *Suc n dvd fls-base-parteval* $(f - g)$ *pp n*
using $*$ *power-add dvd-times-left-cancel-iff* **by** *metis*
with *pp-def* **show** *Rep-prime p* ^ *Suc n dvd fls-base-parteval* $(f - g)$ (*Rep-prime p*) *n*
by *blast*
qed
qed
qed
with *pp-def* **show** *?thesis* **by** *fast*
qed

lemma *p-equal-fls-extended-intsum-iff*:
$f \simeq_p g \longleftrightarrow$
($\forall n \geq N$. (*Rep-prime p*) ^ *Suc* (*nat* $(n - N)$) *dvd*
($\sum k=N..n$. $(f - g)$\$\$$k$ $*$ (*Rep-prime p*) ^ *nat* $(k - N)$))
if $N \leq$ *min* (*fls-subdegree f*) (*fls-subdegree g*)
proof (*standard*, *clarify*)
assume *fg*: $f \simeq_p g$
fix *n*::*int* **assume** *n*: $n \geq N$
define *m* **where** $m \equiv$ *nat* $(n - N)$
from *that fg* **have**
(*Rep-prime p*) ^ *Suc m dvd* ($\sum k \leq m$. $(f - g)$\$\$$(N +$ *int k*$)$ $*$ (*Rep-prime p*)^k)
using *p-equal-fls-extended-sum-iff* **by** *blast*
moreover from *n m-def* **have**
($\sum k=N..n$. $(f - g)$\$\$$k$ $*$ (*Rep-prime p*) ^ *nat*($k - N$)) =
($\sum k \leq m$. $(f - g)$\$\$$(N +$ *int k*$)$ $*$ (*Rep-prime p*)^k)
by (*simp add*: *sum.atLeastAtMost-int-shift-bounds algebra-simps*)
ultimately show
(*Rep-prime p*) ^ *Suc* (*nat* $(n - N)$) *dvd*
($\sum k=N..n$. $(f - g)$\$\$$k$ $*$ (*Rep-prime p*) ^ *nat* $(k - N)$)
using *m-def* **by** *argo*
next
assume *dvd-sum*:
$\forall n \geq N$. (*Rep-prime p*) ^ *Suc* (*nat* $(n - N)$) *dvd*
($\sum k=N..n$. $(f - g)$\$\$$k$ $*$ (*Rep-prime p*) ^ *nat* $(k - N)$)
show $f \simeq_p g$
proof (*rule iffD2*, *rule p-equal-fls-extended-sum-iff*, *rule that*, *safe*)
fix *n*::*nat*
have *N-n*: $N +$ *int n* $\geq N$ **by** *fastforce*
moreover have *nat* $(N +$ *int n* $- N) = n$ **by** *simp*
ultimately have
(*Rep-prime p*) ^ *Suc n dvd*
($\sum k=N..N +$ *int n*. $(f - g)$\$\$$k$ $*$ (*Rep-prime p*) ^ *nat* $(k - N)$)
using *dvd-sum* **by** *fastforce*
moreover from *N-n* **have**
($\sum k \leq n$. $(f - g)$ \$\$ $(N +$ *int k*$)$ $*$ *Rep-prime p* ^ k) =
($\sum k=N..N +$ *int n*. $(f - g)$\$\$$k$ $*$ (*Rep-prime p*) ^ *nat* $(k - N)$)

by (*simp add*: *sum.atLeastAtMost-int-shift-bounds*)
ultimately show
Rep-prime p ^ *Suc n dvd* ($\sum$ *k*≤*n*. (*f* − *g*) $$ (*N* + *int k*) ∗ *Rep-prime p* ^ *k*)
by *presburger*
qed
qed

lemma *fls-p-equal0-extended-sum-iff*:
f ≃$_p$ *0* ⟷
(∀ *n*. (*Rep-prime p*) ^ *Suc n dvd*
($\sum$ *k*≤*n*. *f*$$(*N* + *int k*) ∗ (*Rep-prime p*)^*k*))
if *N* ≤ *fls-subdegree f*
proof−
define *pp* **where** *pp* ≡ *Rep-prime p*
have *d0-case*:
⋀*f N*. *fls-subdegree f* = *0* ⟹ *N* ≤ *0* ⟹
f ≃$_p$ *0* ⟷
(∀ *n*. *pp* ^ *Suc n dvd* ($\sum$ *k*≤*n*. *f*$$(*N* + *int k*) ∗ *pp*^*k*))
proof−
fix *f N* **from** *pp-def*
show *fls-subdegree f* = *0* ⟹ *N* ≤ *0* ⟹ *?thesis f N*
using *p-equal-fls-extended-sum-iff*[*of N f 0*]
by *simp*
qed
have *f* ≃$_p$ *0* ⟷ (*fls-base-factor f*) ≃$_p$ *0*
using *p-equal-fls-shift-iff*[*of f 0*] *fls-base-factor-def* **by** *auto*
also from *that pp-def* **have**
. . . = (∀ *n*. *pp* ^ *Suc n dvd* ($\sum$ *k*≤*n*. *f* $$ ((*N* + *int k*)) ∗ *pp*^*k*))
using *fls-base-factor-subdegree*[*of f*] *d0-case*[*of fls-base-factor f N* − *fls-subdegree f*]
by *auto*
finally show *?thesis* **using** *pp-def* **by** *blast*
qed

lemma *fls-p-equal0-extended-intsum-iff*:
f ≃$_p$ *0* ⟷
(∀ *n*≥*N*. (*Rep-prime p*) ^ *Suc* (*nat* (*n* − *N*)) *dvd*
($\sum$ *k*=*N*..*n*. *f*$$*k* ∗ (*Rep-prime p*) ^ *nat* (*k* − *N*)))
if *N* ≤ *fls-subdegree f*
proof (*standard*, *clarify*)
assume *f*: *f* ≃$_p$ *0*
fix *n*::*int* **assume** *n*: *n* ≥ *N*
define *m* **where** *m* ≡ *nat* (*n* − *N*)
from *that f* **have**
(*Rep-prime p*) ^ *Suc m dvd* ($\sum$ *k*≤*m*. *f*$$(*N* + *int k*) ∗ (*Rep-prime p*)^*k*)
using *fls-p-equal0-extended-sum-iff* **by** *blast*
moreover from *n m-def* **have**
($\sum$ *k*=*N*..*n*. *f*$$*k* ∗ (*Rep-prime p*) ^ *nat* (*k* − *N*)) =
($\sum$ *k*≤*m*. *f*$$(*N* + *int k*) ∗ (*Rep-prime p*)^*k*)

by (*simp add*: *sum.atLeastAtMost-int-shift-bounds algebra-simps*)
ultimately show
(*Rep-prime p*) ^ *Suc* (*nat* (*n* − *N*)) *dvd*
($\sum$ *k*=*N*..*n*. *f*$$*k* ∗ (*Rep-prime p*) ^ *nat* (*k* − *N*))
using *m-def* **by** *argo*
next
assume *dvd-sum*:
∀ *n*≥*N*. (*Rep-prime p*) ^ *Suc* (*nat* (*n* − *N*)) *dvd*
($\sum$ *k*=*N*..*n*. *f*$$*k* ∗ (*Rep-prime p*) ^ *nat* (*k* − *N*))
show *f* $\simeq_p$ *0*
proof (*rule iffD2*, *rule fls-p-equal0-extended-sum-iff*, *rule that*, *safe*)
fix *n*::*nat*
have *N-n*: *N* + *int n* ≥ *N* **by** *fastforce*
moreover have *nat* (*N* + *int n* − *N*) = *n* **by** *simp*
ultimately have
(*Rep-prime p*) ^ *Suc n dvd*
($\sum$ *k*=*N*..*N* + *int n*. *f*$$*k* ∗ (*Rep-prime p*) ^ *nat* (*k* − *N*))
using *dvd-sum* **by** *fastforce*
moreover from *N-n* **have**
($\sum$ *k*≤*n*. *f* $$ (*N* + *int k*) ∗ *Rep-prime p* ^ *k*) =
($\sum$ *k*=*N*..*N* + *int n*. *f*$$*k* ∗ (*Rep-prime p*) ^ *nat* (*k* − *N*))
by (*simp add*: *sum.atLeastAtMost-int-shift-bounds*)
ultimately show
Rep-prime p ^ *Suc n dvd* ($\sum$ *k*≤*n*. *f* $$ (*N* + *int k*) ∗ *Rep-prime p* ^ *k*)
by *presburger*
qed
qed

end

context
fixes *p* :: *'a* :: *nontrivial-factorial-idom prime*
begin

lemma *fls-1-not-p-equal-0*: (*1*::*'a fls*) ¬$\simeq_p$ *0*
using *fps-pvanishes-one-iff not-prime-unit Rep-prime*[*of p*] **unfolding** *p-equal-fls-def*
by *auto*

lemma *p-equal-fls0-nonneg-subdegree*:
f $\simeq_p$ *0* ⟷ *fps-primevanishes p* (*fls-regpart f*)
if *fls-subdegree f* ≥ *0*
proof−
define *n* **where** *n* ≡ *nat* (*fls-subdegree f*)
moreover have
f $\simeq_p$ *0* ⟷
fps-primevanishes p (*fls-base-factor-to-fps f* ∗ *fps-X* ^ *n*)
unfolding *p-equal-fls-def*
using *Rep-prime*[*of p*] *fps-pvanishes-mult-fps-X-power-iff*(*1*)[*of Rep-prime p*]

by *simp*
ultimately show *?thesis*
using *fls-base-factor-subdegree*[*of f*] *fls-regpart-times-fps-X-power*(*1*)[*of fls-base-factor f n*]
by (*auto simp add*: *that*)
qed

lemma *p-equal-fls-nonneg-subdegree*:
$f \simeq_p g \longleftrightarrow$ *fps-primevanishes p* (*fls-regpart* $(f - g)$)
if *fls-subdegree f* ≥ 0 **and** *fls-subdegree g* ≥ 0
proof−
from *that* **have** *fls-subdegree* $(f - g) \geq 0$
using *fls-subdegree-minus*[*of f g*] **by** (*cases* $f = g$) *auto*
thus *?thesis* **using** *p-equal-fls-conv-p-equal-0*[*of p f g*] *p-equal-fls0-nonneg-subdegree*
by *blast*
qed

lemma *p-equal-fls-trans-to-0*:
$f \simeq_p g$ **if** $f \simeq_p 0$ **and** $g \simeq_p 0$ **for** *f g* :: *'a fls*
proof (*rule iffD2*, *rule p-equal-fls-shift-iff*, *rule iffD2*, *rule p-equal-fls-nonneg-subdegree*)
define *n* **where** $n \equiv$ *min* (*fls-subdegree f*) (*fls-subdegree g*)
from *n-def* **show** *fls-subdegree* (*fls-shift n f*) ≥ 0 *fls-subdegree* (*fls-shift n g*) ≥ 0
using *fls-shift-nonneg-subdegree*[*of n f*] *fls-shift-nonneg-subdegree*[*of n g*] **by** *auto*
with *that* **show** *fps-primevanishes p* (*fls-regpart* (*fls-shift n f* − *fls-shift n g*))
using *p-equal-fls0-nonneg-subdegree*[*of fls-shift n f*] *p-equal-fls-shift-iff*[*of p f 0*]
p-equal-fls0-nonneg-subdegree[*of fls-shift n g*] *p-equal-fls-shift-iff*[*of p g 0*]
by (*auto intro*: *fps-pvanishes-minus*)
qed

lemma *p-equal-fls-trans*:
$f \simeq_p g \Longrightarrow g \simeq_p h \Longrightarrow$
$f \simeq_p h$
for *f g h* :: *'a fls*
using *p-equal-fls-conv-p-equal-0*[*of p f g*] *p-equal-fls-conv-p-equal-0*[*of p h g*]
p-equal-fls-sym[*of p g h*] *p-equal-fls-trans-to-0*[*of f−g h−g*]
p-equal-fls-conv-p-equal-0[*of p f−g h−g*] *p-equal-fls-conv-p-equal-0*[*of p f h*]
by *auto*

end

global-interpretation *p-equality-fls*:
p-equality-1
p-equal :: *'a::nontrivial-factorial-idom prime* ⇒
'a fls ⇒ *'a fls* ⇒ *bool*
proof (*unfold-locales*, *intro equivpI*)
fix *p* :: *'a prime*
define *E* :: *'a fls* ⇒ *'a fls* ⇒ *bool* **where** $E \equiv$ *p-equal p*
show *reflp E*
by (*auto intro*: *reflpI simp add*: *E-def p-equal-fls-refl*)

show *symp E* **using** *p-equal-fls-sym* **by** (*auto intro*: *sympI simp add*: *E-def*)
show *transp E*
using *E-def p-equal-fls-trans*[*of p*] **by** (*fastforce intro*: *transpI*)

fix *x y* :: *'a fls*
show $y \simeq_p 0 \Longrightarrow x * y \simeq_p 0$
using *p-equal-flsD fps-pvanishes-mult fls-base-factor-to-fps-mult*[*of x y*]
unfolding *p-equal-fls-def*
by *fastforce*

qed (*rule p-equal-fls-conv-p-equal-0*)

overloading
globally-p-equal-fls ≡
globally-p-equal ::
'a::*nontrivial-factorial-idom fls* ⇒ *'a fls* ⇒ *bool*
begin

definition *globally-p-equal-fls* ::
'a::*nontrivial-factorial-idom fls* ⇒ *'a fls* ⇒ *bool*
where *globally-p-equal-fls*[*simp*]:
globally-p-equal-fls = *p-equality-fls.globally-p-equal*

end

context
fixes *p* :: *'a* :: *nontrivial-factorial-idom prime*
begin

lemma *p-equal-fls-const-iff*:
(*fls-const c*) $\simeq_p$ (*fls-const d*) ⟷ *c* = *d* **for** *c d* :: *'a*
proof
define *pp* **where** *pp* ≡ *Rep-prime p*
assume (*fls-const c*) $\simeq_p$ (*fls-const d*)
with *pp-def* **have** *1*: ∀ *n*::*nat*. *pp* ^ *Suc n dvd* (*c* − *d*)
using *p-equal-flsD* **unfolding** *fps-pvanishes-def* **by** (*fastforce simp add*: *fls-minus-const*)
have *infinite* {*n*. *pp* ^ *n dvd* (*c* − *d*)}
proof
assume *finite* {*n*. *pp* ^ *n dvd* (*c* − *d*)}
moreover have *Suc* (*Max* {*n*. *pp* ^ *n dvd* (*c* − *d*)}) ∈ {*n*. *pp* ^ *n dvd* (*c* − *d*)} **using** *1* **by** *simp*
ultimately have *Suc* (*Max* {*n*. *pp* ^ *n dvd* (*c* − *d*)}) ≤ *Max* {*n*. *pp* ^ *n dvd* (*c* − *d*)} **by** *simp*
thus *False* **by** *simp*
qed
with *pp-def* **show** *c* = *d* **using** *Rep-prime-not-unit finite-divisor-powers*[*of c* − *d*] **by** *fastforce*
qed *simp*

lemma *p-equal-fls-const-0-iff*:
  (*fls-const c*) $\simeq_p$ *0* $\longleftrightarrow$ *c* = *0* **for** *c* :: *'a*
  using *p-equal-fls-const-iff*[*of c 0*] **by** *simp*

lemma *p-equal-fls-X-intpow-iff*:
  ((*fls-X-intpow m* :: *'a fls*) $\simeq_p$ (*fls-X-intpow n*)) = (*m* = *n*)
proof (*induction m n rule*: *linorder-less-wlog*, *standard*)
  define *pp* **and** *X* :: *int* ⇒ *'a fls*
    where *pp* ≡ *Rep-prime p* **and** *X* ≡ *fls-X-intpow*
  fix *m n* :: *int* **assume** *eq*: (*X m*) $\simeq_p$ (*X n*) **and** *mn*: *m* < *n*
  with *pp-def X-def* **have**
    pp ^ *Suc* (*nat* (*n* − *m*)) *dvd* (∑*k*=*m*..*n*. (*X m* − *X n*)$$*k* ∗ *pp* ^ *nat* (*k* − *m*))
    using *p-equal-fls-extended-intsum-iff*[*of m X m X n p*]
    by *auto*
  moreover have
    (∑*k*=*m*..*n*. (*X m* − *X n*)$$*k* ∗ *pp* ^ *nat* (*k* − *m*)) =
      (∑*k*∈{*m*,*n*}. (*X m* − *X n*)$$*k* ∗ *pp* ^ *nat* (*k* − *m*))
  proof (*rule sum.mono-neutral-right*, *safe*)
    from *mn* **show** *m* ∈ {*m*..*n*} **and** *n* ∈ {*m*..*n*} **by** *auto*
    fix *i*
    assume *i* : *i* ∈ {*m*..*n*} *i* ≠ *m*
      and *Xi*: (*X m* − *X n*) $$ *i* ∗ *pp* ^ *nat* (*i* − *m*) ≠ *0*
    with *X-def* **have** *X m* $$ *i* = *0* **by** *simp*
    moreover from *X-def* **have** *i* < *n* ⟹ *X n* $$ *i* = *0* **by** *force*
    ultimately show *i* = *n* **using** *i*(*1*) *Xi* **by** *fastforce*
  qed
  ultimately have *pp* ^ *Suc* (*nat* (*n* − *m*)) *dvd 1* − (*pp* ^ *nat* (*n* − *m*))
    using *mn X-def* **by** *auto*
  from *this* **obtain** *k* **where** *1* − (*pp* ^ *nat* (*n* − *m*)) = *pp* ^ *Suc* (*nat* (*n* − *m*))
∗ *k* **by** *fast*
  hence *1* = *pp* ^ *nat* (*n* − *m*) ∗ (*1* + *pp* ∗ *k*) **by** (*simp add*: *algebra-simps*)
  with *pp-def mn* **show** *m* = *n*
    using *dvdI*[*of 1*] *Rep-prime-not-unit is-unit-power-iff*[*of pp*] **by** *fastforce*
qed (*auto simp add*: *p-equal-fls-sym*)

lemma *fls-X-intpow-nequiv0*: (*fls-X-intpow m* :: *'a fls*) ¬$\simeq_p$ *0*
  using *p-equal-fls-shift-iff fls-1-not-p-equal-0* **by** *fastforce*

end

4.4.2 Depth as maximum subdegree of equivalent series

Each nonzero equivalence class will have a maximum subdegree.

lemma *no-greatest-p-equal-subdegree*:
  f $\simeq_p$ *0*
  if ∀ *n* :: *int*. (∃ *g*. *fls-subdegree g* > *n* ∧ *f* $\simeq_p$ *g*)
  for *p* :: *'a*::*nontrivial-factorial-comm-ring prime* **and** *f* :: *'a fls*
proof (*cases f* = *0*)
  case *False* **show** *?thesis*

proof (*intro p-equal-flsI fps-pvanishesI*)
fix n :: *nat*
define *pp* **where** $pp \equiv$ *Rep-prime p*
from *that* **obtain** g **where** g: *fls-subdegree* $g >$ *fls-subdegree* f + *int* n $f \simeq_p g$
by *blast*
with *False pp-def* **have** *fls-base-parteval* $(f - g)$ *pp* n = *fls-base-parteval* $(f - 0)$ *pp* n
using *fps-parteval-fls-base-factor-to-fps-diff* **by** *auto*
with *False pp-def* $g(2)$ **show** *pp* ^ *Suc n dvd fls-base-parteval* $(f - 0)$ *pp* n
using *p-equal-flsD fps-pvanishesD* **by** *metis*
qed
qed (*simp add*: *p-equal-fls-refl*)

abbreviation
p-equal-fls-simplified-set p f $\equiv$
$\{g.\ f \simeq_p g \wedge$ *fls-subdegree* $g \geq$ *fls-subdegree* $f\}$
— We restrict to those equivalent series that represent a simplification of the given series so that the image of the collection under taking subdegrees is finite.

context
fixes p :: *'a::nontrivial-factorial-comm-ring prime*
and f :: *'a fls*
begin

lemma *p-equal-fls-simplifed-nempty*:
fls-subdegree ‘ (*p-equal-fls-simplified-set p f*) $\neq \{\}$
using *p-equal-fls-refl*[*of p f*] **by** *fast*

lemma *p-equal-fls-simplifed-finite*:
finite (*fls-subdegree* ‘ (*p-equal-fls-simplified-set p f*)) **if** $f \neg\simeq_p 0$
proof−
from *that* **obtain** *n::int*
where $\forall g.\ f \simeq_p g \longrightarrow \neg$ *fls-subdegree* $g > n$
using *no-greatest-p-equal-subdegree*[*of p f*]
by *auto*
hence n: $\forall g.\ f \simeq_p g \longrightarrow$ *fls-subdegree* $g \leq n$ **by** *auto*
define *deg-diff* **where** *deg-diff* $\equiv (\lambda n.\ n -$ *fls-subdegree* $f)$
define D **where** $D \equiv$ *fls-subdegree* ‘ (*p-equal-fls-simplified-set p f*)
show *finite D*
proof (*rule finite-imageD*)
show *finite* (*deg-diff* ‘ D)
proof (*rule finite-imageD*)
have *nat* ‘ *deg-diff* ‘ $D \subseteq \{m::nat.\ m \leq$ *nat* (*deg-diff n*)$\}$
using *n D-def deg-diff-def* **by** *auto*
thus *finite* (*nat* ‘ *deg-diff* ‘ D) **using** *finite-subset* **by** *blast*
from *D-def deg-diff-def* **show** *inj-on nat* (*deg-diff* ‘ D)
by (*auto intro*: *inj-onI*)
qed
from *deg-diff-def* **show** *inj-on deg-diff D* **by** (*auto intro*: *inj-onI*)

qed
qed

end

overloading
p-depth-fls ≡
p-depth :: *'a::nontrivial-factorial-comm-ring prime* ⇒ *'a fls* ⇒ *int*
p-depth-const ≡
p-depth :: *'a::nontrivial-factorial-comm-ring prime* ⇒ *'a* ⇒ *int*
begin

definition
p-depth-fls :: *'a::nontrivial-factorial-comm-ring prime* ⇒ *'a fls* ⇒ *int*
where
p-depth-fls p f ≡
(*if f* $\simeq_p$ *0 then 0 else Max* (*fls-subdegree* ‘ (*p-equal-fls-simplified-set p f*)))

definition
p-depth-const :: *'a::nontrivial-factorial-comm-ring prime* ⇒ *'a* ⇒ *int*
where
p-depth-const p c ≡ *p-depth p* (*fls-const c*)

end

context
fixes *p* :: *'a::nontrivial-factorial-comm-ring prime*
begin

lemma *p-depth-fls-eqI*:
$f^{\circ p}$ = *fls-subdegree g*
if *f* ¬$\simeq_p$ *0* **and** *f* $\simeq_p$ *g*
and $\bigwedge h$. *f* $\simeq_p$ *h* ⟹ *fls-subdegree h* ≤ *fls-subdegree g*
for *f g* :: *'a fls*
proof−
define *M* **where** *M* ≡ *Max* (*fls-subdegree* ‘ (*p-equal-fls-simplified-set p f*))
with *that* **have** *M* = *fls-subdegree g*
using *p-equal-fls-simplifed-finite*
by (*force intro*: *Max-eqI simp add*: *p-equal-fls-refl*)
with *that*(*1*) *M-def* **show** *?thesis* **by** (*auto simp add*: *p-depth-fls-def*)
qed

lemma *p-depth-fls-p-equal0*[*simp*]:
($f^{\circ p}$) = *0* **if** *f* $\simeq_p$ *0* **for** *f* :: *'a fls*
using *that* **by** (*simp add*: *p-depth-fls-def*)

lemma *p-depth-fls0*[*simp*]: (*0*::*'a fls*)$^{\circ p}$ = *0*
by (*simp add*: *p-depth-fls-def p-equal-fls-refl*)

lemma *p-depth-fls-n0D*:
$f^{\circ p}$ = *Max* (*fls-subdegree* ‘ (*p-equal-fls-simplified-set p f*))
if $f \neg\simeq_p 0$
for *f* :: *'a fls*
using *that* **by** (*simp add*: *p-depth-fls-def*)

lemma *p-depth-flsE*:
fixes *f* :: *'a fls*
assumes $f \neg\simeq_p 0$
obtains *g* **where** $f \simeq_p g$ **and** *fls-subdegree g* = $(f^{\circ p})$
and $\forall h.\ f \simeq_p h \longrightarrow$ *fls-subdegree h* $\leq$ *fls-subdegree g*
proof−
define *M* **where** *M* ≡ *Max* (*fls-subdegree* ‘ (*p-equal-fls-simplified-set p f*))
with *assms* **have** *M* ∈ *fls-subdegree* ‘ (*p-equal-fls-simplified-set p f*)
using *p-equal-fls-simplifed-finite*[*of p f*] *p-equal-fls-simplifed-nempty*[*of p f*] **by** *simp*
from *this* **obtain** *g* **where** *g*: *g* ∈ *p-equal-fls-simplified-set p f M* = *fls-subdegree g* **by** *fast*
from *g*(*1*) **have** $f \simeq_p g$ **by** *blast*
moreover from *g*(*2*) *M-def assms* **have** *2*: *fls-subdegree g* = $(f^{\circ p})$
by (*fastforce simp add*: *p-depth-fls-def*)
moreover have
$\forall h.\ f \simeq_p h \longrightarrow$ *fls-subdegree h* $\leq$ *fls-subdegree g*
proof *clarify*
fix *h* **assume** $f \simeq_p h$ **show** *fls-subdegree h* $\leq$ *fls-subdegree g*
proof (*cases fls-subdegree h* $\geq$ *fls-subdegree f*)
case *True* **with** *assms* ‹$f \simeq_p h$› *M-def g* **show** *?thesis*
using *p-equal-fls-simplifed-finite* **by** *force*
next
case *False*
moreover from *assms M-def g* **have** *fls-subdegree f* $\leq$ *fls-subdegree g*
using *p-equal-fls-simplifed-finite* **by** *blast*
ultimately show *?thesis* **by** *simp*
qed
qed
ultimately show *thesis* **using** *that* **by** *simp*
qed

lemma *p-depth-fls-ge-p-equal-subdegree*:
$(f^{\circ p}) \geq$ *fls-subdegree g*
if $f \neg\simeq_p 0$ **and** $f \simeq_p g$
for *f g* :: *'a fls*
proof (*cases fls-subdegree g* $\geq$ *fls-subdegree f*)
case *True* **with** *that* **show** *?thesis*
using *p-equal-fls-simplifed-finite* **by** (*fastforce simp add*: *p-depth-fls-def*)
next
case *False*
moreover from *that*(*1*) **have** *fls-subdegree f* $\leq (f^{\circ p})$
using *p-equal-fls-simplifed-finite* **by** (*fastforce simp add*: *p-equal-fls-refl p-depth-fls-def*)

ultimately show *?thesis* **by** *simp*
qed

lemma *p-depth-fls-ge-p-equal-subdegree′*:
$(f^{\circ p}) \geq$ *fls-subdegree f* **if** $f \neg\simeq_p 0$ **for** *f g* :: *′a fls*
using *that p-depth-fls-ge-p-equal-subdegree p-equal-fls-refl* **by** *blast*

lemma *p-equal-fls-simplified-set-shift*:
p-equal-fls-simplified-set p (*fls-shift n f*) = *fls-shift n* ‘ *p-equal-fls-simplified-set p f*
if $f \neg\simeq_p 0$
for *f* :: *′a fls*
proof−
have ∗:
⋀*f n*. (*f*::*′a fls*) $\neg\simeq_p 0 \Longrightarrow$
fls-shift n ‘ *p-equal-fls-simplified-set p f* ⊆
p-equal-fls-simplified-set p (*fls-shift n f*)
proof (*intro subsetI*, *clarify*)
fix *n* ::*int*
and *f g* :: *′a fls*
assume *f-neq-0*: $f \neg\simeq_p 0$
and *eq* : $f \simeq_p g$
and *deg* : *fls-subdegree f* ≤ *fls-subdegree g*
from *f-neq-0 eq* **have** $f \neq 0$ $g \neq 0$ **using** *p-equal-fls-refl* **by** *auto*
with *deg* **have** *fls-subdegree* (*fls-shift n f*) ≤ *fls-subdegree* (*fls-shift n g*) **by** *simp*
moreover from *eq* **have** (*fls-shift n f*) $\simeq_p$ (*fls-shift n g*)
using *p-equal-fls-shift-iff* **by** *fast*
ultimately show
(*fls-shift n f*) $\simeq_p$ (*fls-shift n g*) ∧
fls-subdegree (*fls-shift n f*) ≤ *fls-subdegree* (*fls-shift n g*)
by *fast*
qed
have $f \simeq_p 0$ ⟷ (*fls-shift n f*) $\simeq_p 0$
using *p-equal-fls-shift-iff*[*of p f 0*] **by** *simp*
with *that* **have**
p-equal-fls-simplified-set p (*fls-shift n f*) ⊆
fls-shift n ‘ *p-equal-fls-simplified-set p f*
using ∗[*of fls-shift n f* −*n*] **by** *force*
with *that* **show**
p-equal-fls-simplified-set p (*fls-shift n f*) = *fls-shift n* ‘ *p-equal-fls-simplified-set p f*
using ∗[*of f n*] **by** *fast*
qed

lemma *p-equal-fls-simplified-set-shift-subdegrees*:
fls-subdegree ‘ *p-equal-fls-simplified-set p* (*fls-shift n f*) =
(λ*x*. *x* − *n*) ‘ *fls-subdegree* ‘ *p-equal-fls-simplified-set p f*
if $f \neg\simeq_p 0$

for f :: $'a$ *fls*
proof −
from *that* **have**
fls-subdegree ` *p-equal-fls-simplified-set* p (*fls-shift* n f) =
fls-subdegree ` *fls-shift* $n$ ` *p-equal-fls-simplified-set* p f
using *p-equal-fls-simplified-set-shift* **by** *fast*
moreover from *that* **have** $\forall$ g∈*p-equal-fls-simplified-set* p f. $g \neq 0$ **by** *auto*
ultimately show *?thesis* **by** *force*
qed

lemma *p-depth-fls-shift*:
(*fls-shift* n f)$^{\circ p}$ = ($f^{\circ p}$) − n **if** f $\neg\simeq_p$ *0*
for f :: $'a$ *fls*
proof −
from *that* **have** *sf-nequiv-0*: (*fls-shift* n f) $\neg\simeq_p$ *0*
using *p-equal-fls-shift-iff* **by** *fastforce*
with *that* **have**
(*fls-shift* n f)$^{\circ p}$ =
Max ((λx. $x - n$) ` *fls-subdegree* ` *p-equal-fls-simplified-set* p f)
using *p-depth-fls-n0D* *p-equal-fls-simplified-set-shift-subdegrees*
by *fastforce*
thus *?thesis*
using *p-equal-fls-simplifed-finite*[*OF that*] *p-equal-fls-simplifed-nempty*
p-depth-fls-n0D[*OF that*]
Max-add-commute[*of*
fls-subdegree ` *p-equal-fls-simplified-set* p f
λx. x
$-n$
]
by *auto*
qed

end

context
fixes p :: $'a$::*nontrivial-factorial-idom prime*
and f g :: $'a$ *fls*
begin

lemma *p-depth-fls-p-equal*: ($f^{\circ p}$) = ($g^{\circ p}$) **if** $f \simeq_p g$
proof (*cases* $f \simeq_p$ *0*)
case *True* **with** *that* **show** *?thesis* **using** *p-equality-fls.trans0-iff* **by** *fastforce*
next
case *False*
with *that* **have** g: g $\neg\simeq_p$ *0* **using** *p-equality-fls.trans0-iff* **by** *blast*
from *this* **obtain** h **where** h:
$g \simeq_p h$ *fls-subdegree* h = ($g^{\circ p}$)
by (*elim p-depth-flsE*)
from *that* h(*1*) **have** $f \simeq_p h$ **using** *p-equality-fls.trans* **by** *fast*

moreover from *that g h*(*2*) **have**
$\bigwedge h'$. *f* $\simeq_p$ *h'* $\Longrightarrow$ *fls-subdegree h'* $\leq$ *fls-subdegree h*
using *p-equality-fls.trans-right p-depth-fls-ge-p-equal-subdegree* **by** *metis*
ultimately show *?thesis*
using *False h*(*2*) *p-depth-fls-eqI* **by** *metis*
qed

lemma *p-depth-fls-can-simplify-iff*:
p pdvd g $$ (*fls-subdegree g*) = (*fls-subdegree g* < ($f^{\circ p}$))
if *f-nequiv0*: *f* $\neg\simeq_p$ *0*
and *fg-equiv* : *f* $\simeq_p$ *g*
proof
define *gdeg* **where** *gdeg* ≡ *fls-subdegree g*
define *gth* **where** *gth* ≡ *g* $$ *gdeg*
define *pp* **where** *pp* ≡ *Rep-prime p*
assume *p-dvd*: *p pdvd gth*
from *this* **obtain** *k* **where** *k*: *gth* = *pp* ∗ *k* **unfolding** *pp-def gth-def gdeg-def* **by** (*elim dvdE*)
define *h* **where**
h ≡ *g* − *fls-shift* (−*gdeg*) (*fls-const gth*) + *fls-shift* (−(*gdeg* + *1*)) (*fls-const k*)
from *f-nequiv0 fg-equiv* **have** *g* ≠ *0* **by** *blast*
with *gth-def gdeg-def* **have**
fls-subdegree (*fls-shift* (−*gdeg*) (*fls-const gth*) − *fls-shift* (−(*gdeg* + *1*)) (*fls-const k*)) =
gdeg
by (*auto intro*: *fls-subdegree-eqI*)
with *h-def* **have** *g-h*: *fls-base-factor-to-fps* (*g* − *h*) = *fps-const gth* − *fps-const k* ∗ *fps-X*
by (*auto intro*: *fps-ext*)
have *p-g-h*: $\bigwedge n$. *pp* ^ *Suc n dvd fls-base-parteval* (*g* − *h*) *pp n*
proof−
fix *n*
from *k p-dvd* **show** *pp* ^ *Suc n dvd fls-base-parteval* (*g* − *h*) *pp n*
using *g-h* **by** (*cases n*) (*auto simp add*: *fps-parteval-mult-fps-X*(*1*))
qed
with *pp-def* **have** *g* $\simeq_p$ *h* **by** (*fastforce intro*: *p-equal-flsI fps-pvanishesI*)
moreover have *gdeg* < *fls-subdegree h*
proof (*intro fls-subdegree-greaterI*)
show *h* ≠ *0*
proof
assume*h* = *0*
with *pp-def* **have** *g* $\simeq_p$ *0*
using *p-g-h* **by** (*fastforce intro*: *p-equal-flsI fps-pvanishesI*)
with *f-nequiv0 fg-equiv* **show** *False* **using** *p-equality-fls.trans0-iff* **by** *blast*
qed
qed (*simp add*: *h-def gdeg-def gth-def*)
ultimately show *gdeg* < ($f^{\circ p}$)
using *gdeg-def that p-depth-fls-ge-p-equal-subdegree*[*of p f h*]
p-equal-fls-trans[*of p f g h*]

by *force*
next
define *pp* **where** $pp \equiv$ *Rep-prime p*
assume *g*: *fls-subdegree g* $<$ $(f^{\circ p})$
with *that* **obtain** *h* **where** *h*: $f \simeq_p h$ *fls-subdegree h* $= (f^{\circ p})$
by (*elim p-depth-flsE*)
from *f-nequiv0 fg-equiv* **have** *g-n0*: $g \neq 0$ **by** *blast*
with *g h*(*2*) **have** *fls-subdegree* $(g - h) =$ *fls-subdegree g* **by** (*auto intro*: *fls-subdegree-eqI*)
moreover from *fg-equiv h*(*1*) *pp-def*
have *pp* ^ *Suc 0 dvd* (*fps-parteval* (*fls-regpart* (*fls-base-factor* $(g - h)$)) *pp 0*)
using *p-equality-fls.trans-right p-equal-flsD fps-pvanishesD*
by *blast*
ultimately show *p pdvd g* $$ (*fls-subdegree g*) **by** (*simp add*: *pp-def g h*(*2*))
qed

lemma *p-depth-fls-rep-iff*:
$(f^{\circ p}) =$ *fls-subdegree g* $\longleftrightarrow$
$\neg$ *p pdvd g* $$ (*fls-subdegree g*)
if $f \neg\simeq_p 0$ **and** $f \simeq_p g$
using *that p-depth-fls-can-simplify-iff p-depth-fls-ge-p-equal-subdegree* **by** *fastforce*

end

global-interpretation *p-depth-fls*:
p-equality-depth-no-zero-divisors-1
p-equal :: *'a::nontrivial-factorial-idom prime* $\Rightarrow$
'a fls $\Rightarrow$ *'a fls* $\Rightarrow$ *bool*
p-depth :: *'a prime* $\Rightarrow$ *'a fls* $\Rightarrow$ *int*
proof (*unfold-locales*, *intro notI*)
fix *p* :: *'a prime* **and** *f g* :: *'a fls*
define *pp* **where** $pp \equiv$ *Rep-prime p*
assume *f-nequiv0*: $f \neg\simeq_p 0$
and *g-nequiv0*: $g \neg\simeq_p 0$
and *fg-equiv0*: $(f * g) \simeq_p 0$
from *f-nequiv0* **obtain** *s-f*
where *s-f*: $f \simeq_p$ *s-f fls-subdegree s-f* $= (f^{\circ p})$
using *p-depth-flsE*
by *blast*
from *g-nequiv0* **obtain** *s-g*
where *s-g*: $g \simeq_p$ *s-g fls-subdegree s-g* $= (g^{\circ p})$
using *p-depth-flsE*
by *blast*
from *s-f*(*1*) *s-g*(*1*) *fg-equiv0* **have** (*s-f* $*$ *s-g*) $\simeq_p 0$
using *p-equality-fls.mult p-equality-fls.trans0-iff* **by** *blast*
hence *fps-primevanishes p* (*fls-base-factor-to-fps* (*s-f* $*$ *s-g*)) **using** *p-equal-flsD*
by *fastforce*
with *pp-def* **have** *pp* ^ *Suc 0 dvd fls-base-parteval* (*s-f* $*$ *s-g*) *pp 0*
using *fps-pvanishesD* **by** *fast*
with *pp-def* **have** *p pdvd s-f* $$ (*fls-subdegree s-f*) $*$ *s-g* $$ (*fls-subdegree s-g*)

using *fls-base-factor-to-fps-mult*[*of s-f s-g*] **by** *fastforce*
hence
p pdvd s-f $$ (*fls-subdegree s-f*) ∨ *p pdvd s-g* $$ (*fls-subdegree s-g*)
using *Rep-prime prime-dvd-multD* **by** *auto*
with *pp-def f-nequiv0 s-f g-nequiv0 s-g* **show** *False* **by** (*metis p-depth-fls-rep-iff*)
next
fix *p* :: *'a prime*
show ∃ *x*::*'a fls*. $x \neg\simeq_p 0$ **using** *fls-1-not-p-equal-0* **by** *auto*
next
fix *p* :: *'a* :: *nontrivial-factorial-idom prime* **and** *f g* :: *'a fls*
assume *fg-nequiv0*: $(f * g) \neg\simeq_p 0$
from *fg-nequiv0*
have *f-nequiv0*: $f \neg\simeq_p 0$
and *g-nequiv0*: $g \neg\simeq_p 0$
by (*auto simp add*: *p-equality-fls.mult-0-left p-equality-fls.mult-0-right*)
from *f-nequiv0* **obtain** *s-f*
where *s-f*: $f \simeq_p$ *s-f fls-subdegree s-f* = $(f^{\circ p})$
using *p-depth-flsE*
by *blast*
from *g-nequiv0* **obtain** *s-g*
where *s-g*: $g \simeq_p$ *s-g fls-subdegree s-g* = $(g^{\circ p})$
using *p-depth-flsE*
by *blast*
from *s-f*(*1*) *s-g*(*1*) **have** *equiv*: $(f * g) \simeq_p$ (*s-f* ∗ *s-g*)
using *p-equality-fls.mult* **by** *fast*
with *fg-nequiv0* **have** *s-f* ≠ *0 s-g* ≠ *0* **by** *auto*
hence *deg*: *fls-subdegree* (*s-f* ∗ *s-g*) = *fls-subdegree s-f* + *fls-subdegree s-g* **by** *simp*
moreover have $(s\text{-}f * s\text{-}g)^{\circ p}$ = *fls-subdegree* (*s-f* ∗ *s-g*)
proof (*rule iffD2*, *rule p-depth-fls-rep-iff*)
from *fg-nequiv0* **show** (*s-f* ∗ *s-g*) $\neg\simeq_p 0$
using *equiv p-equality-fls.trans0-iff* **by** *fast*
from *s-f f-nequiv0* **have** ¬ *p pdvd s-f* $$ (*fls-subdegree s-f*)
by (*metis p-depth-fls-rep-iff*)
moreover from *s-g g-nequiv0* **have** ¬ *p pdvd s-g* $$ (*fls-subdegree s-g*)
by (*metis p-depth-fls-rep-iff*)
moreover have
(*s-f* ∗ *s-g*) $$ (*fls-subdegree* (*s-f* ∗ *s-g*)) =
s-f $$ (*fls-subdegree s-f*) ∗
s-g $$ (*fls-subdegree s-g*)
using *deg* **by** *simp*
ultimately show ¬ *p pdvd* (*s-f* ∗ *s-g*) $$ (*fls-subdegree* (*s-f* ∗ *s-g*))
using *Rep-prime prime-dvd-multD* **by** *auto*
qed *simp*
ultimately show $(f * g)^{\circ p} = (f^{\circ p}) + (g^{\circ p})$
using *s-f*(*2*) *s-g*(*2*) *equiv p-depth-fls-p-equal* **by** *metis*
next
fix *p* :: *'a prime* **and** *f* :: *'a fls*
show $(-f^{\circ p}) = (f^{\circ p})$

proof (*cases* $f \simeq_p 0$)
case *True* **thus** *?thesis* **using** *p-equality-fls.uminus*[*of p f 0*] **by** *simp*
next
case *False*
moreover from *this* **obtain** *h*
where $f \simeq_p h$
and *fls-subdegree* $h = (f^{\circ p})$
by (*elim p-depth-flsE*)
ultimately show $(- f)^{\circ p} = (f^{\circ p})$
using *p-depth-fls-rep-iff p-equality-fls.uminus*[*of p f h*] *p-equality-fls.uminus*[*of p* $-f$ *0*]
p-depth-fls-rep-iff
by *fastforce*
qed
next
fix *p* :: *'a prime* **and** *f g* :: *'a fls*
assume *f*: $f \neg\simeq_p 0$ **and** *d-f-g*: $(f^{\circ p}) < (g^{\circ p})$
show $((f + g)^{\circ p}) = (f^{\circ p})$
proof (*cases* $g \simeq_p 0$)
case *True* **thus** *?thesis* **using** *p-equality-fls.add-0-right*[*of p g f*] *p-depth-fls-p-equal* **by** *auto*
next
case *False*
from *f* **obtain** *s-f*
where *s-f*: $f \simeq_p$ *s-f fls-subdegree s-f* $= (f^{\circ p})$
using *p-depth-flsE*
by *blast*
from *False* **obtain** *s-g*
where *s-g*: $g \simeq_p$ *s-g fls-subdegree s-g* $= (g^{\circ p})$
using *p-depth-flsE*
by *blast*
from *f d-f-g s-f s-g*(*2*) **have** *deg*: *fls-subdegree* (*s-f* + *s-g*) = *fls-subdegree s-f*
using *nth-fls-subdegree-zero-iff*[*of s-f*] **by** (*auto intro*: *fls-subdegree-eqI*)
from *s-f s-g* **have** $(f + g) \simeq_p$ (*s-f* + *s-g*)
using *p-equality-fls.add* **by** *blast*
moreover from *s-f s-g*(*2*) *f d-f-g*
have $*$: ¬ *p pdvd* (*s-f* + *s-g*) $$ (*fls-subdegree* (*s-f* + *s-g*))
using *deg p-depth-fls-rep-iff*
by *fastforce*
moreover have $(f + g) \neg\simeq_p 0$
proof
define *pp* **where** *pp* ≡ *Rep-prime p*
assume $(f + g) \simeq_p 0$
with *s-f*(*1*) *s-g*(*1*) **have** *f-g*: (*s-f* + *s-g*) $\simeq_p 0$
by (*meson p-equality-fls.add p-equality-fls.trans-right*)
with *pp-def* **have** *pp* ^ *Suc 0 dvd fls-base-parteval* (*s-f* + *s-g*) *pp 0*
using *p-equal-flsD*[*of p - 0*] *fps-pvanishesD*[*of pp - 0*] **by** *fastforce*
with *pp-def* **show** *False* **using** $*$ **by** *simp*
qed

```
    ultimately have (f + g)°p = fls-subdegree (s-f + s-g)
      using p-depth-fls-rep-iff by blast
    with s-f(2) show ?thesis using deg by auto
  qed
next
  fix p :: 'a prime and f g :: 'a fls
  assume f    : f ¬≃p 0
    and  fg   : f + g ¬≃p 0
    and  d-f-g: (f°p) = (g°p)
  show (f°p) ≤ ((f + g)°p)
  proof (cases g ≃p 0)
    case True thus ?thesis
     using p-equality-fls.add-0-right[of p g f] p-depth-fls-p-equal[of p] by presburger
  next
    case False
    from f obtain s-f
      where s-f: f ≃p s-f fls-subdegree s-f = (f°p)
      using p-depth-flsE
      by    blast
    from False obtain s-g
      where s-g: g ≃p s-g fls-subdegree s-g = (g°p)
      using p-depth-flsE
      by    blast
    from s-f s-g have (f + g) ≃p (s-f + s-g)
      using p-equality-fls.add by fast
    moreover from this fg d-f-g s-f(2) s-g(2)
      have  fls-subdegree (s-f + s-g) ≥ fls-subdegree s-f
      by    (force intro: fls-subdegree-geI)
    ultimately show ?thesis
      using fg s-f(2) p-depth-fls-ge-p-equal-subdegree by fastforce
  qed
qed (simp, rule p-depth-fls-p-equal)

context
  fixes p :: 'a::nontrivial-factorial-idom prime
begin

lemma p-depth-fls1[simp]: (1::'a fls)°p = 0
  by (rule FLS-Prime-Equiv-Depth.p-depth-fls.depth-of-1)

lemma p-depth-fls-X: (fls-X::'a fls)°p = 1
  using fls-1-not-p-equal-0 p-equal-fls-shift-iff[of p fls-X::'a fls 0 1]
        p-depth-fls-shift[of p - 1]
  by    fastforce

lemma p-depth-fls-X-inv: (fls-X-inv::'a fls)°p = −1
   by (metis fls-1-not-p-equal-0 fls-X-inv-conv-shift-1 p-depth-fls-shift p-depth-fls1
diff-0)
```

lemma *p-depth-fls-X-intpow*: $((\textit{fls-X-intpow } n)::'a\ \textit{fls})^{\circ p} = n$
  using *p-depth-fls-shift*[*of p*] *fls-1-not-p-equal-0* **by** *fastforce*

lemma *p-depth-const*:
  $c^{\circ p}$ = *pmultiplicity p c* **for** *c* :: *'a*
proof (*cases c = 0*)
  define *pp* **where** *pp* ≡ *Rep-prime p*
  case *False*
  moreover from *pp-def* **have** ¬*is-unit pp* **using** *Rep-prime-not-unit* **by** *simp*
  ultimately obtain *k* **where** *k*: *c = pp ^ pmultiplicity p c ∗ k* ¬ *p pdvd k*
    unfolding *pp-def* **by** (*elim multiplicity-decompose'*)
  from *k*(*2*) **have** *k-n0*: *fls-const k* ≠ *0* **using** *fls-const-nonzero*[*of k*] **by** *blast*
  define *g* **and** *S*
    where *g* ≡ *fls-shift* (− *pmultiplicity p c*) (*fls-const k*)
    and *S* ≡ *fls-base-parteval* (*fls-const c* − *g*) *pp*
  have ⋀*n*. *pp ^ Suc n dvd S n*
  proof−
    fix *n* **show** *pp ^ Suc n dvd S n*
    proof (*cases n* < *pmultiplicity p c*)
      case *True*
      with *False pp-def S-def g-def k* **show** *?thesis*
        using *k-n0 fps-parteval-fls-base-factor-to-fps-diff*[*of fls-const c n g pp*]
            fls-const-nonzero[*of c*] *dvd-power-le*[*of pp pp Suc n multiplicity pp c*]
        by *force*
    next
      case *False*
      from *k*(*1*) *g-def* **have** *fls-subdegree* (*fls-const c* − *g*) = *0*
        by (*cases pmultiplicity p c = 0*) (*auto intro*: *fls-subdegree-eqI*)
      with *g-def False S-def k*(*1*) **show** *?thesis*
        using *fls-times-conv-regpart*[*of fls-X ^* (*pmultiplicity p c*) *fls-const k*]
            fps-parteval-mult-fps-X-power(*2*)[*of pmultiplicity p c n fps-const k*]
        by (*simp add*: *fls-X-power-times-conv-shift*(*1*) *fls-X-pow-conv-fps-X-pow*)
    qed
  qed
  with *S-def* **have** *fls-const c* $\simeq_p$ *g* **using** *pp-def p-equal-flsI fps-pvanishesI* **by** *blast*
  moreover from *g-def pp-def k*(*2*) **have** ¬ *p pdvd g* $$ (*fls-subdegree g*) **using** *k-n0* **by** *simp*
  moreover from *False* **have** *fls-const c* ¬$\simeq_p$ *0*
    using *p-equal-fls-const-0-iff* **by** *meson*
  ultimately have $c^{\circ p}$ = *fls-subdegree g*
    by (*metis p-depth-const-def p-depth-fls-rep-iff*)
  with *g-def pp-def* **show** *?thesis* **using** *k-n0* **by** *simp*
qed (*simp add*: *p-depth-const-def*)

lemma *p-depth-prime-const*: $(\textit{Rep-prime } p)^{\circ p} = 1$
  using *Rep-prime*[*of p*] *p-depth-const* **by** *simp*

end

4.5 Reduction relative to a prime

4.5.1 Of scalars

class *nontrivial-euclidean-ring-cancel* = *nontrivial-factorial-comm-ring* + *euclidean-semiring-cancel*
begin
subclass *nontrivial-factorial-idom* **..**
subclass *euclidean-ring-cancel* **..**
end

instance *int* :: *nontrivial-euclidean-ring-cancel* **..**

abbreviation *pdiv* ::
 $'a \Rightarrow 'a$::*nontrivial-euclidean-ring-cancel prime* $\Rightarrow 'a$ (**infix** *pdiv 70*)
 where *a pdiv p* $\equiv$ *a div* (*Rep-prime p*)

consts *p-modulo* :: $'a \Rightarrow 'b$ *prime* $\Rightarrow 'a$ (**infixl** *pmod 70*)

overloading *p-modulo-scalar* $\equiv$ *p-modulo* :: $'a \Rightarrow 'a$ *prime* $\Rightarrow 'a$
begin
definition *p-modulo-scalar* ::
 $'a$::*nontrivial-euclidean-ring-cancel* $\Rightarrow 'a$ *prime* $\Rightarrow 'a$
 where *p-modulo-scalar-def*[*simp*]: *p-modulo-scalar a p* $\equiv$ *a mod* (*Rep-prime p*)
end

class *nontrivial-unique-euclidean-ring* = *nontrivial-factorial-comm-ring* + *unique-euclidean-semiring*
 + **assumes** *division-segment-prime*: *prime p* $\Longrightarrow$ *division-segment p* = *1*
begin
subclass *nontrivial-euclidean-ring-cancel* **..**
end

instance *int* :: *nontrivial-unique-euclidean-ring*
proof
 fix *p*::*int* **assume** *p*: *prime p*
 hence $\neg$ *p* < *0* **using** *normalize-prime*[*of p*] **by** *force*
 thus *division-segment p* = *1* **using** *division-segment-int-def* **by** *fastforce*
qed

lemma *division-segment-1*: *division-segment* (*1*::$'a$::*unique-euclidean-semiring*) =
1
proof−
 have *is-unit* (*division-segment* (*1*::$'a$)) **by** *force*
 from *this* **obtain** *k* **where** *1* = *division-segment* (*1*::$'a$) $*$ *k* **by** *blast*
 moreover have *division-segment* (*1*::$'a$) = *division-segment 1* $*$ *division-segment*
1
  using *division-segment-mult*[*of 1 1*] **by** *force*
 ultimately show *division-segment* (*1*::$'a$) = *1*
  using *mult.assoc mult-1-right* **by** *metis*
qed

lemma *one-pdiv*: $(1::'a)$ *pdiv* $p = 0$
  and *one-pmod*: $(1::'a)$ *pmod* $p = 1$
  for p :: *'a::nontrivial-unique-euclidean-ring prime*
proof−
  define *pp* **where** $pp \equiv$ *Rep-prime p*
  have $((1::'a)$ *pdiv p*, 1 *pmod p*$) = (0, (1::'a))$
    unfolding *p-modulo-scalar-def*
  proof (*intro euclidean-relationI*)
    show *is-unit* (*Rep-prime p*) $\Longrightarrow 1 = 0 \wedge 1 = 0 *$ *Rep-prime p*
      using *Rep-prime*[*of p*] **by** *auto*
    from *pp-def* **have** *division-segment* $(1::'a) =$ *division-segment pp*
      using *Rep-prime division-segment-1 division-segment-prime*[*of pp*] **by** *force*
     moreover from *pp-def* **have** *euclidean-size* $(1::'a) <$ *euclidean-size* $(pp *$
$(1::'a))$
       using *Rep-prime-n0 Rep-prime-not-unit euclidean-size-times-nonunit*[*of pp*
$1::'a$] **by** *auto*
    ultimately show
      division-segment $(1::'a) =$ *division-segment* (*Rep-prime p*) $\wedge$
        euclidean-size $(1::'a) <$ *euclidean-size* (*Rep-prime p*) $\wedge$
        $1 = 0 *$ *Rep-prime p* $+ 1$
      using *pp-def* **by** *auto*
  qed (*simp-all add*: *Rep-prime-n0*)
  thus $(1::'a)$ *pdiv* $p = 0$ **and** $(1::'a)$ *pmod* $p = 1$ **by** *auto*
qed

4.5.2 Inductive partial reduction of formal Laurent series

This inductive function reduces one term at a time. The *nat* argument specifies how far beyond the subdegree to reduce, so that all terms to *one less* than the *nat* argument beyond the subdegree are reduced, and the term at the *nat* argument beyond the subdegree is updated to include the carry.

primrec *fls-partially-p-modulo-rep* ::
  'a::nontrivial-euclidean-ring-cancel fls $\Rightarrow$
    nat $\Rightarrow$ *'a prime* $\Rightarrow$ (*int* $\Rightarrow$ *'a*)
  where *fls-partially-p-modulo-rep f* 0 $p =$ ($$) f |
  fls-partially-p-modulo-rep f (*Suc N*) $p =$
    (*fls-partially-p-modulo-rep f N p*)(
      fls-subdegree f $+$ *int N* :=
        (*fls-partially-p-modulo-rep f N p* (*fls-subdegree f* $+$ *int N*)) *pmod p*,
      fls-subdegree f $+$ *int N* $+ 1$:=
        f $$ (*fls-subdegree f* $+$ *int N* $+ 1$) $+$
        (*fls-partially-p-modulo-rep f N p* (*fls-subdegree f* $+$ *int N*)) *pdiv p*
)

lemma *fls-partially-p-modulo-rep-zero*[*simp*]:
  fls-partially-p-modulo-rep 0 N p $n = 0$
proof−
  have $\forall n$. *fls-partially-p-modulo-rep* 0 N p $n = 0$ **by** (*induct N*) *auto*

thus *?thesis* **by** *blast*
qed

lemma *fls-partially-p-modulo-rep-zeros-below*:
fls-partially-p-modulo-rep f N p n = *0* **if** *n* < *fls-subdegree f*
using *that* **by** (*induct N*) *simp-all*

lemma *fls-partially-p-modulo-rep-func-lower-bound*:
$\forall$ *n*<*fls-subdegree f*. *fls-partially-p-modulo-rep f* (*nat* (*n* + *1* − *fls-subdegree f*)) *p n* = *0*
by *simp*

lemma *fls-partially-p-modulo-rep-agrees*:
fls-partially-p-modulo-rep f N p n = *fls-partially-p-modulo-rep f M p n*
if *n* ≤ *fls-subdegree f* + *int N* − *1* **and** *M* ≥ *N*
proof−
have ∗:
$\bigwedge$*K*. *fls-partially-p-modulo-rep f N p n* = *fls-partially-p-modulo-rep f* (*N* + *K*) *p n*
proof−
fix *K* **from** *that*(*1*) **show**
fls-partially-p-modulo-rep f N p n = *fls-partially-p-modulo-rep f* (*N* + *K*) *p n*
by (*induct K*) *auto*
qed
from *that*(*2*) **show** *?thesis* **using** ∗[*of M* − *N*] **by** *simp*
qed

lemma *fls-partially-p-modulo-rep-func-partially*:
n > *fls-subdegree f* + *int N* ⟹ *fls-partially-p-modulo-rep f N p n* = *f* $$ *n*
by (*induct N*, *simp-all*)

lemma *fls-partially-p-modulo-rep-cong*:
fls-partially-p-modulo-rep f N p n = *fls-partially-p-modulo-rep g N p n*
if *fls-subdegree f* = *fls-subdegree g* **and** $\forall$ *k*≤*n*. *f* $$ *k* = *g* $$ *k*
proof−
have
$\forall$ *k*≤*n*. *fls-partially-p-modulo-rep f N p k* = *fls-partially-p-modulo-rep g N p k*
proof (*induct N*, *safe*)
case (*Suc N*)
fix *k* **assume** *k* ≤ *n*
moreover consider
k = *fls-subdegree f* + *int N* | *k* = *fls-subdegree f* + *int N* + *1* |
k < *fls-subdegree f* + *int N* | *k* > *fls-subdegree f* + *int N* + *1*
by *fastforce*
ultimately
show *fls-partially-p-modulo-rep f* (*Suc N*) *p k* = *fls-partially-p-modulo-rep g* (*Suc N*) *p k*
by *cases* (*auto simp add*: *that Suc*)
qed (*simp add*: *that*(*2*))

thus *?thesis* **by** *auto*
qed

lemma *fls-partially-reduced-nth-vs-rep*:
$N \le n + 1 - \textit{fls-subdegree } f \Longrightarrow \textit{fls-partially-p-modulo-rep } f\ N\ p = (\$\$)\ f$
if $\forall k \le n.\ (f\ \$\$\ k)\ \textit{pdiv}\ p = 0$
proof (*induct N*)
case (*Suc N*) **thus** $\textit{fls-partially-p-modulo-rep } f\ (\textit{Suc } N)\ p = (\$\$)\ f$
using *that Suc div-mult-mod-eq*[*of f* $\$\$$ (*fls-subdegree f* + *int N*) *Rep-prime p*]
by *simp*
qed *simp*

lemma *fls-partially-p-modulo-rep-carry*:
$(\sum k=\textit{fls-subdegree } f..N.$
$(f\$\$k - \textit{fls-partially-p-modulo-rep } f\ (\textit{nat}\ (N - \textit{fls-subdegree } f))\ p\ k) *$
$(\textit{Rep-prime } p)\ \widehat{}\ \textit{nat}\ (k - \textit{fls-subdegree } f)) = 0$
proof (*cases* $N \ge$ *fls-subdegree f*, *induct N rule*: *int-ge-induct*)
case (*step N*)
define *pp* **and** $d :: {'a}\ \textit{fls} \Rightarrow \textit{int}$ **and** *fpmod-rep*
where $pp \equiv \textit{Rep-prime } p$ **and** $d \equiv \textit{fls-subdegree}$
and $\textit{fpmod-rep} \equiv (\lambda n.\ \textit{fls-partially-p-modulo-rep } f\ n\ p)$
define *S* **where**
$S \equiv$
$(\sum k=d(f)..N+1.\ (f\$\$k - \textit{fpmod-rep}\ (\textit{nat}\ (N + 1 - d(f)))\ k) * pp\ \widehat{}\ \textit{nat}\ (k$
$- d(f)))$
from *d-def step*(*1*) **have** *natN*: $\textit{nat}\ (N + 1 - d(f)) = \textit{Suc}\ (\textit{nat}\ (N - d(f)))$
by (*simp add*: *Suc-nat-eq-nat-zadd1*)
from *d-def step*(*1*) **have** $\{d(f)..N + 1\} = \textit{insert}\ (N+1)\ (\textit{insert } N\ \{d(f)..N-1\})$
by *auto*
moreover from *d-def fpmod-rep-def step*(*1*) **have**
$\forall k\le N-1.\ \textit{fpmod-rep}\ (\textit{nat}\ (N + 1 - d(f)))\ k = \textit{fpmod-rep}\ (\textit{nat}\ (N - d(f)))\ k$
using *fls-partially-p-modulo-rep-agrees*[*of - f nat* $(N - d(f))$ *nat* $(N + 1 -$
$d(f))$]
by *simp*
ultimately have
$S =$
$(f\$\$N - ((\textit{fpmod-rep}\ (\textit{nat}\ (N - d(f)))\ N\ \textit{pdiv}\ p) * pp + (\textit{fpmod-rep}\ (\textit{nat}\ (N$
$- d(f)))\ N)\ \textit{pmod}\ p))$
$* pp\ \widehat{}\ \textit{nat}\ (N - d(f)) +$
$(\sum k\in\{d(f)..N-1\}.\ (f\$\$k - \textit{fpmod-rep}\ (\textit{nat}\ (N - d(f)))\ k) * pp\ \widehat{}\ \textit{nat}\ (k -$
$d(f)))$
using *d-def fpmod-rep-def S-def step*(*1*) *natN*
by (*simp add*: *algebra-simps*)
moreover from *d-def step*(*1*) **have** *df-to-N*: $\{d(f)..N\} = \textit{insert } N\ \{d(f)..N-1\}$
by *fastforce*
ultimately have
$S = (\sum k\in\{d(f)..N\}.\ (f\$\$k - \textit{fpmod-rep}\ (\textit{nat}\ (N - d(f)))\ k) * pp\ \widehat{}\ \textit{nat}\ (k -$
$d(f)))$
using *pp-def S-def* **by** (*simp add*: *algebra-simps*)

with *pp-def d-def fpmod-rep-def S-def* **show** *?case* **using** *step*(*2*) **by** *presburger*
qed *simp-all*

4.5.3 Full reduction of formal Laurent series

Now we collect the reduced terms into a formal Laurent series.

overloading *p-modulo-fls* ≡ *p-modulo* :: *'a fls* ⇒ *'a prime* ⇒ *'a fls*
begin
definition *p-modulo-fls* ::
'a::*nontrivial-euclidean-ring-cancel fls* ⇒ *'a prime* ⇒ *'a fls*
where
p-modulo-fls f p ≡
Abs-fls (*λn. fls-partially-p-modulo-rep f* (*nat* (*n* + *1* − *fls-subdegree f*)) *p n*)
end

lemma *p-modulo-fls-nth*:
(*f pmod p*) $$ *n* = *fls-partially-p-modulo-rep f* (*nat* (*n* + *1* − *fls-subdegree f*)) *p n*
using *nth-Abs-fls-lower-bound fls-partially-p-modulo-rep-func-lower-bound*
unfolding *p-modulo-fls-def*
by *meson*

lemma *p-modulo-fls-nth'*:
int N ≥ *n* + *1* − *fls-subdegree f* ⟹
(*f pmod p*) $$ *n* = *fls-partially-p-modulo-rep f N p n*
proof (*induct N*)
case (*Suc N*) **thus** *?case*
by (*cases int* (*Suc N*) = *n* + *1* − *fls-subdegree f*, *metis p-modulo-fls-nth nat-int*, *force*)
qed (*simp add*: *p-modulo-fls-nth*)

context
fixes *p* :: *'a*::*nontrivial-euclidean-ring-cancel prime*
begin

lemma *fls-pmod-reduced*[*simp*]: ((*f pmod p*) $$ *n*) *pdiv p* = *0*
using *fls-partially-p-modulo-rep.simps*(*2*)[*of f nat* (*n* − *fls-subdegree f*) *p*]
by (
cases n fls-subdegree f − *1 rule*: *linorder-cases*,
simp-all add: *p-modulo-fls-nth Suc-nat-eq-nat-zadd1 algebra-simps*
)

lemma *fls-pmod-pmod-nth*[*simp*]: ((*f pmod p*) $$ *n*) *pmod p* = (*f pmod p*) $$ *n* **for** *f* :: *'a fls*
using *div-mult-mod-eq*[*of* (*f pmod p*) $$ *n Rep-prime p*] **by** *auto*

lemma *fls-pmod-carry*:
($\sum$ *k*=*fls-subdegree f*..*N*. (*f* − (*f pmod p*)) $$ *k* ∗ (*Rep-prime p*) ^ *nat* (*k* − *fls-subdegree f*)) =

(*fls-partially-p-modulo-rep f* (*nat* (*N* + *1* − *fls-subdegree f*)) *p* (*N* + *1*) − *f* $$ (*N* + *1*)) ∗
(*Rep-prime p*) ^ *nat* (*N* + *1* − *fls-subdegree f*)
proof (*cases N* ≥ *fls-subdegree f* − *1*)
case *True*
define *d* :: *'a fls* ⇒ *int* **and** *fpmod-rep*
where *d* ≡ *fls-subdegree*
and *fpmod-rep* ≡ (λ*n*. *fls-partially-p-modulo-rep f n p*)
from *d-def True* **have** {*d*(*f*)..*N* + *1*} = *insert* (*N*+*1*) {*d*(*f*)..*N*} **by** *auto*
moreover from *d-def fpmod-rep-def True*
have ∀ *k*≤*N*. *fpmod-rep* (*nat* (*N* + *1* − *d*(*f*))) *k* = (*f pmod p*) $$ *k*
using *p-modulo-fls-nth'*[*of - f nat* (*N* + *1* − *d*(*f*))]
by *force*
ultimately show *?thesis*
using *d-def fpmod-rep-def fls-partially-p-modulo-rep-carry*[*of f N* + *1 p*]
by (*simp add*: *algebra-simps*)
qed *simp*

lemma *fls-pmod-carry'*:
(∑ *k*=*fls-subdegree f*..*N*. (*f* − (*f pmod p*)) $$ *k* ∗ (*Rep-prime p*) ^ *nat* (*k* − *fls-subdegree f*)) =
(*fls-partially-p-modulo-rep f* (*nat* (*N* − *fls-subdegree f*)) *p N pdiv p*) ∗
(*Rep-prime p*) ^ *Suc* (*nat* (*N* − *fls-subdegree f*))
if *N* ≥ *fls-subdegree f*
using *that fls-pmod-carry*[*of f N*]
fls-partially-p-modulo-rep.simps(*2*)[*of f nat* (*N* − *fls-subdegree f*) *p*]
by (*simp add*: *algebra-simps Suc-nat-eq-nat-zadd1*)

lemma *fls-p-reduced-is-zero-iff*:
f = *0* ⟷ *f* $\simeq_p$ *0* ∧ (∀ *n*. *f*$$*n pdiv p* = *0*)
for *f* :: *'a fls*
proof (*standard*, *safe*)
assume *f-0*: *f* $\simeq_p$ *0* **and** *f-reduced*: ∀ *n*. *f*$$*n pdiv p* = *0*
define *pp* **and** *d* :: *'a fls* ⇒ *int*
where *pp* ≡ *Rep-prime p* **and** *d* ≡ *fls-subdegree*
show *f* = *0*
proof (*intro fls-eqI*)
fix *n* **have** ∀ *k*≤*n*. *f*$$*k* = *0*
proof (*cases n* ≥ *d*(*f*) − *1*, *induct n rule*: *int-ge-induct*, *safe*)
case (*step n*)
fix *k* **assume** *k*: *k* ≤ *n* + *1*
define *S* **where** *S* ≡ (∑ *k*=*d*(*f*)..*n* + *1*. *f* $$ *k* ∗ *pp* ^ *nat* (*k* − *d*(*f*)))
with *f-0 d-def pp-def step*(*1*) **have** *pp* ^ *Suc* (*nat* (*n* + *1* − *d f*)) *dvd S*
using *fls-p-equal0-extended-intsum-iff*[*of - f*] **by** *fastforce*
from *this* **obtain** *T* **where** *T*: *S* = *pp* ^ *Suc* (*nat* (*n* + *1* − *d f*)) ∗ *T* **by** (*elim dvdE*)
from *step*(*1*) **have** {*d*(*f*)..*n* + *1*} = *insert* (*n* + *1*) {*d*(*f*)..*n*} **by** *fastforce*
with *S-def pp-def step*(*2*)
have *S* = ((*f*$$(*n*+*1*) *pdiv p*) ∗ *pp* + *f*$$(*n*+*1*) *pmod p*) ∗ *pp* ^ *nat* (*n* + *1*

$-$ $d(f)$)
  by *force*
 moreover from *f-reduced* **have** $f\$\$(n+1)$ *pdiv* $p = 0$ **by** *blast*
 ultimately have $f\$\$(n + 1) = 0$
  using *pp-def T Rep-prime-n0*[*of p*] *dvd-imp-mod-0*[*of pp*] *mod-mod-trivial*[*of* $f\$\$(n + 1)$ *pp*]
  by *force*
 with *k step*(*2*) **show** f $\$\$$ $k = 0$ **by** (*cases* $k = n + 1$) *auto*
 qed (*simp-all add*: *d-def*)
 thus f $\$\$$ $n = 0$ $\$\$$ n **by** *auto*
 qed
qed *simp-all*

lemma *p-modulo-fls-eq0*[*simp*]:
 f pmod p $= 0$ **if** $f \simeq_p 0$ **for** f :: $'a$ *fls*
proof−
 define d :: $'a$ *fls* $\Rightarrow$ *int* **and** *fpmod-rep*
  where $d \equiv$ *fls-subdegree*
  and *fpmod-rep* $\equiv$ ($\lambda n.$ *fls-partially-p-modulo-rep f n p*)
 have $\forall n.$ $\forall k \le n.$ (*f pmod p*) $\$\$$ $k = 0$
 proof
  fix n **show** $\forall k \le n.$ (*f pmod p*) $\$\$$ $k = 0$
  proof (*cases* $n \ge d(f)$, *induct n rule*: *int-ge-induct*, *safe*)
   case *base* **fix** k **assume** k: $k \le d(f)$
   show (*f pmod p*) $\$\$$ $k = 0$
   proof (*cases* $k = d(f)$)
    case *True*
    with *that d-def* **show** *?thesis*
     using *fls-p-equal0-extended-intsum-iff*[*of - f*] *p-modulo-fls-nth*[*of f*] **by** *fastforce*
   next
    case *False* **with** *k d-def* **show** *?thesis* **by** (*simp add*: *p-modulo-fls-nth*)
   qed
  next
   case (*step n*)
   fix k **assume** k: $k \le n + 1$
   define *pp* **where** *pp* $\equiv$ *Rep-prime p*
   define S **where** $S \equiv (\sum k{=}d(f)..n + 1.\ f\ \$\$\ k * pp\ \hat{}\ nat\ (k - d(f)))$
   with *that d-def pp-def step*(*1*) **have** $pp\ \hat{}\ Suc\ (nat\ (n + 1 - d\ f))$ *dvd* S
    using *fls-p-equal0-extended-intsum-iff*[*of - f*] **by** *fastforce*
   from *this* **obtain** T **where** T: $S = pp\ \hat{}\ Suc\ (nat\ (n + 1 - d\ f)) * T$ **by** (*elim dvdE*)
   moreover from *step*(*1*) **have** $\{d(f)..n + 1\} =$ *insert* $(n + 1)$ $\{d(f)..n\}$ **by** *fastforce*
   ultimately have
    (*f pmod p*) $\$\$$ $(n+1) * pp\ \hat{}\ nat\ (n + 1 - d(f)) =$
     $(T -$ *fpmod-rep* $(nat\ (n + 1 - d(f)))\ (n + 1)$ *pdiv* $p) * pp * pp\ \hat{}\ nat\ (n + 1 - d(f))$)
    using *d-def fpmod-rep-def pp-def S-def step fls-pmod-carry'*[*of f n* $+$ *1*]

by (*simp add*: *algebra-simps*)
with *pp-def* **have** (*f pmod p*) $$ (*n*+*1*) = *0*
using *Rep-prime-n0*[*of p*] *dvd-mult-cancel-right fls-pmod-reduced*[*of f n* + *1*]
by *simp*
with *k step*(*2*) **show** (*f pmod p*) $$ *k* = *0* **by** (*cases k* = *n* + *1*) *auto*
qed (*simp add*: *d-def p-modulo-fls-nth*)
qed
thus *f pmod p* = *0* **using** *fls-eqI*[*of f pmod p 0*] **by** *auto*
qed

lemma *fls-pmod-eq0-iff*:
f pmod p = *0* ⟷ *f* $\simeq_p$ *0* **for** *f* :: *'a fls*
proof (*standard*, *rule iffD2*, *rule fls-p-equal0-extended-intsum-iff*, *safe*)
define *pp* **and** *d* :: *'a fls* ⇒ *int*
where *pp* ≡ *Rep-prime p* **and** *d* ≡ *fls-subdegree*
show *d*(*f*) ≤ *d*(*f*) **by** *blast*
fix *n* **assume** *f pmod p* = *0* **and** *n* ≥ *d*(*f*)
with *pp-def d-def*
show *pp* ^ *Suc* (*nat* (*n* − *d*(*f*))) *dvd* (∑*k*=*d*(*f*)..*n*. *f* $$ *k* ∗ *pp* ^ *nat* (*k* −
d(*f*)))
using *fls-pmod-carry'*
by *force*
qed *simp*

lemma *fls-pmod-equiv*: *f* $\simeq_p$ *f pmod p* **for** *f* :: *'a fls*
proof (*cases f pmod p* = *0*)
case *True* **thus** *?thesis* **using** *fls-pmod-eq0-iff* **by** *simp*
next
case *False* **show** *?thesis*
proof (*rule iffD2*, *rule p-equal-fls-extended-intsum-iff*, *safe*)
define *pp* **and** *d* :: *'a fls* ⇒ *int*
where *pp* ≡ *Rep-prime p* **and** *d* ≡ *fls-subdegree*
from *False d-def* **have** *d*(*f*) ≤ *d*(*f pmod p*)
by (*auto intro*: *fls-subdegree-geI simp add*: *p-modulo-fls-nth*)
thus *d*(*f*) ≤ *min* (*d*(*f*)) (*d*(*f pmod p*)) **by** *fastforce*
fix *n* **assume** *n* ≥ *d*(*f*)
with *d-def pp-def* **show**
pp ^ *Suc* (*nat* (*n* − *d*(*f*))) *dvd* (∑*k*=*d*(*f*)..*n*. (*f* − *f pmod p*) $$ *k* ∗ *pp* ^ *nat*
(*k* − *d*(*f*)))
using *fls-pmod-carry'*[*of f*] **by** *auto*
qed
qed

lemma *fls-p-reduced-subdegree-unique*:
fls-subdegree g = ($f^{\circ p}$)
if *f* $\simeq_p$ *g* **and** ∀ *n*. (*g* $$ *n*) *pdiv p* = *0*
for *f g* :: *'a fls*
proof (*cases f* $\simeq_p$ *0*)
case *True*

with *that* **have** $g = 0$ **using** *fls-p-reduced-is-zero-iff p-equality-fls.trans-right* **by** *blast*
with *True* **show** *?thesis* **by** (*simp add*: *p-depth-fls-def*)
next
case *False* **show** *?thesis*
proof (*intro p-depth-fls-eqI*[*symmetric*], *rule False*, *rule that*(*1*))
fix *h* **assume** *h*: $f \simeq_p h$
from *that*(*1*) *h*
have *fls-subdegree h* > *fls-subdegree g* $\Longrightarrow$ *p pdvd g* $$ (*fls-subdegree g*)
using *fls-pmod-equiv p-equality-fls.trans-right*[*of p f*]
p-equal-fls-extended-intsum-iff [*of fls-subdegree g g h*]
by *fastforce*
moreover from *that*(*2*) **have** (*g* $$ (*fls-subdegree g*)) *pdiv p* = *0* **by** *blast*
ultimately show *fls-subdegree h* ≤ *fls-subdegree g*
using *False that*(*1*) *nth-fls-subdegree-nonzero*[*of g*] **by** *fastforce*
qed
qed

lemma *fls-pmod-subdegree*:
fls-subdegree (*f pmod p*) = ($f^{\circ p}$) **for** *f* :: *'a fls*
using *fls-p-reduced-subdegree-unique fls-pmod-reduced fls-pmod-equiv* **by** *blast*

lemma *fls-pmod-depth*:
(*f pmod p*)$^{\circ p}$ = ($f^{\circ p}$) **for** *f* :: *'a fls*
by (*metis p-depth-fls.depth-equiv fls-pmod-equiv*)

lemma *fls-pmod-subdegree-ge*:
fls-subdegree (*f pmod p*) ≥ *fls-subdegree f* **if** $f \neg\simeq_p 0$
for *f* :: *'a fls*
using *that fls-pmod-subdegree p-depth-fls-ge-p-equal-subdegree'* **by** *auto*

lemma *fls-pmod-nth-below-subdegree*:
(*f pmod p*) $$ *n* = *0* **if** *n* < *fls-subdegree f* **for** *f* :: *'a fls*
using *that fls-pmod-subdegree-ge*[*of f*] **by** *fastforce*

lemma *fls-pmod-nth-cong*:
(*f pmod p*) $$ *n* = (*g pmod p*) $$ *n* **if** ∀ *k*≤*n*. *f* $$ *k* = *g* $$ *k* **for** *f g* :: *'a fls*
proof−
consider
f = *0 g* = *0* | (*f-n0*) *f* ≠ *0 g* = *0* | (*g-n0*) *f* = *0 g* ≠ *0* |
(*f-n0-upper*) *f* ≠ *0 n* ≥ *fls-subdegree f* |
(*g-n0-lower*) *g* ≠ *0 n* < *fls-subdegree f*
by *linarith*
thus *?thesis*
proof *cases*
case *f-n0* **with** *that* **show** *?thesis*
using *fls-subdegree-geI*[*of f n* + *1*] **by** (*simp add*: *fls-pmod-nth-below-subdegree*)
next
case *g-n0* **with** *that* **show** *?thesis*

using *fls-subdegree-geI*[*of g n* + *1*] **by** (*simp add*: *fls-pmod-nth-below-subdegree*)
next
case *g-n0-lower* **with** *that* **show** *?thesis*
using *fls-subdegree-geI*[*of g n* + *1*] **by** (*simp add*: *fls-pmod-nth-below-subdegree*)
next
case *f-n0-upper*
from *f-n0-upper*(*2*) **have** $\bigwedge k.\ k <$ *fls-subdegree f* $\Longrightarrow k \leq n$ **by** *simp*
with *that* **have** $\bigwedge k.\ k <$ *fls-subdegree f* $\Longrightarrow g$ \$\$ $k = 0$ **by** *fastforce*
with *that f-n0-upper* **show** *?thesis*
using *nth-fls-subdegree-nonzero fls-subdegree-eqI p-modulo-fls-nth fls-partially-p-modulo-rep-cong*
by *metis*
qed *simp*
qed

lemma *fls-partially-reduced-nth-vs-pmod*:
(*f pmod p*) \$\$ $n = f$ \$\$ n **if** $\forall k \leq n$. (*f* \$\$ *k*) *pdiv p* = *0*
proof (*cases* $n \geq$ *fls-subdegree f*)
case *True* **with** *that* **show** *?thesis*
by (*simp add*: *p-modulo-fls-nth fls-partially-reduced-nth-vs-rep*)
qed (*force simp add*: *fls-pmod-nth-below-subdegree*)

lemma *fls-pmod-eqI*:
f pmod p = *g* **if** $f \simeq_p g$ **and** $\forall n$. (*g* \$\$ *n*) *pdiv p* = *0*
proof (*intro fls-eq-above-subdegreeI*, *safe*)
define *d* :: *'a fls* $\Rightarrow$ *int* **where** $d \equiv$ *fls-subdegree*
with *that* **have** $*$: *d*(*f pmod p*) $\geq$ ($f^{\circ p}$) *d*(*g*) $\geq$ ($f^{\circ p}$)
using *fls-pmod-subdegree fls-p-reduced-subdegree-unique* **by** (*simp*, *presburger*)
thus *d*(*f pmod p*) $\geq$ ($f^{\circ p}$) $-$ *1* *d*(*g*) $\geq$ ($f^{\circ p}$) $-$ *1* **by** *auto*
fix *n* **assume** *n*: $n \geq (f^{\circ p}) - 1$
hence $\forall k \leq n$. (*f pmod p*) \$\$ *k* = *g* \$\$ *k*
proof (*induct n rule*: *int-ge-induct*)
case *base* **from** *that* **show** *?case* **using** *fls-pmod-subdegree fls-p-reduced-subdegree-unique* **by** *simp*
next
case (*step n*) **show** *?case*
proof *clarify*
define *pp* **where** $pp \equiv$ *Rep-prime p*
fix *k* **assume** *k*: $k \leq n + 1$
define *S* **where**
$S \equiv (\sum k{=}(f^{\circ p})..n + 1.$
(*f pmod p* $-$ *g*) \$\$ $k * pp$ ^ *nat* ($k - (f^{\circ p})$)))
define *A*
where $A \equiv$ (*if* $n + 1 = (f^{\circ p})$ *then* {} *else* {($f^{\circ p}$)..*n*})
with *step*(*1*) **have** {($f^{\circ p}$)..*n* + *1*} = *insert* (*n* + *1*) *A* **by** *force*
moreover from *A-def step*(*2*) **have** $\forall k \in A$. (*f pmod p* $-$ *g*) \$\$ *k* = *0* **by** *simp*
ultimately have
S = (*f pmod p* $-$ *g*) \$\$ (*n* + *1*) $*$ *pp* ^ *nat* (*n* + *1* $-$ ($f^{\circ p}$))

using *S-def A-def* **by** *force*
moreover from *step*(*1*) *that*(*1*) *d-def pp-def S-def*
have $pp \,\hat{}\, Suc\ (nat\ (n + 1 - (f^{\circ p})))\ dvd\ S$
using $*$ *p-equality-fls.trans-right fls-pmod-equiv*
p-equal-fls-extended-intsum-iff[*of* $(f^{\circ p})$ *f pmod p g*]
by *force*
ultimately have $p\ pdvd\ ((f\ pmod\ p) - g)\ \$\$\ (n + 1)$ **using** *pp-def Rep-prime-n0* **by** *auto*
with *pp-def that*(*2*)
have $(f\ pmod\ p)\ \$\$\ (n + 1) = ((g\$\$(n{+}1))\ pdiv\ p) * pp + (g\$\$(n{+}1))\ pmod\ p$
using *mod-eq-dvd-iff fls-pmod-reduced div-mult-mod-eq*[*of* $(f\ pmod\ p)\ \$\$\ (n + 1)\ pp$]
by *force*
with *pp-def k* **show** $(f\ pmod\ p)\ \$\$\ k = g\ \$\$\ k$
using *step*(*2*) **by** (*cases* $k = n + 1$) (*auto simp add*: *div-mult-mod-eq*)
qed
qed
thus $(f\ pmod\ p)\ \$\$\ n = g\ \$\$\ n$ **by** *fast*
qed

lemma *fls-pmod-eq-pmod-iff*:
$f\ pmod\ p = g\ pmod\ p \longleftrightarrow f \simeq_p g$ **for** $f\ g :: 'a\ fls$
proof
show $f\ pmod\ p = g\ pmod\ p \Longrightarrow f \simeq_p g$
by (*metis fls-pmod-equiv p-equality-fls.refl p-equality-fls.cong*)
show $f \simeq_p g \Longrightarrow f\ pmod\ p = g\ pmod\ p$
using *fls-pmod-equiv p-equality-fls.trans fls-pmod-reduced fls-pmod-eqI* **by** *meson*
qed

lemma *fls-pmod-equiv-pmod-iff*:
$(f\ pmod\ p) \simeq_p (g\ pmod\ p) \longleftrightarrow f \simeq_p g$
for $f\ g :: 'a\ fls$
using *fls-pmod-equiv p-equality-fls.cong p-equality-fls.sym* **by** *meson*

lemma *fls-p-modulo-iff*:
$f\ pmod\ p = g \longleftrightarrow f \simeq_p g \wedge (\forall n.\ (g\ \$\$\ n)\ pdiv\ p = 0)$
using *fls-pmod-equiv fls-pmod-reduced fls-pmod-eqI* **by** *blast*

lemma *fls-pmod-pmod*[*simp*]: $(f\ pmod\ p)\ pmod\ p = f\ pmod\ p$ **for** $f :: 'a\ fls$
by (*metis fls-pmod-equiv p-equal-fls-trans fls-pmod-reduced fls-pmod-eqI*)

lemma *fls-pmod-cong*:
$f\ pmod\ p = g\ pmod\ p$ **if** $f \simeq_p g$ **for** $f\ g :: 'a\ fls$
using *that fls-pmod-equiv p-equal-fls-trans fls-pmod-reduced fls-pmod-eqI* **by** *meson*

end

4.5.4 Algebraic properties of reduction

lemma *fls-pmod-1* [*simp*]:
  $(1::'a\ fls)$ *pmod* $p = 1$ **for** p :: $'a$::*nontrivial-unique-euclidean-ring prime*
  by (*intro fls-pmod-eqI*, *simp-all add*: *one-pdiv*)

context
  fixes p :: $'a$::*nontrivial-euclidean-ring-cancel prime*
begin

lemma *fls-pmod-0*: $(0::'a\ fls)$ *pmod* $p = 0$
  by *simp*

lemma *fls-pmod-uminus-equiv*: $(-f)$ *pmod* $p \simeq_p -(f$ *pmod* $p)$ **for** f :: $'a$ *fls*
  using *fls-pmod-equiv p-equality-fls.uminus p-equality-fls.trans-right* **by** *blast*

lemma *fls-pmod-equiv-add*:
  $(f + g)$ *pmod* $p = (f' + g')$ *pmod* p **if** $f \simeq_p f'$ **and** $g \simeq_p g'$
  for $f\ f'\ g\ g'$ :: $'a$ *fls*
  using *that p-equality-fls.add fls-pmod-equiv p-equal-fls-trans fls-pmod-reduced fls-pmod-eqI*
  by *meson*

lemma *fls-pmod-const-add*:
  (*fls-const* $c + f$) *pmod* p = *fls-const* $c + f$ *pmod* p **if** *fls-subdegree* $f > 0$ **and** c *pdiv* $p = 0$
  for c :: $'a$ **and** f :: $'a$ *fls*
proof (*intro fls-pmod-eqI*, *safe*)
  fix n **from** *that* **show** (*fls-const* $c + f$ *pmod* p) \$\$ n *pdiv* $p = 0$
   by (*cases n 0::int rule*: *linorder-cases*, *simp-all add*: *fls-pmod-nth-below-subdegree*)
qed (*simp add*: *fls-pmod-equiv p-equality-fls.add*)

lemma *fls-pmod-add-equiv*:
  $(f + g)$ *pmod* $p \simeq_p (f$ *pmod* $p) + (g$ *pmod* $p)$ **for** $f\ g$ :: $'a$ *fls*
  by (*metis fls-pmod-equiv fls-pmod-equiv-add p-equality-fls.trans-left*)

lemma *fls-pmod-mismatched-add-nth*:
  $n <$ *fls-subdegree* $g \Longrightarrow ((f + g)$ *pmod* $p)$ \$\$ $n = (f$ *pmod* $p)$ \$\$ n
  $((f + g)$ *pmod* $p)$ \$\$ (*fls-subdegree* g) =
    $((f$ *pmod* $p)$ \$\$ (*fls-subdegree* g) + g \$\$ (*fls-subdegree* g)) *pmod* p
  if *fls-subdegree* $g >$ *fls-subdegree* f
  for f :: $'a$ *fls*
proof−
  assume n: $n <$ *fls-subdegree* g
  show $((f + g)$ *pmod* $p)$ \$\$ $n = (f$ *pmod* $p)$ \$\$ n
  proof (*cases* $f = 0$)
    case *True* **with** n **show** *?thesis* **using** *fls-pmod-nth-below-subdegree* **by** *auto*
  next
    case *False*
   with *that* **have** *fls-subdegree* $(f + g)$ = *fls-subdegree* f **by** (*simp add*: *fls-subdegree-eqI*)
    with n **show** *?thesis*

using *fls-partially-p-modulo-rep-cong*[*of f* + *g*] **by** (*simp add*: *p-modulo-fls-nth*)
qed
next
define *pp* **and** *d* :: *'a fls* ⇒ *int* **and** *pmod-rep*
where *pp* ≡ *Rep-prime p* **and** *d* ≡ *fls-subdegree*
and *pmod-rep* ≡ (λ*f N*. *fls-partially-p-modulo-rep f N p*)
show
((*f* + *g*) *pmod p*) $$ (*fls-subdegree g*) =
((*f pmod p*) $$ (*fls-subdegree g*) + *g* $$ (*fls-subdegree g*)) *pmod p*
proof (*cases f* = *0*)
case *False*
with *that d-def* **have** *dfg*: *d*(*f* + *g*) = *d*(*f*) **by** (*simp add*: *fls-subdegree-eqI*)
with *that d-def pmod-rep-def pp-def* **show** *?thesis*
using *fls-partially-p-modulo-rep.simps*(*2*)[*of f* + *g nat* (*d*(*g*) − *d*(*f*))]
fls-partially-p-modulo-rep.simps(*2*)[*of f* + *g nat* (*d*(*g*) − *d*(*f*) − *1*)]
fls-partially-p-modulo-rep.simps(*2*)[*of f nat* (*d*(*g*) − *d*(*f*) − *1*)]
fls-partially-p-modulo-rep-cong[*of f* + *g f d*(*g*) − *1 nat* (*d*(*g*) − *d*(*f*) − *1*)]
fls-partially-p-modulo-rep.simps(*2*)[*of f nat* (*d*(*g*) − *d*(*f*))]
by (*simp add*: *p-modulo-fls-nth Suc-nat-eq-nat-zadd1 algebra-simps mod-simps*)
qed (*simp add*: *d-def pmod-rep-def p-modulo-fls-nth*)
qed

lemma *fls-pmod-equiv-minus*:
(*f* − *g*) *pmod p* = (*f'* − *g'*) *pmod p* **if** $f \simeq_p f'$ **and** $g \simeq_p g'$
for *f f' g g'* :: *'a fls*
using *that p-equality-fls.minus fls-pmod-equiv p-equal-fls-trans fls-pmod-reduced fls-pmod-eqI*
by *meson*

lemma *fls-pmod-minus-equiv*:
(*f* − *g*) *pmod p* $\simeq_p$ ((*f pmod p*) − (*g pmod p*)) **for** *f g* :: *'a fls*
by (*metis fls-pmod-equiv fls-pmod-equiv-minus p-equality-fls.trans-left*)

lemma *fls-pmod-equiv-times*:
(*f* ∗ *g*) *pmod p* = (*f'* ∗ *g'*) *pmod p* **if** $f \simeq_p f'$ **and** $g \simeq_p g'$
for *f f' g g'* :: *'a fls*
using *that p-equality-fls.mult fls-pmod-equiv p-equal-fls-trans fls-pmod-reduced fls-pmod-eqI*
by *meson*

lemma *fls-pmod-times-equiv*:
(*f* ∗ *g*) *pmod p* $\simeq_p$ (*f pmod p*) ∗ (*g pmod p*) **for** *f g* :: *'a fls*
by (*metis fls-pmod-equiv fls-pmod-equiv-times p-equality-fls.trans-left*)

lemma *fls-pmod-diff-cancel*:
(*f pmod p*) $$ *n* = (*g pmod p*) $$ *n* **if** *n* < $((f - g)^{\circ p})$
for *f g* :: *'a fls*
proof−

consider
$f \simeq_p 0$ | $g \simeq_p 0$ |
(*nonzero*) $f \neg\simeq_p 0$ $g \neg\simeq_p 0$
by *fast*
thus *?thesis*
using *that*
proof *cases*
case *nonzero*
hence $n < ((f - g)^{\circ p}) \Longrightarrow (f\ pmod\ p)\ \$\$\ n = (g\ pmod\ p)\ \$\$\ n$
proof (*induct f g rule*: *linorder-wlog'*[*of* $\lambda f.\ f^{\circ p}$])
case (*le f g*)
have $*$:
$\bigwedge n.\ n < ((f - g)^{\circ p}) \Longrightarrow$
$((f\ pmod\ p - g\ pmod\ p)\ pmod\ p)\ \$\$\ n = 0$
using *p-depth-fls.depth-diff-equiv fls-pmod-equiv fls-pmod-subdegree fls-eq0-below-subdegree*
by *metis*
from *le*(*2*) **consider**
(*diff-le*) $((f - g)^{\circ p}) \le (f^{\circ p})$ |
(*diff-gt*) $((f - g)^{\circ p}) > (g^{\circ p})$ |
(*diff-betw*)
$(f^{\circ p}) < ((f - g)^{\circ p})$
$((f - g)^{\circ p}) \le (g^{\circ p})$
by *linarith*
thus $(f\ pmod\ p)\ \$\$\ n = (g\ pmod\ p)\ \$\$\ n$
proof *cases*
case *diff-le* **with** *le*(*1*,*2*) **show** *?thesis* **by** (*simp add*: *fls-pmod-subdegree*)
next
case *diff-betw*
from *le*(*2*) *diff-betw*(*2*) **have** $\forall k \le n.\ ((f\ pmod\ p - g\ pmod\ p)\ \$\$\ k)\ pdiv\ p$
$= 0$
by (*auto simp add*: *fls-pmod-subdegree*)
with *le*(*2*) **have** $(f\ pmod\ p - g\ pmod\ p)\ \$\$\ n = 0$
using $*$ *fls-partially-reduced-nth-vs-pmod* **by** *metis*
thus *?thesis* **by** *auto*
next
case *diff-gt*
have $\neg\ (f^{\circ p}) < (g^{\circ p})$
proof
assume *lt*: $(f^{\circ p}) < (g^{\circ p})$
moreover from *this*
have $\forall k \le (f^{\circ p}).\ ((f\ pmod\ p - g\ pmod\ p)\ \$\$\ k)\ pdiv\ p = 0$
by (*auto simp add*: *fls-pmod-subdegree*)
ultimately have $(f\ pmod\ p - g\ pmod\ p)\ \$\$\ (f^{\circ p}) = 0$
using *diff-gt* $*$ *fls-partially-reduced-nth-vs-pmod*[*of* - *f pmod p* $-$ *g pmod*
p]
by *fastforce*
with *le*(*3*) *lt* **show** *False*
using *fls-pmod-subdegree*[*of* - *p*] *fls-pmod-eq0-iff nth-fls-subdegree-nonzero*
by *force*

qed
with *le*(*1*) **have** *df-dg*: $(f^{\circ p}) = (g^{\circ p})$ **by** *linarith*
define *pp* **where** *pp* ≡ *Rep-prime p*
have
$n < ((f - g)^{\circ p}) \Longrightarrow$
$\forall k \le n.$ (*f pmod p*) $$ *k* = (*g pmod p*) $$ *k*
proof (*cases* $n \ge (f^{\circ p})$)
case *True*
have *strong*:
$\bigwedge N.\ N < ((f - g)^{\circ p}) \Longrightarrow$
$\forall k<N.$ (*f pmod p*) $$ *k* = (*g pmod p*) $$ *k* $\Longrightarrow$
¬ (*f pmod p*) $$ *N* ≠ (*g pmod p*) $$ *N*
proof
fix *N*
assume *N* : $N < ((f - g)^{\circ p})$
$\forall k<N.$ (*f pmod p*) $$ *k* = (*g pmod p*) $$ *k*
and *neq*: (*f pmod p*) $$ *N* ≠ (*g pmod p*) $$ *N*
hence (*f pmod p* − *g pmod p*) $$ *N* ≠ *0* **by** *simp*
with *N*(*2*) **have** *fls-subdegree* (*f pmod p* − *g pmod p*) = *N*
using *fls-subdegree-eqI*[*of f pmod p* − *g pmod p N*] **by** *auto*
hence ((*f pmod p* − *g pmod p*) *pmod p*) $$ *N* = ((*f pmod p* − *g pmod p*) $$ *N*) *pmod p*
by (*simp add*: *p-modulo-fls-nth*)
with *N*(*1*) **have** ((*f pmod p* − *g pmod p*) $$ *N*) *pmod p* = *0*
using ∗ **by** *metis*
hence ((*f pmod p*) $$ *N*) *pmod p* = ((*g pmod p*) $$ *N*) *pmod p*
using *mod-eq-dvd-iff* **by** *auto*
with *neq* **show** *False* **using** *fls-pmod-pmod-nth*[*of f*] *fls-pmod-pmod-nth*[*of g*] **by** *metis*
qed
from *True* **show**
$n < ((f - g)^{\circ p}) \Longrightarrow$
$\forall k \le n.$ (*f pmod p*) $$ *k* = (*g pmod p*) $$ *k*
proof (*induct n rule*: *int-ge-induct*, *safe*)
case *base*
fix *k*
from *diff-gt* **have** $(f^{\circ p}) < (f - g^{\circ p})$ **using** *df-dg* **by** *auto*
moreover have *k-lt*: $\forall k<f^{\circ p}.$ (*f pmod p*) $$ *k* = (*g pmod p*) $$ *k*
using *df-dg* **by** (*auto simp add*: *fls-pmod-subdegree*)
ultimately have
¬ (*f pmod p*) $$ $(f^{\circ p})$ ≠
(*g pmod p*) $$ $(f^{\circ p})$
$k < (f^{\circ p}) \Longrightarrow$ (*f pmod p*) $$ *k* = (*g pmod p*) $$ *k*
using *strong* **by** (*presburger*, *blast*)
moreover assume $k \le (f^{\circ p})$
ultimately show (*f pmod p*) $$ *k* = (*g pmod p*) $$ *k* **by** *fastforce*
next
case (*step n*)
fix *k*

from *step*(*2*,*3*) **have** $\neg$ (*f pmod p*) $$ (*n* + *1*) $\neq$ (*g pmod p*) $$ (*n* + *1*)
  using *strong* **by** *auto*
 hence *k* = *n* + *1* $\Longrightarrow$ (*f pmod p*) $$ *k* = (*g pmod p*) $$ *k* **by** *fastforce*
 moreover from *step*(*2*,*3*)
  have *k* $\leq$ *n* $\Longrightarrow$ (*f pmod p*) $$ *k* = (*g pmod p*) $$ *k*
  by *auto*
 moreover assume *k* $\leq$ *n* + *1*
 ultimately show (*f pmod p*) $$ *k* = (*g pmod p*) $$ *k* **by** *fastforce*
qed
next
 case *False* **thus** $\forall$ *k*$\leq$*n*. (*f pmod p*) $$ *k* = (*g pmod p*) $$ *k*
  using *df-dg* **by** (*auto simp add*: *fls-pmod-subdegree*)
qed
with *le*(*2*) **show** *?thesis* **by** *fast*
qed
qed (*simp add*: *p-depth-fls.depth-diff*)
with *that* **show** *?thesis* **by** *blast*
qed (
fastforce simp add: *p-depth-fls.depth-diff-equiv0-left fls-pmod-eq0-iff fls-pmod-subdegree*,
fastforce simp add: *p-depth-fls.depth-diff-equiv0-right fls-pmod-eq0-iff fls-pmod-subdegree*
)
qed

lemma *fls-pmod-diff-cancel-iff*:
((*f* − *g*)$^{\circ p}$) > *n* $\longleftrightarrow$
 ($\forall$ *k*$\leq$*n*. (*f pmod p*) $$ *k* = (*g pmod p*) $$ *k*)
if *f* $\neg\simeq_p$ *g*
for *f g* :: *'a fls*
proof
assume $\forall$ *k*$\leq$*n*. (*f pmod p*) $$ *k* = (*g pmod p*) $$ *k*
moreover from *that* **have** *f pmod p* $\neq$ *g pmod p* **using** *fls-pmod-eq-pmod-iff* **by** *auto*
ultimately have *fls-subdegree* (*f pmod p* − *g pmod p*) > *n*
 using *fls-subdegree-greaterI*[*of f pmod p* − *g pmod p*] **by** *auto*
with *that* **have** *fls-subdegree* ((*f pmod p* − *g pmod p*) *pmod p*) > *n*
 using *fls-pmod-subdegree-ge*[*of p f pmod p* − *g pmod p*] *fls-pmod-equiv-pmod-iff*
  p-equality-fls.conv-0
 by *fastforce*
thus ((*f* − *g*)$^{\circ p}$) > *n*
 using *fls-pmod-subdegree fls-pmod-equiv p-depth-fls.depth-diff-equiv* **by** *metis*
qed (*simp add*: *fls-pmod-diff-cancel*)

end

4.6 Inversion relative to a prime

4.6.1 Of scalars

class *bezout-semiring* = *semiring-gcd* +
assumes *bezout*: $\exists$ *u v*. *u* ∗ *x* + *v* ∗ *y* = *gcd x y*

class *bezout-ring* = *ring-gcd* +
  assumes *bezout*: $\exists$ *u v*. *u* $*$ *x* + *v* $*$ *y* = *gcd x y*
begin
subclass *bezout-semiring* **using** *bezout* **by** *unfold-locales*
subclass *idom* **..**
end

instance *int* :: *bezout-ring* **by** (*standard*, *rule bezout-int*)

class *nontrivial-factorial-euclidean-bezout* = *nontrivial-euclidean-ring-cancel* + *bezout-semiring*
begin
subclass *bezout-ring* **by** (*standard*, *rule bezout*)
end

class *nontrivial-factorial-unique-euclidean-bezout* =
  nontrivial-unique-euclidean-ring + *bezout-semiring*
begin
subclass *nontrivial-factorial-euclidean-bezout* **..**
end

instance *int* :: *nontrivial-factorial-unique-euclidean-bezout* **..**

lemma *prime-residue-inverse-ex1*:
  $\exists$!*x*. *x pdiv p* = *0* $\wedge$ (*x* $*$ *a*) *pmod p* = *1* **if** $\neg$ *p pdvd a*
  for *a* :: *'a::nontrivial-factorial-unique-euclidean-bezout* **and** *p* :: *'a prime*
proof (*intro ex-ex1I*, *safe*)
  define *pp* **where** *pp* $\equiv$ *Rep-prime p*
  with *that* **obtain** *u v* **where** *uv*: *u* $*$ *pp* + *v* $*$ *a* = *1*
   using *Rep-prime*[*of p*] *prime-imp-coprime bezout*[*of pp a*] *coprime-iff-gcd-eq-1*[*of pp a*]
    by *auto*
  define *y* **where** *y* $\equiv$ *v pmod p*
  with *pp-def* **have** (*y* $*$ *a*) *pmod p* = *1*
    by (*metis*
     uv p-modulo-scalar-def mod-add-left-eq mod-mult-self2-is-0 add-0-left mod-mult-left-eq
      one-pmod
)
  moreover from *y-def* **have** *y pdiv p* = *0* **by** *auto*
  ultimately show $\exists$ *x*. *x pdiv p* = *0* $\wedge$ (*x* $*$ *a*) *pmod p* = *1* **by** *fast*

  fix *x x'*
  assume *x* : *x pdiv p* = *0* *x* $*$ *a pmod p* = *1*
    and *x'*: *x' pdiv p* = *0* *x'* $*$ *a pmod p* = *1*
  from *pp-def x*(*1*) **have** *x pmod p* = *x* **using** *div-mult-mod-eq*[*of x pp*] **by** *auto*
  with *x'*(*2*) **have** *x* = ((*x pmod p*) $*$ ((*x'* $*$ *a*) *pmod p*)) *pmod p* **by** *auto*
  also from *pp-def x*(*2*) **have** ... = (*x'* $*$ *1*) *pmod p*
    by (*metis p-modulo-scalar-def mult.assoc mult.commute mod-mult-right-eq*)

finally show $x = x'$ **using** *pp-def* $x'(1)$ *div-mult-mod-eq*[*of* x' *pp*] **by** *simp*
qed

consts *pinverse* :: $'a \Rightarrow 'b$ *prime* $\Rightarrow 'a$
$((\text{-}^{-1}\text{-})\ [51,\ 51]\ 50)$

overloading *pinverse-scalar* $\equiv$ *pinverse* :: $'a \Rightarrow 'a$ *prime* $\Rightarrow 'a$
begin
definition *pinverse-scalar* ::
$'a$::*nontrivial-factorial-unique-euclidean-bezout* $\Rightarrow 'a$ *prime* $\Rightarrow 'a$
where
pinverse-scalar a p $\equiv$
(*if p pdvd a then 0 else*
(*THE x. x pdiv p* $= 0 \wedge (x * a)$ *pmod p* $= 1$))
end

context
fixes p :: $'a$::*nontrivial-factorial-unique-euclidean-bezout prime*
begin

lemma *pinverse-scalar*: $((a^{-1p}) * a)$ *pmod p* $= 1$ **if** $\neg$ *p pdvd a* **for** a :: $'a$
using *that theI'*[*OF prime-residue-inverse-ex1*] **unfolding** *pinverse-scalar-def* **by** *auto*

lemma *pinverse-scalar-reduced*: (a^{-1p}) *pdiv p* $= 0$ **for** a :: $'a$
using *theI'*[*OF prime-residue-inverse-ex1*] **unfolding** *pinverse-scalar-def* **by** *auto*

lemma *pinverse-scalar-nzero*: $(a^{-1p}) \neq 0$ **if** $\neg$ *p pdvd a* **for** a :: $'a$
using *that pinverse-scalar* **by** *fastforce*

lemma *pinverse-scalar-zero*[*simp*]: $((0{::}'a)^{-1p}) = 0$
unfolding *pinverse-scalar-def* **by** *force*

lemma *pinverse-scalar-one*[*simp*]: $((1{::}'a)^{-1p}) = 1$
using *Rep-prime-not-unit the1-equality*[*OF prime-residue-inverse-ex1*, *of p 1*] *one-pdiv one-pmod*
unfolding *pinverse-scalar-def*
by *auto*

end

4.6.2 Inductive partial inversion of formal Laurent series

Inductively define a depth-zero partial inverse representative function. It is effectively a polynomial of degree equal to the *nat* argument.

primrec *fls-partially-pinverse-rep* ::
$'a$::*nontrivial-factorial-unique-euclidean-bezout fls* $\Rightarrow$ *nat* $\Rightarrow$
$'a$ *prime* $\Rightarrow$ (*int* $\Rightarrow 'a$)

where
  fls-partially-pinverse-rep f 0 p n =
    (*if n* = *0*
      then ((*f pmod p*) $$ ($f^{\circ p}$))$^{-1p}$
      else 0)
| *fls-partially-pinverse-rep f* (*Suc N*) *p* =
    (*fls-partially-pinverse-rep f N p*)(
      int (*Suc N*) := −(
        (
          (
            fls-shift ($f^{\circ p}$) (*Abs-fls* (*fls-partially-pinverse-rep f N p*)) ∗
            (*f pmod p*)
) *pmod p*
) $$ (*int* (*Suc N*)) ∗ (((*f pmod p*) $$ ($f^{\circ p}$))$^{-1p}$)
) *pmod p*
)

lemma *fls-partially-pinverse-rep-func-lower-bound*:
  ∀ *n*<*0*. *fls-partially-pinverse-rep f* (*nat n*) *p n* = *0*
  by *fastforce*

lemma *fls-partially-pinverse-rep-at-0*:
  fls-partially-pinverse-rep f N p 0 = (((*f pmod p*) $$ ($f^{\circ p}$))$^{-1p}$)
  by (*induct N*) *simp-all*

lemma *fls-partially-pinverse-rep-func-lower-bound′*:
  ∀ *n*<*0*. *fls-partially-pinverse-rep f N p n* = *0*
  by (*induct N*) *simp-all*

lemmas *Abs-fls-partially-pinverse-rep-nth* =
  nth-Abs-fls-lower-bound[*OF fls-partially-pinverse-rep-func-lower-bound′*]

lemma *fls-partially-pinverse-rep-func-truncates*:
  n > *int N* ⟹ *fls-partially-pinverse-rep f N p n* = *0*
  by (*induct N*, *simp-all*)

context
  fixes *p* :: ′*a*::*nontrivial-factorial-unique-euclidean-bezout prime*
begin

lemma *pinverse-scalar-fls-pmod-base*:
  (
    (((*f pmod p*) $$ ($f^{\circ p}$))$^{-1p}$) ∗ (*f pmod p*) $$ ($f^{\circ p}$)
) *pmod p* = *1*
  if *f* $\neg\simeq_p$ *0*
  for *f* :: ′*a fls*
  using *that fls-pmod-eq0-iff fls-pmod-subdegree*[*of f*]
    nth-fls-subdegree-nonzero[*of f pmod p*]
    pinverse-scalar[*of p* (*f pmod p*) $$ ($f^{\circ p}$)]

fls-pmod-reduced[*of f p* ($f^{\circ p}$)]
by *fastforce*

lemma *fls-partially-pinverse-rep-eq-zero*:
fls-partially-pinverse-rep f N p = *0* **if** $f \simeq_p 0$
for *f* :: *'a fls*
proof (*induct N*)
case *0* **from** *that* **show** *?case* **by** *auto*
next
case (*Suc N*) **show** *?case*
proof
fix *n* **show** *fls-partially-pinverse-rep f* (*Suc N*) *p n* = *0 n*
proof (*cases n* = *int* (*Suc N*))
case *True*
from *Suc* **have**
fls-shift ($f^{\circ p}$) (*Abs-fls* (*fls-partially-pinverse-rep f N p*)) = *0*
using *fls-shift-zero* **unfolding** *zero-fls-def zero-fun-def* **by** *fastforce*
with *True* **show** *?thesis* **using** *fls-pmod-0* **by** *auto*
qed (*simp add*: *Suc*)
qed
qed

lemma *fls-partially-pinverse-rep-nzero-at-0*:
fls-partially-pinverse-rep f N p 0 ≠ *0* **if** $f \neg\simeq_p 0$ **for** *f* :: *'a fls*
using *that fls-partially-pinverse-rep-at-0*[*of f*] *pinverse-scalar-fls-pmod-base* **by** *fastforce*

lemma *fls-partially-pinverse-rep-eq-zero-iff*:
fls-partially-pinverse-rep f N p = *0* ⟷ $f \simeq_p 0$
for *f* :: *'a fls*
using *fls-partially-pinverse-rep-eq-zero*[*of f N*] *fls-partially-pinverse-rep-nzero-at-0*[*of f N*]
by *fastforce*

lemma *Abs-fls-partially-pinverse-rep-nzero*:
Abs-fls (*fls-partially-pinverse-rep f N p*) ≠ *0* **if** $f \neg\simeq_p 0$
for *f* :: *'a fls*
using *that fls-partially-pinverse-rep-nzero-at-0 Abs-fls-partially-pinverse-rep-nth*[*of f N p*]
by *fastforce*

lemma *Abs-fls-partially-pinverse-rep-eq-zero-iff*:
Abs-fls (*fls-partially-pinverse-rep f N p*) = *0* ⟷ $f \simeq_p 0$
for *f* :: *'a fls*
using *fls-eq-iff Abs-fls-partially-pinverse-rep-nth*[*of f*]
fls-partially-pinverse-rep-eq-zero-iff[*of f N*]
by *fastforce*

lemma *Abs-fls-partially-pinverse-rep-subdegree*:

fls-subdegree (*Abs-fls* (*fls-partially-pinverse-rep f N p*)) = *0* **if** $f \neg\simeq_p 0$
for *f* :: *'a fls*
using *that fls-partially-pinverse-rep-nzero-at-0 fls-partially-pinverse-rep-func-lower-bound' Abs-fls-partially-pinverse-rep-nth fls-subdegree-eqI*
by *metis*

lemma *fls-shift-pinv-rep-subdegree*:
fls-subdegree (*fls-shift m* (*Abs-fls* (*fls-partially-pinverse-rep f N p*))) = $-m$
if $f \neg\simeq_p 0$
for *f* :: *'a fls*
using *that Abs-fls-partially-pinverse-rep-nzero Abs-fls-partially-pinverse-rep-subdegree*
by *fastforce*

lemma *fls-partially-pinverse-rep-cong*:
fls-partially-pinverse-rep f N p ($n - K$) = *fls-partially-pinverse-rep g N p* ($n - K$)
if $f^{\circ p} = K$ **and** $g^{\circ p} = K$ **and** $\forall k \le n.$ *f* \$\$ *k* = *g* \$\$ *k*
for *f g* :: *'a fls*
proof−
have *equiv0-case*:
$\bigwedge$*f g* :: *'a fls*.
$f \simeq_p 0 \Longrightarrow g^{\circ p} = 0 \Longrightarrow$
$\forall k \le n.$ *f* \$\$ *k* = *g* \$\$ *k* $\Longrightarrow$
fls-partially-pinverse-rep f N p n = *fls-partially-pinverse-rep g N p n*
proof−
fix *f g* :: *'a fls*
assume *f*: $f \simeq_p 0$ **and** *g*: $g^{\circ p} = 0$
and *fg*: $\forall k \le n.$ *f* \$\$ *k* = *g* \$\$ *k*
show *fls-partially-pinverse-rep f N p n* = *fls-partially-pinverse-rep g N p n*
proof (*cases* $g \simeq_p 0$, *metis f fls-partially-pinverse-rep-eq-zero*)
case *False*
have $\forall k \le n.$ *fls-partially-pinverse-rep g N p k* = *0*
proof (*induct N*)
case *0* **with** *f g fg* **show** *?case* **using** *fls-pmod-nth-cong*[*of 0 g f*] **by** *auto*
next
case (*Suc N*) **show** *?case*
proof *clarify*
fix *k* **assume** *k*: $k \le n$ **show** *fls-partially-pinverse-rep g* (*Suc N*) *p k* = *0*
proof (*cases k* = *int* (*Suc N*))
case *False* **with** *k Suc* **show** *?thesis* **by** *force*
next
case *True*
moreover define *G* **where** $G \equiv$ *Abs-fls* (*fls-partially-pinverse-rep g N p*)
moreover have ((*G* ∗ (*g pmod p*)) *pmod p*) \$\$ *int* (*Suc N*) = (*0 pmod p*) \$\$ *int* (*Suc N*)
proof (*rule fls-pmod-nth-cong*, *clarify*)
fix *j* **assume** $j \le$ *int* (*Suc N*)
with *f g fg G-def True False k* **show** (*G* ∗ (*g pmod p*)) \$\$ *j* = *0* \$\$ *j*

using *fls-times-nth*(*1*)[*of G*] *fls-pmod-subdegree*[*of g p*]
Abs-fls-partially-pinverse-rep-subdegree[*of g*] *fls-pmod-nth-cong*[*of*
- g f]
p-modulo-fls-eq0[*of p f*]
by *auto*
qed
ultimately show *fls-partially-pinverse-rep g* (*Suc N*) *p k* = *0* **using** *g*
by *simp*
qed
qed
qed
with *f* **show** *?thesis* **using** *fls-partially-pinverse-rep-eq-zero* **by** *fastforce*
qed
qed
consider
$f \simeq_p 0$ |
$g \simeq_p 0$ |
(*nonzero*) $f \neg\simeq_p 0$ $g \neg\simeq_p 0$
by *blast*
thus *?thesis*
using *that equiv0-case*
proof *cases*
case *nonzero*
from *that* **have** *base*:
$n \geq K \Longrightarrow$
$(((f\ pmod\ p)\ \$\$\ K)^{-1p}) = (((g\ pmod\ p)\ \$\$\ K)^{-1p})$
using *fls-pmod-nth-cong*[*of K f g*] **by** *simp*
have
$\forall k \leq n.$
fls-partially-pinverse-rep f N p $(k - K)$ = *fls-partially-pinverse-rep g N p* $(k$
$- K)$
proof (*induct N*, *simp add*: *that*(*1*,*2*), *safe*, *rule base*, *simp*)
case (*Suc N*)
fix *k* **assume** *k*: $k \leq n$
show
fls-partially-pinverse-rep f (*Suc N*) *p* $(k - K)$ =
fls-partially-pinverse-rep g (*Suc N*) *p* $(k - K)$
proof (*cases k* = *int* (*Suc N*) + *K*)
case *False* **with** *Suc k* **show** *?thesis* **by** *fastforce*
next
case *True*
with *k* **have** *K*: $n \geq K$ **by** *linarith*
define *A* :: *'a fls* ⇒ *'a fls*
where *A* ≡ (λ*f*. *Abs-fls* (*fls-partially-pinverse-rep f N p*))
have
((*fls-shift K* (*A f*) ∗ (*f pmod p*)) *pmod p*) \$\$ (*int* (*Suc N*)) =
((*fls-shift K* (*A g*) ∗ (*g pmod p*)) *pmod p*) \$\$ (*int* (*Suc N*))
proof (*rule fls-pmod-nth-cong*, *clarify*)
fix *j* **assume** *j*: $j \leq$ *int* (*Suc N*)

with *k True* **have** $\forall\, i{\in}\{0..j\}.\ i + K \le n$ **by** *simp*
with *Suc A-def* **have** $\forall\, i{\in}\{0..j\}.\ A\ f\ \$\$\ i = A\ g\ \$\$\ i$
using *Abs-fls-partially-pinverse-rep-nth*[*of - N p*] **by** *fastforce*
moreover from *that*(*3*) *True j k* **have**
$\forall\, i{\in}\{0..j\}.\ (f\ pmod\ p)\ \$\$\ (K + j - i) = (g\ pmod\ p)\ \$\$\ (K + j - i)$
using *fls-pmod-nth-cong*[*of K + j − - f g p*] **by** *simp*
ultimately show
(*fls-shift K* (*A f*) ∗ (*f pmod p*)) $$ *j* = (*fls-shift K* (*A g*) ∗ (*g pmod p*)) $$ *j*
using *that*(*1*,*2*) *nonzero A-def fls-shift-pinv-rep-subdegree fls-pmod-subdegree*[*of - p*]
fls-times-nth(*1*)[*of fls-shift K* (*A -*)]
by *auto*
qed
with *that*(*1*,*2*) *True A-def* **show**
fls-partially-pinverse-rep f (*Suc N*) *p* (*k* − *K*) =
fls-partially-pinverse-rep g (*Suc N*) *p* (*k* − *K*)
using *base K* **by** *simp*
qed
qed
thus *?thesis* **by** *auto*
qed (*simp-all add*: *p-depth-fls.depth-equiv-0*)
qed

lemma *fls-pinv-rep-times-f-subdegree*:
fls-subdegree (
fls-shift ($f^{\circ p}$) (*Abs-fls* (*fls-partially-pinverse-rep f N p*)) ∗
(*f pmod p*)
) = *0*
if $f \neg\simeq_p 0$
for *f* :: *'a fls*
proof−
from *that*
have *f pmod p* ≠ *0* **and** *Abs-fls* (*fls-partially-pinverse-rep f N p*) ≠ *0*
using *fls-pmod-eq0-iff Abs-fls-partially-pinverse-rep-nzero* **by** (*blast*, *blast*)
with *that* **show** *?thesis*
using *fls-shift-pinv-rep-subdegree fls-pmod-subdegree*
fls-shift-eq0-iff[*of* ($f^{\circ p}$)]
by *fastforce*
qed

lemma *Abs-fls-pinv-rep-Suc*:
fixes *f* :: *'a fls* **and** *N* :: *nat*
defines
c ≡
−(
(
(
fls-shift ($f^{\circ p}$) (*Abs-fls* (*fls-partially-pinverse-rep f N p*)) ∗

$(f\ pmod\ p)$
$)\ pmod\ p$
$)\ \$\$\ (int\ (Suc\ N)) * (((f\ pmod\ p)\ \$\$\ (f^{\circ p}))^{-1p})$
$)\ pmod\ p$
assumes f: $f \neg\simeq_p 0$
shows
Abs-fls (*fls-partially-pinverse-rep f* (*Suc N*) *p*) =
Abs-fls (*fls-partially-pinverse-rep f N p*) + *fls-shift* (− *int* (*Suc N*)) (*fls-const c*)
proof (*intro fls-eqI*)
define *Abs-inv-rep* **and** *s*
where *Abs-inv-rep* ≡ (λ*N*. *Abs-fls* (*fls-partially-pinverse-rep f N p*))
and *s* ≡ *fls-shift* (− *int* (*Suc N*)) (*fls-const c*)
fix *n* **show** *Abs-inv-rep* (*Suc N*) \$\$ *n* = (*Abs-inv-rep N* + *s*) \$\$ *n*
proof−
consider (*zero*) $n = 0$ | (*SucN*) $n = Suc\ N$ | (*neither*) $n \neq 0$ $n \neq Suc\ N$
by *blast*
thus *?thesis*
proof *cases*
case *zero*
moreover from *this Abs-inv-rep-def* **have**
Abs-inv-rep (*Suc N*) \$\$ *n* = $(((f\ pmod\ p)\ \$\$\ (f^{\circ p}))^{-1p})$
by (*metis Abs-fls-partially-pinverse-rep-nth fls-partially-pinverse-rep-at-0*)
ultimately show *?thesis*
using *Abs-inv-rep-def s-def Abs-fls-partially-pinverse-rep-nth*[*of f N*]
fls-partially-pinverse-rep-at-0[*of f N*]
by *force*
next
case *SucN* **with** *Abs-inv-rep-def s-def c-def* **show** *?thesis*
using *Abs-fls-partially-pinverse-rep-nth*[*of f N*]
Abs-fls-partially-pinverse-rep-nth[*of f Suc N*]
fls-partially-pinverse-rep-func-truncates[*of N n*]
by *fastforce*
next
case *neither* **with** *Abs-inv-rep-def s-def* **show** *?thesis*
using *Abs-fls-partially-pinverse-rep-nth*[*of f N*]
Abs-fls-partially-pinverse-rep-nth[*of f Suc N*]
by *fastforce*
qed
qed
qed

lemma *fls-pinv-rep-times-f-nth*:
((
fls-shift $(f^{\circ p})$ (*Abs-fls* (*fls-partially-pinverse-rep f N p*)) ∗
$(f\ pmod\ p)$
$)\ pmod\ p)$ \$\$ $n = 0$
if f: $f \neg\simeq_p 0$ **and** n: $n \geq 1$ $n \leq int\ N$
for f :: $'a$ *fls* **and** n :: *int*

proof −
  define *pp* **where** *pp* ≡ *Rep-prime p*
  define *Abs-inv-rep*
    where *Abs-inv-rep* ≡ (λ*N*. *Abs-fls* (*fls-partially-pinverse-rep f N p*))
  define *g* **where**
    g ≡ (λ*N*. *fls-shift* ($f^{\circ p}$) (*Abs-inv-rep N*) ∗ (*f pmod p*))
  from *f g-def Abs-inv-rep-def* **have** *dg*: ∀ *N*. *fls-subdegree* (*g N*) = *0*
    using *fls-pinv-rep-times-f-subdegree* **by** *presburger*
  have ∀ *k*∈{*1*..*int N*}. (*g N pmod p*) $$ *k* = *0*
  proof (*induct N*)
    case (*Suc N*)
    define *c* **where**
      c ≡ −(
        ((*g N*) *pmod p*) $$ (*int* (*Suc N*)) ∗ (((*f pmod p*) $$ ($f^{\circ p}$))$^{-1p}$)
) *pmod p*
    define *s* **where** *s* ≡ *fls-shift* (($f^{\circ p}$) − *int* (*Suc N*)) (*fls-const c*)
    from *that s-def c-def g-def Abs-inv-rep-def*
      have *g-SucN-decomp*: *g* (*Suc N*) = *g N* + *s* ∗ (*f pmod p*)
      using *Abs-fls-pinv-rep-Suc*[*of f N*]
      by (*simp add*: *algebra-simps*)
    from *pp-def* **have** *fpmod-base*:
      (*f pmod p*) $$ ($f^{\circ p}$) *pmod p* = (*f pmod p*) $$ ($f^{\circ p}$)
      using *fls-pmod-reduced*[*of f p* $f^{\circ p}$]
        div-mult-mod-eq[*of* (*f pmod p*) $$ ($f^{\circ p}$) *pp*]
      by *simp*
    show *?case*
    proof (*cases c* = *0*, *safe*)
      case *True* **fix** *k* **assume** *k*: *k* ∈ {*1*..*int* (*Suc N*)}
      from *f pp-def True c-def* **have**
        (((*g N*) *pmod p*) $$ (*int* (*Suc N*))) *pmod p* = *0*
        using *fpmod-base pinverse-scalar-fls-pmod-base*[*of f*]
          mod-mult-right-eq[*of*
            −((*g N*) *pmod p*) $$ (*int* (*Suc N*))
            ((((*f pmod p*) $$ ($f^{\circ p}$))$^{-1p}$) ∗
              (*f pmod p*) $$ ($f^{\circ p}$))
            pp
          ]
        by (*auto simp add*: *algebra-simps*)
      with *pp-def* **have** ((*g N*) *pmod p*) $$ (*int* (*Suc N*)) = *0*
        using *fls-pmod-reduced div-mult-mod-eq*[*of* ((*g N*) *pmod p*) $$ (*int* (*Suc N*)) *pp*] **by** *simp*
      moreover have {*1*..*int* (*Suc N*)} = *insert* (*int* (*Suc N*)) {*1*..*int N*} **by** *force*
      ultimately show (*g* (*Suc N*) *pmod p*) $$ *k* = *0* **using** *k Suc s-def True g-SucN-decomp* **by** *auto*
    next
      case *False* **fix** *k* **assume** *k*: *k* ∈ {*1*..*int* (*Suc N*)}
      from *False s-def* **have** *s-n0*: *s* ≠ *0* **using** *fls-shift-eq0-iff*[*of* - *fls-const c*] **by** *auto*
      moreover from *False s-def*

have *ds*: *fls-subdegree s* = − ((*f*$^{\circ p}$) − *int* (*Suc N*))
by *simp*
ultimately have *d-s-fpmod*: *fls-subdegree* (*s* ∗ (*f pmod p*)) = *int* (*Suc N*)
using *f fls-pmod-eq0-iff* [*of f p*] *fls-pmod-subdegree*[*of f p*]
fls-subdegree-mult[*of s f pmod p*]
by *auto*
show (*g* (*Suc N*) *pmod p*) $$ *k* = *0*
proof (*cases k* = *int* (*Suc N*))
case *True*
with *pp-def s-def c-def* **have**
(*g* (*Suc N*) *pmod p*) $$ *k* =
(
(*g N pmod p*) $$ (*int* (*Suc N*)) +
(
−(
((*g N*) *pmod p*) $$ (*int* (*Suc N*)) ∗
(((*f pmod p*) $$ (*f*$^{\circ p}$))$^{-1p}$)
) *pmod p*
∗ ((*f pmod p*) $$ (*f*$^{\circ p}$) *pmod p*)
) *pmod p*
) *pmod p*
using *g-SucN-decomp dg ds d-s-fpmod fpmod-base*
fls-pmod-mismatched-add-nth(*2*)[*of g N s* ∗ (*f pmod p*)]
fls-times-base[*of s f pmod p*]
fls-pmod-subdegree[*of f p*]
mod-add-right-eq[*of* (*g N pmod p*) $$ (*int* (*Suc N*)) - *pp*]
by *simp*
with *f pp-def* **show** *?thesis*
using *pinverse-scalar-fls-pmod-base*[*of f*]
mod-mult-right-eq[*of*
− ((*g N*) *pmod p*) $$ (*int* (*Suc N*))
(((*f pmod p*) $$ (*f*$^{\circ p}$))$^{-1p}$) ∗
((*f pmod p*) $$ (*f*$^{\circ p}$))
pp
]
mod-add-right-eq[*of*
(*g N pmod p*) $$ (*int* (*Suc N*)) − ((*g N*) *pmod p*) $$ (*int* (*Suc N*))
pp
]
by (*simp add*: *mod-mult-eq algebra-simps*)
next
case *False*
with *Suc k* **show** *?thesis*
using *g-SucN-decomp dg d-s-fpmod* **by** (*simp add*: *fls-pmod-mismatched-add-nth*(*1*))
qed
qed
qed *simp*
with *n g-def Abs-inv-rep-def* **show** *?thesis* **by** *presburger*
qed

end

4.6.3 Full inversion of formal Laurent series

Now we collect the terms of the infinite sequence of partial inversion polynomials into a formal Laurent series, shifted to the opposite depth as the original.

overloading *pinverse-fls* ≡ *pinverse* :: *'a fls* ⇒ *'a prime* ⇒ *'a fls*
begin
definition *pinverse-fls* ::
  'a::nontrivial-factorial-unique-euclidean-bezout fls ⇒ *'a prime* ⇒ *'a fls*
  where
    pinverse-fls f p ≡
      (*if f* $\simeq_p$ *0 then 0 else*
        fls-shift ($f^{\circ p}$)
          (*Abs-fls* (λ*n*. *fls-partially-pinverse-rep f* (*nat n*) *p n*)))
end

context
  fixes *p* :: *'a::nontrivial-factorial-unique-euclidean-bezout prime*
begin

lemma *pinverse-fls-nth*:
  (f^{-1p}) $$ *n* =
    fls-partially-pinverse-rep f (*nat* (*n* + ($f^{\circ p}$))) *p* (*n* + ($f^{\circ p}$))
  if *f* ¬$\simeq_p$ *0*
 using *that nth-Abs-fls-lower-bound*[*OF fls-partially-pinverse-rep-func-lower-bound*]
  unfolding *pinverse-fls-def*
  by *auto*

lemma *pinverse-fls-nth'*:
  int N ≥ *n* + ($f^{\circ p}$) ⟹
    (f^{-1p}) $$ *n* = *fls-partially-pinverse-rep f N p* (*n* + ($f^{\circ p}$))
  if *f* ¬$\simeq_p$ *0*
proof (*induct N*)
  case *0* **with** *that* **show** *?case* **by** (*simp add*: *pinverse-fls-nth*)
next
  case (*Suc N*) **with** *that* **show** *?case* **using** *nat-int*[*of Suc N*]
    by (*cases n* + ($f^{\circ p}$) = *int* (*Suc N*)) (*auto simp add*: *pinverse-fls-nth*)
qed

lemma *pinverse-fls-eq0*[*simp*]:
  (f^{-1p}) = *0* **if** *f* $\simeq_p$ *0* **for** *f* :: *'a fls*
  using *that* **unfolding** *pinverse-fls-def* **by** *auto*

lemma *fls-pinv-eq0-iff*:
  (f^{-1p}) = *0* ⟷ *f* $\simeq_p$ *0* **for** *f* :: *'a fls*
  using *pinverse-scalar-fls-pmod-base pinverse-fls-nth*[*of f* −($f^{\circ p}$)] **by** *fastforce*

lemma *fls-pinv-subdegree*:
fls-subdegree $(f^{-1p}) = -(f^{\circ p})$ **for** f :: *'a fls*
using *fls-partially-pinverse-rep-nzero-at-0*[*of p f 0*] *fls-partially-pinverse-rep-func-lower-bound*
by (*cases* $f \simeq_p 0$)
(*simp*, *fastforce intro*: *fls-subdegree-eqI simp add*: *pinverse-fls-nth*)

lemma *fls-pinv-reduced*[*simp*]: $((f^{-1p})$ \$\$ $n)$ *pdiv* $p = 0$
proof (*cases* $f \simeq_p 0$)
case *False* **show** *?thesis*
proof (*cases n fls-subdegree* (f^{-1p}) *rule*: *linorder-cases*)
case *greater*
hence $n + (f^{\circ p}) - 1 \geq 0$ **using** *fls-pinv-subdegree*[*of f*] **by** *force*
hence *nat* $(n + (f^{\circ p})) =$ *Suc* (*nat* $(n + (f^{\circ p}) - 1)$) **by** *linarith*
with *False* **show** *?thesis* **by** (*fastforce simp add*: *pinverse-fls-nth*)
qed (*simp*, *fastforce simp add*: *False pinverse-fls-nth fls-pinv-subdegree pinverse-scalar-reduced*)
qed *simp*

lemma *pinverse-times-p-modulo-fls*:
($(f^{-1p}) * (f$ *pmod* $p)$) *pmod* $p = 1$ **if** $f \neg\simeq_p 0$
for f :: *'a fls*
proof (*intro fls-eq-above-subdegreeI*, *safe*)
define *f-0* **where** *f-0* $\equiv$ (f *pmod* p) \$\$ $(f^{\circ p})$
from *that* **have** *d-prod*: *fls-subdegree* $((f^{-1p}) * (f$ *pmod* $p)) = 0$
using *fls-pmod-eq0-iff*[*of f*] *fls-pinv-eq0-iff*[*of f*]
by (*auto simp add*: *fls-pmod-subdegree fls-pinv-subdegree*)
with *that* **have** *base*: $((f^{-1p}) * (f$ *pmod* $p)$ *pmod* $p)$ \$\$ $0 = 1$
using *pinverse-scalar-fls-pmod-base fls-times-base*[*of* f^{-1p} *f pmod p*]
by (*auto simp add*: *p-modulo-fls-nth pinverse-fls-nth fls-pmod-subdegree fls-pinv-subdegree*)
hence $((f^{-1p}) * (f$ *pmod* $p)) \neg\simeq_p 0$ **by** *fastforce*
thus $0 \leq$ *fls-subdegree* $((f^{-1p}) * (f$ *pmod* $p)$ *pmod* $p)$
using *d-prod fls-pmod-subdegree-ge* **by** *metis*
fix k::*int* **assume** k: $k \geq 0$
show $((f^{-1p}) * (f$ *pmod* $p)$ *pmod* $p)$ \$\$ $k = 1$ \$\$ k
proof (*cases* $k = 0$)
case *False*
define s **where**
$s \equiv$
fls-shift $(f^{\circ p})$ (*Abs-fls* (*fls-partially-pinverse-rep f* (*nat k*) *p*))
from *that s-def* **have** *fls-subdegree* $s = -(f^{\circ p})$
using *Abs-fls-partially-pinverse-rep-nzero Abs-fls-partially-pinverse-rep-subdegree*
by *fastforce*
with *that s-def*
have $((f^{-1p}) * (f$ *pmod* $p)$ *pmod* $p)$ \$\$ $k = (s * (f$ *pmod* $p)$ *pmod* $p)$ \$\$ k
using *fls-times-nth*(*1*)[*of* f^{-1p}] *fls-times-nth*(*1*)[*of s*]
pinverse-fls-nth'[*of f - nat k*] *Abs-fls-partially-pinverse-rep-nth*[*of f*]
by (*auto intro*: *fls-pmod-nth-cong simp add*: *fls-pmod-subdegree fls-pinv-subdegree*)
with *that k False s-def* **show** *?thesis* **using** *fls-pinv-rep-times-f-nth* **by** *fastforce*

qed (*simp add*: *base*)
qed *simp*

lemma *pinverse-fls*:
$((f^{-1p}) * f)$ *pmod* $p = 1$ **if** $f \neg\simeq_p 0$ **for** f :: *'a fls*
using *that pinverse-times-p-modulo-fls fls-pmod-equiv*[*of p f*]
fls-pmod-equiv-times[*of p* f^{-1p} f^{-1p}]
by *fastforce*

lemma *pinverse-fls-mult-equiv1*:
$((f^{-1p}) * f) \simeq_p 1$ **if** $f \neg\simeq_p 0$ **for** f :: *'a fls*
by (*metis that pinverse-fls fls-pmod-equiv*)

lemma *pinverse-fls-mult-equiv1'*:
$(f * (f^{-1p})) \simeq_p 1$ **if** $f \neg\simeq_p 0$ **for** f :: *'a fls*
by (*metis that pinverse-fls-mult-equiv1 mult.commute*)

lemma *p-modulo-pinverse-fls*[*simp*]:
(f^{-1p}) *pmod* $p = (f^{-1p})$ **for** f :: *'a fls*
using *fls-pmod-reduced fls-pmod-eqI* **by** *force*

lemma *fls-pinv-equiv0-iff*:
$(f^{-1p}) \simeq_p 0 \longleftrightarrow f \simeq_p 0$
for f :: *'a fls*
proof
assume $((f^{-1p}) \simeq_p 0)$
hence (f^{-1p}) *pmod* $p = 0$ **by** (*simp del*: *p-modulo-pinverse-fls*)
thus $(f \simeq_p 0)$ **using** *fls-pinv-eq0-iff* **by** *auto*
qed *simp*

lemma *fls-pinv-depth*:
$(f^{-1p})^{\circ p} = -(f^{\circ p})$ **for** f :: *'a fls*
proof (*cases* $f \simeq_p 0$)
case *False*
hence $(f^{-1p})^{\circ p}$ = *fls-subdegree* (f^{-1p})
using *fls-pinv-equiv0-iff*[*of f*] *fls-pinv-reduced*[*of f fls-subdegree* (f^{-1p})]
fls-pinv-eq0-iff[*of f*] *nth-fls-subdegree-nonzero*[*of* f^{-1p}]
p-depth-fls-rep-iff[*of p - * f^{-1p}]
by *fastforce*
thus *?thesis* **using** *fls-pinv-subdegree* **by** *presburger*
qed *simp*

lemma *fls-pinv-eqI*:
$(f^{-1p}) = g$ **if** $g * f \simeq_p 1$ **and** $\forall n.\ (g\ \$\$\ n)$ *pdiv* $p = 0$
for *f g* :: *'a fls*
proof−
from *that*(*1*) **have** $f \neg\simeq_p 0$
using *p-equality-fls.mult-0-right fls-1-not-p-equal-0 p-equality-fls.trans-right* **by** *blast*

moreover from *that*(*1*)
have $g * ((f^{-1p}) * f) \simeq_p (f^{-1p})$
using *p-equality-fls.mult-one-left*[*of p g* $*$ *f* f^{-1p}]
by (*simp add*: *ac-simps*)
ultimately show *?thesis*
using *that*(*2*) *fls-pmod-eqI*[*of p - g*] *p-equality-fls.mult-left pinverse-fls-mult-equiv1*
p-equality-fls.trans-right[*of p - - g*]
by *fastforce*
qed

lemma *fls-pinv-cong*:
$(f^{-1p}) = (g^{-1p})$ **if** $f \simeq_p g$ **for** f :: *'a fls*
proof (*cases* $g \simeq_p 0$)
case *True* **with** *that* **show** *?thesis* **using** *p-equal-fls-trans* **by** *fastforce*
next
case *False*
moreover from *that* **have** $(g^{-1p}) * f \simeq_p (g^{-1p}) * g$
using *p-equality-fls.mult-left* **by** *fast*
ultimately show *?thesis*
using *pinverse-fls-mult-equiv1 p-equal-fls-trans fls-pinv-eqI fls-pinv-reduced* **by**
blast
qed

lemma *pinverse-p-modulo-fls*:
$(f\ pmod\ p)^{-1p} = (f^{-1p})$ **for** f :: *'a fls*
using *fls-pinv-cong p-equal-fls-sym fls-pmod-equiv* **by** *blast*

lemma *fls-pinv-X-intpow*: $(fls\text{-}X\text{-}intpow\ n :: 'a\ fls)^{-1p} = fls\text{-}X\text{-}intpow\ (-n)$
by (*intro fls-pinv-eqI*, *simp add*: *fls-times-both-shifted-simp*, *auto simp add*: *one-pdiv*)

lemma *fls-pinv-uminus*: $(-f)^{-1p} = (-\ (f^{-1p}))\ pmod\ p$ **for** f :: *'a fls*
proof (*cases* $f \simeq_p 0$)
case *True* **thus** *?thesis* **using** *p-equality-fls.uminus* **by** *fastforce*
next
case *False*
hence $(\ (-\ (f^{-1p}))\ pmod\ p * -\ f) \simeq_p 1$
using *fls-pmod-equiv*[*of p* $(-\ (f^{-1p}))$ *pmod p* $* -\ f$]
fls-pmod-equiv[*of p* $(-\ (f^{-1p}))$] *p-equal-fls-refl*[*of p* $-f$]
p-equal-fls-sym[*of p* $(-\ (f^{-1p}))$ $(-\ (f^{-1p}))$ *pmod p*]
fls-pmod-equiv-times[*of p -* $(-\ (f^{-1p}))$] *pinverse-fls*
by *fastforce*
thus *?thesis* **using** *fls-pinv-eqI fls-pmod-reduced* **by** *blast*
qed

end

4.6.4 Algebraic properties of inversion

context

fixes *p* :: *'a::nontrivial-factorial-unique-euclidean-bezout prime*
begin

lemma *fls-pinv0*: $(0 :: 'a\ fls)^{-1p} = 0$
by *simp*

lemma *fls-pinv1* [*simp*]: $(1 :: 'a\ fls)^{-1p} = 1$
by (*intro fls-pinv-eqI*, *simp-all*, *rule one-pdiv*)

lemma *fls-pinv-pinv*:
$((f^{-1p})^{-1p}) = f\ pmod\ p$ **for** $f :: 'a\ fls$
proof (*cases* $f \simeq_p 0$)
case *False* **thus** *?thesis*
using *pinverse-times-p-modulo-fls fls-pmod-equiv mult.commute fls-pmod-reduced fls-pinv-eqI*
by *metis*
qed *simp*

lemma *fls-pinv-pinv-equiv*:
$((f^{-1p})^{-1p}) \simeq_p f$ **for** $f :: 'a\ fls$
using *fls-pinv-pinv*[*of f*] *fls-pmod-equiv p-equal-fls-sym* **by** *fastforce*

lemma *fls-pinv-mult*:
$(f * g)^{-1p} = ((f^{-1p}) * (g^{-1p}))\ pmod\ p$
for $f\ g :: 'a\ fls$
proof−
have *case0*:
$\bigwedge f\ g :: 'a\ fls.$
$f \simeq_p 0 \Longrightarrow$
$(f * g)^{-1p} = ((f^{-1p}) * (g^{-1p}))\ pmod\ p$
proof−
fix $f\ g :: 'a\ fls$ **assume** $f \simeq_p 0$
thus $(f * g)^{-1p} = ((f^{-1p}) * (g^{-1p}))\ pmod\ p$
using *p-equality-fls.mult-0-left* **by** *fastforce*
qed
consider
$f \simeq_p 0 \mid g \simeq_p 0 \mid$
(*neither*) $f \neg\simeq_p 0\ g \neg\simeq_p 0$
by *blast*
thus *?thesis*
proof (*cases*, *metis case0*, *metis case0 mult.commute*, *intro fls-pinv-eqI*)
case *neither*
hence $((f^{-1p}) * f * ((g^{-1p}) * g)) \simeq_p 1$
using *p-equality-fls.mult pinverse-fls-mult-equiv1*
by *fastforce*
moreover have
$(((f^{-1p}) * (g^{-1p}))\ pmod\ p * (f * g)) \simeq_p$
$((f^{-1p}) * f * ((g^{-1p}) * g))$
using *fls-pmod-equiv p-equal-fls-sym p-equality-fls.mult-right*[*of p - - f * g*]

by (*fastforce simp add*: *algebra-simps*)
ultimately
show $((f^{-1p}) * (g^{-1p})\ pmod\ p * (f * g)) \simeq_p 1$
using *p-equal-fls-trans*
by *blast*
show $\forall n.\ ((f^{-1p}) * (g^{-1p})\ pmod\ p)\ \$\$\ n\ pdiv\ p = 0$
using *fls-pmod-reduced* **by** *fast*
qed
qed

lemma *fls-pinv-mult-equiv*:
$((f * g)^{-1p}) \simeq_p (f^{-1p}) * (g^{-1p})$
for *f g* :: *'a fls*
by (*metis fls-pinv-mult fls-pmod-equiv p-equal-fls-sym*)

lemma *fls-pinv-pow*:
$(f \frown n)^{-1p} = ((f^{-1p}) \frown n)\ pmod\ p$
for *f g* :: *'a fls*
proof (*induct n*)
case (*Suc n*) **thus** *?case*
using *fls-pinv-mult fls-pmod-equiv*[*of p* $(f^{-1p}) \frown n$]
fls-pmod-equiv-times[*of p* f^{-1p} f^{-1p}]
by *simp*
qed *simp*

lemma *fls-pinv-pow-equiv*:
$((f \frown n)^{-1p}) \simeq_p (f^{-1p}) \frown n$ **for** *f* :: *'a fls*
by (*metis fls-pinv-pow fls-pmod-equiv p-equal-fls-sym*)

lemma *fls-pinv-diff-cancel-lead-coeff*:
$(((f^{-1p}) - (g^{-1p}))^{\circ p}) > -n$
if *f* : $f \neg\simeq_p 0\ f^{\circ p} = n$
and *g* : $g \neg\simeq_p 0\ g^{\circ p} = n$
and *fg*: $f \neg\simeq_p g\ ((f - g)^{\circ p}) > n$
for *f g* :: *'a fls*
proof−
from *fg*(*1*)
have *inv-diff-n0*: $((f^{-1p}) - (g^{-1p})) \neg\simeq_p 0$
by (*metis p-equality-fls.conv-0 fls-pinv-cong fls-pinv-pinv fls-pmod-eq-pmod-iff*)
moreover have *fls-subdegree* $((f^{-1p}) - (g^{-1p})) > -n$
proof (*rule fls-subdegree-greaterI*)
show $(f^{-1p}) - (g^{-1p}) \neq 0$ **using** *inv-diff-n0* **by** *fastforce*
fix *k* :: *int*
from *fg*(*2*) **have** *fg'*: $((f\ pmod\ p - g\ pmod\ p)^{\circ p}) > n$
using *fls-pmod-depth fls-pmod-equiv fls-pmod-equiv-minus*[*of p f f pmod p g g pmod p*]
by *metis*
hence *fls-subdegree* $((f\ pmod\ p - g\ pmod\ p)\ pmod\ p) > n$ **by** (*metis fls-pmod-subdegree*)
hence $((f\ pmod\ p - g\ pmod\ p)\ pmod\ p)\ \$\$\ n = 0$ **by** *simp*

moreover have
$((f\ pmod\ p - g\ pmod\ p)\ pmod\ p)$ \$\$ $n =$
fls-partially-p-modulo-rep
$(f\ pmod\ p - g\ pmod\ p)$ $(nat\ (n + 1 -$ *fls-subdegree* $(f\ pmod\ p - g\ pmod\ p)))$ $p\ n$
using *p-modulo-fls-nth*[*of f pmod p − g pmod p p n*] **by** *fastforce*
moreover from $f(2)$ $g(2)$ fg' **have** *fls-subdegree* $(f\ pmod\ p - g\ pmod\ p) \geq n$
using *fls-subdegree-minus*[*of f pmod p g pmod p*]
fls-pmod-subdegree[*of f p*] *fls-pmod-subdegree*[*of g p*]
by *force*
ultimately have $((f\ pmod\ p)$ \$\$ $n)$ $pmod\ p = ((g\ pmod\ p)$ \$\$ $n)$ $pmod\ p$
using *mod-eq-dvd-iff* **by** (*cases fls-subdegree* $(f\ pmod\ p - g\ pmod\ p) = n$, *force*, *simp*)
hence $(f\ pmod\ p)$ \$\$ $n = (g\ pmod\ p)$ \$\$ n **using** *fls-pmod-pmod-nth* **by** *metis*
moreover from $f(2)$ $g(2)$
have $k < -n \Longrightarrow ((f^{-1p}) - (g^{-1p}))$ \$\$ $k = 0$
by (*simp add*: *fls-pinv-subdegree*)
ultimately
show $k \leq -n \Longrightarrow ((f^{-1p}) - (g^{-1p}))$ \$\$ $k = 0$
using *f g*
by (*cases* $k = -n$, *simp add*: *pinverse-fls-nth*, *simp*)
qed
ultimately show *?thesis* **using** *p-depth-fls-ge-p-equal-subdegree'* **by** *fastforce*
qed

end

4.7 Topology by place relative to indexed depth for formal Laurent series

4.7.1 General pattern for constructing sequence limits

definition *fls-condition-lim* ::
'a::nontrivial-euclidean-ring-cancel prime ⇒ (*nat* ⇒ *'a fls*) ⇒
(*nat* ⇒ *nat* ⇒ *bool*) ⇒ *'a fls*
where
fls-condition-lim p F P =
Abs-fls (λ*n*. (*F* (*LEAST K*. *P* (*nat n*) *K*) *pmod p*) \$\$ *n*)

lemma *fls-condition-lim-fun-ex-lower-bound*:
fixes *p* :: *'a::nontrivial-euclidean-ring-cancel prime*
and *F* :: *nat* ⇒ *'a fls*
and *P* :: *int* ⇒ *nat* ⇒ *bool*
defines
f ≡ (λ*n*. (*F* (*LEAST K*. *P* (*nat n*) *K*) *pmod p*) \$\$ *n*)
and
N ≡ *min 0* (*fls-subdegree* ((*F* (*LEAST K*. *P 0 K*)) *pmod p*))
shows ∀ *n*<*N*. *f n* = *0*
using *assms* **by** *simp*

lemma *fls-condition-lim-nth*:
fls-condition-lim p F P $$ n = ((*F* (*LEAST K*. *P* (*nat n*) *K*)) *pmod p*) $$ n
using *nth-Abs-fls-lower-bound*[*OF fls-condition-lim-fun-ex-lower-bound*, *of F*]
fls-condition-lim-def[*of p F*]
by *auto*

lemma *fls-pmod-condition-lim*: (*fls-condition-lim p F P*) *pmod p* = *fls-condition-lim p F P*
by (*intro fls-pmod-eqI*, *simp-all add*: *fls-condition-lim-nth*)

4.7.2 Completeness

abbreviation *fls-p-limseq-condition* ≡ *p-depth-fls.p-limseq-condition*
abbreviation *fls-p-limseq* ≡ *p-depth-fls.p-limseq*
abbreviation *fls-p-cauchy-condition* ≡ *p-depth-fls.p-cauchy-condition*
abbreviation *fls-p-cauchy* ≡ *p-depth-fls.p-cauchy*

lemma *fls-p-cauchy-condition-pmod-uniformity*:
((*F k*) *pmod p*) $$ m = ((*F k*′) *pmod p*) $$ m
if *fls-p-cauchy-condition p F n K* **and** $k \geq K$ **and** $k' \geq K$ **and** $m \leq n$
for *p* :: ′*a*::*nontrivial-euclidean-ring-cancel prime* **and** *F* :: *nat* ⇒ ′*a fls*
using *that fls-pmod-cong*[*of p*] *fls-pmod-diff-cancel*[*of m p F k F k*′]
p-depth-fls.p-cauchy-conditionD
by *fastforce*

abbreviation
fls-p-cauchy-lim p X ≡
fls-condition-lim p X (λ*n*. *fls-p-cauchy-condition p X* (*int n*))

lemma *fls-p-cauchy-limseq*: *fls-p-limseq p F* (*fls-p-cauchy-lim p F*) **if** *fls-p-cauchy p F*
proof
fix *n*
define *K* **where** *K* ≡ (λ*n*. *LEAST K*. *fls-p-cauchy-condition p F* (*int* (*nat n*)) *K*)
have *fls-p-limseq-condition p F* (*fls-p-cauchy-lim p F*) *n* (*K n*)
proof
fix *k* **assume** *k*: $k \geq K\ n$ *F k* $\neg\simeq_p$ (*fls-p-cauchy-lim p F*)
have *fls-subdegree* ((*F k pmod p*) − *fls-p-cauchy-lim p F*) > *n*
proof (*intro fls-subdegree-greaterI*)
from *k*(*2*) **show** (*F k*) *pmod p* − (*fls-p-cauchy-lim p F*) ≠ *0*
using *fls-p-modulo-iff* **by** *auto*
fix *j* **assume** *j*: $j \leq n$
have ((*F* (*K j*)) *pmod p*) $$ j = ((*F k*) *pmod p*) $$ j
proof (*intro fls-p-cauchy-condition-pmod-uniformity*)
from *that K-def* **show** *fls-p-cauchy-condition p F* (*int* (*nat j*)) (*K j*)
using *p-depth-fls.p-cauchy-condition-LEAST*[*of p F*] **by** *blast*
from *j* **consider** $j < 0$ $n < 0$ | $j < 0$ $n \geq 0$ | $j \geq 0$ $n \geq 0$
by *linarith*

```
        thus k ≥ K j
          using that K-def k(1) j p-depth-fls.p-cauchy-condition-LEAST-mono
          by   cases (auto, fastforce, fastforce)
      qed simp-all
      with K-def have (F k pmod p) $$ j = fls-p-cauchy-lim p F $$ j
        by (metis fls-condition-lim-nth)
      thus (F k pmod p − fls-p-cauchy-lim p F) $$ j = 0 by simp
    qed
    moreover from k(2) have
      ((F k − fls-p-cauchy-lim p F)°p) ≥
        fls-subdegree ((F k pmod p) − fls-p-cauchy-lim p F)
      using p-equality-fls.conv-0[of p] p-equality-fls.minus[of p] fls-pmod-equiv[of p]
      by   (auto intro: p-depth-fls-ge-p-equal-subdegree)
    ultimately show ((F k − fls-p-cauchy-lim p F)°p) > n by force
  qed
  thus ∃ K. fls-p-limseq-condition p F (fls-p-cauchy-lim p F) n K by blast
qed

lemma fls-p-cauchy-lim-unique:
  fls-p-cauchy-lim p F = fls-p-cauchy-lim p G
  if fls-p-cauchy p G and ∀ n≥N. (F n) ≃p (G n)
proof−
  have fls-p-cauchy p F
  proof
    fix n
    from that(1) obtain M where M: fls-p-cauchy-condition p G n M by force
    define K where K ≡ max M N
    have fls-p-cauchy-condition p F n K
    proof (intro p-depth-fls.p-cauchy-conditionI)
      fix k k′ assume kk′: k ≥ K k′ ≥ K (F k) ¬≃p (F k′)
      moreover from that(2) K-def kk′(1,2)
        have  F-kk′: (F k) ≃p (G k) (F k′) ≃p (G k′)
        by    auto
      ultimately have k ≥ M and k′ ≥ M and (G k) ¬≃p (G k′)
          using K-def p-equality-fls.sym p-equality-fls.cong by (fastforce, fastforce,
metis)
      with M show ((F k − F k′)°p) > n
      using F-kk′ p-depth-fls.p-cauchy-conditionD[of p G n M k k′] p-depth-fls.depth-diff-equiv
        by   metis
    qed
    thus ∃ K. fls-p-cauchy-condition p F n K by blast
  qed
  moreover from that have fls-p-limseq p F (fls-p-cauchy-lim p G)
    using fls-p-cauchy-limseq[of p G]
          p-depth-fls.p-limseq-p-cong[of N p F G fls-p-cauchy-lim p G fls-p-cauchy-lim
p G]
    by   auto
  ultimately show ?thesis
   by (metis fls-p-cauchy-limseq p-depth-fls.p-limseq-unique fls-pmod-cong fls-pmod-condition-lim)
```

qed

4.8 Transfer to prime-indexed sequences of formal Laurent series

4.8.1 Equivalence and depth

The equivalence is by divisibility of the difference of each pair of corresponding terms by the index for those terms.

type-synonym *'a fls-pseq = 'a prime ⇒ 'a fls*

overloading
  p-equal-fls-pseq ≡
    p-equal :: 'a::nontrivial-factorial-semiring prime ⇒
      'a fls-pseq ⇒ 'a fls-pseq ⇒ bool
  p-restrict-fls-pseq ≡
    p-restrict ::
      'a fls-pseq ⇒ ('a prime ⇒ bool) ⇒ 'a fls-pseq
  p-depth-fls-pseq ≡
    p-depth :: 'a prime ⇒ 'a fls-pseq ⇒ int
  global-unfrmzr-pows-fls-pseq ≡
    global-unfrmzr-pows :: ('a prime ⇒ int) ⇒ 'a fls-pseq
begin

definition *p-equal-fls-pseq ::*
  'a::nontrivial-factorial-semiring prime ⇒
    'a fls-pseq ⇒ 'a fls-pseq ⇒ bool
  where *p-equal-fls-pseq-def*[*simp*]: *p-equal-fls-pseq p X Y ≡* $(X\ p) \simeq_p (Y\ p)$

definition *p-restrict-fls-pseq ::*
  'a::nontrivial-factorial-semiring fls-pseq ⇒
    ('a prime ⇒ bool) ⇒ 'a fls-pseq
  where
    p-restrict-fls-pseq X P ≡ (λp::'a prime. if P p then X p else 0)

definition *p-depth-fls-pseq ::*
  'a::nontrivial-factorial-semiring prime ⇒ 'a fls-pseq ⇒ int
  where *p-depth-fls-pseq-def*[*simp*]: *p-depth-fls-pseq p X ≡* $(X\ p)^{\circ p}$

definition *global-unfrmzr-pows-fls-pseq ::*
  ('a::nontrivial-factorial-semiring prime ⇒ int) ⇒ 'a fls-pseq
  where *global-unfrmzr-pows-fls-pseq f ≡ (λp::'a prime. fls-X-intpow (f p))*

end

global-interpretation *p-equality-depth-fls-pseq*:
  global-p-equality-depth-no-zero-divisors-1
    p-equal :: 'a::nontrivial-factorial-idom prime ⇒
      'a fls-pseq ⇒ 'a fls-pseq ⇒ bool

p-restrict ::
'*a fls-pseq* ⇒ ('*a prime* ⇒ *bool*) ⇒ '*a fls-pseq*
p-depth :: '*a prime* ⇒ '*a fls-pseq* ⇒ *int*
proof

fix *p* :: '*a prime*
define
E :: '*a fls-pseq* ⇒ '*a fls-pseq* ⇒ *bool*
where $E \equiv$ *p-equal p*
thus *equivp E*
using *equivp-transfer p-equality-fls.equivp*
unfolding *p-equal-fls-pseq-def*
by *fast*

fix *X Y* :: '*a fls-pseq*
show $(X \simeq_p Y) = ((X - Y) \simeq_p 0)$ **using** *p-equality-fls.conv-0* **by** *auto*
have $(\lambda q$::'*a prime*. $(1$::'*a fls*$)) \neg\simeq_p 0$
by (*auto simp add*: *fls-1-not-p-equal-0*)
thus $\exists X$::'*a fls-pseq*. $X \neg\simeq_p 0$ **by** *auto*

show
⟦
$X \neg\simeq_p 0$; $X + Y \neg\simeq_p 0$;
$(X^{\circ p}) = (Y^{\circ p})$
⟧ $\Longrightarrow (X^{\circ p}) \leq ((X + Y)^{\circ p})$
using *p-depth-fls.depth-pre-nonarch*(*2*) **by** *fastforce*

qed (
simp-all add:
p-equality-fls.mult-0-right p-depth-fls.no-zero-divisors p-restrict-fls-pseq-def
p-depth-fls.depth-equiv p-depth-fls.depth-uminus p-depth-fls.depth-pre-nonarch(*1*)
p-depth-fls.depth-mult-additive
)

overloading
globally-p-equal-fls-pseq ≡
globally-p-equal :: '*a*::*nontrivial-factorial-idom fls-pseq* ⇒
'*a fls-pseq* ⇒ *bool*
p-depth-set-fls-pseq ≡
p-depth-set ::
'*a*::*nontrivial-factorial-idom prime* ⇒ *int* ⇒ '*a fls-pseq set*
global-depth-set-fls-pseq ≡
global-depth-set :: *int* ⇒ '*a fls-pseq set*
begin

definition *globally-p-equal-fls-pseq* ::
'*a*::*nontrivial-factorial-idom fls-pseq* ⇒ '*a fls-pseq* ⇒ *bool*
where *globally-p-equal-fls-pseq-def*[*simp*]:
globally-p-equal-fls-pseq = *p-equality-depth-fls-pseq.globally-p-equal*

definition *p-depth-set-fls-pseq* ::
  'a::nontrivial-factorial-idom prime ⇒ *int* ⇒ *'a fls-pseq set*
  where *p-depth-set-fls-pseq-def* [*simp*]:
    p-depth-set-fls-pseq = *p-equality-depth-fls-pseq.p-depth-set*

definition *global-depth-set-fls-pseq* ::
  int ⇒ *'a::nontrivial-factorial-idom fls-pseq set*
  where *global-depth-set-fls-pseq-def* [*simp*]:
    global-depth-set-fls-pseq = *p-equality-depth-fls-pseq.global-depth-set*

end

lemma *fls-pseq-globally-reduced*:
  $X \simeq_{\forall\, p} (\lambda p.\ (X\ p)\ pmod\ p)$
  for *X* :: *'a::nontrivial-euclidean-ring-cancel fls-pseq*
  unfolding *p-equal-fls-pseq-def* **using** *fls-pmod-equiv* **by** *fastforce*

lemma *global-unfrmzr-pows0-fls-pseq*:
  ($\mathfrak{p}$ (*0* :: *'a::nontrivial-factorial-idom prime* ⇒ *int*) :: *'a fls-pseq*) = *1*
  unfolding *global-unfrmzr-pows-fls-pseq-def* **by** *auto*

context
  fixes *p* :: *'a::nontrivial-factorial-idom prime*
begin

lemma *global-unfrmzr-pows-fls-pseq-nequiv0*:
  ($\mathfrak{p}$ *f* :: *'a fls-pseq*) $\neg\simeq_p$ *0* **for** *f* :: *'a prime* ⇒ *int*
  by (*simp add*: *global-unfrmzr-pows-fls-pseq-def fls-X-intpow-nequiv0*)

lemma *global-unfrmzr-pows-fls-pseq*:
  ($\mathfrak{p}$ *f* :: *'a fls-pseq*)$^{\circ p}$ = *f p* **for** *f* :: *'a prime* ⇒ *int*
  by (*simp add*: *global-unfrmzr-pows-fls-pseq-def p-depth-fls-X-intpow*)

lemma *global-unfrmzr-pows-fls-pseq-pequiv-iff*:
  ($\mathfrak{p}$ *f*:: *'a fls-pseq*) $\simeq_p$ ($\mathfrak{p}$ *g*) ⟷ *f p* = *g p*
  for *f g* :: *'a prime* ⇒ *int*
  using *p-equal-fls-X-intpow-iff* **unfolding** *global-unfrmzr-pows-fls-pseq-def* **by** *auto*

end

lemma *global-unfrmzr-pows-prod-fls-pseq*:
  ($\mathfrak{p}$ *f* :: *'a fls-pseq*) ∗ ($\mathfrak{p}$ *g*) = $\mathfrak{p}$ (*f* + *g*)
  for *f g* :: *'a::nontrivial-factorial-idom prime* ⇒ *int*
proof (*standard*)
  fix *p* :: *'a prime*
  define $\mathfrak{p}f$ $\mathfrak{p}g$ :: *'a fls-pseq*
    where $\mathfrak{p}f \equiv \mathfrak{p}\ f$ **and** $\mathfrak{p}g \equiv \mathfrak{p}\ g$
  hence $\mathfrak{p}f$ *p* ∗ $\mathfrak{p}g$ *p* = *fls-shift* (− *f p* + − *g p*) (*1* ∗ *1*)

using *fls-times-both-shifted-simp* **unfolding** *global-unfrmzr-pows-fls-pseq-def* **by** *metis*
thus $(\mathfrak{p}f * \mathfrak{p}g)\ p = \mathfrak{p}\ (f + g)\ p$ **unfolding** *global-unfrmzr-pows-fls-pseq-def* **by** *simp*
qed

context
fixes p :: *'a::nontrivial-factorial-idom prime*
begin

lemma *global-unfrmzr-pows-fls-pseq-nzero*:
$(\mathfrak{p}\ f :: {'a}\ \textit{fls-pseq}) \neg\simeq_p 0$ **for** f :: *'a prime* $\Rightarrow$ *int*
proof−
define P :: (*'a prime* $\Rightarrow$ *int*) $\Rightarrow$ *'a fls-pseq* **where** $P \equiv \mathfrak{p}$
hence ∗: $\bigwedge f.\ P\ f \simeq_p 0 \Longrightarrow f\ p = 0$
using *global-unfrmzr-pows-fls-pseq p-equality-depth-fls-pseq.depth-equiv-0* **by** *metis*
from *P-def* **have** $\neg\ P\ f \simeq_p 0$
using ∗[*of f*] ∗[*of f* + (λ*p. 1*)] *global-unfrmzr-pows-prod-fls-pseq*[*of f*]
p-equality-depth-fls-pseq.mult-0-left[*of p - P* (λ*p. 1*)]
by *fastforce*
with *P-def* **show** *?thesis* **by** *blast*
qed

lemma *prod-w-global-unfrmzr-pows-fls-pseq*:
$(X * \mathfrak{p}\ f)^{\circ p} = (X^{\circ p}) + f\ p$ **if** $X \neg\simeq_p 0$
for X :: *'a fls-pseq* **and** f :: *'a prime* $\Rightarrow$ *int*
using *that global-unfrmzr-pows-fls-pseq-nzero p-equality-depth-fls-pseq.no-zero-divisors global-unfrmzr-pows-fls-pseq p-equality-depth-fls-pseq.depth-mult-additive*
by *metis*

lemma *normalize-depth-fls-pseq*:
$(X * \mathfrak{p}\ (\lambda p{::}'a\ prime.\ -(X^{\circ p})))^{\circ p} = 0$
for X :: *'a fls-pseq*
using *prod-w-global-unfrmzr-pows-fls-pseq p-equality-depth-fls-pseq.depth-equiv-0 p-equality-depth-fls-pseq.mult-0-left*[*of p X*]
by *fastforce*

end

lemma *normalized-depth-fls-pseq-product-additive*:
$(X * \mathfrak{p}\ (\lambda p{::}'a\ prime.\ -(X^{\circ p}))) *$
$(Y * \mathfrak{p}\ (\lambda p{::}'a\ prime.\ -(Y^{\circ p}))) =$
$((X * Y) * \mathfrak{p}\ (\lambda p{::}'a\ prime.\ -((X^{\circ p}) + (Y^{\circ p}))))$
for $X\ Y$ :: *'a::nontrivial-factorial-idom fls-pseq*
by (*simp add*: *algebra-simps global-unfrmzr-pows-prod-fls-pseq plus-fun-def*)

context
fixes p :: *'a::nontrivial-factorial-idom prime*

begin

lemma *normalized-depth-fls-pseq-product-equiv*:
$(X * \mathfrak{p}\ (\lambda p::'a\ prime.\ -(X^{\circ p}))) *$
$(Y * \mathfrak{p}\ (\lambda p::'a\ prime.\ -(Y^{\circ p}))) \simeq_p$
$((X * Y) * \mathfrak{p}\ (\lambda p::'a\ prime.\ -((X * Y)^{\circ p})))$
for $X\ Y$ :: *'a fls-pseq*
proof (*cases* $X * Y \simeq_p 0$)
case *True* **thus** *?thesis*
using *normalized-depth-fls-pseq-product-additive*[*of* $X\ Y$]
p-equality-depth-fls-pseq.trans-left[*of p - 0*]
p-equality-depth-fls-pseq.mult-0-left[*of p*]
by *presburger*
next
case *False* **thus** *?thesis*
using *global-unfrmzr-pows-fls-pseq-pequiv-iff* [*of p*]
p-equality-depth-fls-pseq.depth-mult-additive[*of p X Y*]
normalized-depth-fls-pseq-product-additive[*of X Y*]
p-equality-depth-fls-pseq.mult-left[*of p - - X* $*$ *Y*]
by *simp*
qed

lemma *trivial-global-unfrmzr-pows-fls-pseq*:
$(\mathfrak{p}\ f$ :: *'a fls-pseq*$) \simeq_p 1$ **if** $f\ p = 0$ **for** f :: *'a prime* $\Rightarrow$ *int*
using *that* **unfolding** *global-unfrmzr-pows-fls-pseq-def* **by** *auto*

lemma *prod-w-trivial-global-unfrmzr-pows-fls-pseq*:
$X * \mathfrak{p}\ f \simeq_p X$ **if** $f\ p = 0$
for f :: *'a prime* $\Rightarrow$ *int* **and** X :: *'a fls-pseq*
using *that trivial-global-unfrmzr-pows-fls-pseq p-equality-depth-fls-pseq.mult-left*
by *fastforce*

end

lemma *pow-global-unfrmzr-pows-fls-pseq*:
$(\mathfrak{p}\ f$ :: *'a fls-pseq*$)\ \hat{}\ n = (\mathfrak{p}\ (\lambda p::'a\ prime.\ int\ n * f\ p))$
for f :: *'a::nontrivial-factorial-idom prime* $\Rightarrow$ *int*
by (
induct n, *simp flip*: *zero-fun-def one-fun-def*, *rule global-unfrmzr-pows0-fls-pseq*,
simp add: *global-unfrmzr-pows-prod-fls-pseq plus-fun-def algebra-simps*
)

lemma *global-unfrmzr-pows-fls-pseq-inv*:
$(\mathfrak{p}\ (-f)$ :: *'a fls-pseq*$) * (\mathfrak{p}\ f) = 1$
for f :: *'a::nontrivial-factorial-idom prime* $\Rightarrow$ *int*
using *global-unfrmzr-pows-prod-fls-pseq*[*of* $-f\ f$] *global-unfrmzr-pows0-fls-pseq* **by** *fastforce*

lemma *global-unfrmzr-pows-fls-pseq-decomp*:

$X = ($
  $(X * \mathfrak{p}\ (\lambda p{::}'a\ prime.\ -(X^{\circ p}))) *$
  $\mathfrak{p}\ (\lambda p{::}'a\ prime.\ X^{\circ p})$
$)$
for X :: $'a$::*nontrivial-factorial-idom fls-pseq*
proof−
  have
    $-\ (\lambda p{::}'a\ prime.\ X^{\circ p}) = (\lambda p{::}'a\ prime.\ -(X^{\circ p}))$
    by *force*
  thus *?thesis*
    using *global-unfrmzr-pows-fls-pseq-inv*[*of* $\lambda p{::}'a\ prime.\ (X^{\circ p})$]
    by (*simp add*: *algebra-simps*)
qed

context
  fixes p :: $'a$::*nontrivial-factorial-idom prime*
begin

lemma *global-unfrmzr-pth-fls-pseq*:
  $(\mathfrak{p}\ (1 :: ('a\ prime \Rightarrow int))\ p :: 'a\ fls) \simeq_p$
    (*fls-const* (*Rep-prime p*))
proof (*rule iffD2*, *rule p-equal-fls-extended-intsum-iff*, *safe*)
  define *pp* **where** *pp* ≡ *Rep-prime p*
  define $\mathfrak{p}$*1p* :: $'a$ *fls* **where** $\mathfrak{p}$*1p* ≡ $\mathfrak{p}$ $(1 :: 'a\ prime \Rightarrow int)$ p
  from *pp-def* $\mathfrak{p}$*1p-def*
    show $0 \le$ *min* (*fls-subdegree* $\mathfrak{p}$*1p*) (*fls-subdegree* (*fls-const pp*))
    unfolding *global-unfrmzr-pows-fls-pseq-def*
    by *simp*
  fix $n{::}int$ **assume** $n \ge 0$
  moreover have
    $n \ge 1 \Longrightarrow$
      $(\sum k{=}0..n.\ (\mathfrak{p}1p - \textit{fls-const pp})\ \$\$\ k * pp\ \widehat{}\ nat\ (k - 0)) = 0$
  proof (*induct n rule*: *int-ge-induct*)
    case *base*
    have $\{0..1{::}int\} = \{0,1\}$ **by** *fastforce*
    thus *?case* **unfolding** $\mathfrak{p}$*1p-def* *global-unfrmzr-pows-fls-pseq-def* **by** *fastforce*
  next
    case (*step n*)
    moreover from *step*(*1*) **have** $\{0..n + 1\} = insert\ (n + 1)\ \{0..n\}$ **by** *fastforce*
    ultimately show *?case* **unfolding** $\mathfrak{p}$*1p-def global-unfrmzr-pows-fls-pseq-def* **by**
fastforce
  qed
  ultimately show
    $pp\ \widehat{}\ Suc\ (nat\ (n - 0))\ dvd\ (\sum k{=}0..n.\ (\mathfrak{p}1p - \textit{fls-const pp})\ \$\$\ k * pp\ \widehat{}\ nat$
$(k - 0))$
    by (
      cases $n = 0$ $n = 1$ *rule*: *case-split*[*case-product case-split*],
      simp-all add: $\mathfrak{p}$*1p-def global-unfrmzr-pows-fls-pseq-def*
)

qed

lemma *global-unfrmzr-fls-pseq*:
$(\mathfrak{p}\ (1 :: ('a\ prime \Rightarrow int)) :: 'a\ fls\text{-}pseq) \simeq_p$
$(\lambda p::'a\ prime.\ fls\text{-}const\ (Rep\text{-}prime\ p))$
using *p-equal-fls-pseq-def global-unfrmzr-pth-fls-pseq* **by** *fast*

end

lemma *normalize-depth-fls-pseqE*:
fixes $X :: 'a::nontrivial\text{-}factorial\text{-}idom\ fls\text{-}pseq$
obtains *X-0*
where $\forall p::'a\ prime.\ X\text{-}0^{\circ p} = 0$
and $X = X\text{-}0 * \mathfrak{p}\ (\lambda p::'a\ prime.\ X^{\circ p})$
using *normalize-depth-fls-pseq global-unfrmzr-pows-fls-pseq-decomp*
by *blast*

4.8.2 Inversion

abbreviation *global-pinverse-fls-pseq* ::
$'a::nontrivial\text{-}factorial\text{-}comm\text{-}ring\ fls\text{-}pseq \Rightarrow 'a\ fls\text{-}pseq$
$((\text{-}^{-1\forall\, p})\ [51]\ 50)$
where $global\text{-}pinverse\text{-}fls\text{-}pseq\ X \equiv (\lambda p.\ (X\ p)^{-1p})$

context
fixes $X :: 'a::nontrivial\text{-}factorial\text{-}unique\text{-}euclidean\text{-}bezout\ fls\text{-}pseq$
begin

lemma *global-pinverse-fls-pseq-eq0*[*simp*]:
$(X^{-1\forall\, p}) = 0$ **if** $X \simeq_{\forall\, p} 0$
using *that p-equality-depth-fls-pseq.globally-p-equalD* **by** *fastforce*

lemma *global-pinverse-fls-pseq-eq0-iff*:
$(X^{-1\forall\, p}) = 0 \longleftrightarrow$
$X \simeq_{\forall\, p} 0$
proof
assume X: $(X^{-1\forall\, p}) = 0$
have $\forall p.\ X\ p \simeq_p 0$
proof
fix $p :: 'a\ prime$
have $\neg\ (X \neg\simeq_p 0)$
proof
assume $X \neg\simeq_p 0$
hence $(X^{-1\forall\, p})\ p \neq 0$ **using** *fls-pinv-eq0-iff* **by** *auto*
with X **show** *False* **using** *zero-fun-def* **by** *metis*
qed
thus $X\ p \simeq_p 0$ **by** *fastforce*
qed
thus $X \simeq_{\forall\, p} 0$ **by** *simp*

qed (*rule global-pinverse-fls-pseq-eq0*)

lemma *global-pinverse-fls-pseq-equiv0-iff*:
$(X^{-1\forall\, p}) \simeq_{\forall\, p}$ *0* $\longleftrightarrow$
$X \simeq_{\forall\, p}$ *0*
using *fls-pinv-equiv0-iff*
unfolding *globally-p-equal-fls-pseq-def p-equality-depth-fls-pseq.globally-p-equal-def*
by *fastforce*

lemma *global-left-pinverse-fls-pseq*:
$(X^{-1\forall\, p}) * X \simeq_{\forall\, p}$
$(\lambda p.$ *of-bool* $(X \neg\simeq_p$ *0*$))$
using *pinverse-fls-mult-equiv1* **by** *fastforce*

lemma *global-right-pinverse-fls-pseq*:
$X * (X^{-1\forall\, p}) \simeq_{\forall\, p}$
$(\lambda p.$ *of-bool* $(X \neg\simeq_p$ *0*$))$
using *global-left-pinverse-fls-pseq mult.commute*[*of X*] **by** *presburger*

lemma *global-left-pinverse-fls-pseq'*:
$(X^{-1\forall\, p}) * X \simeq_p$ *1* **if** $X \neg\simeq_p$ *0*
for *p* :: *'a prime*
using *that global-left-pinverse-fls-pseq p-equality-depth-fls-pseq.globally-p-equalD*[*of*
- - *p*]
by *fastforce*

lemma *global-right-pinverse-fls-pseq'*:
$X * (X^{-1\forall\, p}) \simeq_p$ *1* **if** $X \neg\simeq_p$ *0*
for *p* :: *'a prime*
using *that global-left-pinverse-fls-pseq'* **by** (*simp add*: *mult.commute*)

lemma *global-pinverse-pinverse-fls-pseq*:
$((X^{-1\forall\, p})^{-1\forall\, p})$
$\simeq_{\forall\, p} X$
using *fls-pinv-pinv-equiv* **by** *fastforce*

lemma *global-pinverse-mult-fls-pseq*:
$((X * Y)^{-1\forall\, p}) \simeq_{\forall\, p}$
$(X^{-1\forall\, p}) * (Y^{-1\forall\, p})$
using *fls-pinv-mult-equiv* **by** *fastforce*

lemma *global-pinverse-pow-fls-pseq*:
$((X \,\hat{}\, n)^{-1\forall\, p}) \simeq_{\forall\, p}$
$(X^{-1\forall\, p}) \,\hat{}\, n$
by (*simp*, *standard*, *metis pow-fun-apply fls-pinv-pow-equiv*)

lemma *global-pinverse-uminus-fls-pseq*:
$((-X)^{-1\forall\, p}) \simeq_{\forall\, p}$
$-\, (X^{-1\forall\, p})$

by (*simp*, *standard*, *metis fls-pinv-uminus fls-pmod-equiv p-equal-fls-sym*)

end

lemma *globally-pinverse-pcong*:
$(X^{-1\forall\, p}) \simeq_p (Y^{-1\forall\, p})$
if $X \simeq_p Y$
for p :: *'a::nontrivial-factorial-unique-euclidean-bezout prime* **and** X Y :: *'a fls-pseq*
using *that fls-pinv-cong*
by *fastforce*

lemma *globally-pinverse-cong*:
$(X^{-1\forall\, p}) \simeq_{\forall\, p}$
$(Y^{-1\forall\, p})$
if $X \simeq_{\forall\, p} Y$
for X Y :: *'a::nontrivial-factorial-unique-euclidean-bezout fls-pseq*
using *that p-equality-depth-fls-pseq.globally-p-equalD globally-pinverse-pcong* **by** *fastforce*

lemma *globally-pinverse-global-unfrmzr-pows*:
$(\mathfrak{p}\ f$ :: *'a fls-pseq*$)^{-1\forall\, p} = \mathfrak{p}\ (-\ f)$
for f :: *'a::nontrivial-factorial-unique-euclidean-bezout prime* $\Rightarrow$ *int*
using *fls-pinv-X-intpow* **unfolding** *global-unfrmzr-pows-fls-pseq-def* **by** *auto*

lemma *global-pinverse-fls-pseq0*:
(
(*0::'a::nontrivial-factorial-unique-euclidean-bezout fls-pseq*)$^{-1\forall\, p}$
) = *0*
by *fastforce*

lemma *global-pinverse-fls-pseq1*:
(
(*1::'a::nontrivial-factorial-unique-euclidean-bezout fls-pseq*)$^{-1\forall\, p}$
) = *1*
by *auto*

lemma *global-pinverse-diff-cancel-lead-coeff*:
((
$(X^{-1\forall\, p})$ −
$(Y^{-1\forall\, p})$
$)^{\circ p}) > -n$
if $X \neg\simeq_p 0$ **and** $X^{\circ p} = n$
and $Y \neg\simeq_p 0$ **and** $Y^{\circ p} = n$
and $X \neg\simeq_p Y$ **and** $((X - Y)^{\circ p}) > n$
for p :: *'a::nontrivial-factorial-unique-euclidean-bezout prime*
and X Y :: *'a fls-pseq*
using *that fls-pinv-diff-cancel-lead-coeff* **by** *auto*

4.8.3 Topology via global bounded depth

abbreviation *fls-pseq-bdd-global-depth* ≡ *p-equality-depth-fls-pseq.bdd-global-depth*
abbreviation *fls-pseq-bdd-global-dist* ≡ *p-equality-depth-fls-pseq.bdd-global-dist*
abbreviation *fls-pseq-globally-cauchy* ≡ *p-equality-depth-fls-pseq.globally-cauchy*
abbreviation
fls-pseq-global-cauchy-condition ≡ *p-equality-depth-fls-pseq.global-cauchy-condition*

lemma *fls-pseq-global-cauchy-condition-pmod-uniformity*:
$((X\ k\ p)\ pmod\ p)$ \$\$ $m = ((X\ k'\ p)\ pmod\ p)$ \$\$ m
if *fls-pseq-global-cauchy-condition* $X\ n\ K$ **and** $k \geq K$ **and** $k' \geq K$ **and** $m \leq int$
n
for p :: $'a$::*nontrivial-euclidean-ring-cancel prime* **and** X :: $nat \Rightarrow$ $'a$ *fls-pseq*
using *that fls-pmod-cong*[*of p*] *fls-pmod-diff-cancel*[*of m p X k p X k' p*]
p-equality-depth-fls-pseq.global-cauchy-conditionD
by *fastforce*

abbreviation
fls-pseq-cauchy-lim X ≡
λp. *fls-condition-lim p* (λn. $X\ n\ p$) (*fls-pseq-global-cauchy-condition X*)

lemma *fls-pseq-cauchy-condition-lim*:
$\forall_F$ *k in sequentially. fls-pseq-bdd-global-dist* (*X k*) (*fls-pseq-cauchy-lim X*) $< e$
if *pos*: $e > 0$ **and** *cauchy*: *fls-pseq-globally-cauchy X*
proof−
define *limval* **where**
limval ≡
λp. *fls-condition-lim p* (λn. $(X\ n)\ p$) (*fls-pseq-global-cauchy-condition X*)
show $\forall_F$ *k in sequentially. fls-pseq-bdd-global-dist* (*X k*) *limval* $< e$
proof (*cases* $e > 1$)
case *True*
hence $\bigwedge n$. *fls-pseq-bdd-global-dist* (*X n*) *limval* $< e$
using *p-equality-depth-fls-pseq.bdd-global-dist-bdd order-le-less-subst1*[*of - id 1*
e]
by *fastforce*
thus *?thesis* **by** *simp*
next
case *False* **show** *?thesis*
proof (*standard*, *intro p-equality-depth-fls-pseq.bdd-global-dist-lessI pos*)
fix n :: *nat* **and** p :: $'a$ *prime*
define d **where** $d \equiv \lfloor log\ 2\ (inverse\ e)\rfloor$
define K :: $int \Rightarrow nat$
where $K \equiv \lambda n$. (*LEAST K*. *fls-pseq-global-cauchy-condition X* (*nat n*) *K*)
assume n: $n \geq K\ d\ X\ n \neg\simeq_p$ *limval*
have *fls-subdegree* $(((X\ n\ p)\ pmod\ p) - limval\ p) \geq d$
proof (*intro fls-subdegree-geI*)
from $n(2)$ **show** $(X\ n\ p)\ pmod\ p - limval\ p \neq 0$ **using** *fls-p-modulo-iff* **by**
auto
fix j **assume** j: $j < d$
have $((X\ (K\ j)\ p)\ pmod\ p)$ \$\$ $j = ((X\ n\ p)\ pmod\ p)$ \$\$ j

proof (*intro fls-pseq-global-cauchy-condition-pmod-uniformity*)
from *cauchy K-def* **show** *fls-pseq-global-cauchy-condition X* (*nat j*) (*K j*)
using *p-equality-depth-fls-pseq.global-cauchy-condition-LEAST*[*of X nat j*]
by *fastforce*
from *cauchy n j K-def* **show** $K\ j \leq n$
using *p-equality-depth-fls-pseq.global-cauchy-condition-LEAST-mono*[*of X nat j nat d*]
by *fastforce*
qed *simp-all*
with *K-def limval-def* **have** ($X\ n\ p\ pmod\ p$) \$\$ j = *limval p* \$\$ j
by (*metis fls-condition-lim-nth*)
thus ($X\ n\ p\ pmod\ p$ − *limval p*) \$\$ j = *0* **by** *auto*
qed
moreover from *n*(*2*) **have**
(($X\ n$ − *limval*)$^{\circ p}$) ≥
fls-subdegree (($X\ n\ p\ pmod\ p$) − *limval p*)
using *p-equality-fls.conv-0*[*of p*] *p-equality-fls.minus*[*of p*] *fls-pmod-equiv*[*of p*]
by (*auto intro*: *p-depth-fls-ge-p-equal-subdegree*)
ultimately show (($X\ n$ − *limval*)$^{\circ p}$) ≥ d **by** *simp*
qed
qed
qed

end

theory *pAdic-Product*

imports
FLS-Prime-Equiv-Depth
HOL−Computational-Algebra.Fraction-Field
HOL−Analysis.Elementary-Metric-Spaces

begin

5 p-Adic fields as equivalence classes of sequences of formal Laurent series

5.1 Preliminaries

lemma *inj-on-vimage-image-eq*:
$(f -` (f ` B)) \cap A = B$ **if** *inj-on f A* **and** $B \subseteq A$
using *that* **unfolding** *inj-on-def* **by** *fast*

5.2 The type definition as a quotient

quotient-type (**overloaded**) *'a p-adic-prod*
= *'a::nontrivial-factorial-idom fls-pseq* / *globally-p-equal*
by (*simp*, *rule p-equality-depth-fls-pseq.globally-p-equal-equivp*)

lemmas *rep-p-adic-prod-inverse* = *Quotient3-abs-rep*[*OF Quotient3-p-adic-prod*]
and *p-adic-prod-lift-globally-p-equal* = *Quotient3-rel-abs*[*OF Quotient3-p-adic-prod*]
and *globally-p-equal-fls-pseq-equals-rsp* = *equals-rsp*[*OF Quotient3-p-adic-prod*]
and *p-adic-prod-eq-iff*
= *Quotient3-rel-rep*[*OF Quotient3-p-adic-prod*, *symmetric*]
and *globally-p-equal-fls-pseq-rep-p-adic-prod-refl*
= *Quotient3-rep-reflp*[*OF Quotient3-p-adic-prod*]
and *globally-p-equal-fls-pseq-rep-abs-p-adic-prod*
= *rep-abs-rsp*[*OF Quotient3-p-adic-prod*] *rep-abs-rsp-left*[*OF Quotient3-p-adic-prod*]

lemma *p-adic-prod-globally-p-equal-self*:
(*rep-p-adic-prod* (*abs-p-adic-prod r*)) $\simeq_{\forall p}$ *r*
using *Quotient3-rep-abs*[*OF Quotient3-p-adic-prod*]
p-equality-depth-fls-pseq.globally-p-equal-refl
by *auto*

lemma *rep-p-adic-prod-p-equal-self*: *rep-p-adic-prod* (*abs-p-adic-prod r*) $\simeq_p$ *r*
for *p* :: *'a::nontrivial-factorial-idom prime* **and** *r* :: *'a fls-pseq*
using *p-adic-prod-globally-p-equal-self p-equality-depth-fls-pseq.globally-p-equalD*
by *auto*

lemma *rep-p-adic-prod-set-inverse*: *abs-p-adic-prod* ' (*rep-p-adic-prod* ' *A*) = *A*
proof
show *abs-p-adic-prod* ' *rep-p-adic-prod* ' *A* ⊆ *A*
proof
fix *x* **assume** *x* ∈ *abs-p-adic-prod* ' *rep-p-adic-prod* ' *A*
from *this* **obtain** *a* **where** *a* ∈ *A* *x* = *abs-p-adic-prod* (*rep-p-adic-prod a*) **by** *fast*
thus *x* ∈ *A* **using** *rep-p-adic-prod-inverse* **by** *metis*
qed
show *A* ⊆ *abs-p-adic-prod* ' *rep-p-adic-prod* ' *A*
proof
fix *a* **assume** *a* ∈ *A*
thus *a* ∈ *abs-p-adic-prod* ' *rep-p-adic-prod* ' *A*
using *rep-p-adic-prod-inverse*[*of a*] **by** *force*
qed
qed

lemma *p-adic-prod-cases* [*case-names abs-p-adic-prod*, *cases type*: *p-adic-prod*]:
C **if** $\bigwedge X$. *x* = *abs-p-adic-prod X* ⟹ *C*
by (*metis that rep-p-adic-prod-inverse*)

lemmas *two-p-adic-prod-cases* = *p-adic-prod-cases*[*case-product p-adic-prod-cases*]

and *three-p-adic-prod-cases* =
p-adic-prod-cases[*case-product p-adic-prod-cases*[*case-product p-adic-prod-cases*]]

lemma *p-adic-prod-reduced-cases* [*case-names abs-p-adic-prod*]:
fixes x :: *'a::nontrivial-euclidean-ring-cancel p-adic-prod*
obtains X **where** x = *abs-p-adic-prod* X **and** $\forall p.$ (X p) *pmod* p = X p
proof (*cases* x)
case (*abs-p-adic-prod* X)
define Y **where** $Y \equiv (\lambda p.$ (X p) *pmod* p)
with *abs-p-adic-prod* **have** x = *abs-p-adic-prod* Y
using *fls-pseq-globally-reduced p-adic-prod-lift-globally-p-equal* **by** *blast*
moreover from *Y-def* **have** $\forall p.$ (Y p) *pmod* p = Y p **by** *fastforce*
ultimately show *?thesis* **using** *that* **by** *blast*
qed

lemma *p-adic-prod-unique-rep*:
$\exists!X.$ x = *abs-p-adic-prod* X $\wedge$ ($\forall p.$ (X p) *pmod* p = X p)
for x :: *'a::nontrivial-euclidean-ring-cancel p-adic-prod*
proof (*intro ex-ex1I*, *safe*)
obtain X **where** x = *abs-p-adic-prod* X $\wedge$ ($\forall p.$ (X p) *pmod* p = X p)
using *p-adic-prod-reduced-cases* **by** *meson*
thus $\exists X.$ x = *abs-p-adic-prod* X $\wedge$ ($\forall p.$ X p *pmod* p = X p) **by** *fast*
next
fix X X' :: *'a fls-pseq*
assume $\forall p.$ X p *pmod* p = X p
and $\forall p.$ X' p *pmod* p = X' p
and *abs-p-adic-prod* X = *abs-p-adic-prod* X'
thus $X = X'$ **using** *p-adic-prod.abs-eq-iff*[*of* X X'] *fls-pmod-cong* **by** *fastforce*
qed

abbreviation
reduced-rep-p-adic-prod $x \equiv$
(*THE* $X.$ x = *abs-p-adic-prod* X $\wedge$ ($\forall p.$ (X p) *pmod* p = X p))

lemma *reduced-rep-p-adic-prod-is-rep*:
x = *abs-p-adic-prod* (*reduced-rep-p-adic-prod* x)
for x :: *'a::nontrivial-euclidean-ring-cancel p-adic-prod*
using *theI'*[*OF p-adic-prod-unique-rep*] **by** *fast*

lemma *reduced-rep-p-adic-prod-is-reduced*:
(*reduced-rep-p-adic-prod* x p) *pmod* p = *reduced-rep-p-adic-prod* x p
for p :: *'a::nontrivial-euclidean-ring-cancel prime* **and** X :: *'a p-adic-prod*
using *theI'*[*OF p-adic-prod-unique-rep*] **by** *fast*

lemma *abs-p-adic-prod-inverse*:
reduced-rep-p-adic-prod (*abs-p-adic-prod* X) = X **if** $\forall p.$ (X p) *pmod* p = X p
for X :: *'a::nontrivial-euclidean-ring-cancel fls-pseq*
by (*intro the1-equality p-adic-prod-unique-rep*, *simp add*: *that*)

lemma *p-adic-prod-seq-cases* [*case-names abs-p-adic-prod*]:
  C **if** $\bigwedge F.\ X = (\lambda n.\ abs\text{-}p\text{-}adic\text{-}prod\ (F\ n)) \Longrightarrow C$
proof−
  define F **where** $F \equiv \lambda n.\ rep\text{-}p\text{-}adic\text{-}prod\ (X\ n)$
  hence $X = (\lambda n.\ abs\text{-}p\text{-}adic\text{-}prod\ (F\ n))$
    using *rep-p-adic-prod-inverse* **by** *metis*
  with *that* **show** C **by** *auto*
qed

lemma *p-adic-prod-seq-reduced-cases* [*case-names abs-p-adic-prod*]:
  fixes $X :: nat \Rightarrow 'a::nontrivial\text{-}euclidean\text{-}ring\text{-}cancel\ p\text{-}adic\text{-}prod$
  obtains F
    where $X = (\lambda n.\ abs\text{-}p\text{-}adic\text{-}prod\ (F\ n))$
    and $\forall (n::nat)\ (p::'a\ prime).\ (F\ n\ p)\ pmod\ p = F\ n\ p$
proof−
  define F **where** $F \equiv \lambda n.\ reduced\text{-}rep\text{-}p\text{-}adic\text{-}prod\ (X\ n)$
  from *F-def* **have** $X = (\lambda n.\ abs\text{-}p\text{-}adic\text{-}prod\ (F\ n))$
    using *reduced-rep-p-adic-prod-is-rep* **by** *blast*
  moreover from *F-def* **have** $\forall (n::nat)\ (p::'a\ prime).\ (F\ n\ p)\ pmod\ p = F\ n\ p$
    using *reduced-rep-p-adic-prod-is-reduced* **by** *blast*
  ultimately show *thesis* **using** *that* **by** *simp*
qed

5.3 Algebraic instantiations

The product of p-adic fields form a ring.

instantiation *p-adic-prod* :: (*nontrivial-factorial-idom*) *comm-ring-1*
begin

lift-definition $zero\text{-}p\text{-}adic\text{-}prod :: 'a\ p\text{-}adic\text{-}prod$ **is** $0::'a\ fls\text{-}pseq$ **.**

lift-definition $one\text{-}p\text{-}adic\text{-}prod :: 'a\ p\text{-}adic\text{-}prod$ **is** $1::'a\ fls\text{-}pseq$ **.**

lift-definition *plus-p-adic-prod* ::
  $'a\ p\text{-}adic\text{-}prod \Rightarrow 'a\ p\text{-}adic\text{-}prod \Rightarrow 'a\ p\text{-}adic\text{-}prod$
  is $\lambda X\ Y.\ X + Y$
  using *p-equality-depth-fls-pseq.globally-p-equal-add* **by** *force*

lift-definition $uminus\text{-}p\text{-}adic\text{-}prod :: 'a\ p\text{-}adic\text{-}prod \Rightarrow 'a\ p\text{-}adic\text{-}prod$
  is $\lambda X.\ -\ X$
using *p-equality-depth-fls-pseq.globally-p-equal-uminus* **by** *auto*

lift-definition *minus-p-adic-prod* ::
  $'a\ p\text{-}adic\text{-}prod \Rightarrow 'a\ p\text{-}adic\text{-}prod \Rightarrow 'a\ p\text{-}adic\text{-}prod$
  is $\lambda X\ Y.\ X - Y$
  using *p-equality-depth-fls-pseq.globally-p-equal-minus* **by** *force*

lift-definition *times-p-adic-prod* ::
  $'a\ p\text{-}adic\text{-}prod \Rightarrow 'a\ p\text{-}adic\text{-}prod \Rightarrow 'a\ p\text{-}adic\text{-}prod$

is $\lambda X\ Y.\ X * Y$
using *p-equality-depth-fls-pseq.globally-p-equal-mult* **by** *fastforce*

instance
proof
fix *a b c* :: *'a p-adic-prod*
show $a + b + c = a + (b + c)$ **by** *transfer* (*simp add*: *add.assoc*)
show $a + b = b + a$ **by** *transfer* (*simp add*: *add.commute*)
show $1 * a = a$ **by** *transfer simp*
show $0 + a = a$ **by** *transfer simp*
show $- a + a = 0$ **by** *transfer simp*
show $a - b = a + - b$ **by** *transfer simp*
show $a * b * c = a * (b * c)$ **by** *transfer* (*simp add*: *mult.assoc*)
show $(a + b) * c = a * c + b * c$ **by** *transfer* (*simp add*: *distrib-right*)
show $a * b = b * a$ **by** *transfer* (*simp add*: *mult.commute*)
have ¬ (*0::'a fls-pseq*) $\simeq_{\forall p}$ (*1::'a fls-pseq*)
using *p-equality-depth-fls-pseq.globally-p-equalD p-equal-fls-sym fls-1-not-p-equal-0*
by *fastforce*
thus (*0::'a p-adic-prod*) $\neq$ (*1::'a p-adic-prod*) **by** *transfer simp*
qed

end

lemma *pow-p-adic-prod-abs-eq*:
(*abs-p-adic-prod X*) ^ *n* = *abs-p-adic-prod* (*X* ^ *n*)
for *X* :: *'a::nontrivial-factorial-idom fls-pseq*
proof (*induct n*)
case *0*
have (*abs-p-adic-prod X*) ^ *0* = (*1* :: *'a p-adic-prod*) **by** *auto*
moreover have *abs-p-adic-prod* (*X* ^ *0*) = (*1* :: *'a p-adic-prod*)
using *one-p-adic-prod.abs-eq* **by** *simp*
ultimately show *?case* **by** *presburger*
next
case (*Suc n*) **thus** (*abs-p-adic-prod X*) ^ *Suc n* = *abs-p-adic-prod* (*X* ^ *Suc n*)
using *times-p-adic-prod.abs-eq* **by** *auto*
qed

We can create inverses at the nonzero places.

instantiation *p-adic-prod* ::
(*nontrivial-factorial-unique-euclidean-bezout*) {*divide-trivial*, *inverse*}
begin

lift-definition *divide-p-adic-prod* ::
'a p-adic-prod ⇒ *'a p-adic-prod* ⇒ *'a p-adic-prod*
is $\lambda X\ Y.\ X * (Y^{-1\forall p})$
proof−
fix *X X' Y Y'* :: *'a fls-pseq*
assume $X \simeq_{\forall p} X'$ **and** $Y \simeq_{\forall p} Y'$
thus

$X * (Y^{-1\forall\, p}) \simeq_{\forall\, p}$
$X' * (Y'^{-1\forall\, p})$
using *globally-pinverse-cong*[*of* *Y* *Y'*] *p-equality-depth-fls-pseq.globally-p-equal-mult*[*of* *X* *X'*]
by *fastforce*
qed

lift-definition *inverse-p-adic-prod* :: *'a p-adic-prod* ⇒ *'a p-adic-prod*
is *global-pinverse-fls-pseq*
by (*rule globally-pinverse-cong*)

instance
proof
show $\bigwedge$*a*::*'a p-adic-prod*. *a div 0* = *0* **by** *transfer* (*simp flip*: *zero-fun-def*)
show $\bigwedge$*a*::*'a p-adic-prod*. *a div 1* = *a* **by** *transfer* (*simp flip*: *one-fun-def*)
show $\bigwedge$*a*::*'a p-adic-prod*. *0 div a* = *0* **by** *transfer* (*simp flip*: *zero-fun-def*)
qed

end

5.4 Equivalence and depth relative to a prime

overloading
p-equal-p-adic-prod ≡
p-equal :: *'a*::*nontrivial-factorial-idom prime* ⇒ *'a p-adic-prod* ⇒
'a p-adic-prod ⇒ *bool*
p-restrict-p-adic-prod ≡
p-restrict :: *'a p-adic-prod* ⇒
(*'a prime* ⇒ *bool*) ⇒ *'a p-adic-prod*
p-depth-p-adic-prod ≡ *p-depth* :: *'a prime* ⇒ *'a p-adic-prod* ⇒ *int*
global-unfrmzr-pows-p-adic-prod ≡
global-unfrmzr-pows :: (*'a prime* ⇒ *int*) ⇒ *'a p-adic-prod*
begin

lift-definition *p-equal-p-adic-prod* ::
'a::*nontrivial-factorial-idom prime* ⇒
'a p-adic-prod ⇒ *'a p-adic-prod* ⇒ *bool*
is *p-equal*
by (
simp only: *globally-p-equal-fls-pseq-def*,
rule p-equality-depth-fls-pseq.globally-p-equal-transfer-equals-rsp, *simp-all*
)

lift-definition *p-restrict-p-adic-prod* ::
'a::*nontrivial-factorial-idom p-adic-prod* ⇒
(*'a prime* ⇒ *bool*) ⇒ *'a p-adic-prod*
is $\lambda X\ P\ p.$ *if* $P\ p$ *then* $X\ p$ *else* *0*
using *p-equality-depth-fls-pseq.globally-p-equalD* **by** *fastforce*

lift-definition *p-depth-p-adic-prod* ::
  'a::nontrivial-factorial-idom prime ⇒ *'a p-adic-prod* ⇒ *int*
  is *p-depth*
  by (
    simp only: *globally-p-equal-fls-pseq-def*,
    rule p-equality-depth-fls-pseq.globally-p-equal-p-depth
)

lift-definition *global-unfrmzr-pows-p-adic-prod* ::
  (*'a::nontrivial-factorial-idom prime* ⇒ *int*) ⇒ *'a p-adic-prod*
  is *global-unfrmzr-pows* **.**

end

lemma *p-equal-p-adic-prod-abs-eq0*: (*abs-p-adic-prod X* $\simeq_p$ *0*) = (*X* $\simeq_p$ *0*)
  for *p* :: *'a::nontrivial-factorial-idom prime* **and** *X* :: *'a fls-pseq*
  using *p-equal-p-adic-prod.abs-eq*[*of p X 0*::*'a fls-pseq*]
  by (*simp add*: *zero-p-adic-prod.abs-eq*)

lemma *p-equal-p-adic-prod-abs-eq1*: (*abs-p-adic-prod X* $\simeq_p$ *1*) = (*X* $\simeq_p$ *1*)
  for *p* :: *'a::nontrivial-factorial-idom prime* **and** *X* :: *'a fls-pseq*
  using *p-equal-p-adic-prod.abs-eq*[*of p X 1*::*'a fls-pseq*]
  by (*simp add*: *one-p-adic-prod.abs-eq*)

global-interpretation *p-equality-p-adic-prod*:
  p-equality-no-zero-divisors-1
    p-equal :: *'a::nontrivial-factorial-idom prime* ⇒
      'a p-adic-prod ⇒ *'a p-adic-prod* ⇒ *bool*
proof

  fix *p* :: *'a prime*

  define *E* :: *'a p-adic-prod* ⇒ *'a p-adic-prod* ⇒ *bool*
    where *E* ≡ *p-equal p*

  show *equivp E*
    unfolding *E-def p-equal-p-adic-prod-def*
  proof−
    define *F* :: *'a fls-pseq* ⇒ *'a fls-pseq* ⇒ *bool*
      and *G* :: *'a p-adic-prod* ⇒ *'a p-adic-prod* ⇒ *bool*
      where *F* ≡ *p-equal p*
      and
        G ≡
          map-fun id (*map-fun rep-p-adic-prod* (*map-fun rep-p-adic-prod id*)) *p-equal*
p
    hence *G* = (λ*x y*. *F* (*rep-p-adic-prod x*) (*rep-p-adic-prod y*)) **by** *force*
   with *F-def* **show** *equivp G* **using** *equivp-transfer p-equality-depth-fls-pseq.equivp*
by *fast*
  qed

show $\exists x{::}'a$ *p-adic-prod*. $x \neg\simeq_p 0$
by (*transfer*, *rule p-equality-depth-fls-pseq.nontrivial*)

fix x y :: *'a p-adic-prod*
show $(x \simeq_p y) = (x - y \simeq_p 0)$
and $y \simeq_p 0 \Longrightarrow x * y \simeq_p 0$
by (
transfer, *rule p-equality-depth-fls-pseq.conv-0*,
transfer, *rule p-equality-depth-fls-pseq.mult-0-right*
)

assume $x \neg\simeq_p 0$ **and** $y \neg\simeq_p 0$
thus $x * y \neg\simeq_p 0$
using *p-depth-fls.no-zero-divisors*[*of p*]
by (*cases x*, *cases y*)
(*auto simp add*: *p-equal-p-adic-prod-abs-eq0 times-p-adic-prod.abs-eq*)

qed

overloading
globally-p-equal-p-adic-prod ≡
globally-p-equal ::
'a::nontrivial-factorial-idom p-adic-prod ⇒ *'a p-adic-prod* ⇒ *bool*
begin

definition *globally-p-equal-p-adic-prod* ::
'a::nontrivial-factorial-idom p-adic-prod ⇒ *'a p-adic-prod* ⇒ *bool*
where *globally-p-equal-p-adic-prod-def*[*simp*]:
globally-p-equal-p-adic-prod = *p-equality-p-adic-prod.globally-p-equal*

end

global-interpretation *global-p-depth-p-adic-prod*:
global-p-equal-depth-no-zero-divisors-w-inj-index-consts
p-equal :: *'a::nontrivial-factorial-idom prime* ⇒
'a p-adic-prod ⇒ *'a p-adic-prod* ⇒ *bool*
p-restrict ::
'a p-adic-prod ⇒ (*'a prime* ⇒ *bool*) ⇒ *'a p-adic-prod*
p-depth :: *'a prime* ⇒ *'a p-adic-prod* ⇒ *int*
λf. *abs-p-adic-prod* (λp. *fls-const* (*f p*))
Rep-prime
proof (*unfold-locales*, *fold globally-p-equal-p-adic-prod-def*)

fix p :: *'a prime*

fix x y :: *'a p-adic-prod*

show $x \simeq_{\forall p} y \Longrightarrow x = y$ **by** (*simp*, *transfer*, *simp*)

```
  fix P :: 'a prime ⇒ bool
  show P p ⟹ x prestrict P ≃p x by transfer simp
  show ¬ P p ⟹ x prestrict P ≃p 0 by transfer simp

  show ((0::'a p-adic-prod)°p) = 0 by transfer simp
  show x ≃p y ⟹ (x°p) = (y°p)
    by (transfer, rule p-equality-depth-fls-pseq.depth-equiv, simp)
  show   (− x°p) = (x°p)
    by (transfer, rule p-equality-depth-fls-pseq.depth-uminus)
  show
    x ¬≃p 0 ⟹ (x°p) < (y°p)
      ⟹ (x + y°p) = (x°p)
    by (transfer, rule p-equality-depth-fls-pseq.depth-pre-nonarch(1), simp, simp)
  show
    ⟦
      x ¬≃p 0; x + y ¬≃p 0;
      (x°p) = (y°p)
    ⟧ ⟹ (x°p) ≤ ((x + y)°p)
     using p-equality-depth-fls-pseq.depth-pre-nonarch(2)[of p] by (transfer, simp
add: mult.commute)
  show
    x ∗ y ¬≃p 0 ⟹
      (x ∗ y°p) = (x°p) + (y°p)
    by (transfer, rule p-equality-depth-fls-pseq.depth-mult-additive, simp)

  define F :: ('a prime ⇒ 'a) ⇒ 'a p-adic-prod
    where F ≡ λf. abs-p-adic-prod (λp. fls-const (f p))

  show F 1 = 1 by (simp add: F-def one-p-adic-prod.abs-eq flip: one-fun-def)

  fix f g :: 'a prime ⇒ 'a
  show  F (f − g) = F f − F g
    and F (f ∗ g) = F f ∗ F g
    and (F f ≃p 0) = (f p = 0)
    and ((F f)°p) = int (pmultiplicity p (f p))
    by  (simp-all
     add: F-def fls-minus-const fun-diff-def minus-p-adic-prod.abs-eq times-p-adic-prod.abs-eq
          times-fun-def p-equal-p-adic-prod-abs-eq0 p-equal-fls-const-0-iff p-depth-const-def
              p-depth-p-adic-prod.abs-eq
      flip: p-depth-const
    )

qed (
  metis Rep-prime-inject injI, rule Rep-prime-n0, rule Rep-prime-not-unit,
  rule multiplicity-distinct-primes
)

lemma p-depth-p-adic-prod-diff-abs-eq:
```

$((abs\text{-}p\text{-}adic\text{-}prod\ X\ -\ abs\text{-}p\text{-}adic\text{-}prod\ Y)^{\circ p}) = ((X\ -\ Y)^{\circ p})$
for *p* :: *'a::nontrivial-factorial-idom prime* **and** *X Y* :: *'a fls-pseq*
using *minus-p-adic-prod.abs-eq p-depth-p-adic-prod.abs-eq* **by** *metis*

lemma *p-depth-p-adic-prod-diff-abs-eq'*:
$((abs\text{-}p\text{-}adic\text{-}prod\ X\ -\ abs\text{-}p\text{-}adic\text{-}prod\ Y)^{\circ p}) = ((X\ p\ -\ Y\ p)^{\circ p})$
for *p* :: *'a::nontrivial-factorial-idom prime* **and** *X Y* :: *'a fls-pseq*
using *p-depth-p-adic-prod-diff-abs-eq* **by** *auto*

overloading
p-depth-set-p-adic-prod ≡
p-depth-set ::
'a::nontrivial-factorial-idom prime ⇒ *int* ⇒ *'a p-adic-prod set*
global-depth-set-p-adic-prod ≡
global-depth-set :: *int* ⇒ *'a p-adic-prod set*
begin

definition *p-depth-set-p-adic-prod* ::
'a::nontrivial-factorial-idom prime ⇒ *int* ⇒ *'a p-adic-prod set*
where *p-depth-set-p-adic-prod-def* [*simp*]:
p-depth-set-p-adic-prod = *global-p-depth-p-adic-prod.p-depth-set*

definition *global-depth-set-p-adic-prod* ::
int ⇒ *'a::nontrivial-factorial-idom p-adic-prod set*
where *global-depth-set-p-adic-prod-def* [*simp*]:
global-depth-set-p-adic-prod = *global-p-depth-p-adic-prod.global-depth-set*

end

lemma *p-depth-set-p-adic-prod-eq-projection*:
$((\mathcal{P}_{@p}{}^n)$ :: *'a p-adic-prod set*) =
abs-p-adic-prod ‘ $(\mathcal{P}_{@p}{}^n)$
for *p* :: *'a::nontrivial-factorial-idom prime*
proof *safe*
fix *x* :: *'a p-adic-prod*
assume *x*: $x \in \mathcal{P}_{@p}{}^n$
show $x \in$ *abs-p-adic-prod* ‘ $\mathcal{P}_{@p}{}^n$
proof (*cases x*)
case (*abs-p-adic-prod X*)
with *x* **have** $X \in \mathcal{P}_{@p}{}^n$
using *p-depth-set-p-adic-prod-def global-p-depth-p-adic-prod.p-depth-setD*
p-equal-p-adic-prod-abs-eq0 p-depth-p-adic-prod.abs-eq
p-equality-depth-fls-pseq.p-depth-setI p-depth-set-fls-pseq-def
by *metis*
with *abs-p-adic-prod* **show** $x \in$ *abs-p-adic-prod* ‘ $\mathcal{P}_{@p}{}^n$ **by** *fast*
qed
qed (
metis p-equality-depth-fls-pseq.p-depth-setD p-depth-set-fls-pseq-def p-equal-p-adic-prod-abs-eq0
p-depth-p-adic-prod.abs-eq p-depth-set-p-adic-prod-def

global-p-depth-p-adic-prod.p-depth-setI
)

lemma *global-depth-set-p-adic-prod-eq-projection*:
$((\mathcal{P}_{\forall\, p}{}^{n})$ :: *'a p-adic-prod set*$) =$
abs-p-adic-prod ‘ $(\mathcal{P}_{\forall\, p}{}^{n})$
for p :: *'a::nontrivial-factorial-idom prime*
proof *safe*
fix x :: *'a p-adic-prod*
assume x: $x \in \mathcal{P}_{\forall\, p}{}^{n}$
show $x \in$ *abs-p-adic-prod* ‘ $\mathcal{P}_{\forall\, p}{}^{n}$
proof (*cases x*)
case (*abs-p-adic-prod X*)
with x **have** $X \in \mathcal{P}_{\forall\, p}{}^{n}$
using *global-depth-set-p-adic-prod-def global-p-depth-p-adic-prod.global-depth-setD p-equal-p-adic-prod-abs-eq0 p-depth-p-adic-prod.abs-eq p-equality-depth-fls-pseq.global-depth-setI global-depth-set-fls-pseq-def*
by *metis*
with *abs-p-adic-prod* **show** $x \in$ *abs-p-adic-prod* ‘ $\mathcal{P}_{\forall\, p}{}^{n}$
by *fast*
qed
qed (
metis p-equality-depth-fls-pseq.global-depth-setD global-depth-set-fls-pseq-def p-equal-p-adic-prod-abs-eq0 p-depth-p-adic-prod.abs-eq global-depth-set-p-adic-prod-def global-p-depth-p-adic-prod.global-depth-setI
)

context
fixes x :: *'a::nontrivial-factorial-idom p-adic-prod*
begin

lemma *p-adic-prod-p-restrict-depth*:
(*x prestrict P*)$^{\circ p}$ = (*if P p then* $x^{\circ p}$ *else 0*)
for P :: *'a prime* $\Rightarrow$ *bool*
by (*transfer*, *simp*)

lemma *p-adic-prod-global-depth-setI*:
$x \in \mathcal{P}_{\forall\, p}{}^{n}$
if $\bigwedge p$::*'a prime*. $x \neg\simeq_p 0 \Longrightarrow (x^{\circ p}) \geq n$
using *that global-p-depth-p-adic-prod.global-depth-setI global-depth-set-p-adic-prod-def*
by *auto*

lemma *p-adic-prod-global-depth-setI'*:
$x \in \mathcal{P}_{\forall\, p}{}^{n}$
if $\bigwedge p$::*'a prime*. $(x^{\circ p}) \geq n$
using *that global-p-depth-p-adic-prod.global-depth-setI' global-depth-set-p-adic-prod-def*
by *auto*

lemma *p-adic-prod-global-depth-set-p-restrictI*:

p-restrict x P $\in \mathcal{P}_{\forall\, p}{}^{n}$
if $\bigwedge p$::*'a prime. P p* $\Longrightarrow x \in \mathcal{P}_{@p}{}^{n}$
using *that global-p-depth-p-adic-prod.global-depth-set-p-restrictI global-depth-set-p-adic-prod-def*
by *auto*

lemma *p-adic-prod-p-depth-setI*:
$x \in \mathcal{P}_{@p}{}^{n}$
if $x \neg\simeq_p 0 \longrightarrow (x^{\circ p}) \geq n$
for *p* :: *'a prime*
using *that global-p-depth-p-adic-prod.p-depth-setI p-depth-set-p-adic-prod-def* **by** *auto*

end

context
fixes *p* :: *'a::nontrivial-euclidean-ring-cancel prime*
begin

lemma *p-adic-prod-diff-depth-gt-equiv-trans*:
$((x - z)^{\circ p}) > n$
if *xy*: $x \simeq_p y$
and *yz*: $y \neg\simeq_p z$ $((y - z)^{\circ p}) > n$
for *x y z* :: *'a p-adic-prod*
proof (*cases x y z rule*: *three-p-adic-prod-cases*)
case (*abs-p-adic-prod-abs-p-adic-prod-abs-p-adic-prod X Y Z*)
from *xy abs-p-adic-prod-abs-p-adic-prod-abs-p-adic-prod*(*1*,*2*)
have *XY*: $X \simeq_p Y$ **using** *p-equal-p-adic-prod.abs-eq* **by** *auto*
moreover from *yz*(*1*) *abs-p-adic-prod-abs-p-adic-prod-abs-p-adic-prod*(*2*,*3*)
have *YZ*: $Y \neg\simeq_p Z$ **using** *p-equal-p-adic-prod.abs-eq* **by** *auto*
moreover from *yz*(*2*) *abs-p-adic-prod-abs-p-adic-prod-abs-p-adic-prod*(*2*,*3*)
have $((Y\ p - Z\ p)^{\circ p}) > n$ **using** *p-depth-p-adic-prod-diff-abs-eq'* **by** *metis*
ultimately have $\forall k \leq n.$ ((*X p*) *pmod p*) \$\$ *k* = ((*Z p*) *pmod p*) \$\$ *k*
using *fls-pmod-eq-pmod-iff*[*of X p p Y p*] *fls-pmod-diff-cancel-iff* **by** *fastforce*
moreover have (*X p*) $\neg\simeq_p$ (*Z p*) **using** *XY YZ p-equality-fls.trans-right* **by** *auto*
ultimately show $n < ((x - z)^{\circ p})$
using *abs-p-adic-prod-abs-p-adic-prod-abs-p-adic-prod*(*1*,*3*) *fls-pmod-diff-cancel-iff p-depth-p-adic-prod-diff-abs-eq'*
by *metis*
qed

lemma *p-adic-prod-diff-depth-gt0-equiv-trans*:
$((x - z)^{\circ p}) > n$
if $n \geq 0$ **and** $x \simeq_p y$ **and** $((y - z)^{\circ p}) > n$
for *x y z* :: *'a p-adic-prod*
using *that p-equality-p-adic-prod.conv-0 p-adic-prod-diff-depth-gt-equiv-trans global-p-depth-p-adic-prod.depth-equiv-0*[*of p y* − *z*]
by *fastforce*

lemma *p-adic-prod-diff-depth-gt-trans*:
  $((x - z)^{\circ p}) > n$
  if *xy*: $x \neg\simeq_p y$ $((x - y)^{\circ p}) > n$
  and *yz*: $y \neg\simeq_p z$ $((y - z)^{\circ p}) > n$
  and *xz*: $x \neg\simeq_p z$
  for *x y z* :: *'a p-adic-prod*
proof (*cases x y z rule*: *three-p-adic-prod-cases*)
  case (*abs-p-adic-prod-abs-p-adic-prod-abs-p-adic-prod X Y Z*)
  moreover from *this xy*(*2*) *yz*(*2*)
    have $((X\ p - Y\ p)^{\circ p}) > n$
    and $((Y\ p - Z\ p)^{\circ p}) > n$
    using *p-depth-p-adic-prod-diff-abs-eq'*
    by (*metis*, *metis*)
  ultimately have $\forall k \le n.$ ((*X p*) *pmod p*) $$ *k* = ((*Z p*) *pmod p*) $$ *k*
   using *xy*(*1*) *yz*(*1*) **by** (*simp add*: *p-equal-p-adic-prod.abs-eq fls-pmod-diff-cancel-iff*)
  with *xz abs-p-adic-prod-abs-p-adic-prod-abs-p-adic-prod*(*1*,*3*)
    show $n < ((x - z)^{\circ p})$
    using *p-equal-p-adic-prod.abs-eq*[*of p X Z*] *fls-pmod-diff-cancel-iff*[*of p X p Z p*]
          p-depth-p-adic-prod-diff-abs-eq[*of p X Z*]
    by *simp*
qed

lemma *p-adic-prod-diff-depth-gt0-trans*:
  $((x - z)^{\circ p}) > n$
  if $n \ge 0$
  and $((x - y)^{\circ p}) > n$
  and $((y - z)^{\circ p}) > n$
  and $x \neg\simeq_p z$
  for *x y z* :: *'a p-adic-prod*
  using *that p-adic-prod-diff-depth-gt-trans*
       p-equality-p-adic-prod.conv-0[*of p x y*] *p-equality-p-adic-prod.conv-0*[*of p y z*]
        global-p-depth-p-adic-prod.depth-equiv-0[*of p x − y*]
        global-p-depth-p-adic-prod.depth-equiv-0[*of p y − z*]
  by *auto*

end

lemma *p-adic-prod-diff-cancel-lead-coeff*:
  $((inverse\ x - inverse\ y)^{\circ p}) > -n$
  if $x : x \neg\simeq_p 0\ x^{\circ p} = n$
  and $y : y \neg\simeq_p 0\ y^{\circ p} = n$
  and *xy*: $x \neg\simeq_p y$ $((x - y)^{\circ p}) > n$
  for *p* :: *'a::nontrivial-factorial-unique-euclidean-bezout prime*
  and *x y* :: *'a p-adic-prod*
proof (*cases x y rule*: *two-p-adic-prod-cases*)
  case (*abs-p-adic-prod-abs-p-adic-prod X Y*)
  with *x*(*1*) *y*(*1*) *xy*(*1*)
    have $X \neg\simeq_p 0$
    and $Y \neg\simeq_p 0$

and $X \neg\simeq_p Y$
using *p-equal-p-adic-prod-abs-eq0 p-equal-p-adic-prod.abs-eq*
by (*fast*, *fast*, *fast*)
moreover from *abs-p-adic-prod-abs-p-adic-prod x(2) y(2) xy(2)*
have $X^{\circ p} = n$
and $Y^{\circ p} = n$
and $((X - Y)^{\circ p}) > n$
using *p-depth-p-adic-prod.abs-eq*
by (*metis*, *metis*, *metis minus-p-adic-prod.abs-eq*)
moreover from *abs-p-adic-prod-abs-p-adic-prod* **have**
$((\mathit{inverse}\ x - \mathit{inverse}\ y)^{\circ p}) =$
((
$(X^{-1\forall\, p}) -$
$(Y^{-1\forall\, p})$
$)^{\circ p})$
using *inverse-p-adic-prod.abs-eq*[*of X*] *inverse-p-adic-prod.abs-eq*[*of Y*]
p-depth-p-adic-prod-diff-abs-eq[*of*
$p\ X^{-1\forall\, p}\ Y^{-1\forall\, p}$
]
by *argo*
ultimately show *?thesis* **using** *global-pinverse-diff-cancel-lead-coeff* **by** *auto*
qed

lemma *global-unfrmzr-pows-p-adic-prod0*:
$(\mathfrak{p}\ (0 :: 'a{::}\mathit{nontrivial\text{-}factorial\text{-}idom}\ \mathit{prime} \Rightarrow \mathit{int}) :: 'a\ \mathit{p\text{-}adic\text{-}prod}) = 1$
by (
transfer,
metis p-equality-depth-fls-pseq.globally-p-equal-refl globally-p-equal-fls-pseq-def
global-unfrmzr-pows0-fls-pseq
)

context
fixes $p :: 'a{::}\mathit{nontrivial\text{-}factorial\text{-}idom}\ \mathit{prime}$
begin

lemma *global-unfrmzr-pows-p-adic-prod-nequiv0*:
$(\mathfrak{p}\ f :: 'a\ \mathit{p\text{-}adic\text{-}prod}) \neg\simeq_p 0$ **for** $f :: 'a\ \mathit{prime} \Rightarrow \mathit{int}$
by (*transfer*, *rule global-unfrmzr-pows-fls-pseq-nequiv0*)

lemma *global-unfrmzr-pows-p-adic-prod*:
$(\mathfrak{p}\ f :: 'a\ \mathit{p\text{-}adic\text{-}prod})^{\circ p} = f\ p$ **for** $f :: 'a\ \mathit{prime} \Rightarrow \mathit{int}$
by (*transfer*, *rule global-unfrmzr-pows-fls-pseq*)

lemma *global-unfrmzr-pows-p-adic-prod-depth-set*:
fixes $f :: 'a\ \mathit{prime} \Rightarrow \mathit{int}$
defines $n \equiv f\ p$
shows $(\mathfrak{p}\ f :: 'a\ \mathit{p\text{-}adic\text{-}prod}) \in \mathcal{P}_{@p}{}^{n}$
using *p-adic-prod-p-depth-setI*[*of p*] *n-def global-unfrmzr-pows-p-adic-prod*[*of f*]
by *force*

lemma *global-unfrmzr-pows-p-adic-prod-pequiv-iff*:
$((\mathfrak{p}\ f :: 'a\ p\text{-}adic\text{-}prod) \simeq_p (\mathfrak{p}\ g)) = (f\ p = g\ p)$
for $f\ g :: 'a\ prime \Rightarrow int$
by (*transfer*, *rule global-unfrmzr-pows-fls-pseq-pequiv-iff*)

end

lemma *global-unfrmzr-pows-p-adic-prod-global-depth-set*:
$(\mathfrak{p}\ f :: 'a\ p\text{-}adic\text{-}prod) \in \mathcal{P}_{\forall\ p}{}^n$
if $\forall\ p.\ f\ p \geq n$ **for** $f :: 'a::nontrivial\text{-}factorial\text{-}idom\ prime \Rightarrow int$
by (
metis that global-depth-set-p-adic-prod-def global-p-depth-p-adic-prod.global-depth-setI global-unfrmzr-pows-p-adic-prod
)

lemma *global-unfrmzr-pows-p-adic-prod-global-depth-set-0*:
$(\mathfrak{p}\ (int \circ f) :: 'a\ p\text{-}adic\text{-}prod) \in \mathcal{O}_{\forall\ p}$
for $f :: 'a::nontrivial\text{-}factorial\text{-}idom\ prime \Rightarrow nat$
using *global-unfrmzr-pows-p-adic-prod-global-depth-set*[*of 0 int* ∘ *f*] **by** *simp*

lemma *global-unfrmzr-pows-prod-p-adic-prod*:
$(\mathfrak{p}\ f :: 'a\ p\text{-}adic\text{-}prod) * (\mathfrak{p}\ g) = \mathfrak{p}\ (f + g)$
for $f\ g :: 'a::nontrivial\text{-}factorial\text{-}idom\ prime \Rightarrow int$
by (
transfer, *simp*, *standard*,
simp only: *global-unfrmzr-pows-prod-fls-pseq p-equality-fls.refl flip*: *times-fun-apply*
)

context
fixes $p :: 'a::nontrivial\text{-}factorial\text{-}idom\ prime$
begin

lemma *global-unfrmzr-pows-p-adic-prod-nzero*:
$(\mathfrak{p}\ f :: 'a\ p\text{-}adic\text{-}prod) \neg\simeq_p 0$ **for** $f :: 'a\ prime \Rightarrow int$
by (*transfer*, *rule global-unfrmzr-pows-fls-pseq-nzero*)

lemma *prod-w-global-unfrmzr-pows-p-adic-prod*:
$x \neg\simeq_p 0 \Longrightarrow$
$(x * \mathfrak{p}\ f)^{\circ p} = (x^{\circ p}) + f\ p$
for $f :: 'a\ prime \Rightarrow int$ **and** $x :: 'a\ p\text{-}adic\text{-}prod$
by (*transfer*, *rule prod-w-global-unfrmzr-pows-fls-pseq*, *simp*)

lemma *p-adic-prod-normalize-depth*:
$(x * \mathfrak{p}\ (\lambda p::'a\ prime.\ -(x^{\circ p})))^{\circ p} = 0$
for $x :: 'a\ p\text{-}adic\text{-}prod$
by (*transfer*, *rule normalize-depth-fls-pseq*)

end

lemma *p-adic-prod-normalized-depth-product-equiv*:
$(x * \mathfrak{p}\ (\lambda p{::}'a\ prime.\ -(x^{\circ p}))) *$
$(y * \mathfrak{p}\ (\lambda p{::}'a\ prime.\ -(y^{\circ p}))) =$
$((x * y) * \mathfrak{p}\ (\lambda p{::}'a\ prime.\ -((x * y)^{\circ p})))$
for $x\ y$ :: $'a$::*nontrivial-factorial-idom p-adic-prod*
proof *transfer*
fix $X\ Y$ :: $'a$ *fls-pseq*
define d :: $'a$ *fls-pseq* $\Rightarrow$ $'a$ *prime* $\Rightarrow$ *int*
where $d \equiv \lambda X\ p.\ -\ (X^{\circ p})$
thus
$(X * \mathfrak{p}\ (d\ X)) * (Y * \mathfrak{p}\ (d\ Y)) \simeq_{\forall\, p}$
$(X * Y * \mathfrak{p}\ (d\ (X * Y)))$
using *times-fun-apply p-equal-fls-pseq-def normalized-depth-fls-pseq-product-equiv*
by *fastforce*
qed

context
fixes p :: $'a$::*nontrivial-factorial-idom prime*
begin

lemma *trivial-global-unfrmzr-pows-p-adic-prod*:
$f\ p = 0 \Longrightarrow (\mathfrak{p}\ f$ :: $'a$ *p-adic-prod*$) \simeq_p 1$
for f :: $'a$ *prime* $\Rightarrow$ *int*
by (*transfer*, *rule trivial-global-unfrmzr-pows-fls-pseq*, *simp*)

lemma *prod-w-trivial-global-unfrmzr-pows-p-adic-prod*:
$f\ p = 0 \Longrightarrow x * \mathfrak{p}\ f \simeq_p x$
for f :: $'a$ *prime* $\Rightarrow$ *int* **and** x :: $'a$ *p-adic-prod*
by (*transfer*, *rule prod-w-trivial-global-unfrmzr-pows-fls-pseq*, *simp*)

end

lemma *global-unfrmzr-pows-p-adic-prod-pow*:
$(\mathfrak{p}\ f$ :: $'a$ *p-adic-prod*$)\ \widehat{}\ n = (\mathfrak{p}\ (\lambda p.\ int\ n * f\ p))$
for f :: $'a$::*nontrivial-factorial-idom prime* $\Rightarrow$ *int*
by (
transfer,
metis globally-p-equal-fls-pseq-def p-equality-depth-fls-pseq.globally-p-equal-refl pow-global-unfrmzr-pows-fls-pseq
)
lemma *global-unfrmzr-pows-p-adic-prod-inv*:
$(\mathfrak{p}\ (-f)$ :: $'a$ *p-adic-prod*$) * (\mathfrak{p}\ f) = 1$
for f :: $'a$::*nontrivial-factorial-idom prime* $\Rightarrow$ *int*
by (
transfer,
metis globally-p-equal-fls-pseq-def p-equality-depth-fls-pseq.globally-p-equal-refl global-unfrmzr-pows-fls-pseq-inv
)

lemma *global-unfrmzr-pows-p-adic-prod-inverse*:
inverse ($\mathfrak{p}$ *f* :: *'a p-adic-prod*) = $\mathfrak{p}$ $(-f)$
for *f* :: *'a::nontrivial-factorial-unique-euclidean-bezout prime* ⇒ *int*
proof (*transfer*)
fix *f* :: *'a::nontrivial-factorial-unique-euclidean-bezout prime* ⇒ *int*
have $((\mathfrak{p}\ f)^{-1\forall p}) = (\mathfrak{p}\ (-\ f)$ :: *'a fls-pseq*)
using *global-unfrmzr-pows-fls-pseq-def*[*of f*] *global-unfrmzr-pows-fls-pseq-def*[*of* −*f*]
fls-pinv-X-intpow
by *auto*
thus
$((\mathfrak{p}\ f)^{-1\forall p}) \simeq_{\forall p}$
($\mathfrak{p}$ $(-\ f)$ :: *'a fls-pseq*)
by *auto*
qed

lemma *global-unfrmzr-pows-p-adic-prod-decomp*:
$x = ($
$(x * \mathfrak{p}\ (\lambda p$::*'a prime*. $-(x^{\circ p}))) *$
$\mathfrak{p}\ (\lambda p$::*'a prime*. $x^{\circ p})$
)
for *p* :: *'a::nontrivial-factorial-idom prime* **and** *x* :: *'a p-adic-prod*
by (
transfer,
metis global-unfrmzr-pows-fls-pseq-decomp p-equality-depth-fls-pseq.globally-p-equal-refl
globally-p-equal-fls-pseq-def
)

lemma *p-adic-prod-normalize-depthE*:
fixes *x* :: *'a::nontrivial-factorial-idom p-adic-prod*
obtains *x-0*
where ∀ *p*::*'a prime*. $x\text{-}0^{\circ p} = 0$
and $x = x\text{-}0 * \mathfrak{p}\ (\lambda p$::*'a prime*. $x^{\circ p})$
using *p-adic-prod-normalize-depth global-unfrmzr-pows-p-adic-prod-decomp*
by *blast*

global-interpretation *p-adic-prod-div-inv*:
global-p-equal-depth-div-inv
p-equal :: *'a::nontrivial-factorial-unique-euclidean-bezout prime* ⇒
'a p-adic-prod ⇒ *'a p-adic-prod* ⇒ *bool*
p-depth :: *'a prime* ⇒ *'a p-adic-prod* ⇒ *int*
p-restrict ::
'a p-adic-prod ⇒ (*'a prime* ⇒ *bool*) ⇒ *'a p-adic-prod*
proof
fix *p* :: *'a prime* **and** *x y* :: *'a p-adic-prod*
show $x\ /\ y = x * inverse\ y$ **by** *transfer simp*
show *inverse* (*inverse x*) = *x* **by** *transfer* (*rule global-pinverse-pinverse-fls-pseq*)
show *inverse* $(x * y)$ = *inverse x* ∗ *inverse y* **by** *transfer* (*rule global-pinverse-mult-fls-pseq*)

show (*inverse* $x \simeq_p 0$) = ($x \simeq_p 0$)
by *transfer* (*simp add*: *fls-pinv-equiv0-iff*)
show $x \neg\simeq_p 0 \Longrightarrow x / x \simeq_p 1$
by *transfer* (*simp add*: *pinverse-fls-mult-equiv1'*)
qed

5.5 Embeddings of the base ring, *nat*, *int*, and *rat*

5.5.1 Embedding of the base ring

abbreviation *p-adic-prod-consts* ≡ *global-p-depth-p-adic-prod.consts*
abbreviation *p-adic-prod-const* ≡ *global-p-depth-p-adic-prod.const*

lemma *p-depth-p-adic-prod-consts*:
(p-*adic-prod-consts* $f)^{\circ p} = ((f\ p)^{\circ p})$
for p :: *'a::nontrivial-factorial-idom prime* **and** f :: *'a prime* ⇒ *'a*
by (*simp add*: *p-depth-p-adic-prod.abs-eq flip*: *p-depth-const-def*)

lemmas *p-depth-p-adic-prod-const* = *p-depth-p-adic-prod-consts*[*of* - λp. -]

lemma *p-adic-prod-global-unfrmzr*:
($\mathfrak{p}$ (*1* :: *'a::nontrivial-factorial-idom prime* ⇒ *int*) :: *'a p-adic-prod*) =
p-adic-prod-consts Rep-prime
proof (*rule global-p-depth-p-adic-prod.global-imp-eq*, *standard*)
fix p :: *'a prime*
define $\mathfrak{p}$*1-fls-pseq* :: *'a fls-pseq* **and** $\mathfrak{p}$*1-pre* :: *'a p-adic-prod*
where $\mathfrak{p}$*1-fls-pseq* ≡ $\mathfrak{p}$ (*1* :: *'a prime* ⇒ *int*)
and $\mathfrak{p}$*1-pre* ≡ $\mathfrak{p}$ (*1* :: *'a prime* ⇒ *int*)
have
(*abs-p-adic-prod* $\mathfrak{p}$*1-fls-pseq*) $\simeq_p$
(*abs-p-adic-prod* (λp::*'a prime*. *fls-const* (*Rep-prime* p)))
unfolding $\mathfrak{p}$*1-fls-pseq-def p-equal-p-adic-prod.abs-eq*
by (*rule global-unfrmzr-fls-pseq*)
thus $\mathfrak{p}$*1-pre* $\simeq_p$ (*p-adic-prod-consts Rep-prime*)
unfolding $\mathfrak{p}$*1-fls-pseq-def* $\mathfrak{p}$*1-pre-def global-unfrmzr-pows-p-adic-prod-def* **by**
simp
qed

5.5.2 Embedding of the field of fractions of the base ring

global-interpretation *p-adic-prod-embeds*:
global-p-equal-div-inv-w-inj-idom-consts
p-equal :: *'a::nontrivial-factorial-unique-euclidean-bezout prime* ⇒
'a p-adic-prod ⇒ *'a p-adic-prod* ⇒ *bool*
p-restrict ::
'a p-adic-prod ⇒ (*'a prime* ⇒ *bool*) ⇒ *'a p-adic-prod*
λf. *abs-p-adic-prod* (λp. *fls-const* ($f\ p$))
..

global-interpretation *p-adic-prod-depth-embeds*:

global-p-equal-depth-div-inv-w-inj-index-consts
p-equal :: *'a::nontrivial-factorial-unique-euclidean-bezout prime* ⇒
'a p-adic-prod ⇒ *'a p-adic-prod* ⇒ *bool*
p-depth :: *'a prime* ⇒ *'a p-adic-prod* ⇒ *int*
p-restrict ::
'a p-adic-prod ⇒ (*'a prime* ⇒ *bool*) ⇒ *'a p-adic-prod*
λ*f*. *abs-p-adic-prod* (λ*p*. *fls-const* (*f p*))
Rep-prime
..

abbreviation *p-adic-prod-shift-p-depth* ≡ *p-adic-prod-depth-embeds.shift-p-depth*
abbreviation *p-adic-prod-of-fract* ≡ *p-adic-prod-embeds.const-of-fract*

lemma *p-adic-prod-of-fract-depth*:
(*p-adic-prod-of-fract* (*Fraction-Field.Fract a b*) :: *'a p-adic-prod*)$^{\circ p}$ =
($a^{\circ p}$) − ($b^{\circ p}$)
if $a \neq 0$ **and** $b \neq 0$
for *p* :: *'a::nontrivial-factorial-unique-euclidean-bezout prime*
and *a b* :: *'a*
using *that global-p-depth-p-adic-prod.p-equal-func-of-const-0*
p-adic-prod-div-inv.divide-equiv-0-iff p-adic-prod-embeds.const-of-fract-Fract
p-adic-prod-div-inv.divide-depth p-depth-p-adic-prod-const
by *metis*

Product of all p-adic embeddings of the field of fractions of the base ring.

abbreviation *p-adic-Fracts-prod* ::
'a::nontrivial-factorial-unique-euclidean-bezout p-adic-prod set
($\mathcal{Q}_{\forall\, p}$)
where $\mathcal{Q}_{\forall\, p}$ ≡ *p-adic-prod-embeds.range-const-of-fract*

5.5.3 Characteristic zero embeddings

class *nontrivial-factorial-idom-char-0* = *nontrivial-factorial-idom* + *semiring-char-0*
begin
subclass *ring-char-0* **..**
end

instance *int* :: *nontrivial-factorial-idom-char-0* **..**

class *nontrivial-factorial-unique-euclidean-bezout-char-0* =
nontrivial-factorial-unique-euclidean-bezout + *nontrivial-factorial-idom-char-0*

instance *int* :: *nontrivial-factorial-unique-euclidean-bezout-char-0* **..**

instance *p-adic-prod* :: (*nontrivial-factorial-idom-char-0*) *ring-char-0*
by (
standard, *standard*,
metis global-p-depth-p-adic-prod.const-of-nat global-p-depth-p-adic-prod.const-eq-iff
of-nat-eq-iff

)

global-interpretation *p-adic-prod-consts-char-0*:
global-p-equality-w-inj-consts-char-0
p-equal :: *'a::nontrivial-factorial-idom-char-0 prime* ⇒
'a p-adic-prod ⇒ *'a p-adic-prod* ⇒ *bool*
p-restrict ::
'a p-adic-prod ⇒ (*'a prime* ⇒ *bool*) ⇒ *'a p-adic-prod*
λf. *abs-p-adic-prod* (λp. *fls-const* (*f p*))
..

global-interpretation *p-adic-prod-div-inv-consts-char-0*:
global-p-equal-div-inv-w-inj-consts-char-0
p-equal :: *'a::nontrivial-factorial-unique-euclidean-bezout-char-0 prime* ⇒
'a p-adic-prod ⇒ *'a p-adic-prod* ⇒ *bool*
p-restrict ::
'a p-adic-prod ⇒ (*'a prime* ⇒ *bool*) ⇒ *'a p-adic-prod*
λf. *abs-p-adic-prod* (λp. *fls-const* (*f p*))
..

lemma *p-adic-prod-of-nat-depth*:
(*of-nat n* :: *'a p-adic-prod*)$^{\circ p}$ = ((*of-nat n* :: *'a*)$^{\circ p}$)
for *p* :: *'a::nontrivial-factorial-idom prime*
by (*metis global-p-depth-p-adic-prod.const-of-nat p-depth-p-adic-prod-const*)

lemma *p-adic-prod-of-int-depth*:
(*of-int z* :: *'a p-adic-prod*)$^{\circ p}$ = ((*of-int z* :: *'a*)$^{\circ p}$)
for *p* :: *'a::nontrivial-factorial-idom prime*
by (*metis global-p-depth-p-adic-prod.const-of-int p-depth-p-adic-prod-const*)

abbreviation *p-adic-prod-of-rat* ≡ *p-adic-prod-div-inv-consts-char-0.const-of-rat*

lemma *p-adic-prod-of-rat-depth*:
(*p-adic-prod-of-rat* (*Rat.Fract a b*) :: *'a p-adic-prod*)$^{\circ p}$ =
((*of-int a* :: *'a*)$^{\circ p}$) − ((*of-int b* :: *'a*)$^{\circ p}$)
if $a \neq 0$ **and** $b \neq 0$
for *p* :: *'a::nontrivial-factorial-unique-euclidean-bezout-char-0 prime*
and *a b* :: *int*
using *that p-adic-prod-consts-char-0.const-of-int-p-equal-0-iff*
p-adic-prod-div-inv.divide-equiv-0-iff p-adic-prod-div-inv-consts-char-0.const-of-rat-rat
p-adic-prod-div-inv.divide-depth p-adic-prod-of-int-depth
by *metis*

5.6 Topologies on products of p-adic fields

5.6.1 By local place

Completeness

abbreviation *p-adic-prod-p-open-nbhd* ≡ *global-p-depth-p-adic-prod.p-open-nbhd*

abbreviation *p-adic-prod-p-open-nbhds* ≡ *global-p-depth-p-adic-prod.p-open-nbhds*
abbreviation
p-adic-prod-p-limseq-condition ≡ *global-p-depth-p-adic-prod.p-limseq-condition*
abbreviation
p-adic-prod-p-cauchy-condition ≡ *global-p-depth-p-adic-prod.p-cauchy-condition*
abbreviation *p-adic-prod-p-limseq* ≡ *global-p-depth-p-adic-prod.p-limseq*
abbreviation *p-adic-prod-p-cauchy* ≡ *global-p-depth-p-adic-prod.p-cauchy*
abbreviation
p-adic-prod-p-open-nbhds-limseq ≡ *global-p-depth-p-adic-prod.p-open-nbhds-LIMSEQ*
abbreviation
p-adic-prod-local-p-open-nbhds ≡ *global-p-depth-p-adic-prod.local-p-open-nbhds*

lemma *abs-p-adic-prod-seq-cases* [*case-names abs-p-adic-prod*]:
C **if** $\bigwedge F.\ X = (\lambda n.\ \textit{abs-p-adic-prod}\ (F\ n)) \Longrightarrow C$
proof−
have $X = (\lambda n.\ \textit{abs-p-adic-prod}\ (\textit{rep-p-adic-prod}\ (X\ n)))$
using *rep-p-adic-prod-inverse* **by** *metis*
with *that* **show** C **by** *fast*
qed

lemma *abs-p-adic-prod-p-limseq-condition*:
p-adic-prod-p-limseq-condition p $(\lambda n.\ \textit{abs-p-adic-prod}\ (F\ n))$ $(\textit{abs-p-adic-prod}\ X)$
$n\ K$ =
fls-p-limseq-condition p $(\lambda n.\ F\ n\ p)$ $(X\ p)$ $n\ K$
unfolding *global-p-depth-p-adic-prod.p-limseq-condition-def p-depth-fls.p-limseq-condition-def*
by (*auto simp add*: *p-equal-p-adic-prod.abs-eq p-depth-p-adic-prod-diff-abs-eq*)

lemma *abs-p-adic-prod-p-limseq*:
p-adic-prod-p-limseq p $(\lambda n.\ \textit{abs-p-adic-prod}\ (F\ n))$ $(\textit{abs-p-adic-prod}\ X)$ =
fls-p-limseq p $(\lambda n.\ F\ n\ p)$ $(X\ p)$
using *abs-p-adic-prod-p-limseq-condition* **by** *fast*

lemma *abs-p-adic-prod-p-cauchy-condition*:
p-adic-prod-p-cauchy-condition p $(\lambda n.\ \textit{abs-p-adic-prod}\ (F\ n))$ $n\ K$ =
fls-p-cauchy-condition p $(\lambda n.\ F\ n\ p)$ $n\ K$
unfolding *global-p-depth-p-adic-prod.p-cauchy-condition-def p-depth-fls.p-cauchy-condition-def*
by (*auto simp add*: *p-equal-p-adic-prod.abs-eq p-depth-p-adic-prod-diff-abs-eq*)

lemma *abs-p-adic-prod-p-cauchy*:
p-adic-prod-p-cauchy p $(\lambda n.\ \textit{abs-p-adic-prod}\ (F\ n))$ = *fls-p-cauchy* p $(\lambda n.\ F\ n\ p)$
using *abs-p-adic-prod-p-cauchy-condition* **by** *blast*

lemma *p-adic-prod-p-restrict-cauchy-lim*:
p-adic-prod-p-limseq q
$(\lambda n.\ \textit{p-restrict}\ (X\ n)\ ((=)\ p))$
$(\textit{abs-p-adic-prod}\ (\lambda q.$
if $q = p$ *then* *fls-p-cauchy-lim* p $(\lambda n.\ \textit{rep-p-adic-prod}\ (X\ n)\ p)$ *else* 0
$))$
if *p-adic-prod-p-cauchy* $p\ X$

proof (*cases X rule*: *abs-p-adic-prod-seq-cases*)
  case (*abs-p-adic-prod F*)
  define *limval* **where**
    limval ≡
      abs-p-adic-prod (λq.
        if $q = p$ *then fls-p-cauchy-lim p* (λn. *rep-p-adic-prod* (*X n*) *p*) *else 0*
)
  show *p-adic-prod-p-limseq q* (λn. *X n prestrict* (=) *p*) *limval*
  proof (*cases* $p = q$)
    case *True*
    with *that abs-p-adic-prod* **have** *F*: *fls-p-cauchy q* (λn. *F n q*)
      using *abs-p-adic-prod-p-cauchy* **by** *blast*
    have *fls-p-limseq q* (λn. *F n q*) (*fls-p-cauchy-lim q* (λn. *F n q*))
      using *F fls-p-cauchy-limseq* **by** *blast*
    moreover have
      fls-p-cauchy-lim q (λn. *rep-p-adic-prod* (*abs-p-adic-prod* (*F n*)) *q*) =
        fls-p-cauchy-lim q (λn. *F n q*)
      by (
        intro fls-p-cauchy-lim-unique F, *clarify*,
      use abs-p-adic-prod p-adic-prod-globally-p-equal-self globally-p-equal-fls-pseq-def
        in *auto*
)
    ultimately have *p-adic-prod-p-limseq q X limval*
      using *True limval-def abs-p-adic-prod abs-p-adic-prod-p-limseq* **by** *force*
    moreover from *True* **have** ∀ *n*. (*p-restrict* (*X n*) ((=) *p*)) $\simeq_q$ (*X n*)
      using *global-p-depth-p-adic-prod.p-restrict-equiv* **by** *blast*
    ultimately show *?thesis*
      using *global-p-depth-p-adic-prod.p-limseq-p-cong*[*of 0 q - X limval limval*] **by**
auto
  next
    case *False*
    with *abs-p-adic-prod limval-def* **show** *?thesis*
      using *global-p-depth-p-adic-prod.p-restrict-equiv0*[*of* (=) *p q*]
            p-equal-p-adic-prod-abs-eq0[*of q*]
            global-p-depth-p-adic-prod.p-limseq-p-cong[*of*
              0 q λn. (*X n*) *prestrict* (=) *p 0 limval*
            ]
            global-p-depth-p-adic-prod.p-limseq-0-iff[*of q limval*]
      by *fastforce*
  qed
qed

global-interpretation *p-complete-p-adic-prod*:
  p-complete-global-p-equal-depth
    p-equal :: *'a*::*nontrivial-euclidean-ring-cancel prime* ⇒
      'a p-adic-prod ⇒ *'a p-adic-prod* ⇒ *bool*
    p-restrict ::
      'a p-adic-prod ⇒ (*'a prime* ⇒ *bool*) ⇒ *'a p-adic-prod*

p-depth :: *'a prime* ⇒ *'a p-adic-prod* ⇒ *int*
proof
fix *p* :: *'a prime* **and** *X* :: *nat* ⇒ *'a p-adic-prod*
define *res-X* :: *nat* ⇒ *'a p-adic-prod* **and** *limval* :: *'a p-adic-prod*
where *res-X* ≡ *λn. p-restrict* (*X n*) ((=) *p*)
and
limval ≡
abs-p-adic-prod (*λq.*
if q = *p then fls-p-cauchy-lim p* (*λn. rep-p-adic-prod* (*X n*) *p*) *else 0*
)
moreover assume *p-adic-prod-p-cauchy p X*
ultimately have ∀ *q. p-adic-prod-p-limseq q res-X limval*
using *p-adic-prod-p-restrict-cauchy-lim* **by** *blast*
hence *p-adic-prod-p-open-nbhds-limseq res-X limval*
using *global-p-depth-p-adic-prod.globally-limseq-iff-locally-limseq* **by** *fast*
thus *global-p-depth-p-adic-prod.p-open-nbhds-convergent res-X*
using *global-p-depth-p-adic-prod.t2-p-open-nbhds.convergent-def* **by** *auto*
qed

global-interpretation *p-complete-p-adic-prod-div-inv*:
p-complete-global-p-equal-depth-div-inv
p-equal :: *'a::nontrivial-factorial-unique-euclidean-bezout prime* ⇒
'a p-adic-prod ⇒ *'a p-adic-prod* ⇒ *bool*
p-depth :: *'a prime* ⇒ *'a p-adic-prod* ⇒ *int*
p-restrict ::
'a p-adic-prod ⇒ (*'a prime* ⇒ *bool*) ⇒ *'a p-adic-prod*
..

lemmas *p-adic-prod-hensel* = *p-complete-p-adic-prod-div-inv.hensel*

5.6.2 By bounded global depth

The metric

instantiation *p-adic-prod* :: (*nontrivial-factorial-idom*) *metric-space*
begin

definition *dist* = *global-p-depth-p-adic-prod.bdd-global-dist*
declare *dist-p-adic-prod-def* [*simp*]

definition *uniformity* = *global-p-depth-p-adic-prod.bdd-global-uniformity*
declare *uniformity-p-adic-prod-def* [*simp*]

definition *open-p-adic-prod* :: *'a p-adic-prod set* ⇒ *bool*
where
open-p-adic-prod U = (∀ *x*∈*U*.
eventually (λ(*x'*, *y*). *x'* = *x* ⟶ *y* ∈ *U*) *uniformity*
)

instance

by (
standard,
simp-all add: *global-p-depth-p-adic-prod.bdd-global-uniformity-def open-p-adic-prod-def*
global-p-depth-p-adic-prod.metric-space-by-bdd-depth.dist-triangle2
)

end

abbreviation *p-adic-prod-global-depth* $\equiv$ *global-p-depth-p-adic-prod.bdd-global-depth*

lemma *bdd-global-depth-p-adic-prod-abs-eq*:
p-adic-prod-global-depth (*abs-p-adic-prod X*) = *fls-pseq-bdd-global-depth X*
proof (*cases X* $\simeq_{\forall\, p}$ *0*)
case *True*
hence *abs-p-adic-prod X* = *0*
using *p-adic-prod-lift-globally-p-equal zero-p-adic-prod.abs-eq* **by** *metis*
with *True* **show** *?thesis* **by** *force*
next
case *False*
hence *abs-p-adic-prod X* $\neq$ *0*
using *p-adic-prod.abs-eq-iff zero-p-adic-prod.abs-eq* **by** *metis*
hence $\neg$ *abs-p-adic-prod X* $\simeq_{\forall\, p}$ *0*
using *global-p-depth-p-adic-prod.global-eq-iff* **by** *auto*
hence
p-adic-prod-global-depth (*abs-p-adic-prod X*) =
(*INF p*$\in${*p*::'*a prime. abs-p-adic-prod X* $\neg\simeq_p$ *0*}.
nat (((*abs-p-adic-prod X*)$^{\circ p}$) + *1*))
using *global-p-depth-p-adic-prod.bdd-global-depthD-as-image*[*of abs-p-adic-prod X*]
by *fastforce*
also have
... = (*INF p*$\in${*p*::'*a prime. X* $\neg\simeq_p$ *0*}.
nat (((*abs-p-adic-prod X*)$^{\circ p}$) + *1*))
using *p-equal-p-adic-prod.abs-eq zero-p-adic-prod.abs-eq* **by** *metis*
finally show *?thesis*
using *False p-depth-p-adic-prod.abs-eq*[*of - X*]
p-equality-depth-fls-pseq.bdd-global-depthD-as-image
by *force*
qed

lemma *dist-p-adic-prod-abs-eq*:
dist (*abs-p-adic-prod X*) (*abs-p-adic-prod Y*) = *fls-pseq-bdd-global-dist X Y*
proof (*cases X* $\simeq_{\forall\, p}$ *Y*)
case *True*
hence *abs-p-adic-prod X* = *abs-p-adic-prod Y* **using** *p-adic-prod-lift-globally-p-equal* **by** *metis*
with *True* **show** *?thesis* **by** *fastforce*
next
case *False*

from *False* **have** *abs-p-adic-prod X* $\neq$ *abs-p-adic-prod Y*
  using *p-adic-prod.abs-eq-iff* **by** *metis*
hence $\neg$ *abs-p-adic-prod X* $\simeq_{\forall\, p}$ *abs-p-adic-prod Y*
  using *global-p-depth-p-adic-prod.global-eq-iff* **by** *auto*
hence
  dist (*abs-p-adic-prod X*) (*abs-p-adic-prod Y*) =
   inverse (*2* ^ *p-adic-prod-global-depth* (*abs-p-adic-prod X* − *abs-p-adic-prod Y*))
  using *global-p-depth-p-adic-prod.bdd-global-distD* **by** *auto*
also have ... = *inverse* (*2* ^ *p-adic-prod-global-depth* (*abs-p-adic-prod* (*X* − *Y*)))
  using *minus-p-adic-prod.abs-eq* **by** *metis*
finally show *?thesis*
  using *False bdd-global-depth-p-adic-prod-abs-eq p-equality-depth-fls-pseq.bdd-global-distD*
  by *fastforce*
qed

lemma *p-adic-prod-metric-is-finer*:
  generate-topology p-adic-prod-p-open-nbhds U $\Longrightarrow$ *open U*
proof (*induct U rule*: *generate-topology.induct*)
  case (*Basis U*) **show** *open U*
  proof (*intro Elementary-Metric-Spaces.openI*)
    fix *u* **assume** *u* $\in$ *U*
    moreover from *Basis* **obtain** *p n x* **where** *U* = *p-adic-prod-p-open-nbhd p n x*
      by *fast*
    ultimately have *U*: *U* = *p-adic-prod-p-open-nbhd p n u*
      using *global-p-depth-p-adic-prod.p-open-nbhd-circle-multicentre* **by** *fast*
    have *ball u* (*inverse* (*2* ^ *nat n*)) $\subseteq$ *U*
      unfolding *U global-p-depth-p-adic-prod.p-open-nbhd-eq-circle ball-def*
      using *global-p-depth-p-adic-prod.bdd-global-dist-sym*[*of u*]
        global-p-depth-p-adic-prod.bdd-global-dist-less-pow2-iff[*of - u nat n*]
      by *auto*
    thus $\exists$ *e*>*0*. *ball u e* $\subseteq$ *U* **by** *force*
  qed
qed (*simp*, *fast*, *fast*)

lemma *p-adic-prod-metric-is-finer-than-purely-local*:
  generate-topology (*p-adic-prod-local-p-open-nbhds p*) *U* $\Longrightarrow$
    open U
  for *p* :: *'a*::*nontrivial-factorial-idom prime*
  and *U* :: *'a p-adic-prod set*
  using *global-p-depth-p-adic-prod.local-p-open-nbhds-are-coarser p-adic-prod-metric-is-finer*
  by *blast*

lemma *p-adic-prod-limseq-abs-eq*:
  (λ*n*. *abs-p-adic-prod* (*X n*)) $\longrightarrow$ *abs-p-adic-prod x*
    $\longleftrightarrow$
    ($\forall$ *e*>*0*. $\forall_F$ *n in sequentially*. *fls-pseq-bdd-global-dist* (*X n*) *x* < *e*)
proof−

have
∀ n. dist (abs-p-adic-prod (X n)) (abs-p-adic-prod x) =
fls-pseq-bdd-global-dist (X n) x
using *dist-p-adic-prod-abs-eq[of X - x]* **by** *blast*
thus *?thesis*
using *tendsto-iff[of λn. abs-p-adic-prod (X n) abs-p-adic-prod x]* **by** *presburger*
qed

lemma *p-adic-prod-bdd-metric-LIMSEQ*:
X ⟶ x = global-p-depth-p-adic-prod.metric-space-by-bdd-depth.LIMSEQ X x
for *X :: nat ⇒ 'a::nontrivial-factorial-idom p-adic-prod* **and** *x :: 'a p-adic-prod*
unfolding *tendsto-iff global-p-depth-p-adic-prod.metric-space-by-bdd-depth.tendsto-iff dist-p-adic-prod-def*
by *fast*

lemma *p-adic-prod-global-depth-set-lim-closed*:
$x \in \mathcal{P}_{\forall\, p}{}^{n}$
if *lim* : $X \longrightarrow x$
and *depth*: $\forall_F$ *k in sequentially.* $X\ k \in \mathcal{P}_{\forall\, p}{}^{n}$
for *X :: nat ⇒ 'a::nontrivial-factorial-unique-euclidean-bezout p-adic-prod*
and *x :: 'a p-adic-prod*
using *that p-adic-prod-bdd-metric-LIMSEQ p-adic-prod-depth-embeds.global-depth-set-LIMSEQ-closed*
by *fastforce*

lemma *p-adic-prod-metric-ball-multicentre*:
ball y e = ball x e **if** *y ∈ ball x e*
for *x y :: 'a::nontrivial-factorial-idom p-adic-prod*
using *that global-p-depth-p-adic-prod.bdd-global-dist-ball-multicentre*
unfolding *ball-def dist-p-adic-prod-def*
by *blast*

lemma *p-adic-prod-metric-ball-at-0*:
ball (0::'a::nontrivial-factorial-idom p-adic-prod) (inverse (2 ^ n)) = global-depth-set (int n)
using *global-p-depth-p-adic-prod.bdd-global-dist-ball-at-0*
unfolding *ball-def dist-p-adic-prod-def global-depth-set-p-adic-prod-def*
by *auto*

lemma *p-adic-prod-metric-ball-UNIV*:
ball x e = UNIV **if** *e > 1*
for *x :: 'a::nontrivial-factorial-idom p-adic-prod*
using *that global-p-depth-p-adic-prod.bdd-global-dist-ball-UNIV*
unfolding *ball-def dist-p-adic-prod-def*
by *blast*

lemma *p-adic-prod-metric-ball-at-0-normalize*:
ball (0::'a::nontrivial-factorial-idom p-adic-prod) e =
global-depth-set ⌊log 2 (inverse e)⌋
if *e > 0* **and** *e ≤ 1*

using *that global-p-depth-p-adic-prod.bdd-global-dist-ball-at-0-normalize*
unfolding *ball-def dist-p-adic-prod-def global-depth-set-p-adic-prod-def*
by *blast*

lemma *p-adic-prod-metric-ball-translate*:
ball x e = *x* +*o ball 0 e* **for** *x* :: *'a::nontrivial-factorial-idom p-adic-prod*
using *global-p-depth-p-adic-prod.bdd-global-dist-ball-translate*
unfolding *ball-def dist-p-adic-prod-def*
by *blast*

lemma *p-adic-prod-left-translate-metric-continuous*:
continuous-on UNIV ((+) *x*) **for** *x* :: *'a::nontrivial-factorial-idom p-adic-prod*
proof
fix *e* :: *real* **and** *z* :: *'a p-adic-prod*
assume *e*: $e > 0$
moreover have $\forall y.$ *dist y z* $< e \longrightarrow$ *dist* $(x + y)$ $(x + z) \leq e$
using *global-p-depth-p-adic-prod.bdd-global-dist-left-translate-continuous*[*of*
- - e x
]
unfolding *dist-p-adic-prod-def*
by *fastforce*
ultimately show
$\exists d>0. \forall x'\in UNIV.$
dist x' $z < d \longrightarrow$ *dist* $(x + x')$ $(x + z) \leq e$
by *auto*
qed

lemma *p-adic-prod-right-translate-metric-continuous*:
continuous-on UNIV $(\lambda z.\ z + x)$ **for** *x* :: *'a::nontrivial-factorial-idom p-adic-prod*
proof−
have $(\lambda z.\ z + x) = (+)\ x$ **by** (*auto simp add*: *add.commute*)
thus *?thesis* **using** *p-adic-prod-left-translate-metric-continuous* **by** *metis*
qed

lemma *p-adic-prod-left-bdd-mult-bdd-metric-continuous*:
continuous-on UNIV $((*)\ x)$
if *bdd*:
$\forall p$::*'a prime*. $x \neg\simeq_p 0 \longrightarrow$
$(x^{\circ p}) \geq n$
for *x* :: *'a::nontrivial-factorial-idom p-adic-prod*
proof
fix *e* :: *real* **and** *z* :: *'a p-adic-prod*
assume *e*: $e > 0$
moreover define *d* **where** $d \equiv min\ (e * 2\ powi\ (n - 1))\ 1$
ultimately have $\forall y.$ *dist y z* $< d \longrightarrow$ *dist* $(x * y)$ $(x * z) \leq e$
using *bdd global-p-depth-p-adic-prod.bdd-global-dist-bdd-mult-continuous* **by** *fast-force*
moreover from *e d-def* **have** $d > 0$ **by** *auto*
ultimately show

$\exists\, d{>}0.\ \forall\, y{\in}UNIV.\ dist\ y\ z < d \longrightarrow dist\ (x * y)\ (x * z) \leq e$
by *blast*
qed

lemma *p-adic-prod-bdd-metric-limseq-bdd-mult*:
$(\lambda k.\ w * X\ k) \longrightarrow (w * x)$
if *bdd*:
$\forall\, p{::}'a\ prime.\ w \neg\simeq_p 0 \longrightarrow$
$(w^{\circ p}) \geq n$
and *seq*: $X \longrightarrow x$
for $w\ x$ :: $'a$::*nontrivial-factorial-idom p-adic-prod* **and** X :: $nat \Rightarrow\ 'a$ *p-adic-prod*
using *bdd seq p-adic-prod-left-bdd-mult-bdd-metric-continuous continuous-on-tendsto-compose*
by *fastforce*

lemma *p-adic-prod-nonpos-global-depth-set-open*:
$ball\ x\ 1 \subseteq (\mathcal{P}_{\forall\, p}{}^{n})$
if $n \leq 0$ **and** $x \in (\mathcal{P}_{\forall\, p}{}^{n})$
for x :: $'a$::*nontrivial-factorial-idom p-adic-prod*
proof−
have $ball\ x\ 1 = x +o\ ball\ 0\ 1$ **using** *p-adic-prod-metric-ball-translate* **by** *blast*
also have $\ldots = x +o\ (\mathcal{O}_{\forall\, p})$
using *p-adic-prod-metric-ball-at-0*[*of 0*] **by** *auto*
finally show *?thesis*
using *that global-p-depth-p-adic-prod.global-depth-set-elt-set-plus-closed*[*of n 0*]
by *auto*
qed

lemma *p-adic-prod-global-depth-set-open*:
open $((\mathcal{P}_{\forall\, p}{}^{n})$ :: $'a$::*nontrivial-factorial-idom p-adic-prod set*)
proof (*cases* $n \geq 0$)
case *True*
hence $(\mathcal{P}_{\forall\, p}{}^{n}) = ball\ (0{::}'a\ p\text{-}adic\text{-}prod)\ (inverse\ (2\ \hat{}\ nat\ n))$
using *p-adic-prod-metric-ball-at-0* **by** *fastforce*
thus *?thesis* **by** *simp*
next
case *False* **thus** *?thesis*
using *p-adic-prod-nonpos-global-depth-set-open*[*of n*]
Elementary-Metric-Spaces.openI[*of*
$(\mathcal{P}_{\forall\, p}{}^{n})$ :: $'a$ *p-adic-prod set*
]
by *force*
qed

lemma *p-adic-prod-ball-abs-eq*:
abs-p-adic-prod ‘ $\{Y.\ fls\text{-}pseq\text{-}bdd\text{-}global\text{-}dist\ X\ Y < e\} =$
ball (*abs-p-adic-prod X*) *e*
for x :: $'a$::*nontrivial-factorial-idom p-adic-prod*
proof−
define $B\ B'$

where $B \equiv \{Y.$ *fls-pseq-bdd-global-dist* $X\ Y < e\}$
and $B' \equiv$ *ball* (*abs-p-adic-prod* X) e
moreover have $\bigwedge y.\ y \in B' \Longrightarrow y \in$ *abs-p-adic-prod* ‘ B
proof−
fix y **assume** y: $y \in B'$
show $y \in$ *abs-p-adic-prod* ‘ B
proof (*cases* y)
case (*abs-p-adic-prod* Y)
moreover from *this* B'*-def* y **have** *fls-pseq-bdd-global-dist* $X\ Y < e$
using *mem-ball dist-p-adic-prod-abs-eq* **by** *metis*
ultimately show *?thesis* **using** *B-def* **by** *blast*
qed
qed
ultimately show *abs-p-adic-prod* ‘ $B = B'$
using *dist-p-adic-prod-abs-eq*[*of* X] **by** *fastforce*
qed

Completeness

abbreviation *p-adic-prod-globally-cauchy* ≡ *global-p-depth-p-adic-prod.globally-cauchy*
abbreviation
p-adic-prod-global-cauchy-condition ≡ *global-p-depth-p-adic-prod.global-cauchy-condition*

lemma *p-adic-prod-global-cauchy-condition-abs-eq*:
p-adic-prod-global-cauchy-condition ($\lambda n.$ *abs-p-adic-prod* ($X\ n$)) =
fls-pseq-global-cauchy-condition X
unfolding *global-p-depth-p-adic-prod.global-cauchy-condition-def*
p-equality-depth-fls-pseq.global-cauchy-condition-def
by (
standard, *standard*,
metis p-equal-p-adic-prod.abs-eq p-depth-p-adic-prod-diff-abs-eq
)

lemma *p-adic-prod-globally-cauchy-abs-eq*:
p-adic-prod-globally-cauchy ($\lambda n.$ *abs-p-adic-prod* ($X\ n$)) = *fls-pseq-globally-cauchy*
X
using *p-adic-prod-global-cauchy-condition-abs-eq* **by** *metis*

lemma *p-adic-prod-globally-cauchy-vs-bdd-metric-Cauchy*:
p-adic-prod-globally-cauchy X = *Cauchy* X
for X :: *nat* ⇒ ′*a*::*nontrivial-factorial-idom p-adic-prod*
using *global-p-depth-p-adic-prod.globally-cauchy-vs-bdd-metric-Cauchy*
unfolding *global-p-depth-p-adic-prod.metric-space-by-bdd-depth.Cauchy-def Cauchy-def*
by *auto*

abbreviation
p-adic-prod-cauchy-lim X ≡
abs-p-adic-prod ($\lambda p.$
fls-condition-lim p
($\lambda n.$ *reduced-rep-p-adic-prod* ($X\ n$) p)

(*p-adic-prod-global-cauchy-condition X*)
)

lemma *p-adic-prod-bdd-metric-complete'*:
X $\longrightarrow$ *p-adic-prod-cauchy-lim X* **if** *Cauchy X*
proof (*cases X rule*: *p-adic-prod-seq-reduced-cases*)
case (*abs-p-adic-prod F*) **with** *that* **show** *?thesis*
using *p-adic-prod-globally-cauchy-vs-bdd-metric-Cauchy*[*of* λn. *abs-p-adic-prod* (*F n*)]
p-adic-prod-globally-cauchy-abs-eq fls-pseq-cauchy-condition-lim
p-adic-prod-global-cauchy-condition-abs-eq[*of F*] *abs-p-adic-prod-inverse*[*of F -*]
p-adic-prod-limseq-abs-eq[*of F*]
by *fastforce*
qed

lemma *p-adic-prod-bdd-metric-complete*:
complete (*UNIV* :: *'a::nontrivial-euclidean-ring-cancel p-adic-prod set*)
using *p-adic-prod-bdd-metric-complete' completeI* **by** *blast*

5.7 Hiding implementation details

lifting-update *p-adic-prod.lifting*
lifting-forget *p-adic-prod.lifting*

5.8 The subring of adelic integers

5.8.1 Type definition as a subtype of adeles

typedef (**overloaded**) *'a adelic-int* =
$\mathcal{O}_{\forall p}$ :: *'a::nontrivial-factorial-idom p-adic-prod set*
using *global-p-depth-p-adic-prod.global-depth-set-0* **by** *auto*

lemma *Abs-adelic-int-inverse'*:
Rep-adelic-int (*Abs-adelic-int x*) = *x* **if** $x \in \mathcal{O}_{\forall p}$
using *that Abs-adelic-int-inverse* **by** *fast*

lemma *Abs-adelic-int-cases* [*case-names Abs-adelic-int*, *cases type*: *adelic-int*]:
P **if**
$\bigwedge x.\ z = $ *Abs-adelic-int x* $\Longrightarrow$
$x \in \mathcal{O}_{\forall p} \Longrightarrow P$
proof−
define *x* **where** $x \equiv$ *Rep-adelic-int z*
hence *z* = *Abs-adelic-int x* **using** *Rep-adelic-int-inverse* **by** *metis*
moreover from *x-def* **have** $x \in \mathcal{O}_{\forall p}$ **using** *Rep-adelic-int* **by** *blast*
ultimately show *P* **using** *that* **by** *fast*
qed

lemmas *two-Abs-adelic-int-cases* = *Abs-adelic-int-cases*[*case-product Abs-adelic-int-cases*]
and *three-Abs-adelic-int-cases* =

Abs-adelic-int-cases[*case-product Abs-adelic-int-cases*[*case-product Abs-adelic-int-cases*]]

lemma *adelic-int-seq-cases* [*case-names Abs-adelic-int*]:
fixes $X :: nat \Rightarrow 'a::nontrivial\text{-}factorial\text{-}idom\ adelic\text{-}int$
obtains $F :: nat \Rightarrow 'a\ p\text{-}adic\text{-}prod$
where $X = (\lambda n.\ Abs\text{-}adelic\text{-}int\ (F\ n))$
and $range\ F \subseteq \mathcal{O}_{\forall\ p}$
proof−
define F **where** $F \equiv \lambda n.\ Rep\text{-}adelic\text{-}int\ (X\ n)$
hence $X = (\lambda n.\ Abs\text{-}adelic\text{-}int\ (F\ n))$
using *Rep-adelic-int-inverse* **by** *metis*
moreover from *F-def* **have** $range\ F \subseteq \mathcal{O}_{\forall\ p}$
using *Rep-adelic-int* **by** *blast*
ultimately show *thesis* **using** *that* **by** *presburger*
qed

5.8.2 Algebraic instantiaions

instantiation *adelic-int* :: (*nontrivial-factorial-idom*) *comm-ring-1*
begin

definition $0 = Abs\text{-}adelic\text{-}int\ 0$

lemma *zero-adelic-int-rep-eq*: $Rep\text{-}adelic\text{-}int\ 0 = 0$
using *global-p-depth-p-adic-prod.global-depth-set-0*[*of 0*] *Abs-adelic-int-inverse*
by (*simp add*: *zero-adelic-int-def*)

definition $1 \equiv Abs\text{-}adelic\text{-}int\ 1$

lemma *one-adelic-int-rep-eq*: $Rep\text{-}adelic\text{-}int\ 1 = 1$
using *global-p-depth-p-adic-prod.global-depth-set-1 Abs-adelic-int-inverse*
by (*simp add*: *one-adelic-int-def*)

definition $x + y = Abs\text{-}adelic\text{-}int\ (Rep\text{-}adelic\text{-}int\ x + Rep\text{-}adelic\text{-}int\ y)$

lemma *plus-adelic-int-rep-eq*: $Rep\text{-}adelic\text{-}int\ (a + b) = Rep\text{-}adelic\text{-}int\ a + Rep\text{-}adelic\text{-}int\ b$
using *Rep-adelic-int global-p-depth-p-adic-prod.global-depth-set-add Abs-adelic-int-inverse*
unfolding *plus-adelic-int-def*
by *fastforce*

lemma *plus-adelic-int-abs-eq*:
$x \in \mathcal{O}_{\forall\ p} \Longrightarrow$
$y \in \mathcal{O}_{\forall\ p} \Longrightarrow$
$Abs\text{-}adelic\text{-}int\ x + Abs\text{-}adelic\text{-}int\ y = Abs\text{-}adelic\text{-}int\ (x + y)$
using *Abs-adelic-int-inverse*[*of x*] *Abs-adelic-int-inverse*[*of y*]
unfolding *plus-adelic-int-def*
by *simp*

definition $-x = $ *Abs-adelic-int* (− *Rep-adelic-int x*)

lemma *uminus-adelic-int-rep-eq*: *Rep-adelic-int* (− *x*) = − *Rep-adelic-int x*
using *Rep-adelic-int global-p-depth-p-adic-prod.global-depth-set-uminus*
Abs-adelic-int-inverse
unfolding *uminus-adelic-int-def*
by *auto*

lemma *uminus-adelic-int-abs-eq*:
$x \in \mathcal{O}_{\forall\, p} \Longrightarrow$ − *Abs-adelic-int x* = *Abs-adelic-int* (− *x*)
using *Abs-adelic-int-inverse*[*of x*] **unfolding** *uminus-adelic-int-def* **by** *simp*

definition *minus-adelic-int* ::
'a adelic-int ⇒ *'a adelic-int* ⇒ *'a adelic-int*
where *minus-adelic-int* ≡ (λ*x y*. *x* + − *y*)

lemma *minus-adelic-int-rep-eq*: *Rep-adelic-int* (*x* − *y*) = *Rep-adelic-int x* − *Rep-adelic-int*
y
using *plus-adelic-int-rep-eq uminus-adelic-int-rep-eq* **by** (*simp add*: *minus-adelic-int-def*)

lemma *minus-adelic-int-abs-eq*:
$x \in \mathcal{O}_{\forall\, p} \Longrightarrow$
$y \in \mathcal{O}_{\forall\, p} \Longrightarrow$
Abs-adelic-int x − *Abs-adelic-int y* = *Abs-adelic-int* (*x* − *y*)
using *global-p-depth-p-adic-prod.global-depth-set-uminus*[*of y*]
by (*simp add*:
minus-adelic-int-def uminus-adelic-int-abs-eq
plus-adelic-int-abs-eq
)

definition $x * y = $ *Abs-adelic-int* (*Rep-adelic-int x* ∗ *Rep-adelic-int y*)

lemma *times-adelic-int-rep-eq*: *Rep-adelic-int* (*x* ∗ *y*) = *Rep-adelic-int x* ∗ *Rep-adelic-int*
y
using *Rep-adelic-int global-p-depth-p-adic-prod.global-depth-set-times Abs-adelic-int-inverse*
unfolding *times-adelic-int-def*
by *force*

lemma *times-adelic-int-abs-eq*:
$x \in \mathcal{O}_{\forall\, p} \Longrightarrow$
$y \in \mathcal{O}_{\forall\, p} \Longrightarrow$
Abs-adelic-int x ∗ *Abs-adelic-int y* = *Abs-adelic-int* (*x* ∗ *y*)
using *Abs-adelic-int-inverse*[*of x*] *Abs-adelic-int-inverse*[*of y*]
unfolding *times-adelic-int-def*
by *simp*

instance
proof

fix *a b c* :: *'a adelic-int*

show $a + b = b + a$ **by** (*cases a*, *cases b*) (*simp add*: *plus-adelic-int-abs-eq add.commute*)
show $0 + a = a$
using *global-p-depth-p-adic-prod.global-depth-set-0 plus-adelic-int-abs-eq*[*of 0*]
unfolding *zero-adelic-int-def*
by (*cases a*) *auto*
show $1 * a = a$
using *global-p-depth-p-adic-prod.global-depth-set-1 times-adelic-int-abs-eq*[*of 1*]
by (*cases a*) (*simp add*: *one-adelic-int-def*)
show $- a + a = 0$
using *global-p-depth-p-adic-prod.global-depth-set-uminus uminus-adelic-int-abs-eq plus-adelic-int-abs-eq*
unfolding *zero-adelic-int-def*
by (*cases a*) *fastforce*
show $a - b = a + - b$ **by** (*simp add*: *minus-adelic-int-def*)
show $a * b = b * a$ **by** (*cases a*, *cases b*) (*simp add*: *times-adelic-int-abs-eq mult.commute*)

show $a + b + c = a + (b + c)$
proof (*cases a b c rule*: *three-Abs-adelic-int-cases*)
case (*Abs-adelic-int-Abs-adelic-int-Abs-adelic-int x y z*) **thus** *?thesis*
using *global-p-depth-p-adic-prod.global-depth-set-add*[*of x*]
global-p-depth-p-adic-prod.global-depth-set-add[*of y*]
by (*simp add*: *plus-adelic-int-abs-eq add.assoc*)
qed

show $a * b * c = a * (b * c)$
proof (*cases a b c rule*: *three-Abs-adelic-int-cases*)
case (*Abs-adelic-int-Abs-adelic-int-Abs-adelic-int x y z*) **thus** $a * b * c = a * (b * c)$
using *global-p-depth-p-adic-prod.global-depth-set-times*[*of 0 x*]
global-p-depth-p-adic-prod.global-depth-set-times[*of 0 y*]
by (*simp add*: *times-adelic-int-abs-eq mult.assoc*)
qed

show $(a + b) * c = a * c + b * c$
proof (*cases a b c rule*: *three-Abs-adelic-int-cases*)
case (*Abs-adelic-int-Abs-adelic-int-Abs-adelic-int x y z*) **thus** $(a + b) * c = a * c + b * c$
using *global-p-depth-p-adic-prod.global-depth-set-add*[*of x*]
global-p-depth-p-adic-prod.global-depth-set-times[*of 0 x*]
global-p-depth-p-adic-prod.global-depth-set-times[*of 0 y*]
by (*simp add*: *plus-adelic-int-abs-eq times-adelic-int-abs-eq distrib-right*)
qed

show (*0::'a adelic-int*) $\neq$ *1*

using *Abs-adelic-int-inject*[*of 0*::*'a p-adic-prod 1*]
global-p-depth-p-adic-prod.global-depth-set-0
global-p-depth-p-adic-prod.global-depth-set-1
unfolding *zero-adelic-int-def one-adelic-int-def*
by *auto*

qed

end

lemma *pow-adelic-int-rep-eq*: *Rep-adelic-int* $(x \mathbin{\widehat{}} n)$ = (*Rep-adelic-int* x) $\widehat{}$ n
by (*induct n*, *simp-all add*: *one-adelic-int-rep-eq times-adelic-int-rep-eq*)

lemma *pow-adelic-int-abs-eq*:
(*Abs-adelic-int* x) $\widehat{}$ n = *Abs-adelic-int* $(x \mathbin{\widehat{}} n)$ **if** $x \in \mathcal{O}_{\forall\, p}$
using *that global-p-depth-p-adic-prod.global-depth-set-0-pow times-adelic-int-abs-eq*
by (*induct n*, *simp add*: *one-adelic-int-def*, *fastforce*)

5.8.3 Equivalence and depth relative to a prime

overloading
p-equal-adelic-int ≡
p-equal :: *'a*::*nontrivial-factorial-idom prime* ⇒
'a adelic-int ⇒ *'a adelic-int* ⇒ *bool*
p-restrict-adelic-int ≡
p-restrict :: *'a adelic-int* ⇒
(*'a prime* ⇒ *bool*) ⇒ *'a adelic-int*
p-depth-adelic-int ≡
p-depth :: *'a*::*nontrivial-factorial-idom prime* ⇒ *'a adelic-int* ⇒ *int*
global-unfrmzr-pows-adelic-int ≡
global-unfrmzr-pows :: (*'a prime* ⇒ *nat*) ⇒ *'a adelic-int*
begin

definition *p-equal-adelic-int* ::
'a::*nontrivial-factorial-idom prime* ⇒
'a adelic-int ⇒ *'a adelic-int* ⇒ *bool*
where *p-equal-adelic-int p x y* ≡ (*Rep-adelic-int x*) $\simeq_p$ (*Rep-adelic-int y*)

definition *p-restrict-adelic-int* ::
'a::*nontrivial-factorial-idom adelic-int* ⇒
(*'a prime* ⇒ *bool*) ⇒ *'a adelic-int*
where
p-restrict-adelic-int x P ≡ *Abs-adelic-int* ((*Rep-adelic-int x*) *prestrict P*)

definition *p-depth-adelic-int* ::
'a::*nontrivial-factorial-idom prime* ⇒ *'a adelic-int* ⇒ *int*
where *p-depth-adelic-int p x* ≡ ((*Rep-adelic-int x*)$^{\circ p}$)

definition *global-unfrmzr-pows-adelic-int* ::

$('a::nontrivial\text{-}factorial\text{-}idom\ prime \Rightarrow nat) \Rightarrow 'a\ adelic\text{-}int$
where
global-unfrmzr-pows-adelic-int f $\equiv$ *Abs-adelic-int* (*global-unfrmzr-pows* (*int* $\circ$ *f*))

end

context
fixes p :: $'a::nontrivial\text{-}factorial\text{-}idom\ prime$
begin

lemma *p-equal-adelic-int-abs-eq*:
$(Abs\text{-}adelic\text{-}int\ x) \simeq_p (Abs\text{-}adelic\text{-}int\ y) \longleftrightarrow$
$x \simeq_p y$
if $x \in \mathcal{O}_{\forall\ p}$
and $y \in \mathcal{O}_{\forall\ p}$
for $x\ y$ :: $'a\ p\text{-}adic\text{-}prod$
using *that*
by (*simp add*: *p-equal-adelic-int-def Abs-adelic-int-inverse*)

lemma *p-equal-adelic-int-abs-eq0*:
$(Abs\text{-}adelic\text{-}int\ x) \simeq_p 0 \longleftrightarrow x \simeq_p 0$
if $x \in \mathcal{O}_{\forall\ p}$ **for** x :: $'a\ p\text{-}adic\text{-}prod$
using *that p-equal-adelic-int-abs-eq global-p-depth-p-adic-prod.global-depth-set-0*
unfolding *zero-adelic-int-def*
by *auto*

lemma *p-equal-adelic-int-rep-eq0*:
$(Rep\text{-}adelic\text{-}int\ x) \simeq_p 0 \longleftrightarrow x \simeq_p 0$
for x :: $'a\ adelic\text{-}int$
using *p-equal-adelic-int-def zero-adelic-int-rep-eq* **by** *metis*

lemma *p-equal-adelic-int-abs-eq1*:
$(Abs\text{-}adelic\text{-}int\ x) \simeq_p 1 \longleftrightarrow x \simeq_p 1$
if $x \in \mathcal{O}_{\forall\ p}$ **for** x :: $'a\ p\text{-}adic\text{-}prod$
using *that p-equal-adelic-int-abs-eq global-p-depth-p-adic-prod.global-depth-set-1*
unfolding *one-adelic-int-def*
by *auto*

lemma *p-depth-adelic-int-abs-eq*:
$(Abs\text{-}adelic\text{-}int\ x)^{\circ p} = (x^{\circ p})$
if $x \in \mathcal{O}_{\forall\ p}$ **for** x :: $'a\ p\text{-}adic\text{-}prod$
using *that* **by** (*simp add*: *p-depth-adelic-int-def Abs-adelic-int-inverse*)

lemma *adelic-int-depth*: $(x^{\circ p}) \geq 0$ **for** x :: $'a\ adelic\text{-}int$
using *global-p-depth-p-adic-prod.nonpos-global-depth-setD p-depth-adelic-int-abs-eq*
by (*cases x*) *auto*

end

lemma *p-restrict-adelic-int-rep-eq*:
Rep-adelic-int (*x prestrict P*) = (*Rep-adelic-int x*) *prestrict P*
for *x* :: *'a::nontrivial-factorial-idom adelic-int*
and *P* :: *'a prime* ⇒ *bool*
using *Rep-adelic-int global-p-depth-p-adic-prod.global-depth-set-p-restrict Abs-adelic-int-inverse*
unfolding *p-restrict-adelic-int-def*
by *fastforce*

lemma *p-restrict-adelic-int-abs-eq*:
(*Abs-adelic-int x*) *prestrict P* = *Abs-adelic-int* (*x prestrict P*)
if $x \in \mathcal{O}_{\forall\, p}$
for *x* :: *'a::nontrivial-factorial-idom p-adic-prod* **and** *P* :: *'a prime* ⇒ *bool*
using *that Abs-adelic-int-inverse' p-restrict-adelic-int-def* **by** *metis*

lemma *global-unfrmzr-pows-adelic-int-rep-eq*:
Rep-adelic-int ($\mathfrak{p}$ *f* :: *'a adelic-int*) = $\mathfrak{p}$ (*int* ∘ *f*)
for *f* :: *'a::nontrivial-factorial-idom prime* ⇒ *nat*
using *global-unfrmzr-pows-p-adic-prod-global-depth-set-0 Abs-adelic-int-inverse' global-unfrmzr-pows-adelic-int-def*
by *metis*

global-interpretation *p-equality-adelic-int*:
p-equality-no-zero-divisors-1
p-equal :: *'a::nontrivial-factorial-idom prime* ⇒
'a adelic-int ⇒ *'a adelic-int* ⇒ *bool*
proof

fix *p* :: *'a prime*

show *equivp* (*p-equal p* :: *'a adelic-int* ⇒ *'a adelic-int* ⇒ *bool*)
using *equivp-transfer p-equality-p-adic-prod.equivp* **unfolding** *p-equal-adelic-int-def*
by *fast*

fix *x y* :: *'a adelic-int*

show $(x \simeq_p y) = (x - y \simeq_p 0)$
using *p-equality-p-adic-prod.conv-0*
by (*simp add*: *p-equal-adelic-int-def minus-adelic-int-rep-eq zero-adelic-int-rep-eq*)

show $y \simeq_p 0 \Longrightarrow x * y \simeq_p 0$
using *p-equality-p-adic-prod.mult-0-right*
by (*simp add*: *p-equal-adelic-int-def times-adelic-int-rep-eq zero-adelic-int-rep-eq*)

have (*1::'a adelic-int*) $\neg\simeq_p 0$
unfolding *p-equal-adelic-int-def*
by (*metis*
zero-adelic-int-rep-eq one-adelic-int-rep-eq

p-equality-p-adic-prod.one-p-nequal-zero
)
thus $\exists x::'a$ *adelic-int*. $x \neg\simeq_p 0$ **by** *blast*

assume $x \neg\simeq_p 0$ **and** $y \neg\simeq_p 0$
thus $x * y \neg\simeq_p 0$
using *p-equality-p-adic-prod.no-zero-divisors*
by (*simp add*: *p-equal-adelic-int-def times-adelic-int-rep-eq zero-adelic-int-rep-eq*)

qed

overloading
globally-p-equal-adelic-int ≡
globally-p-equal ::
'a::nontrivial-factorial-idom adelic-int ⇒ *'a adelic-int* ⇒ *bool*
begin

definition *globally-p-equal-adelic-int* ::
'a::nontrivial-factorial-idom adelic-int ⇒ *'a adelic-int* ⇒ *bool*
where *globally-p-equal-adelic-int-def*[*simp*]:
globally-p-equal-adelic-int = *p-equality-adelic-int.globally-p-equal*

end

global-interpretation *global-p-depth-adelic-int*:
global-p-equal-depth-no-zero-divisors-w-inj-index-consts
p-equal :: *'a::nontrivial-factorial-idom prime* ⇒
'a adelic-int ⇒ *'a adelic-int* ⇒ *bool*
p-restrict ::
'a adelic-int ⇒ (*'a prime* ⇒ *bool*) ⇒ *'a adelic-int*
p-depth :: *'a prime* ⇒ *'a adelic-int* ⇒ *int*
λf. *Abs-adelic-int* (*p-adic-prod-consts f*)
Rep-prime
proof (*unfold-locales*, *fold globally-p-equal-adelic-int-def*)

fix *p* :: *'a prime*

fix *x y* :: *'a adelic-int*

show $x \simeq_{\forall p} y \Longrightarrow x = y$
by (
cases x, *cases y*,
use p-equal-adelic-int-abs-eq global-p-depth-p-adic-prod.global-imp-eq **in** *fast-force*
)

fix *P* :: *'a prime* ⇒ *bool*
show $P\ p \Longrightarrow x$ *prestrict* $P \simeq_p x$
by (

cases x,
metis global-depth-set-p-adic-prod-def p-restrict-adelic-int-abs-eq p-equal-adelic-int-abs-eq global-p-depth-p-adic-prod.global-depth-set-p-restrict global-p-depth-p-adic-prod.p-restrict-equiv
)
show $\neg\ P\ p \Longrightarrow x\ prestrict\ P \simeq_p 0$
by (
cases x,
metis global-depth-set-p-adic-prod-def p-restrict-adelic-int-abs-eq p-equal-adelic-int-abs-eq0 global-p-depth-p-adic-prod.global-depth-set-p-restrict global-p-depth-p-adic-prod.p-restrict-equiv0
)

show $((0::'a\ adelic\text{-}int)^{\circ p}) = 0$
using *global-p-depth-p-adic-prod.depth-of-0*
by (*simp add*: *p-depth-adelic-int-def zero-adelic-int-rep-eq*)

show $x \simeq_p y \Longrightarrow (x^{\circ p}) = (y^{\circ p})$
using *global-p-depth-p-adic-prod.depth-equiv*
by (*simp add*: *p-equal-adelic-int-def p-depth-adelic-int-def*)

show $(-\ x^{\circ p}) = (x^{\circ p})$
using *global-p-depth-p-adic-prod.depth-uminus*
by (*simp add*: *p-depth-adelic-int-def uminus-adelic-int-rep-eq*)

show
$x \neg\simeq_p 0 \Longrightarrow$
$(x^{\circ p}) < (y^{\circ p}) \Longrightarrow$
$((x + y)^{\circ p}) = (x^{\circ p})$
using *global-p-depth-p-adic-prod.depth-pre-nonarch*(*1*)
by (
simp add: *p-equal-adelic-int-def p-depth-adelic-int-def zero-adelic-int-rep-eq plus-adelic-int-rep-eq*
)

show
$\llbracket$
$x \neg\simeq_p 0$;
$(x + y) \neg\simeq_p 0$;
$(x^{\circ p}) = (y^{\circ p})$
$\rrbracket \Longrightarrow (x + y^{\circ p}) \geq (x^{\circ p})$
using *global-p-depth-p-adic-prod.depth-pre-nonarch*(*2*)
by (
fastforce
simp add: *p-equal-adelic-int-def p-depth-adelic-int-def zero-adelic-int-rep-eq plus-adelic-int-rep-eq*
)

show

$(x * y) \neg\simeq_p 0 \Longrightarrow$
  $(x * y^{\circ p}) = (x^{\circ p}) + (y^{\circ p})$
using *global-p-depth-p-adic-prod.depth-mult-additive*
by (
simp add: *p-equal-adelic-int-def p-depth-adelic-int-def zero-adelic-int-rep-eq times-adelic-int-rep-eq*
)

show *Abs-adelic-int* (*p-adic-prod-consts 1*) = *1*
by (*metis global-p-depth-p-adic-prod.consts-1 one-adelic-int-def*)

fix *f g* :: *'a prime* ⇒ *'a*

show
Abs-adelic-int (*p-adic-prod-consts* (*f* − *g*)) =
Abs-adelic-int (*p-adic-prod-consts f*) − *Abs-adelic-int* (*p-adic-prod-consts g*)
by (
metis global-depth-set-p-adic-prod-def global-p-depth-p-adic-prod.consts-diff global-p-depth-p-adic-prod.global-depth-set-consts minus-adelic-int-abs-eq
)

show
Abs-adelic-int (*p-adic-prod-consts* (*f* ∗ *g*)) =
Abs-adelic-int (*p-adic-prod-consts f*) ∗ *Abs-adelic-int* (*p-adic-prod-consts g*)
by (
metis global-depth-set-p-adic-prod-def global-p-depth-p-adic-prod.consts-mult global-p-depth-p-adic-prod.global-depth-set-consts times-adelic-int-abs-eq
)

show (*Abs-adelic-int* (*p-adic-prod-consts f*) $\simeq_p$ *0*) = (*f p* = *0*)
by (
metis global-depth-set-p-adic-prod-def global-p-depth-p-adic-prod.global-depth-set-consts p-equal-adelic-int-abs-eq0 global-p-depth-p-adic-prod.p-equal-func-of-consts-0
)

show (*Abs-adelic-int* (*p-adic-prod-consts f*)$^{\circ p}$) = *int* (*pmultiplicity p* (*f p*))
by (
metis global-depth-set-p-adic-prod-def global-p-depth-p-adic-prod.global-depth-set-consts p-depth-adelic-int-abs-eq global-p-depth-p-adic-prod.p-depth-func-of-consts
)

qed
(
metis Rep-prime-inject injI, *rule Rep-prime-n0*, *rule Rep-prime-not-unit*, *rule multiplicity-distinct-primes*
)

lemma *p-depth-adelic-int-diff-abs-eq*:
((*Abs-adelic-int x* − *Abs-adelic-int y*)$^{\circ p}$) = ((*x* − *y*)$^{\circ p}$)

if $x \in \mathcal{O}_{\forall\, p}$ **and** $y \in \mathcal{O}_{\forall\, p}$
for p :: *'a::nontrivial-factorial-idom prime* **and** $x\ y$ :: *'a p-adic-prod*
using *that global-depth-set-p-adic-prod-def minus-adelic-int-abs-eq p-depth-adelic-int-abs-eq global-p-depth-p-adic-prod.global-depth-set-minus*
by *metis*

lemma *p-depth-adelic-int-diff-abs-eq'*:
$((\mathit{Abs\text{-}adelic\text{-}int}\ (\mathit{abs\text{-}p\text{-}adic\text{-}prod}\ X) - \mathit{Abs\text{-}adelic\text{-}int}\ (\mathit{abs\text{-}p\text{-}adic\text{-}prod}\ Y))^{\circ p}) = ((X - Y)^{\circ p})$
if X: $X \in \mathcal{O}_{\forall\, p}$ **and** Y: $Y \in \mathcal{O}_{\forall\, p}$
for p :: *'a::nontrivial-factorial-idom prime* **and** $X\ Y$ :: *'a fls-pseq*
proof−
have *abs-p-adic-prod* $X \in \mathcal{O}_{\forall\, p}$
by (
simp, *intro global-p-depth-p-adic-prod.global-depth-setI*,
metis X p-equal-p-adic-prod-abs-eq0 p-equality-depth-fls-pseq.global-depth-setD global-depth-set-fls-pseq-def p-depth-p-adic-prod.abs-eq
)
moreover have *abs-p-adic-prod* $Y \in \mathcal{O}_{\forall\, p}$
by (
simp, *intro global-p-depth-p-adic-prod.global-depth-setI*,
metis Y p-equal-p-adic-prod-abs-eq0 p-equality-depth-fls-pseq.global-depth-setD global-depth-set-fls-pseq-def p-depth-p-adic-prod.abs-eq
)
ultimately show *?thesis* **by** (*metis p-depth-adelic-int-diff-abs-eq p-depth-p-adic-prod-diff-abs-eq*)
qed

context
fixes p :: *'a::nontrivial-euclidean-ring-cancel prime*
begin

lemma *adelic-int-diff-depth-gt-equiv-trans*:
$((a - c)^{\circ p}) > n$
if $a \simeq_p b$ **and** $b \neg\simeq_p c$ **and** $((b - c)^{\circ p}) > n$
for $a\ b\ c$ :: *'a adelic-int*
by (
cases a, *cases b*, *cases c*,
metis that p-depth-adelic-int-diff-abs-eq p-equal-adelic-int-abs-eq p-adic-prod-diff-depth-gt-equiv-trans
)

lemma *adelic-int-diff-depth-gt0-equiv-trans*:
$((a - c)^{\circ p}) > n$
if $n \geq 0$ **and** $a \simeq_p b$ **and** $((b - c)^{\circ p}) > n$
for $a\ b\ c$ :: *'a adelic-int*
using *that p-equality-adelic-int.conv-0 adelic-int-diff-depth-gt-equiv-trans global-p-depth-adelic-int.depth-equiv-0[of p b − c]*
by *fastforce*

lemma *adelic-int-diff-depth-gt-trans*:
$((a - c)^{\circ p}) > n$
if $a \neg\simeq_p b$ **and** $((a - b)^{\circ p}) > n$
and $b \neg\simeq_p c$ **and** $((b - c)^{\circ p}) > n$
and $a \neg\simeq_p c$
for $a\ b\ c$ :: *'a adelic-int*
by (
cases a, *cases b*, *cases c*,
metis that p-depth-adelic-int-diff-abs-eq p-equal-adelic-int-abs-eq p-adic-prod-diff-depth-gt-trans
)

lemma *adelic-int-diff-depth-gt0-trans*:
$((a - c)^{\circ p}) > n$
if $n \geq 0$
and $((a - b)^{\circ p}) > n$
and $((b - c)^{\circ p}) > n$
and $a \neg\simeq_p c$
for $a\ b\ c$ :: *'a adelic-int*
using *that adelic-int-diff-depth-gt-trans*
p-equality-adelic-int.conv-0[*of p a b*] *p-equality-adelic-int.conv-0*[*of p b c*]
global-p-depth-adelic-int.depth-equiv-0[*of p* $a - b$]
global-p-depth-adelic-int.depth-equiv-0[*of p* $b - c$]
by *auto*

lemma *adelic-int-prestrict-zero-depth*:
$1 \leq ((x\ prestrict\ (\lambda p{::}'a\ prime.\ (x^{\circ p}) = 0) - x)^{\circ p})$
if $x\ prestrict\ (\lambda p{::}'a\ prime.\ (x^{\circ p}) = 0) \neg\simeq_p x$
for x :: *'a adelic-int*
using *that adelic-int-depth*[*of p x*] *global-p-depth-adelic-int.depth-equiv*[*of p*]
global-p-depth-adelic-int.p-restrict-equiv[*of* $\lambda p.\ (x^{\circ p}) = 0$]
global-p-depth-adelic-int.p-restrict-equiv0[*of* $\lambda p.\ (x^{\circ p}) = 0\ p\ x$]
p-equality-adelic-int.minus-right[*of p - - x*] *global-p-depth-adelic-int.depth-uminus*[*of p x*]
by *force*

end

lemma *adelic-int-normalize-depth*:
$(x * \mathfrak{p}\ (\lambda p{::}'a\ prime.\ -(x^{\circ p}))) \in \mathcal{O}_{\forall\ p}$
for x :: *'a::nontrivial-factorial-idom p-adic-prod*
using *p-adic-prod-normalize-depth*[*of - x*] **by** (*fastforce intro*: *p-adic-prod-global-depth-setI'*)

lemma *global-unfrmzr-pows0-adelic-int*:
$(\mathfrak{p}\ (0 :: 'a{::}nontrivial\text{-}factorial\text{-}idom\ prime \Rightarrow nat) :: 'a\ adelic\text{-}int) = 1$
proof−
have $int \circ (0 :: 'a\ prime \Rightarrow nat) = (0 :: 'a\ prime \Rightarrow int)$ **by** *auto*
thus *?thesis*
using *global-unfrmzr-pows-p-adic-prod0*

unfolding *global-unfrmzr-pows-adelic-int-def one-adelic-int-def*
by *metis*
qed

context
fixes p :: $'a$::*nontrivial-factorial-idom prime*
begin

lemma *global-unfrmzr-pows-adelic-int*:
$(\mathfrak{p}\ f :: 'a\ adelic\text{-}int)^{\circ p} = int\ (f\ p)$ **for** f :: $'a\ prime \Rightarrow nat$
using *global-unfrmzr-pows-p-adic-prod-global-depth-set-0 p-depth-adelic-int-abs-eq*[*of - p*]
global-unfrmzr-pows-p-adic-prod[*of p*]
unfolding *global-unfrmzr-pows-adelic-int-def*
by *fastforce*

lemma *global-unfrmzr-pows-adelic-int-pequiv-iff*:
$((\mathfrak{p}\ f :: 'a\ adelic\text{-}int) \simeq_p (\mathfrak{p}\ g)) = (f\ p = g\ p)$
for $f\ g$ :: $'a\ prime \Rightarrow nat$
proof−
have $((int \circ f)\ p = (int \circ g)\ p) = (f\ p = g\ p)$ **by** *fastforce*
thus *?thesis*
using *global-unfrmzr-pows-p-adic-prod-global-depth-set-0 p-equal-adelic-int-abs-eq*
global-unfrmzr-pows-p-adic-prod-pequiv-iff
unfolding *global-unfrmzr-pows-adelic-int-def*
by *blast*
qed

end

lemma *global-unfrmzr-pows-prod-adelic-int*:
$(\mathfrak{p}\ f :: 'a\ adelic\text{-}int) * (\mathfrak{p}\ g) = \mathfrak{p}\ (f + g)$
for $f\ g$ :: $'a$::*nontrivial-factorial-idom prime* $\Rightarrow$ *nat*
proof−
have $(int \circ f) + (int \circ g) = int \circ (f + g)$ **by** *fastforce*
thus *?thesis*
using *global-unfrmzr-pows-p-adic-prod-global-depth-set-0 times-adelic-int-abs-eq*
global-unfrmzr-pows-prod-p-adic-prod
unfolding *global-unfrmzr-pows-adelic-int-def*
by *metis*
qed

context
fixes p :: $'a$::*nontrivial-factorial-idom prime*
begin

lemma *global-unfrmzr-pows-adelic-int-nzero*:
$(\mathfrak{p}\ f :: 'a\ adelic\text{-}int) \neg\simeq_p 0$ **for** f :: $'a\ prime \Rightarrow nat$
using *global-unfrmzr-pows-p-adic-prod-global-depth-set-0 p-equal-adelic-int-abs-eq0*

global-unfrmzr-pows-p-adic-prod-nzero
unfolding *global-unfrmzr-pows-adelic-int-def*
by *blast*

lemma *prod-w-global-unfrmzr-pows-adelic-int*:
$(x * \mathfrak{p}\ f)^{\circ p} = (x^{\circ p}) + int\ (f\ p)$
if $x \neg\simeq_p 0$ **for** $f :: 'a\ prime \Rightarrow nat$ **and** $x :: 'a\ adelic\text{-}int$
proof (*cases x*)
case (*Abs-adelic-int a*)
moreover have $(\lambda x.\ int\ (f\ x)) = int \circ f$ **by** *auto*
moreover from *that Abs-adelic-int* **have** $a \neg\simeq_p 0$
using *p-equal-adelic-int-abs-eq0*[*of a p*] **by** *blast*
ultimately have
$((x * \mathfrak{p}\ f)^{\circ p}) =$
$(a^{\circ p}) + int\ (f\ p)$
using *that times-adelic-int-abs-eq global-unfrmzr-pows-p-adic-prod-global-depth-set-0 p-depth-adelic-int-abs-eq global-depth-set-p-adic-prod-def global-p-depth-p-adic-prod.global-depth-set-times*[*of 0 - $\mathfrak{p}$ (int $\circ$ f)*] *prod-w-global-unfrmzr-pows-p-adic-prod*[*of p a f*]
unfolding *global-unfrmzr-pows-adelic-int-def*
by *fastforce*
with *Abs-adelic-int* **show** *?thesis* **using** *p-depth-adelic-int-abs-eq* **by** *fastforce*
qed

lemma *trivial-global-unfrmzr-pows-adelic-int*:
$(\mathfrak{p}\ f :: 'a\ adelic\text{-}int) \simeq_p 1$ **if** $f\ p = 0$ **for** $f :: 'a\ prime \Rightarrow nat$
using *that p-equal-adelic-int-abs-eq1 global-unfrmzr-pows-p-adic-prod-global-depth-set-0 trivial-global-unfrmzr-pows-p-adic-prod*[*of - p*]
unfolding *global-unfrmzr-pows-adelic-int-def*
by *force*

lemma *prod-w-trivial-global-unfrmzr-pows-adelic-int*:
$x * \mathfrak{p}\ f \simeq_p x$ **if** $f\ p = 0$
for $f :: 'a\ prime \Rightarrow nat$ **and** $x :: 'a\ adelic\text{-}int$
using *that p-equality-adelic-int.mult-one-right trivial-global-unfrmzr-pows-adelic-int*
by *blast*

end

lemma *pow-global-unfrmzr-pows-adelic-int*:
$(\mathfrak{p}\ f :: 'a\ adelic\text{-}int)\ \hat{}\ n = (\mathfrak{p}\ ((\lambda p.\ n) * f))$
for $f :: 'a{::}nontrivial\text{-}factorial\text{-}idom\ prime \Rightarrow nat$
proof−
have $(\lambda p.\ int\ n * (int \circ f)\ p) = int \circ ((\lambda p.\ n) * f)$ **by** *auto*
thus *?thesis*
using *global-unfrmzr-pows-p-adic-prod-global-depth-set-0 pow-adelic-int-abs-eq global-unfrmzr-pows-p-adic-prod-pow*
unfolding *global-unfrmzr-pows-adelic-int-def*
by *metis*

qed

abbreviation *adelic-int-consts* $\equiv$ *global-p-depth-adelic-int.consts*
abbreviation *adelic-int-const* $\equiv$ *global-p-depth-adelic-int.const*

lemma *adelic-int-p-depth-consts*:
$(\textit{adelic-int-consts } f)^{\circ p} = ((f\ p)^{\circ p})$
for p :: *'a::nontrivial-factorial-idom prime* **and** f :: *'a prime* $\Rightarrow$ *'a*
using *global-depth-set-p-adic-prod-def global-p-depth-p-adic-prod.global-depth-set-consts p-depth-adelic-int-abs-eq p-depth-p-adic-prod-consts*
by *metis*

lemmas *adelic-int-p-depth-const* = *adelic-int-p-depth-consts*[*of* - λp. -]

lemma *adelic-int-global-unfrmzr*:
($\mathfrak{p}$ (*1* :: *'a::nontrivial-factorial-idom prime* $\Rightarrow$ *nat*) :: *'a adelic-int*)
= *adelic-int-consts Rep-prime*
proof−
define *natfun-1* :: *'a prime* $\Rightarrow$ *nat* **and** *intfun-1* :: *'a prime* $\Rightarrow$ *int*
where *natfun-1* $\equiv$ *1* **and** *intfun-1* $\equiv$ *1*
moreover from *this* **have** *int* $\circ$ *natfun-1* = *intfun-1* **by** *auto*
ultimately show *?thesis*
using *p-adic-prod-global-unfrmzr* **unfolding** *global-unfrmzr-pows-adelic-int-def*
by *metis*
qed

lemma *adelic-int-normalize-depthE*:
fixes x :: *'a::nontrivial-factorial-idom adelic-int*
obtains *x-0*
where $\forall$ *p::'a prime.* $\textit{x-0}^{\circ p} = 0$
and x = *x-0* $*$ $\mathfrak{p}$ (λ*p::'a prime. nat* $(x^{\circ p})$)
proof (*cases x*)
define f :: *'a prime* $\Rightarrow$ *nat*
where $f \equiv \lambda$*p::'a prime. nat* $(x^{\circ p})$
case (*Abs-adelic-int a*)
obtain *a-0* **where** *a-0*:
$\forall$ *p::'a prime.* $\textit{a-0}^{\circ p} = 0$
a = *a-0* $*$ $\mathfrak{p}$ (λ*p::'a prime.* $(a^{\circ p})$)
by (*elim p-adic-prod-normalize-depthE*)
from *a-0*(*1*) **have** *a-0-int*: *a-0* $\in \mathcal{O}_{\forall p}$
using *p-adic-prod-global-depth-setI'*[*of 0 a-0*] **by** *auto*
moreover from *f-def Abs-adelic-int*
have (λ*p::'a prime.* $(a^{\circ p})$) = *int* $\circ$ f
using *p-depth-adelic-int-abs-eq adelic-int-depth*
by *fastforce*
ultimately have x = *Abs-adelic-int a-0* $*$ $\mathfrak{p}$ f
using *f-def Abs-adelic-int*(*1*) *a-0*(*2*) *times-adelic-int-abs-eq global-unfrmzr-pows-p-adic-prod-global-depth-set*[*of 0 int* $\circ$ *f*] *global-unfrmzr-pows-adelic-int-def*[*of f*]

by *fastforce*
moreover from *a-0(1)* **have** $\forall p::'a\ prime.\ (Abs\text{-}adelic\text{-}int\ (a\text{-}0))^{\circ p} = 0$
using *a-0-int p-depth-adelic-int-abs-eq* **by** *metis*
ultimately show *thesis* **using** *that f-def* **by** *presburger*
qed

5.8.4 Inverses

We can create inverses at depth-zero places.

instantiation *adelic-int* ::
(*nontrivial-factorial-unique-euclidean-bezout*) {*divide-trivial*, *inverse*}
begin

definition *inverse-adelic-int* ::
'a adelic-int ⇒ *'a adelic-int*
where
inverse-adelic-int x =
Abs-adelic-int (
(*inverse* (*Rep-adelic-int x*)) *prestrict* ($\lambda p::'a\ prime.\ x^{\circ p} = 0$)
)

definition *divide-adelic-int* ::
'a adelic-int ⇒ *'a adelic-int* ⇒ *'a adelic-int*
where *divide-adelic-int-def*[*simp*]: *divide-adelic-int x y* = *x* ∗ *inverse y*

instance
proof
fix *x* :: *'a adelic-int*
have ($\lambda p::'a\ prime.\ (0::'a\ adelic\text{-}int)^{\circ p} = 0$) = ($\lambda p.\ True$)
using *global-p-depth-adelic-int.depth-of-0* **by** *fast*
hence *x div 0* = *x* ∗ *Abs-adelic-int* (*0 prestrict* ($\lambda p::'a\ prime.\ True$))
using *inverse-adelic-int-def*[*of 0*]
unfolding *zero-adelic-int-rep-eq p-adic-prod-div-inv.inverse-0*
by *simp*
thus *x div 0* = *0*
by (*metis global-p-depth-p-adic-prod.p-restrict-true mult-zero-right zero-adelic-int-def*)
have ($\lambda p::'a\ prime.\ (1::'a\ adelic\text{-}int)^{\circ p} = 0$) = ($\lambda p.\ True$)
using *global-p-depth-adelic-int.p-depth-1* **by** *fast*
hence *x div 1* = *x* ∗ *Abs-adelic-int* (*1 prestrict* ($\lambda p::'a\ prime.\ True$))
using *inverse-adelic-int-def*[*of 1*]
unfolding *one-adelic-int-rep-eq p-adic-prod-div-inv.inverse-1*
by *simp*
thus *x div 1* = *x*
by (*metis global-p-depth-p-adic-prod.p-restrict-true mult-1-right one-adelic-int-def*)
qed *simp*

end

lemma *inverse-adelic-int-abs-eq*:

inverse (*Abs-adelic-int x*) =
Abs-adelic-int (*inverse x prestrict* (λp::*'a prime*. $x^{\circ p}$ = *0*))
if $x \in \mathcal{O}_{\forall\, p}$
for *x* :: *'a*::*nontrivial-factorial-unique-euclidean-bezout p-adic-prod*
using *that Abs-adelic-int-inverse*[*of x*] *p-depth-adelic-int-abs-eq*[*of x*]
unfolding *inverse-adelic-int-def*
by *auto*

lemma *divide-adelic-int-abs-eq*:
Abs-adelic-int x / *Abs-adelic-int y* =
Abs-adelic-int ((*x* / *y*) *prestrict* (λp::*'a prime*. $y^{\circ p}$ = *0*))
if $x \in \mathcal{O}_{\forall\, p}$ **and** $y \in \mathcal{O}_{\forall\, p}$
for *x y* :: *'a*::*nontrivial-factorial-unique-euclidean-bezout p-adic-prod*
using *that divide-adelic-int-def inverse-adelic-int-abs-eq global-depth-set-p-adic-prod-def*
p-adic-prod-div-inv.global-depth-set-inverse times-adelic-int-abs-eq
global-p-depth-p-adic-prod.times-p-restrict p-adic-prod-div-inv.divide-inverse
by *metis*

lemma *p-depth-adelic-int-inverse-diff-abs-eq*:
((*inverse* (*Abs-adelic-int x*) − *inverse* (*Abs-adelic-int y*))$^{\circ p}$) =
((*inverse x* − *inverse y*)$^{\circ p}$)
if $x \in \mathcal{O}_{\forall\, p}$ $x^{\circ p}$ = *0*
and $y \in \mathcal{O}_{\forall\, p}$ $y^{\circ p}$ = *0*
for *p* :: *'a*::*nontrivial-factorial-unique-euclidean-bezout prime* **and** *x y* :: *'a p-adic-prod*
using *that inverse-adelic-int-abs-eq p-adic-prod-div-inv.global-depth-set-inverse*
p-depth-adelic-int-diff-abs-eq[*of*
inverse x prestrict (λp::*'a prime*. ($x^{\circ p}$) = *0*)
]
global-depth-set-p-adic-prod-def global-p-depth-p-adic-prod.depth-diff-equiv[*of*
p]
global-p-depth-p-adic-prod.p-restrict-equiv[*of - p*]
by *fastforce*

context
fixes *x* :: *'a*::*nontrivial-factorial-unique-euclidean-bezout adelic-int*
begin

lemma *inverse-adelic-int-rep-eq*:
Rep-adelic-int (*inverse x*) =
(*inverse* (*Rep-adelic-int x*)) *prestrict* (λp::*'a prime*. $x^{\circ p}$ = *0*)
using *Rep-adelic-int p-adic-prod-div-inv.global-depth-set-inverse Abs-adelic-int-inverse*
unfolding *inverse-adelic-int-def*
by (*auto simp add*: *p-depth-adelic-int-def*)

lemma *adelic-int-inverse-equiv0-iff*:
inverse x $\simeq_p$ *0* $\longleftrightarrow$
($x \simeq_p$ *0* $\vee$ ($x^{\circ p}$) $\neq$ *0*)
for *p* :: *'a prime*
proof (*cases x*)

case (*Abs-adelic-int a*)
thus *?thesis*
using *inverse-adelic-int-abs-eq*[*of a*] *p-adic-prod-div-inv.global-depth-set-inverse*[*of a*]
p-equal-adelic-int-abs-eq0[*of - p*]
global-p-depth-p-adic-prod.p-restrict-equiv0[*of - p inverse a*]
global-p-depth-p-adic-prod.p-restrict-equiv[*of - p inverse a*]
p-equality-p-adic-prod.trans0-iff[*of p - inverse a*]
p-adic-prod-div-inv.inverse-equiv0-iff[*of p*] *p-depth-adelic-int-abs-eq*[*of a*]
by *auto*
qed

lemma *adelic-int-inverse-equiv0-iff'*:
$inverse\ x \simeq_p 0 \longleftrightarrow$
$(x \simeq_p 0 \vee (x^{\circ p}) \geq 1)$
for p :: *'a prime*
using *adelic-int-inverse-equiv0-iff adelic-int-depth*[*of p x*] **by** *force*

end

lemma *adelic-int-inverse-eq0-iff*:
$inverse\ x = 0 \longleftrightarrow$
$(\forall p::'a\ prime.\ x^{\circ p} = 0 \longrightarrow x \simeq_p 0)$
for x :: *'a::nontrivial-factorial-unique-euclidean-bezout adelic-int*
proof (*standard*, *clarify*)
fix p :: *'a prime* **assume** $inverse\ x = 0$ $x^{\circ p} = 0$ **thus** $x \simeq_p 0$
using *adelic-int-inverse-equiv0-iff* **by** *fastforce*
next
assume x: $\forall p::'a\ prime.\ x^{\circ p} = 0 \longrightarrow x \simeq_p 0$
show $inverse\ x = 0$
proof (*intro global-p-depth-adelic-int.global-imp-eq*, *standard*)
fix p :: *'a prime*
from x **show** $inverse\ x \simeq_p 0$
using *adelic-int-depth*[*of p x*] *adelic-int-inverse-equiv0-iff*
by (*cases* $x^{\circ p} = 0$, *fast*, *fastforce*)
qed
qed

lemma *divide-adelic-int-rep-eq*:
Rep-adelic-int $(x\ /\ y) =$
(*Rep-adelic-int x* / *Rep-adelic-int y*) *prestrict* $(\lambda p::'a\ prime.\ y^{\circ p} = 0)$
for $x\ y$ :: *'a::nontrivial-factorial-unique-euclidean-bezout adelic-int*
using *divide-adelic-int-def times-adelic-int-rep-eq inverse-adelic-int-rep-eq*
global-p-depth-p-adic-prod.times-p-restrict p-adic-prod-div-inv.divide-inverse
by *metis*

lemma *adelic-int-divide-by-pequiv0*:
$x\ /\ y \simeq_p 0$ **if** $y \simeq_p 0$
for p :: *'a::nontrivial-factorial-unique-euclidean-bezout prime* **and** $x\ y$ :: *'a adelic-int*

using *that divide-adelic-int-def adelic-int-inverse-equiv0-iff p-equality-adelic-int.mult-0-right*
by *metis*

lemma *adelic-int-divide-by-pos-depth*:
$x\ /\ y \simeq_p 0$ **if** $(y^{\circ p}) > 0$
for *p* :: *'a::nontrivial-factorial-unique-euclidean-bezout prime* **and** *x y* :: *'a adelic-int*
using *that adelic-int-inverse-equiv0-iff p-equality-adelic-int.mult-0-right*[*of p*] **by** *force*

lemma *adelic-int-inverse0*[*simp*]:
inverse (*0*::*'a::nontrivial-factorial-unique-euclidean-bezout adelic-int*) = *0*
using *adelic-int-inverse-eq0-iff* **by** *force*

lemma *adelic-int-inverse1*[*simp*]:
inverse (*1*::*'a::nontrivial-factorial-unique-euclidean-bezout adelic-int*) = *1*
proof−
have $(\lambda p{::}'a\ prime.\ (1{::}'a\ p\text{-}adic\text{-}prod)^{\circ p} = 0) = (\lambda p.\ True)$
by *simp*
thus *?thesis*
using *global-depth-set-p-adic-prod-def one-adelic-int-def inverse-adelic-int-abs-eq global-p-depth-p-adic-prod.global-depth-set-1 p-adic-prod-div-inv.inverse-1 global-p-depth-p-adic-prod.p-restrict-true*
by *metis*
qed

lemma *adelic-int-inverse-mult*:
inverse $(x * y)$ = *inverse* $x *$ *inverse* y
for *x y* :: *'a::nontrivial-factorial-unique-euclidean-bezout adelic-int*
proof (
intro global-p-depth-adelic-int.global-imp-eq, *standard*, *cases x y rule*: *two-Abs-adelic-int-cases*
)
case (*Abs-adelic-int-Abs-adelic-int a b*)
fix *p* :: *'a prime*
define *P* :: *'a p-adic-prod* ⇒ *'a prime* ⇒ *bool*
where $P \equiv (\lambda z\ p.\ z^{\circ p} = 0)$
define *iab iab' ia ib*
where *iab* ≡ *inverse* $(a * b)$ *prestrict* $(P\ (a * b))$
and *iab'* ≡ *inverse* $(a * b)$ *prestrict* $(\lambda p.\ P\ a\ p \wedge P\ b\ p)$
and *ia* ≡ *inverse a prestrict* $(P\ a)$
and *ib* ≡ *inverse b prestrict* $(P\ b)$
from *P-def Abs-adelic-int-Abs-adelic-int*(*2*,*4*) **have**
$P\ (a * b)\ p \Longrightarrow$ *inverse* $(a * b) \neg\simeq_p 0 \Longrightarrow$
$P\ a\ p \wedge P\ b\ p$
using *p-adic-prod-div-inv.inverse-equiv0-iff global-p-depth-p-adic-prod.depth-mult-additive global-p-depth-p-adic-prod.nonpos-global-depth-setD*[*of 0 a p*] *global-p-depth-p-adic-prod.nonpos-global-depth-setD*[*of 0 b p*] *global-depth-set-p-adic-prod-def*
by *force*
moreover from *P-def* **have**

$\neg$ *P* (*a* ∗ *b*) *p* ⟹ *P a p* ⟹ *P b p* ⟹
inverse (*a* ∗ *b*) $\simeq_p$ *0*
using *global-p-depth-p-adic-prod.depth-mult-additive p-adic-prod-div-inv.inverse-equiv0-iff*
by *force*
ultimately have *iab* $\simeq_p$ *iab′*
using *global-p-depth-p-adic-prod.p-restrict-p-equal-p-restrictI*[*of - p*]
unfolding *iab-def iab′-def*
by *force*
moreover from *iab′-def ia-def ib-def* **have** *iab′* $\simeq_p$ *ia* ∗ *ib*
using *global-p-depth-p-adic-prod.p-restrict-times-equiv*[*of p - P a - P b*]
p-adic-prod-div-inv.inverse-mult p-equality-p-adic-prod.sym **by** *metis*
ultimately have *iab* $\simeq_p$ *ia* ∗ *ib* **using** *p-equality-p-adic-prod.trans* **by** *metis*
with *Abs-adelic-int-Abs-adelic-int P-def iab-def ia-def ib-def* **show**
inverse (*x* ∗ *y*) $\simeq_p$
(*inverse x* ∗ *inverse y*)
using *global-depth-set-p-adic-prod-def p-adic-prod-div-inv.global-depth-set-inverse*
global-p-depth-p-adic-prod.global-depth-set-times times-adelic-int-abs-eq
p-equal-adelic-int-abs-eq[*of*
inverse (*a* ∗ *b*) *prestrict* (*P* (*a* ∗ *b*))
(*inverse a prestrict* (*P a*)) ∗ (*inverse b prestrict* (*P b*))
]
inverse-adelic-int-abs-eq[*of a*] *inverse-adelic-int-abs-eq*[*of b*]
inverse-adelic-int-abs-eq[*of a* ∗ *b*]
by *fastforce*
qed

lemma *adelic-int-inverse-pcong*:
(*inverse x*) $\simeq_p$ (*inverse y*) **if** *x* $\simeq_p$ *y*
for *p* :: *′a*::*nontrivial-factorial-unique-euclidean-bezout prime* **and** *x y* :: *′a adelic-int*
proof (*cases x y rule*: *two-Abs-adelic-int-cases*)
case (*Abs-adelic-int-Abs-adelic-int a b*)
with *that* **have** *a* $\simeq_p$ *b* **using** *p-equal-adelic-int-abs-eq* **by** *blast*
moreover define *d* :: *′a p-adic-prod* ⇒ *′a prime* ⇒ *bool*
where $d \equiv \lambda z\ p.\ z^{\circ p} = 0$
ultimately have (*inverse a prestrict* (*d a*)) $\simeq_p$ (*inverse b prestrict* (*d b*))
by (
auto
intro : *global-p-depth-p-adic-prod.p-restrict-p-equal-p-restrictI*
simp add: *p-adic-prod-div-inv.inverse-pcong global-p-depth-p-adic-prod.depth-equiv*
)
with *Abs-adelic-int-Abs-adelic-int d-def* **show** *?thesis*
using *inverse-adelic-int-abs-eq*[*of a*] *inverse-adelic-int-abs-eq*[*of b*] *p-equal-adelic-int-abs-eq*
p-adic-prod-div-inv.global-depth-set-inverse[*of a*]
p-adic-prod-div-inv.global-depth-set-inverse[*of b*]
by *auto*
qed

context
fixes *x* :: *′a*::*nontrivial-factorial-unique-euclidean-bezout adelic-int*

begin

lemma *adelic-int-inverse-uminus*: *inverse* $(-x)$ = $-$ *inverse x*
proof (*cases x*)
  case (*Abs-adelic-int a*) **thus** *?thesis*
   using *global-depth-set-p-adic-prod-def global-p-depth-p-adic-prod.global-depth-set-uminus*[*of a*]
         uminus-adelic-int-abs-eq[*of a*] *p-adic-prod-div-inv.inverse-uminus*[*of a*]
         uminus-adelic-int-abs-eq[*of inverse a prestrict -*]
         p-adic-prod-div-inv.global-depth-set-inverse[*of a*]
         inverse-adelic-int-abs-eq[*of* $-a$] *inverse-adelic-int-abs-eq*[*of a*]
         global-p-depth-p-adic-prod.p-restrict-uminus[*of inverse a*]
         global-p-depth-p-adic-prod.depth-uminus[*of - a*]
    by *fastforce*
qed

lemma *adelic-int-inverse-inverse*:
  inverse (*inverse x*) = *x prestrict* (λp::'*a prime*. $x^{\circ p}$ = *0*)
proof (*cases x*)
  case (*Abs-adelic-int a*)
  moreover define *P* **where** $P \equiv$ (λp::'*a prime*. $a^{\circ p}$ = *0*)
  moreover from *this*
    have (λp::'*a prime*. *P p* $\wedge$ (*inverse a prestrict P*)$^{\circ p}$ = *0*) = *P*
    using *global-p-depth-p-adic-prod.p-restrict-depth*[*of P*]
         p-adic-prod-div-inv.inverse-depth[*of - a*]
    by *auto*
  ultimately show *?thesis*
    using *inverse-adelic-int-abs-eq p-adic-prod-div-inv.global-depth-set-inverse*[*of a*]
         p-adic-prod-div-inv.inverse-inverse[*of a*]
         inverse-adelic-int-abs-eq[*of inverse a prestrict P*] *p-depth-adelic-int-abs-eq*
       p-adic-prod-div-inv.inverse-p-restrict[*of inverse a P*] *p-restrict-adelic-int-abs-eq*
         global-p-depth-p-adic-prod.p-restrict-restrict[*of a P*]
    by *fastforce*
qed

lemma *adelic-int-inverse-pow*: *inverse* (x ^ n) = (*inverse x*) ^ *n*
  by (*induct n*, *simp-all add*: *adelic-int-inverse-mult*)

lemma *adelic-int-divide-self'*:
  x / $x \simeq_p$ *1* **if** $x \neg\simeq_p$ *0* **and** $x^{\circ p}$ = *0*
  for *p* :: '*a prime*
proof (*cases x*)
  case (*Abs-adelic-int a*)
  define *P* :: '*a prime* $\Rightarrow$ *bool*
    where $P \equiv$ (λp. ($a^{\circ p}$) = *0*)
  moreover from *that Abs-adelic-int* **have** $a \neg\simeq_p$ *0* $a^{\circ p}$ = *0*
     using *that p-equal-adelic-int-abs-eq0 p-depth-adelic-int-abs-eq* **by** (*blast*, *fastforce*)
  ultimately

have *Abs-adelic-int* $(a$ / a *prestrict* $P) \simeq_p$ (*Abs-adelic-int 1*)
using *Abs-adelic-int(2) global-p-depth-p-adic-prod.p-restrict-equiv*[*of P p*]
p-adic-prod-div-inv.divide-right-equiv[*of p - a*]
p-adic-prod-div-inv.divide-self-equiv[*of p*] *p-equality-p-adic-prod.trans*[*of p - a / a 1*]
p-adic-prod-div-inv.divide-p-restrict-right[*of a a*]
p-adic-prod-div-inv.global-depth-set-divide[*of a 0*]
p-equal-adelic-int-abs-eq global-p-depth-p-adic-prod.global-depth-set-1
by *fastforce*
with *Abs-adelic-int P-def* **show** *?thesis* **using** *divide-adelic-int-abs-eq one-adelic-int-def* **by** *metis*
qed

lemma *adelic-int-global-divide-self*:
x / $x = 1$
if $\forall p{::}'a$ *prime*. $x \neg\simeq_p 0$
and $\forall p{::}'a$ *prime*. $x^{\circ p} = 0$
using *that global-p-depth-adelic-int.global-imp-eq adelic-int-divide-self'* **by** *fastforce*

lemma *adelic-int-divide-self*:
x / $x =$
1 prestrict $(\lambda p{::}'a$ *prime*. $x \neg\simeq_p 0 \wedge x^{\circ p} = 0)$
proof (*intro global-p-depth-adelic-int.global-imp-eq*, *standard*)
define P :: *'a prime* $\Rightarrow$ *bool*
where $P \equiv (\lambda p.\ x \neg\simeq_p 0 \wedge x^{\circ p} = 0)$
fix p :: *'a prime* **show** x / $x \simeq_p$ *1 prestrict P*
proof (*cases P p*)
case *True* **with** *P-def* **show** *?thesis*
using *p-equality-adelic-int.trans-left adelic-int-divide-self'*[*of p*]
global-p-depth-adelic-int.p-restrict-equiv[*of - p*]
by *force*
next
case *False*
moreover from *this P-def* **have** x / $x \simeq_p 0$
using *divide-adelic-int-def adelic-int-inverse-equiv0-iff*
p-equality-adelic-int.mult-0-right
by *metis*
ultimately show *?thesis*
using *global-p-depth-adelic-int.p-restrict-equiv0 p-equality-adelic-int.trans-left* **by** *metis*
qed
qed

lemma *adelic-int-p-restrict-inverse*:
(*inverse x*) *prestrict* $P =$ *inverse* (x *prestrict* P) **for** P :: *'a prime* $\Rightarrow$ *bool*
proof (*cases x*)
case (*Abs-adelic-int a*)
moreover have

$(\lambda p::'a\ prime.\ a^{\circ p} = 0 \wedge P\ p) =$
$(\lambda p::'a\ prime.\ P\ p \wedge (a\ prestrict\ P)^{\circ p} = 0)$
using *global-p-depth-p-adic-prod.p-restrict-equiv global-p-depth-p-adic-prod.depth-equiv*
by *metis*
ultimately show *?thesis*
using *inverse-adelic-int-abs-eq*[*of a*] *inverse-adelic-int-abs-eq*[*of a prestrict P*]
p-adic-prod-div-inv.global-depth-set-inverse[*of a*] *p-restrict-adelic-int-abs-eq*[*of - P*]
p-adic-prod-div-inv.inverse-p-restrict[*of a*]
global-p-depth-p-adic-prod.p-restrict-restrict[*of inverse a*]
global-p-depth-p-adic-prod.global-depth-set-p-restrict[*of a 0 P*]
by *force*
qed

lemma *adelic-int-inverse-depth*: $(inverse\ x)^{\circ p} = 0$ **for** $p :: 'a\ prime$
proof (*cases x*)
case (*Abs-adelic-int X*)
hence
$(inverse\ x)^{\circ p} =$
$((inverse\ X\ prestrict\ (\lambda p::'a\ prime.\ X^{\circ p} = 0))^{\circ p})$
using *inverse-adelic-int-abs-eq p-adic-prod-div-inv.global-depth-set-inverse*
p-depth-adelic-int-abs-eq
by *fastforce*
thus *?thesis*
using *global-p-depth-p-adic-prod.p-restrict-equiv*[*of - p inverse X*]
global-p-depth-p-adic-prod.depth-equiv[*of p - inverse X*]
p-adic-prod-div-inv.inverse-depth[*of p X*] *global-p-depth-p-adic-prod.depth-equiv-0*[*of p*]
global-p-depth-p-adic-prod.p-restrict-equiv0[*of - p inverse X*]
by (*cases* $X^{\circ p} = 0$, *presburger*, *presburger*)
qed

end

lemma *adelic-int-consts-divide-self*:
$(adelic\text{-}int\text{-}consts\ f)\ /\ (adelic\text{-}int\text{-}consts\ f) =$
$1\ prestrict\ (\lambda p::'a\ prime.\ f\ p \neq 0 \wedge (f\ p)^{\circ p} = 0)$
for f :: *'a::nontrivial-factorial-unique-euclidean-bezout prime* $\Rightarrow$ *'a*
using *adelic-int-divide-self*[*of adelic-int-consts f*] *adelic-int-p-depth-consts*[*of - f*]
global-p-depth-adelic-int.p-equal-func-of-consts-0[*of - f*]
by *presburger*

lemma *adelic-int-const-divide-self*:
$(adelic\text{-}int\text{-}const\ c)\ /\ (adelic\text{-}int\text{-}const\ c) =$
$1\ prestrict\ (\lambda p::'a\ prime.\ c^{\circ p} = 0)$
if $c \neq 0$ **for** c :: *'a::nontrivial-factorial-unique-euclidean-bezout*
using *that* **by** (*simp del*: *divide-adelic-int-def add*: *adelic-int-consts-divide-self*)

lemma *adelic-int-consts-divide-self'*:

(*adelic-int-consts f*) / (*adelic-int-consts f*) $\simeq_p$ *1*
if *f p* $\neq$ *0* **and** $(f\ p)^{\circ p}$ = *0*
for *f* :: *'a::nontrivial-factorial-unique-euclidean-bezout prime* $\Rightarrow$ *'a*
using *that adelic-int-consts-divide-self[of f] global-p-depth-adelic-int.p-restrict-equiv[of - p]*
by *presburger*

lemma *adelic-int-divide-depth*:
$(x\ /\ y)^{\circ p} = (x^{\circ p})$ **if** *x / y* $\neg\simeq_p$ *0*
for *x y* :: *'a::nontrivial-factorial-unique-euclidean-bezout adelic-int* **and** *p* :: *'a prime*
proof (*cases x y rule*: *two-Abs-adelic-int-cases*)
case (*Abs-adelic-int-Abs-adelic-int a b*)
from *that Abs-adelic-int-Abs-adelic-int(3,4)* **have** *db*: $b^{\circ p}$ = *0*
using *p-depth-adelic-int-abs-eq adelic-int-depth[of p y] adelic-int-divide-by-pos-depth[of p y]*
by *fastforce*
moreover have *a / b* $\neg\simeq_p$ *0*
proof
assume *a / b* $\simeq_p$ *0* **with** *that Abs-adelic-int-Abs-adelic-int db* **show** *False*
using *global-p-depth-p-adic-prod.p-restrict-equiv[of - p a / b]*
p-equality-p-adic-prod.trans[of p - a / b 0]
p-adic-prod-div-inv.global-depth-set-divide[of a 0 b]
p-equal-adelic-int-abs-eq0 divide-adelic-int-abs-eq[of a b]
by *force*
qed
ultimately show $(x\ /\ y)^{\circ p} = (x^{\circ p})$
using *Abs-adelic-int-Abs-adelic-int divide-adelic-int-abs-eq*
p-adic-prod-div-inv.global-depth-set-divide[of - 0] p-depth-adelic-int-abs-eq[of - p]
p-adic-prod-div-inv.divide-depth[of p a b]
global-p-depth-p-adic-prod.depth-equiv[of p - a / b]
global-p-depth-p-adic-prod.p-restrict-equiv[of - p a / b]
by *fastforce*
qed

lemma *global-unfrmzr-pows-adelic-int-inverse*:
inverse ($\mathfrak{p}$ *f* :: *'a adelic-int*) = *1 prestrict* (λp::*'a prime*. *f p* = *0*)
for *f* :: *'a::nontrivial-factorial-unique-euclidean-bezout prime* $\Rightarrow$ *nat*
proof (*intro global-p-depth-adelic-int.global-imp-eq*, *standard*)
fix *p* :: *'a prime*
define *P* :: *'a prime* $\Rightarrow$ *bool* **where** *P* $\equiv$ (λp. *f p* = *0*)
define $\mathfrak{p}$*nif* :: *'a p-adic-prod* **and** $\mathfrak{p}$*f* :: *'a adelic-int*
where $\mathfrak{p}$*nif* $\equiv$ $\mathfrak{p}$ ($-$ (*int* $\circ$ *f*)) **and** $\mathfrak{p}$*f* $\equiv$ $\mathfrak{p}$ *f*
from *P-def* $\mathfrak{p}$*nif-def* $\mathfrak{p}$*f-def* **have** *inverse* $\mathfrak{p}$*f* = *Abs-adelic-int* ($\mathfrak{p}$*nif prestrict P*)
using *global-unfrmzr-pows-p-adic-prod-global-depth-set-0[of f]*
inverse-adelic-int-abs-eq[of $\mathfrak{p}$ (*int* $\circ$ *f*)]
global-unfrmzr-pows-p-adic-prod[of - int $\circ$ *f*]
global-unfrmzr-pows-p-adic-prod-inverse[of int $\circ$ *f*]

unfolding *global-unfrmzr-pows-adelic-int-def*
by *fastforce*
moreover from *P-def* $\mathfrak{p}$*nif-def* **have** $\mathfrak{p}$*nif prestrict P* $\simeq_p$ *1 prestrict P*
using *global-p-depth-p-adic-prod.p-restrict-p-equal-p-restrict-by-sameI*[*of P p*]
trivial-global-unfrmzr-pows-p-adic-prod[*of - p*]
by *force*
moreover have $\mathfrak{p}$*nif prestrict P* $\in \mathcal{O}_{\forall\, p}$
proof (*intro p-adic-prod-global-depth-set-p-restrictI*)
fix p :: *'a prime* **assume** *P p*
moreover from $\mathfrak{p}$*nif-def* **have** $\mathfrak{p}$*nif* $\in$ *p-depth-set p* ($-$ (*int* $\circ$ *f*) *p*)
using *global-unfrmzr-pows-p-adic-prod-depth-set*[*of* $-$ (*int* $\circ$ *f*)] **by** *simp*
ultimately show $\mathfrak{p}$*nif* $\in \mathcal{O}_{@p}$ **using** *P-def* **by** *fastforce*
qed
ultimately show (*inverse* $\mathfrak{p}$*f*) $\simeq_p$ (*1 prestrict P*)
using *one-adelic-int-def global-p-depth-p-adic-prod.global-depth-set-1*
p-equal-adelic-int-abs-eq p-restrict-adelic-int-abs-eq global-depth-set-p-adic-prod-def
global-p-depth-p-adic-prod.global-depth-set-p-restrict
by *metis*
qed

lemma *adelic-int-diff-cancel-lead-coeff*:
((*inverse x* $-$ *inverse y*)$^{\circ p}$) $>$ *0*
if *x* : *x* $\neg\simeq_p$ *0* $x^{\circ p}$ = *0*
and *y* : *y* $\neg\simeq_p$ *0* $y^{\circ p}$ = *0*
and *xy*: *x* $\neg\simeq_p$ *y* ((*x* $-$ *y*)$^{\circ p}$) $>$ *0*
for *p* :: *'a::nontrivial-factorial-unique-euclidean-bezout prime*
and *x y* :: *'a adelic-int*
proof (*cases x y rule*: *two-Abs-adelic-int-cases*)
define *P* :: *'a p-adic-prod* $\Rightarrow$ *'a prime* $\Rightarrow$ *bool*
where *P* $\equiv$ λ*a p*. $a^{\circ p}$ = *0*
case (*Abs-adelic-int-Abs-adelic-int a b*)
moreover from *this x*(*1*) *y*(*1*) *xy*(*1*)
have *a* $\neg\simeq_p$ *0*
and *b* $\neg\simeq_p$ *0*
and *a* $\neg\simeq_p$ *b*
using *p-equal-adelic-int-abs-eq0 p-equal-adelic-int-abs-eq*
by (*fast*, *fast*, *fast*)
moreover from *Abs-adelic-int-Abs-adelic-int x*(*2*) *y*(*2*) *xy*(*2*)
have $a^{\circ p}$ = *0*
and $b^{\circ p}$ = *0*
and ((*a* $-$ *b*)$^{\circ p}$) $>$ *0*
by (
metis p-depth-adelic-int-abs-eq, *metis p-depth-adelic-int-abs-eq*,
metis p-depth-adelic-int-diff-abs-eq
)
ultimately show *?thesis*
using *p-depth-adelic-int-inverse-diff-abs-eq p-adic-prod-diff-cancel-lead-coeff* **by**
fastforce
qed

5.8.5 The ideal of primes

overloading
p-depth-set-adelic-int $\equiv$
p-depth-set ::
$'a$::*nontrivial-factorial-idom prime* $\Rightarrow$ *int* $\Rightarrow$ $'a$ *adelic-int set*
global-depth-set-adelic-int $\equiv$
global-depth-set :: *int* $\Rightarrow$ $'a$ *adelic-int set*
begin

definition *p-depth-set-adelic-int* ::
$'a$::*nontrivial-factorial-idom prime* $\Rightarrow$ *int* $\Rightarrow$ $'a$ *adelic-int set*
where *p-depth-set-adelic-int-def*[*simp*]:
p-depth-set-adelic-int = *global-p-depth-adelic-int.p-depth-set*

definition *global-depth-set-adelic-int* ::
int $\Rightarrow$ $'a$::*nontrivial-factorial-idom adelic-int set*
where *global-depth-set-adelic-int-def*[*simp*]:
global-depth-set-adelic-int = *global-p-depth-adelic-int.global-depth-set*

end

context
fixes x :: $'a$::*nontrivial-factorial-idom adelic-int*
begin

lemma *adelic-int-global-depth-setI*:
$x \in \mathcal{P}_{\forall\, p}{}^{n}$
if $\bigwedge p$::$'a$ *prime*. $x \neg\simeq_p 0 \Longrightarrow (x^{\circ p}) \geq n$
using *that global-p-depth-adelic-int.global-depth-setI* **by** *auto*

lemma *adelic-int-global-depth-setI'*:
$x \in \mathcal{P}_{\forall\, p}{}^{n}$
if $\bigwedge p$::$'a$ *prime*. $(x^{\circ p}) \geq n$
using *that global-p-depth-adelic-int.global-depth-setI'* **by** *auto*

lemma *adelic-int-global-depth-set-p-restrictI*:
p-restrict x $P \in \mathcal{P}_{\forall\, p}{}^{n}$
if $\bigwedge p$::$'a$ *prime*. $P\ p \Longrightarrow x \in \mathcal{P}_{@p}{}^{n}$
using *that global-p-depth-adelic-int.global-depth-set-p-restrictI* **by** *auto*

end

context
fixes p :: $'a$::*nontrivial-factorial-idom prime*
begin

lemma *adelic-int-p-depth-setI*:
$x \in \mathcal{P}_{@p}{}^{n}$
if $x \neg\simeq_p 0 \longrightarrow (x^{\circ p}) \geq n$

for x :: $'a$ *adelic-int*
using *that global-p-depth-adelic-int.p-depth-setI* **by** *auto*

lemma *nonpos-p-depth-set-adelic-int*:
$(\mathcal{P}_{@p}{}^{n}) = (UNIV :: 'a::nontrivial\text{-}factorial\text{-}idom\ adelic\text{-}int\ set)$
if $n \le 0$
proof *safe*
fix x :: $'a$ *adelic-int* **show** $x \in (\mathcal{P}_{@p}{}^{n})$
proof (*intro adelic-int-p-depth-setI*, *clarify*)
from *that* **show** $n \le (x^{\circ p})$
using *adelic-int-depth*[*of p x*] **by** *fastforce*
qed
qed *simp*

lemma *p-depth-set-adelic-int-eq-projection*:
$((\mathcal{P}_{@p}{}^{n}) :: 'a::nontrivial\text{-}factorial\text{-}idom\ adelic\text{-}int\ set) =$
Abs-adelic-int ‘ $((\mathcal{P}_{@p}{}^{n}) \cap \mathcal{O}_{\forall\ p})$
proof *safe*
fix x :: $'a$ *adelic-int* **assume** $x \in \mathcal{P}_{@p}{}^{n}$
thus
$x \in$
Abs-adelic-int ‘ $((\mathcal{P}_{@p}{}^{n}) \cap \mathcal{O}_{\forall\ p})$
using *p-equal-adelic-int-abs-eq0 global-p-depth-adelic-int.p-depth-setD*
p-depth-adelic-int-abs-eq p-adic-prod-p-depth-setI
by (*cases x*) *fastforce*
next
fix a :: $'a$ *p-adic-prod*
assume $a \in \mathcal{P}_{@p}{}^{n}$ $a \in \mathcal{O}_{\forall\ p}$
thus *Abs-adelic-int* $a \in \mathcal{P}_{@p}{}^{n}$
using *p-equal-adelic-int-abs-eq0 global-p-depth-p-adic-prod.p-depth-setD*
p-depth-adelic-int-abs-eq
by (*fastforce intro*: *adelic-int-p-depth-setI*)
qed

lemma *lift-p-depth-set-adelic-int*:
Rep-adelic-int ‘
$((\mathcal{P}_{@p}{}^{n}) :: 'a::nontrivial\text{-}factorial\text{-}idom\ adelic\text{-}int\ set)$
$= (\mathcal{P}_{@p}{}^{n}) \cap \mathcal{O}_{\forall\ p}$
proof−
have
Rep-adelic-int ‘ *Abs-adelic-int* ‘
$((\mathcal{P}_{@p}{}^{n}) \cap \mathcal{O}_{\forall\ p})$
$= (\mathcal{P}_{@p}{}^{n}) \cap \mathcal{O}_{\forall\ p}$
using *Abs-adelic-int-inverse* **by** *force*
thus *?thesis* **using** *p-depth-set-adelic-int-eq-projection* **by** *metis*
qed

end

lemma *nonpos-global-depth-set-adelic-int*:
$(\mathcal{P}_{\forall\, p}{}^{n}) = (\mathit{UNIV} :: {}'a{::}\mathit{nontrivial\text{-}factorial\text{-}idom}\ \mathit{adelic\text{-}int}\ \mathit{set})$
if $n \leq 0$
using *that nonpos-p-depth-set-adelic-int global-p-depth-adelic-int.global-depth-set*
by *fastforce*

lemma *nonneg-global-depth-set-adelic-int-eq-projection*:
$((\mathcal{P}_{\forall\, p}{}^{n}) :: {}'a{::}\mathit{nontrivial\text{-}factorial\text{-}idom}\ \mathit{adelic\text{-}int}\ \mathit{set}) =$
$\mathit{Abs\text{-}adelic\text{-}int}$ ‘ $\mathcal{P}_{\forall\, p}{}^{n}$
if *n*: $n \geq 0$
proof *safe*
fix $x :: {}'a\ \mathit{adelic\text{-}int}$ **assume** *x*: $x \in \mathcal{P}_{\forall\, p}{}^{n}$
show $x \in \mathit{Abs\text{-}adelic\text{-}int}$ ‘ $\mathcal{P}_{\forall\, p}{}^{n}$
proof (*cases x*)
case (*Abs-adelic-int a*)
moreover from *this x* **have** $a \in \mathcal{P}_{\forall\, p}{}^{n}$
using *p-equal-adelic-int-abs-eq0 global-p-depth-adelic-int.global-depth-setD p-depth-adelic-int-abs-eq*
by (*fastforce intro*: *p-adic-prod-global-depth-setI*)
ultimately show *?thesis* **by** *fast*
qed
next
fix $a :: {}'a\ \mathit{p\text{-}adic\text{-}prod}$ **assume** $a \in \mathcal{P}_{\forall\, p}{}^{n}$
moreover from *this n* **have** $a \in \mathcal{O}_{\forall\, p}$
using *global-p-depth-p-adic-prod.global-depth-set-antimono* **by** *auto*
ultimately show *Abs-adelic-int a* $\in \mathcal{P}_{\forall\, p}{}^{n}$
using *n p-equal-adelic-int-abs-eq0 global-p-depth-p-adic-prod.global-depth-setD p-depth-adelic-int-abs-eq*
by (*fastforce intro*: *adelic-int-global-depth-setI*)
qed

lemma *lift-nonneg-global-depth-set-adelic-int*:
Rep-adelic-int ‘
$((\mathcal{P}_{\forall\, p}{}^{n}) :: {}'a{::}\mathit{nontrivial\text{-}factorial\text{-}idom}\ \mathit{adelic\text{-}int}\ \mathit{set})$
$= \mathcal{P}_{\forall\, p}{}^{n}$
if *n*: $n \geq 0$
proof−
from *n* **have**
Rep-adelic-int ‘ *Abs-adelic-int* ‘
$(\mathcal{P}_{\forall\, p}{}^{n})$
$= \mathcal{P}_{\forall\, p}{}^{n}$
using *global-p-depth-p-adic-prod.global-depth-set-antimono Abs-adelic-int-inverse*
by *force*
with *n* **show** *?thesis* **using** *nonneg-global-depth-set-adelic-int-eq-projection* **by** *metis*
qed

5.8.6 Topology p-open neighbourhoods

General properties of p-open sets

abbreviation *adelic-int-p-open-nbhd* ≡ *global-p-depth-adelic-int.p-open-nbhd*
abbreviation *adelic-int-p-open-nbhds* ≡ *global-p-depth-adelic-int.p-open-nbhds*
abbreviation *adelic-int-local-p-open-nbhds* ≡ *global-p-depth-adelic-int.local-p-open-nbhds*

lemma *adelic-int-nonpos-p-open-nbhd*:
  adelic-int-p-open-nbhd p n x = UNIV **if** $n \le 0$
  for *p* :: *'a::nontrivial-factorial-idom prime*
proof−
  from *that* **have** *adelic-int-p-open-nbhd p n x = adelic-int-p-open-nbhd p n 0*
    using *global-p-depth-adelic-int.p-open-nbhd-eq-circle*[*of 0 p n*]
        adelic-int-depth[*of p x*]
        global-p-depth-adelic-int.p-open-nbhd-circle-multicentre[*of x 0 p n*]
    by *simp*
  moreover have *adelic-int-p-open-nbhd p n 0 = UNIV*
  proof *safe*
    fix *y* :: *'a adelic-int* **from** *that* **show** *y* ∈ *adelic-int-p-open-nbhd p n 0*
    using *global-p-depth-adelic-int.p-open-nbhd-eq-circle*[*of 0 p n*] *adelic-int-depth*[*of p y*]
      by *auto*
  qed *simp*
  ultimately show *?thesis* **by** *presburger*
qed

lemma *adelic-int-p-open-nbhd-bound*:
  adelic-int-p-open-nbhd p n x = adelic-int-p-open-nbhd p (int (nat n)) x
  using *adelic-int-nonpos-p-open-nbhd*[*of n x p*] *adelic-int-nonpos-p-open-nbhd*[*of 0 x p*] **by** *simp*

lemma *adelic-int-p-open-nbhd-abs-eq*:
  adelic-int-p-open-nbhd p n (Abs-adelic-int x) =
    Abs-adelic-int ‘ (p-adic-prod-p-open-nbhd p n x ∩ $\mathcal{O}_{\forall\, p}$)
  if $x \in \mathcal{O}_{\forall\, p}$
  for *p* :: *'a::nontrivial-factorial-idom prime*
proof *safe*
  fix *y* **assume** *y*: *y* ∈ *adelic-int-p-open-nbhd p n (Abs-adelic-int x)*
  show
    y ∈ *Abs-adelic-int ‘ (p-adic-prod-p-open-nbhd p n x* ∩ $\mathcal{O}_{\forall\, p}$)
  proof (*cases y*)
    case (*Abs-adelic-int z*)
    with *that y* **have** *z* ∈ *p-adic-prod-p-open-nbhd p n x*
    using *global-p-depth-adelic-int.p-open-nbhd-eq-circle p-equal-adelic-int-abs-eq*[*of z x p*]
        p-depth-adelic-int-diff-abs-eq[*of z x p*]
        global-p-depth-p-adic-prod.p-open-nbhd-eq-circle
      by *fastforce*
    with *Abs-adelic-int* **show** *?thesis* **by** *blast*

qed
next
fix z **assume** z: $z \in$ *p-adic-prod-p-open-nbhd* p n x $z \in \mathcal{O}_{\forall\, p}$
with *that* **show** *Abs-adelic-int* $z \in$ *adelic-int-p-open-nbhd* p n (*Abs-adelic-int* x)
using *global-p-depth-adelic-int.p-open-nbhd-eq-circle p-equal-adelic-int-abs-eq*[*of*
z x p]
p-depth-adelic-int-diff-abs-eq[*of* z x p] *global-p-depth-p-adic-prod.p-open-nbhd-eq-circle*
by *fastforce*
qed

lemma *adelic-int-p-open-nbhd-rep-eq*:
Rep-adelic-int ‘ *adelic-int-p-open-nbhd* p n x =
p-adic-prod-p-open-nbhd p n (*Rep-adelic-int* x) $\cap\ \mathcal{O}_{\forall\, p}$
for p :: '*a*::*nontrivial-factorial-idom prime*
proof (*cases* x)
case (*Abs-adelic-int* z)
hence
Rep-adelic-int ‘ *adelic-int-p-open-nbhd* p n x =
Rep-adelic-int ‘ *Abs-adelic-int* ‘
(*p-adic-prod-p-open-nbhd* p n z $\cap\ \mathcal{O}_{\forall\, p}$)
using *adelic-int-p-open-nbhd-abs-eq* **by** *metis*
also have ... = *p-adic-prod-p-open-nbhd* p n z $\cap\ \mathcal{O}_{\forall\, p}$
using *Abs-adelic-int-inverse* **by** *force*
finally show *?thesis* **using** *Abs-adelic-int Abs-adelic-int-inverse* **by** *force*
qed

lemma *adelic-int-local-depth-ring-lift-cover*:
fixes n :: *int*
and p :: '*a*::*nontrivial-factorial-unique-euclidean-bezout prime*
and $\mathcal{A}$ :: '*a adelic-int set set*
defines
($\mathcal{A}'$ :: '*a p-adic-prod set set*) ≡
(λA.
(λx. x *prestrict* ((=) p)) ‘ *Rep-adelic-int* ‘ A +
range (λx. x *prestrict* (($\neq$) p))
) ‘ $\mathcal{A}$
assumes *nonneg*: $n \geq 0$
and *cover*:
(λx. x *prestrict* ((=) p)) ‘ ($\mathcal{P}_{\forall\, p}{}^{n}$) ⊆
$\bigcup\ \mathcal{A}$
shows
(λx. x *prestrict* ((=) p)) ‘ ($\mathcal{P}_{\forall\, p}{}^{n}$) ⊆
$\bigcup\ \mathcal{A}'$
proof *clarify*
fix x :: '*a p-adic-prod*
assume x: $x \in (\mathcal{P}_{\forall\, p}{}^{n})$
with *nonneg* **have** *x-O*: $x \in \mathcal{O}_{\forall\, p}$
using *global-p-depth-p-adic-prod.global-depth-set-antimono* **by** *auto*
from x *nonneg* **have** *res-x-P*: x *prestrict* ((=) p) $\in (\mathcal{P}_{\forall\, p}{}^{n})$

using *global-p-depth-p-adic-prod.global-depth-set-closed-under-p-restrict-image*
by *fastforce*
with *nonneg* **have** *res-x-O*: x *prestrict* $((=)\ p) \in \mathcal{O}_{\forall\ p}$
using *global-p-depth-p-adic-prod.global-depth-set-antimono* **by** *auto*
from *nonneg*
have *Abs-adelic-int* (x *prestrict* $((=)\ p)) \in (\mathcal{P}_{\forall\ p}{}^{n})$
using *res-x-P nonneg-global-depth-set-adelic-int-eq-projection*
by *fast*
moreover have
Abs-adelic-int (x *prestrict* $((=)\ p)) =$
(*Abs-adelic-int* (x *prestrict* $((=)\ p)))$ *prestrict* $((=)\ p)$
using *res-x-O p-restrict-adelic-int-abs-eq* **by** *fastforce*
ultimately obtain A **where** A: $A \in \mathcal{A}$ *Abs-adelic-int* (x *prestrict* $((=)\ p)) \in A$
using *cover* **by** *blast*
from $A(2)$ **have** x *prestrict* $((=)\ p) \in$ *Rep-adelic-int* ‘ A
using *res-x-O Abs-adelic-int-inverse* **by** *force*
moreover have x *prestrict* $((=)\ p) = (x$ *prestrict* $((=)\ p))$ *prestrict* $((=)\ p)$
using *global-p-depth-p-adic-prod.p-restrict-restrict′* **by** *simp*
moreover have *restr-0*: $(0$ :: $'a$ *p-adic-prod*$) = 0$ *prestrict* $((\neq)\ p)$
using *global-p-depth-p-adic-prod.p-restrict-zero* **by** *metis*
ultimately have
x *prestrict* $((=)\ p) + 0$ *prestrict* $((\neq)\ p) \in$
$(\lambda x.\ x$ *prestrict* $((=)\ p))$ ‘ *Rep-adelic-int* ‘ A +
range $(\lambda x.\ x$ *prestrict* $((\neq)\ p))$
by *fast*
with $A(1)$ $\mathcal{A}'$*-def* **show** x *prestrict* $((=)\ p) \in \bigcup \mathcal{A}'$ **using** *restr-0* **by** *auto*
qed

lemma *adelic-int-lift-local-p-open*:
fixes p :: $'a$::*nontrivial-factorial-unique-euclidean-bezout prime*
and A :: $'a$ *adelic-int set*
defines
$A' \equiv$
$(\lambda x.\ x$ *prestrict* $((=)\ p))$ ‘ *Rep-adelic-int* ‘ A +
range $(\lambda x.\ x$ *prestrict* $((\neq)\ p))$
assumes *adelic-int-p-open*: *generate-topology* (*adelic-int-local-p-open-nbhds* p) A
shows *generate-topology* (*p-adic-prod-local-p-open-nbhds* p) A'
unfolding A'*-def set-plus-def*
proof (*rule iffD2*, *rule global-p-depth-p-adic-prod.p-open-nbhds-open-subopen*, *clarify*)
define $f\ f'$ :: $'a$ *p-adic-prod* $\Rightarrow$ $'a$ *p-adic-prod*
where $f \equiv \lambda x.\ x$ *prestrict* $((=)\ p)$
and $f' \equiv \lambda x.\ x$ *prestrict* $((\neq)\ p)$
fix a b **assume** a: $a \in A$
with *adelic-int-p-open* **obtain** n **where** n: *adelic-int-p-open-nbhd* p (*int* (*nat* n)) $a \subseteq A$
using *global-p-depth-adelic-int.p-open-nbhds-open-subopen*[*of* p A]
adelic-int-p-open-nbhd-bound[*of* a p]
by *presburger*

hence ∗:
  p-adic-prod-p-open-nbhd p (*int* (*nat n*)) (*Rep-adelic-int a*)
    $\cap$ $\mathcal{O}_{\forall\,p}$
    $\subseteq$ *Rep-adelic-int* ‘ *A*
  using *adelic-int-p-open-nbhd-rep-eq* **by** *blast*
have
  p-adic-prod-p-open-nbhd p (*int* (*nat n*)) (*f* (*Rep-adelic-int a*) + *f′ b*) $\subseteq$
    f ‘ *Rep-adelic-int* ‘ *A* + *range f′*
proof
  fix *y*
  assume *y* $\in$ *p-adic-prod-p-open-nbhd p* (*int* (*nat n*)) (*f* (*Rep-adelic-int a*) + *f′*
b)
  with *f-def f′-def* **have** *y*: *y* $\in$ *p-adic-prod-p-open-nbhd p* (*int* (*nat n*)) (*Rep-adelic-int*
a)
    by (*simp add*: *global-p-depth-p-adic-prod.p-open-nbhd-p-restrict-add-mixed-drop-right*)
  have *f-y*: *f y* $\in$ $\mathcal{O}_{\forall\,p}$
    unfolding *f-def global-depth-set-p-adic-prod-def*
  proof (
    intro global-p-depth-p-adic-prod.global-depth-set-p-restrictI
        global-p-depth-p-adic-prod.p-depth-setI,
    clarify
)
    assume *y-n0*: *y* $\neg\simeq_p$ *0*
    show ($y^{\circ p}$) $\geq$ *0*
    proof (*cases y* $\simeq_p$ (*Rep-adelic-int a*))
      case *True* **thus** *?thesis*
      using *global-p-depth-p-adic-prod.depth-equiv p-depth-adelic-int-def adelic-int-depth*
        by *metis*
    next
      case *y-a-nequiv*: *False*
      hence *y-a-depth*: ((y − *Rep-adelic-int a*)$^{\circ p}$) $\geq$ *int* (*nat n*)
        using *y global-p-depth-p-adic-prod.p-open-nbhd-eq-circle* **by** *blast*
      show *?thesis*
      proof (*cases Rep-adelic-int a* $\simeq_p$ *0*)
        case *True* **thus** *?thesis*
          using *y-a-depth*
              global-p-depth-p-adic-prod.depth-diff-equiv0-right[*of p Rep-adelic-int*
a y]
          by *linarith*
      next
        case *False*
        from *y-n0* **have**
          ($y^{\circ p}$) < ((*Rep-adelic-int a*)$^{\circ p}$) $\Longrightarrow$ *?thesis*
          using *y-a-depth*
              global-p-depth-p-adic-prod.depth-pre-nonarch-diff-left[
                of p y Rep-adelic-int a
              ]
          by *linarith*
        moreover from *False* **have**

$(y^{\circ p}) \geq ((\mathit{Rep\text{-}adelic\text{-}int}\ a)^{\circ p}) \Longrightarrow$
  ?thesis
using *global-p-depth-p-adic-prod.depth-pre-nonarch-diff-right*[
    of p Rep-adelic-int a y
  ]
  p-depth-adelic-int-def[*of p a*] *adelic-int-depth*[*of p a*]
by *simp*
ultimately show *?thesis* **by** *fastforce*
qed
qed
qed
with *y f-def* **have** $f\ y \in \mathit{Rep\text{-}adelic\text{-}int}$ ‘ A
using ∗ *global-p-depth-p-adic-prod.p-open-nbhd-circle-multicentre′*[*of y*]
  global-p-depth-p-adic-prod.p-open-nbhd-circle-multicentre′[*of f y*]
  global-p-depth-p-adic-prod.p-open-nbhd-p-restrict
by *fast*
moreover from *f-def f′-def* **have** $y = f\ (f\ y) + f'\ y$
using *global-p-depth-p-adic-prod.p-restrict-restrict′*
  global-p-depth-p-adic-prod.p-restrict-decomp
by *auto*
ultimately show $y \in f$ ‘ *Rep-adelic-int* ‘ A + *range* f'
using *set-plus-def* **by** *fast*
qed
thus
$\exists n.$ *p-adic-prod-p-open-nbhd p n* (*f* (*Rep-adelic-int a*) + *f′ b*)
$\subseteq$ {*c*. ∃ *a*∈*f* ‘ *Rep-adelic-int* ‘ *A*. ∃ *b*∈*range f′*. *c* = *a* + *b*}
using *set-plus-def* **by** *blast*
qed

Sequences

abbreviation
adelic-int-p-limseq-condition ≡ *global-p-depth-adelic-int.p-limseq-condition*
abbreviation
adelic-int-p-cauchy-condition ≡ *global-p-depth-adelic-int.p-cauchy-condition*
abbreviation *adelic-int-p-limseq* ≡ *global-p-depth-adelic-int.p-limseq*
abbreviation *adelic-int-p-cauchy* ≡ *global-p-depth-adelic-int.p-cauchy*
abbreviation *adelic-int-p-open-nbhds-limseq* ≡ *global-p-depth-adelic-int.p-open-nbhds-LIMSEQ*

context
fixes *p* :: ′*a*::*nontrivial-factorial-idom prime*
and *X* :: *nat* ⇒ ′*a p-adic-prod*
and *Y* :: *nat* ⇒ ′*a adelic-int*
defines *Y* ≡ λ*n*. *Abs-adelic-int* (*X n*)
assumes *range-X*: *range X* ⊆ $\mathcal{O}_{\forall\ p}$
begin

lemma *adelic-int-p-limseq-condition-abs-eq*:
adelic-int-p-limseq-condition p Y (*Abs-adelic-int x*) =
p-adic-prod-p-limseq-condition p X x

if $x \in \mathcal{O}_{\forall\, p}$
proof (*standard*, *standard*)
fix n :: *int* **and** N :: *nat*
from *range-X* **have** $\forall\, n.\ X\ n \in \mathcal{O}_{\forall\, p}$ **by** *blast*
with *that Y-def* **have**
adelic-int-p-limseq-condition $p\ Y$ (*Abs-adelic-int* x) $n\ N$ =
$(\forall\, k \geq N.\ (X\ k)\ \neg\simeq_p\ x \longrightarrow$
$((X\ k\ -\ x)^{\circ p}) > n)$
using *global-p-depth-adelic-int.p-limseq-condition-def p-equal-adelic-int-abs-eq p-depth-adelic-int-diff-abs-eq*
by *metis*
thus
adelic-int-p-limseq-condition $p\ Y$ (*Abs-adelic-int* x) $n\ N$ =
p-adic-prod-p-limseq-condition $p\ X\ x\ n\ N$
using *global-p-depth-p-adic-prod.p-limseq-condition-def* **by** *blast*
qed

lemma *adelic-int-p-limseq-abs-eq*:
adelic-int-p-limseq $p\ Y$ (*Abs-adelic-int* x) = *p-adic-prod-p-limseq* $p\ X\ x$
if $x \in \mathcal{O}_{\forall\, p}$
using *that adelic-int-p-limseq-condition-abs-eq* **by** *auto*

lemma *adelic-int-p-cauchy-condition-abs-eq*:
adelic-int-p-cauchy-condition $p\ Y$ = *p-adic-prod-p-cauchy-condition* $p\ X$
proof (*standard*, *standard*)
fix n :: *int* **and** K :: *nat*
from *range-X* **have** $\forall\, n.\ X\ n \in \mathcal{O}_{\forall\, p}$ **by** *blast*
moreover from *this Y-def*
have $\forall\, n\ n'.\ ((Y\ n\ -\ Y\ n')^{\circ p}) = ((X\ n\ -\ X\ n')^{\circ p})$
using *p-equal-adelic-int-abs-eq p-depth-adelic-int-diff-abs-eq*
by *blast*
ultimately have
adelic-int-p-cauchy-condition $p\ Y\ n\ K$ =
$(\forall\, k1 \geq K.\ \forall\, k2 \geq K.$
$(X\ k1)\ \neg\simeq_p\ (X\ k2) \longrightarrow$
$((X\ k1\ -\ X\ k2)^{\circ p}) > n)$
using *Y-def global-p-depth-adelic-int.p-cauchy-condition-def p-equal-adelic-int-abs-eq*
by *metis*
thus *adelic-int-p-cauchy-condition* $p\ Y\ n\ K$ = *p-adic-prod-p-cauchy-condition* p
$X\ n\ K$
using *global-p-depth-p-adic-prod.p-cauchy-condition-def* **by** *blast*
qed

lemma *adelic-int-p-cauchy-abs-eq*: *adelic-int-p-cauchy* $p\ Y$ = *p-adic-prod-p-cauchy*
$p\ X$
using *adelic-int-p-cauchy-condition-abs-eq* **by** *simp*

end

lemma *adelic-int-p-open-nbhds-limseq-abs-eq*:
adelic-int-p-open-nbhds-limseq ($\lambda n.$ *Abs-adelic-int* (X n)) (*Abs-adelic-int* x) =
p-adic-prod-p-open-nbhds-limseq X x
if *range* $X \subseteq \mathcal{O}_{\forall\, p}$ **and** $x \in \mathcal{O}_{\forall\, p}$
using *that adelic-int-p-limseq-abs-eq*
global-p-depth-p-adic-prod.globally-limseq-iff-locally-limseq[*of* X x]
global-p-depth-adelic-int.globally-limseq-iff-locally-limseq[*of*
$\lambda n.$ *Abs-adelic-int* (X n) *Abs-adelic-int* x
]
by *blast*

5.8.7 Topology via global depth

The metric

instantiation *adelic-int* :: (*nontrivial-factorial-idom*) *metric-space*
begin

definition *dist* = *global-p-depth-adelic-int.bdd-global-dist*
declare *dist-adelic-int-def* [*simp*]

definition *uniformity* = *global-p-depth-adelic-int.bdd-global-uniformity*
declare *uniformity-adelic-int-def* [*simp*]

definition *open-adelic-int* :: *'a adelic-int set* $\Rightarrow$ *bool*
where
open-adelic-int U =
($\forall\, x \in U$.
eventually ($\lambda(x', y).\ x' = x \longrightarrow y \in U$) *uniformity*
)

instance
by (
standard,
simp-all add: *global-p-depth-adelic-int.bdd-global-uniformity-def open-adelic-int-def*
global-p-depth-adelic-int.metric-space-by-bdd-depth.dist-triangle2
)

end

abbreviation *adelic-int-global-depth* $\equiv$ *global-p-depth-adelic-int.bdd-global-depth*

lemma *adelic-int-bdd-global-depth-abs-eq*:
adelic-int-global-depth (*Abs-adelic-int* x) = *p-adic-prod-global-depth* x
if $x \in \mathcal{O}_{\forall\, p}$ **for** x :: *'a::nontrivial-factorial-idom p-adic-prod*
proof (*cases* $x = 0$)
case *True* **thus** *?thesis*
using *global-p-depth-adelic-int.bdd-global-depth-0*
global-p-depth-p-adic-prod.bdd-global-depth-0
unfolding *zero-adelic-int-def*

by *force*
next
case *False*
moreover from *this that* **have** *Abs-adelic-int* $x \neq 0$
using *Abs-adelic-int-inject global-p-depth-p-adic-prod.global-depth-set-0*
unfolding *zero-adelic-int-def*
by *auto*
ultimately show *?thesis*
using *that Abs-adelic-int-inject global-p-depth-adelic-int.bdd-global-depthD p-equal-adelic-int-abs-eq0 p-depth-adelic-int-abs-eq global-p-depth-p-adic-prod.bdd-global-depthD global-p-depth-adelic-int.global-eq-iff*
by *fastforce*
qed

lemma *dist-adelic-int-abs-eq*:
dist (*Abs-adelic-int* x) (*Abs-adelic-int* y) = *dist* x y
if $x \in \mathcal{O}_{\forall\, p}$ **and** $y \in \mathcal{O}_{\forall\, p}$
for x y :: *'a::nontrivial-factorial-idom p-adic-prod*
proof (*cases* $x = y$)
case *True* **with** *that* **show** *?thesis*
using *Abs-adelic-int-inject global-p-depth-adelic-int.bdd-global-dist-eqD global-p-depth-p-adic-prod.bdd-global-dist-eqD*
by *auto*
next
case *False* **with** *that* **show** *?thesis*
using *dist-adelic-int-def Abs-adelic-int-inject global-p-depth-adelic-int.bdd-global-distD minus-adelic-int-abs-eq global-p-depth-p-adic-prod.global-depth-set-minus adelic-int-bdd-global-depth-abs-eq global-depth-set-p-adic-prod-def global-p-depth-p-adic-prod.bdd-global-distD dist-p-adic-prod-def global-p-depth-p-adic-prod.global-eq-iff global-p-depth-adelic-int.global-eq-iff*
by *metis*
qed

lemma *adelic-int-limseq-abs-eq*:
($\lambda n.$ *Abs-adelic-int* (X n)) $\longrightarrow$ *Abs-adelic-int* x
$\longleftrightarrow$ $X \longrightarrow x$
if *range* $X \subseteq \mathcal{O}_{\forall\, p}$ **and** $x \in \mathcal{O}_{\forall\, p}$
proof−
from *that* **have** $\forall\, n.$ *dist* (*Abs-adelic-int* (X n)) (*Abs-adelic-int* x) = *dist* (X n) x
using *dist-adelic-int-abs-eq*[*of X - x*] **by** *blast*
thus *?thesis*
using *tendsto-iff*[*of* $\lambda n.$ *Abs-adelic-int* (X n) *Abs-adelic-int* x] *tendsto-iff*[*of* X x]
by *presburger*
qed

lemma *adelic-int-bdd-metric-LIMSEQ*:

```
  X ⟶ x = global-p-depth-adelic-int.metric-space-by-bdd-depth.LIMSEQ X x
  for X :: nat ⇒ 'a::nontrivial-factorial-idom adelic-int and x :: 'a adelic-int
  unfolding tendsto-iff global-p-depth-adelic-int.metric-space-by-bdd-depth.tendsto-iff
      dist-adelic-int-def
  by fast

lemma adelic-int-global-depth-set-lim-closed:
  x ∈ 𝒫∀p^n
  if lim : X ⟶ x
  and depth: ∀F k in sequentially. X k ∈ 𝒫∀p^n
  for X :: nat ⇒ 'a::nontrivial-factorial-unique-euclidean-bezout adelic-int
  and x :: 'a adelic-int
proof (
  cases n ≤ 0, use nonpos-global-depth-set-adelic-int in blast,
  cases X x rule: adelic-int-seq-cases[case-product Abs-adelic-int-cases]
)
  case False case (Abs-adelic-int-Abs-adelic-int F a)
  with lim have F ⟶ a using adelic-int-limseq-abs-eq by blast
  moreover have
    ∀F k in sequentially. F k ∈ 𝒫∀p^n
  proof−
    from depth obtain K where K: ∀k≥K. X k ∈ 𝒫∀p^n
      using eventually-sequentially by auto
    have ∀k≥K. F k ∈ 𝒫∀p^n
    proof clarify
      fix k assume k ≥ K
      with False Abs-adelic-int-Abs-adelic-int(1) K obtain z
        where z ∈ 𝒫∀p^n
        and   Abs-adelic-int (F k) = Abs-adelic-int z
        using nonneg-global-depth-set-adelic-int-eq-projection[of n]
        by    auto
      with False Abs-adelic-int-Abs-adelic-int(2)
        show  F k ∈ 𝒫∀p^n
        using Abs-adelic-int-inject[of F k z]
              global-p-depth-p-adic-prod.global-depth-set-antimono[of 0 n]
        by    auto
    qed
    thus ?thesis using eventually-sequentially by blast
  qed
  ultimately have a ∈ 𝒫∀p^n
    using p-adic-prod-global-depth-set-lim-closed by auto
  with False Abs-adelic-int-Abs-adelic-int(3) show ?thesis
    using nonneg-global-depth-set-adelic-int-eq-projection[of n] by auto
qed

lemma adelic-int-metric-ball-multicentre:
  ball y e = ball x e if y ∈ ball x e for x y :: 'a::nontrivial-factorial-idom adelic-int
  using     that global-p-depth-adelic-int.bdd-global-dist-ball-multicentre
  unfolding ball-def dist-adelic-int-def
```

by *blast*

lemma *adelic-int-metric-ball-at-0*:
ball (*0*::*'a*::*nontrivial-factorial-idom adelic-int*) (*inverse* (*2* ^ *n*)) = *global-depth-set* (*int n*)
using *global-p-depth-adelic-int.bdd-global-dist-ball-at-0*
unfolding *ball-def dist-adelic-int-def global-depth-set-adelic-int-def*
by *auto*

lemma *adelic-int-metric-ball-at-0-normalize*:
ball (*0*::*'a*::*nontrivial-factorial-idom adelic-int*) *e* =
global-depth-set $\lfloor$*log 2* (*inverse e*)$\rfloor$
if $e > 0$ **and** $e \leq 1$
using *that global-p-depth-adelic-int.bdd-global-dist-ball-at-0-normalize*
unfolding *ball-def dist-adelic-int-def global-depth-set-adelic-int-def*
by *blast*

lemma *adelic-int-metric-ball-translate*:
ball x e = *x* +*o ball 0 e* **for** *x* :: *'a*::*nontrivial-factorial-idom adelic-int*
using *global-p-depth-adelic-int.bdd-global-dist-ball-translate*
unfolding *ball-def dist-adelic-int-def*
by *blast*

lemma *adelic-int-metric-ball-UNIV*:
ball x e = *UNIV* **if** $e \geq 1$ **for** *x* :: *'a*::*nontrivial-factorial-idom adelic-int*
proof (*cases e* = *1*)
case *False* **with** *that* **have** $e > 1$ **by** *linarith*
thus *?thesis*
using *that global-p-depth-adelic-int.bdd-global-dist-ball-UNIV*
unfolding *ball-def dist-adelic-int-def*
by *blast*
next
case *True*
have *ball x e* = *x* +*o ball 0 e* **using** *adelic-int-metric-ball-translate* **by** *auto*
also from *True* **have** ... = *x* +*o* $\mathcal{O}_{\forall p}$
using *adelic-int-metric-ball-at-0-normalize* **by** *force*
finally show *?thesis*
using *global-p-depth-adelic-int.global-depth-set-elt-set-plus nonpos-global-depth-set-adelic-int*
by *fastforce*
qed

lemma *adelic-int-left-translate-metric-continuous*:
continuous-on UNIV ((+) *x*) **for** *x* :: *'a*::*nontrivial-factorial-idom adelic-int*
proof
fix *e* :: *real* **and** *z* :: *'a adelic-int*
assume *e*: $e > 0$
moreover have $\forall y.$ *dist y z* $< e \longrightarrow$ *dist* (*x* + *y*) (*x* + *z*) $\leq e$
using *global-p-depth-adelic-int.bdd-global-dist-left-translate-continuous*[*of*
- - *e x*

]
unfolding *dist-p-adic-prod-def*
by *fastforce*
ultimately show
$\exists\, d{>}0.\ \forall\, x'{\in}UNIV.$
$dist\ x'\ z < d \longrightarrow dist\ (x + x')\ (x + z) \leq e$
by *auto*
qed

lemma *adelic-int-right-translate-metric-continuous*:
continuous-on UNIV $(\lambda z.\ z + x)$ **for** x :: *'a::nontrivial-factorial-idom adelic-int*
proof−
have $(\lambda z.\ z + x) = (+)\ x$ **by** (*auto simp add*: *add.commute*)
thus *?thesis* **using** *adelic-int-left-translate-metric-continuous* **by** *metis*
qed

lemma *adelic-int-left-mult-bdd-metric-continuous*:
continuous-on UNIV $((*)\ x)$
for x :: *'a::nontrivial-factorial-idom adelic-int*
proof
fix e :: *real* **and** z :: *'a adelic-int*
assume e: $e > 0$
moreover define d **where** $d \equiv min\ (e * inverse\ 2)\ 1$
moreover have $inverse\ (2{::}real) = 2\ powi\ (-1)$ **using** *power-int-def* **by** *force*
ultimately have $\forall\, y.\ dist\ y\ z < d \longrightarrow dist\ (x * y)\ (x * z) \leq e$
using *global-p-depth-adelic-int.bdd-global-dist-bdd-mult-continuous*[*of e x 0*] *adelic-int-depth*
by *fastforce*
moreover from e *d-def* **have** $d > 0$ **by** *auto*
ultimately show
$\exists\, d{>}0.\ \forall\, y{\in}UNIV.\ dist\ y\ z < d \longrightarrow dist\ (x * y)\ (x * z) \leq e$
by *blast*
qed

lemma *adelic-int-bdd-metric-limseq-mult*:
$(\lambda k.\ w * X\ k) \longlongrightarrow (w * x)$ **if** $X \longlongrightarrow x$
for $w\ x$ :: *'a::nontrivial-factorial-idom adelic-int* **and** X :: $nat \Rightarrow$ *'a adelic-int*
using *that adelic-int-left-mult-bdd-metric-continuous continuous-on-tendsto-compose*
by *fastforce*

lemma *adelic-int-global-depth-set-open*:
open $((\mathcal{P}_{\forall\, p}{}^{n})$ :: *'a::nontrivial-factorial-idom adelic-int set*)
proof ($cases\ n \geq 0$)
case *True*
hence $(\mathcal{P}_{\forall\, p}{}^{n}) = ball\ (0{::}'a\ adelic\text{-}int)\ (inverse\ (2\ \hat{}\ nat\ n))$
using *adelic-int-metric-ball-at-0* **by** *fastforce*
thus *?thesis* **by** *simp*
next
case *False*
hence $(\mathcal{P}_{\forall\, p}{}^{n}) = (UNIV$ :: *'a adelic-int set*)

using *nonpos-global-depth-set-adelic-int*[*of n*] **by** *simp*
thus *?thesis* **using** *open-UNIV* **by** *fastforce*
qed

lemma *adelic-int-ball-abs-eq*:
Abs-adelic-int ‘ ball x e = ball (Abs-adelic-int x) e
if $x \in \mathcal{O}_{\forall\, p}$ **and** $e \leq 1$
for *x* :: *'a::nontrivial-factorial-idom p-adic-prod*
proof *safe*
fix *y* **assume** $y \in$ *ball x e*
moreover from *this that(2)* **have** $y \in$ *ball x 1* **by** *fastforce*
ultimately show *Abs-adelic-int y* $\in$ *ball (Abs-adelic-int x) e*
using *that(1) p-adic-prod-nonpos-global-depth-set-open mem-ball dist-adelic-int-abs-eq*
by *fastforce*
next
fix *y* **assume** $y \in$ *ball (Abs-adelic-int x) e*
with *that(1)* **show** $y \in$ *Abs-adelic-int ‘ ball x e*
unfolding *ball-def* **using** *dist-adelic-int-abs-eq* **by** *(cases y) fastforce*
qed

lemma *open-adelic-int-abs-eq*:
open U = open (Abs-adelic-int ‘ U) **if** $U \subseteq (\mathcal{O}_{\forall\, p})$
for *U* :: *'a::nontrivial-factorial-idom p-adic-prod set*
proof *(standard, standard, clarify)*
fix *u* **assume** *u*: $u \in U$
moreover assume *open U*
ultimately obtain *e* **where** *e*: $e > 0$ *ball u e* $\subseteq U$
by *(elim Elementary-Metric-Spaces.openE)*
define e' **where** $e' \equiv min\ e\ 1$
with *e(2)* **have** *ball u* $e' \subseteq U$ **by** *force*
with *that u e'-def* **have** *ball (Abs-adelic-int u)* $e' \subseteq$ *Abs-adelic-int ‘ U*
using *adelic-int-ball-abs-eq*[*of u e'*] **by** *auto*
moreover from *e(1) e'-def* **have** $e' > 0$ **by** *auto*
ultimately show $\exists\, e>0.$ *ball (Abs-adelic-int u) e* $\subseteq$ *Abs-adelic-int ‘ U*
using *e(1) e'-def* **by** *blast*
next
assume *U*: *open (Abs-adelic-int ‘ U)*
show *open U*
proof
have *inj-abs*: *inj-on Abs-adelic-int* $(\mathcal{O}_{\forall\, p})$
using *Abs-adelic-int-inject inj-onI* **by** *meson*
fix *u* **assume** *u*: $u \in U$
with *U* **obtain** *e* **where** *e*: $e > 0$ *ball (Abs-adelic-int u) e* $\subseteq$ *Abs-adelic-int ‘ U*
using *Elementary-Metric-Spaces.openE* **by** *blast*
define e' **where** $e' \equiv min\ e\ 1$
with *that e(2) u* **have** *Abs-adelic-int ‘ ball u* $e' \subseteq$ *Abs-adelic-int ‘ U*
using *adelic-int-ball-abs-eq*[*of u e'*] **by** *fastforce*
moreover from *that u e'-def* **have** *ball u* $e' \subseteq (\mathcal{O}_{\forall\, p})$

using *p-adic-prod-nonpos-global-depth-set-open*[*of 0*] **by** *fastforce*
ultimately have *ball u e'* $\subseteq$ *U* **using** *that inj-on-vimage-image-eq*[*OF inj-abs*]
by *blast*
moreover from *e*(*1*) *e'-def* **have** *e'* $>$ *0* **by** *simp*
ultimately show $\exists$ *e*>*0*. *ball u e* $\subseteq$ *U* **by** *blast*
qed
qed

Completeness

abbreviation *adelic-int-globally-cauchy* $\equiv$ *global-p-depth-adelic-int.globally-cauchy*
abbreviation
adelic-int-global-cauchy-condition $\equiv$ *global-p-depth-adelic-int.global-cauchy-condition*

lemma *adelic-int-global-cauchy-condition-abs-eq*:
adelic-int-global-cauchy-condition (λ*n*. *Abs-adelic-int* (*X n*)) =
p-adic-prod-global-cauchy-condition X
if *range X* $\subseteq$ $\mathcal{O}_{\forall\, p}$
unfolding *global-p-depth-adelic-int.global-cauchy-condition-def*
global-p-depth-p-adic-prod.global-cauchy-condition-def
by (
standard, *standard*,
metis that subsetD rangeI p-equal-adelic-int-abs-eq p-depth-adelic-int-diff-abs-eq
)

lemma *adelic-int-globally-cauchy-abs-eq*:
adelic-int-globally-cauchy (λ*n*. *Abs-adelic-int* (*X n*)) = *p-adic-prod-globally-cauchy*
X
if *range X* $\subseteq$ $\mathcal{O}_{\forall\, p}$
using *that adelic-int-global-cauchy-condition-abs-eq* **by** *metis*

lemma *adelic-int-globally-cauchy-vs-bdd-metric-Cauchy*:
adelic-int-globally-cauchy X = *Cauchy X*
for *X* :: *nat* $\Rightarrow$ *'a::nontrivial-factorial-idom adelic-int*
using *global-p-depth-adelic-int.globally-cauchy-vs-bdd-metric-Cauchy*
unfolding *global-p-depth-adelic-int.metric-space-by-bdd-depth.Cauchy-def Cauchy-def*
by *auto*

lemma *adelic-int-Cauchy-abs-eq*:
Cauchy (λ*n*. *Abs-adelic-int* (*X n*)) = *Cauchy X*
if *range X* $\subseteq$ $\mathcal{O}_{\forall\, p}$
for *X* :: *nat* $\Rightarrow$ *'a::nontrivial-factorial-idom p-adic-prod*
using *that adelic-int-globally-cauchy-vs-bdd-metric-Cauchy adelic-int-globally-cauchy-abs-eq*
p-adic-prod-globally-cauchy-vs-bdd-metric-Cauchy
by *fast*

abbreviation
adelic-int-cauchy-lim X $\equiv$
Abs-adelic-int (*p-adic-prod-cauchy-lim* (λ*n*. *Rep-adelic-int* (*X n*)))

lemma *adelic-int-bdd-metric-complete'*:
X $\longrightarrow$ *adelic-int-cauchy-lim X* **if** *Cauchy X*
for *X* :: *nat* $\Rightarrow$ *'a::nontrivial-factorial-unique-euclidean-bezout adelic-int*
proof (*cases X rule*: *adelic-int-seq-cases*)
case (*Abs-adelic-int F*)
with *that* **have** *F* $\longrightarrow$ *p-adic-prod-cauchy-lim F*
using *adelic-int-Cauchy-abs-eq p-adic-prod-bdd-metric-complete'* **by** *blast*
moreover from *this Abs-adelic-int*(*2*)
have *p-adic-prod-cauchy-lim F* $\in \mathcal{O}_{\forall\, p}$
using *eventually-sequentially p-adic-prod-global-depth-set-lim-closed*[*of F*]
by *force*
ultimately have *X* $\longrightarrow$ *Abs-adelic-int* (*p-adic-prod-cauchy-lim F*)
using *Abs-adelic-int adelic-int-limseq-abs-eq* **by** *blast*
moreover have $F = (\lambda n.$ *Rep-adelic-int* (*X n*))
unfolding *Abs-adelic-int*(*1*)
proof
fix *n* **from** *Abs-adelic-int*(*2*) **show** *F n* = *Rep-adelic-int* (*Abs-adelic-int* (*F n*))
using *Abs-adelic-int-inverse*[*of F n*] **by** *fastforce*
qed
ultimately show *?thesis* **by** *meson*
qed

lemma *adelic-int-bdd-metric-complete*:
complete (*UNIV* :: *'a::nontrivial-factorial-unique-euclidean-bezout adelic-int set*)
using *adelic-int-bdd-metric-complete' completeI* **by** *blast*

end

theory *Fin-Field-Product*

imports
Distinguished-Quotients
pAdic-Product

begin

6 Finite fields of prime order as quotients of the ring of integers of p-adic fields

6.1 The type

6.1.1 The prime ideal as the distinguished ideal of the ring of adelic integers

instantiation *adelic-int* ::
(*nontrivial-factorial-idom*) *comm-ring-1-with-distinguished-ideal*

begin

definition *distinguished-subset-adelic-int* :: *'a adelic-int set*
  where *distinguished-subset-adelic-int-def* [*simp*]:
    distinguished-subset-adelic-int $\equiv \mathcal{P}_{\forall\, p}$

instance
proof (*standard*, *safe*)
  show ($\mathcal{N}$ ::*'a adelic-int set*) = {} $\Longrightarrow$ *False*
    using *global-p-depth-adelic-int.global-depth-set-0* **by** *auto*
next
  fix *g h* :: *'a adelic-int*
  assume $g \in \mathcal{N}$ **and** $h \in \mathcal{N}$
  thus $g - h \in \mathcal{N}$
    using *global-p-depth-adelic-int.global-depth-set-minus* **by** *auto*
next
  fix *r x* :: *'a adelic-int*
  assume $x \in \mathcal{N}$
  thus $r * x \in \mathcal{N}$
   using *global-p-depth-adelic-int.global-depth-set-ideal nonpos-global-depth-set-adelic-int*
    by *fastforce*
next
  define *n* **where** $n \equiv$ (*1*::*'a adelic-int*)
  moreover assume $n \in \mathcal{N}$
  ultimately show *False* **using** *global-p-depth-adelic-int.pos-global-depth-set-1* **by** *force*
qed

end

lemma *distinguished-subset-adelic-int-inverse*:
  inverse $x = 0$ **if** $x \in \mathcal{P}_{\forall\, p}$
  for *x* :: *'a*::*nontrivial-factorial-unique-euclidean-bezout adelic-int*
  using *that global-p-depth-adelic-int.global-depth-setD adelic-int-inverse-eq0-iff* **by** *fastforce*

6.1.2 The finite field product type as a quotient

Create type wrapper *coset-by-dist-sbgrp* over *adelic-int*.

typedef (**overloaded**) *'a adelic-int-quot* =
  UNIV :: *'a*::*nontrivial-factorial-idom adelic-int coset-by-dist-sbgrp set*
  ..

abbreviation
  adelic-int-quot $\equiv \lambda x.$ *Abs-adelic-int-quot* (*distinguished-quotient-coset x*)
abbreviation *p-adic-prod-int-quot* $\equiv \lambda x.$ *adelic-int-quot* (*Abs-adelic-int x*)

lemma *adelic-int-quot-cases* [*case-names adelic-int-quot*]:
  obtains *y* **where** *x* = *adelic-int-quot y*

proof (*cases x*)
  case (*Abs-adelic-int-quot y*) **with** *that* **show** *thesis* **by** (*cases y*) *blast*
qed

lemma *Abs-adelic-int-quot-cases* [*case-names adelic-int-quot-Abs-adelic-int*]:
  obtains *z* **where** $x = $ *p-adic-prod-int-quot z* $z \in \mathcal{O}_{\forall\, p}$
proof (*cases x rule*: *adelic-int-quot-cases*)
  case (*adelic-int-quot y*) **with** *that* **show** *thesis* **by** (*cases y*) *blast*
qed

lemmas *two-adelic-int-quot-cases* = *adelic-int-quot-cases*[*case-product adelic-int-quot-cases*]
  and *three-adelic-int-quot-cases* =
   adelic-int-quot-cases[*case-product adelic-int-quot-cases*[*case-product adelic-int-quot-cases*]]

lemma *abs-adelic-int-eq-iff*:
  adelic-int-quot x = *adelic-int-quot y* $\longleftrightarrow$
    $y - x \in \mathcal{P}_{\forall\, p}$
  using *Abs-adelic-int-quot-inject distinguished-quotient-coset-eq-iff* **by** *force*

lemma *p-adic-prod-int-quot-eq-iff*:
  p-adic-prod-int-quot x = *p-adic-prod-int-quot y* $\longleftrightarrow$
    $y - x \in \mathcal{P}_{\forall\, p}$
  if $x \in \mathcal{O}_{\forall\, p}$ **and** $y \in \mathcal{O}_{\forall\, p}$
 using *that abs-adelic-int-eq-iff*[*of Abs-adelic-int x Abs-adelic-int y*] *Abs-adelic-int-inject*
     minus-adelic-int-abs-eq[*of y x*] *nonneg-global-depth-set-adelic-int-eq-projection*[*of*
1]
       global-p-depth-p-adic-prod.global-depth-set-minus[*of y 0 x*]
       global-p-depth-p-adic-prod.global-depth-set-antimono[*of 0 1*]
  by   *fastforce*

6.1.3 Algebraic instantiations

instantiation *adelic-int-quot* :: (*nontrivial-factorial-idom*) *comm-ring-1*
begin

definition *0* = *Abs-adelic-int-quot 0*

lemma *zero-adelic-int-quot-rep-eq*: *Rep-adelic-int-quot 0* = *0*
  using *Abs-adelic-int-quot-inverse* **by** (*simp add*: *zero-adelic-int-quot-def*)

lemma *zero-adelic-int-quot*: *0* = *adelic-int-quot 0*
  using *zero-adelic-int-quot-def zero-coset-by-dist-sbgrp.abs-eq* **by** *simp*

definition *1* ≡ *Abs-adelic-int-quot 1*

lemma *one-adelic-int-quot-rep-eq*: *Rep-adelic-int-quot 1* = *1*
  using *Abs-adelic-int-quot-inverse* **by** (*simp add*: *one-adelic-int-quot-def*)

lemma *one-adelic-int-quot*: *1* = *adelic-int-quot 1*

using *one-adelic-int-quot-def one-coset-by-dist-sbgrp.abs-eq* **by** *simp*

definition $x + y = \textit{Abs-adelic-int-quot}\ (\textit{Rep-adelic-int-quot}\ x + \textit{Rep-adelic-int-quot}\ y)$

lemma *plus-adelic-int-quot-rep-eq*:
$\textit{Rep-adelic-int-quot}\ (a + b) = \textit{Rep-adelic-int-quot}\ a + \textit{Rep-adelic-int-quot}\ b$
using *Rep-adelic-int-quot Abs-adelic-int-quot-inverse*
unfolding *plus-adelic-int-quot-def*
by *blast*

lemma *plus-adelic-int-quot-abs-eq*:
$\textit{Abs-adelic-int-quot}\ x + \textit{Abs-adelic-int-quot}\ y = \textit{Abs-adelic-int-quot}\ (x + y)$
using *Abs-adelic-int-quot-inverse*[*of x*] *Abs-adelic-int-quot-inverse*[*of y*]
unfolding *plus-adelic-int-quot-def*
by *simp*

lemma *plus-adelic-int-quot*: $\textit{adelic-int-quot}\ x + \textit{adelic-int-quot}\ y = \textit{adelic-int-quot}\ (x + y)$
using *plus-adelic-int-quot-abs-eq plus-coset-by-dist-sbgrp.abs-eq* **by** *fastforce*

definition $-x = \textit{Abs-adelic-int-quot}\ (- \textit{Rep-adelic-int-quot}\ x)$

lemma *uminus-adelic-int-quot-rep-eq*: $\textit{Rep-adelic-int-quot}\ (- x) = - \textit{Rep-adelic-int-quot}\ x$
using *Rep-adelic-int-quot Abs-adelic-int-quot-inverse*
unfolding *uminus-adelic-int-quot-def*
by *blast*

lemma *uminus-adelic-int-quot-abs-eq*: $- \textit{Abs-adelic-int-quot}\ x = \textit{Abs-adelic-int-quot}\ (- x)$
using *Abs-adelic-int-quot-inverse*[*of x*] **unfolding** *uminus-adelic-int-quot-def* **by** *simp*

lemma *uminus-adelic-int-quot*: $- \textit{adelic-int-quot}\ x = \textit{adelic-int-quot}\ (-x)$
using *uminus-adelic-int-quot-abs-eq uminus-coset-by-dist-sbgrp.abs-eq* **by** *fastforce*

definition *minus-adelic-int-quot* ::
$'a\ \textit{adelic-int-quot} \Rightarrow\ 'a\ \textit{adelic-int-quot} \Rightarrow\ 'a\ \textit{adelic-int-quot}$
where $\textit{minus-adelic-int-quot} \equiv (\lambda x\ y.\ x + - y)$

lemma *minus-adelic-int-quot-rep-eq*:
$\textit{Rep-adelic-int-quot}\ (x - y) = \textit{Rep-adelic-int-quot}\ x - \textit{Rep-adelic-int-quot}\ y$
using *plus-adelic-int-quot-rep-eq uminus-adelic-int-quot-rep-eq*
by (*simp add*: *minus-adelic-int-quot-def*)

lemma *minus-adelic-int-quot-abs-eq*:
$\textit{Abs-adelic-int-quot}\ x - \textit{Abs-adelic-int-quot}\ y = \textit{Abs-adelic-int-quot}\ (x - y)$

by (*simp add*:
minus-adelic-int-quot-def uminus-adelic-int-quot-abs-eq
plus-adelic-int-quot-abs-eq
)

lemma *minus-adelic-int-quot*: *adelic-int-quot x* − *adelic-int-quot y* = *adelic-int-quot* (*x* − *y*)
using *minus-adelic-int-quot-abs-eq minus-coset-by-dist-sbgrp.abs-eq* **by** *fastforce*

definition *x* ∗ *y* = *Abs-adelic-int-quot* (*Rep-adelic-int-quot x* ∗ *Rep-adelic-int-quot y*)

lemma *times-adelic-int-quot-rep-eq*:
Rep-adelic-int-quot (*x* ∗ *y*) = *Rep-adelic-int-quot x* ∗ *Rep-adelic-int-quot y*
using *Rep-adelic-int-quot Abs-adelic-int-quot-inverse*
unfolding *times-adelic-int-quot-def*
by *blast*

lemma *times-adelic-int-quot-abs-eq*:
Abs-adelic-int-quot x ∗ *Abs-adelic-int-quot y* = *Abs-adelic-int-quot* (*x* ∗ *y*)
using *Abs-adelic-int-quot-inverse*[*of x*] *Abs-adelic-int-quot-inverse*[*of y*]
unfolding *times-adelic-int-quot-def*
by *simp*

lemma *times-adelic-int-quot*: *adelic-int-quot x* ∗ *adelic-int-quot y* = *adelic-int-quot* (*x* ∗ *y*)
using *times-adelic-int-quot-abs-eq times-coset-by-dist-sbgrp.abs-eq* **by** *auto*

instance
proof

fix *a b c* :: *'a adelic-int-quot*

show *a* + *b* + *c* = *a* + (*b* + *c*)
by (*cases a*, *cases b*, *cases c*) (*simp add*: *plus-adelic-int-quot-abs-eq add.assoc*)
show *a* + *b* = *b* + *a* **by** (*cases a*, *cases b*) (*simp add*: *plus-adelic-int-quot-abs-eq add.commute*)
show *0* + *a* = *a*
using *plus-adelic-int-quot-abs-eq*[*of 0*] **by** (*cases a*, *simp add*: *zero-adelic-int-quot-def*)
show *1* ∗ *a* = *a*
using *times-adelic-int-quot-abs-eq*[*of 1*] **by** (*cases a*) (*simp add*: *one-adelic-int-quot-def*)

show − *a* + *a* = *0*
by (*cases a*)
(*simp add*:
uminus-adelic-int-quot-abs-eq plus-adelic-int-quot-abs-eq zero-adelic-int-quot-def
)

show *a* − *b* = *a* + − *b* **by** (*simp add*: *minus-adelic-int-quot-def*)

show $a * b * c = a * (b * c)$
  by (*cases a*, *cases b*, *cases c*)
    (*simp add*: *times-adelic-int-quot-abs-eq mult.assoc*)

show $a * b = b * a$ **by** (*cases a*, *cases b*) (*simp add*: *times-adelic-int-quot-abs-eq mult.commute*)

show $(a + b) * c = a * c + b * c$
  by (*cases a*, *cases b*, *cases c*)
    (*simp add*: *plus-adelic-int-quot-abs-eq times-adelic-int-quot-abs-eq distrib-right*)

show $(0::'a\ adelic\text{-}int\text{-}quot) \neq 1$
  using *Abs-adelic-int-quot-inject*[*of 0*::'*a adelic-int coset-by-dist-sbgrp 1*]
  by (*simp add*: *zero-adelic-int-quot-def one-adelic-int-quot-def*)

qed

end

instantiation *adelic-int-quot* ::
  (*nontrivial-factorial-unique-euclidean-bezout*) {*divide-trivial*, *inverse*}
begin

lift-definition *inverse-adelic-int-coset* ::
  '*a adelic-int coset-by-dist-sbgrp* ⇒ '*a adelic-int coset-by-dist-sbgrp*
  is *inverse*
  unfolding *distinguished-subset-adelic-int-def*
proof
  fix *x y* :: '*a adelic-int*
  assume *xy*: $- y + x \in \mathcal{P}_{\forall p}$
  show $-$ *inverse y* $+$ *inverse x* $\in \mathcal{P}_{\forall p}$
  proof (*intro adelic-int-global-depth-setI*)
    fix *p* :: '*a prime*
    assume *xy*'$: -$ *inverse y* $+$ *inverse x* $\neg\simeq_p 0$
    have $((inverse\ x - inverse\ y)^{\circ p}) > 0$
    proof (*intro adelic-int-diff-cancel-lead-coeff*)
      from *xy*' **show** $x \neg\simeq_p y$
        using *p-equality-adelic-int.conv-0 adelic-int-inverse-pcong* **by** *fastforce*
      with *xy* **show** $((x - y)^{\circ p}) > 0$
        using *p-equality-adelic-int.conv-0 global-p-depth-adelic-int.global-depth-setD*
        by *fastforce*
      moreover from *this xy*' **show** $x \neg\simeq_p 0$ **and** $y \neg\simeq_p 0$
      using *adelic-int-inverse-equiv0-iff global-p-depth-adelic-int.depth-diff-equiv0-left*[*of p x*]
          global-p-depth-adelic-int.depth-diff-equiv0-right[*of p y*]
          p-equality-adelic-int.minus-0[*of p inverse x inverse y*]
        by (*force*, *force*)
      ultimately show $(x^{\circ p}) = 0$ **and** $(y^{\circ p}) = 0$

using *xy′ adelic-int-depth adelic-int-inverse-equiv0-iff*
global-p-depth-adelic-int.depth-pre-nonarch-diff-left[*of p x y*]
global-p-depth-adelic-int.depth-pre-nonarch-diff-right[*of p y x*]
p-equality-adelic-int.minus-0[*of p inverse x inverse y*]
by (*force*, *force*)
qed
thus $((-\ inverse\ y + inverse\ x)^{\circ p}) \geq 1$ **by** *force*
qed
qed *simp*

definition *inverse-adelic-int-quot* :: *′a adelic-int-quot* ⇒ *′a adelic-int-quot*
where
inverse-adelic-int-quot x ≡
Abs-adelic-int-quot (*inverse-adelic-int-coset* (*Rep-adelic-int-quot x*))

lemma *inverse-adelic-int-quot*: *inverse* (*adelic-int-quot x*) = *adelic-int-quot* (*inverse x*)
unfolding *inverse-adelic-int-quot-def*
by (*simp add*: *Abs-adelic-int-quot-inverse inverse-adelic-int-coset.abs-eq*)

lemma *inverse-adelic-int-quot-0*[*simp*]: *inverse* (*0* :: *′a adelic-int-quot*) = *0*
by (*metis zero-adelic-int-quot inverse-adelic-int-quot adelic-int-inverse0*)

lemma *inverse-adelic-int-quot-1*[*simp*]: *inverse* (*1* :: *′a adelic-int-quot*) = *1*
by (*metis one-adelic-int-quot inverse-adelic-int-quot adelic-int-inverse1*)

definition *divide-adelic-int-quot* ::
′a adelic-int-quot ⇒ *′a adelic-int-quot* ⇒ *′a adelic-int-quot*
where *divide-adelic-int-quot-def*[*simp*]: *divide-adelic-int-quot x y* ≡ *x* ∗ *inverse y*

instance by *standard simp-all*

end

lemma *divide-adelic-int-quot*: *adelic-int-quot x* / *adelic-int-quot y* = *adelic-int-quot* (*x* / *y*)
for *x y* :: *′a::nontrivial-factorial-unique-euclidean-bezout adelic-int*
using *times-adelic-int-quot inverse-adelic-int-quot*
unfolding *divide-adelic-int-quot-def divide-adelic-int-def*
by *metis*

6.2 Equivalence relative to a prime

6.2.1 Equivalence

overloading
p-equal-adelic-int-coset ≡
p-equal :: *′a::nontrivial-factorial-idom prime* ⇒
′a adelic-int coset-by-dist-sbgrp ⇒
′a adelic-int coset-by-dist-sbgrp ⇒ *bool*

p-equal-adelic-int-quot $\equiv$
p-equal :: *'a::nontrivial-factorial-idom prime* $\Rightarrow$
'a adelic-int-quot $\Rightarrow$ *'a adelic-int-quot* $\Rightarrow$ *bool*
p-restrict-adelic-int-coset $\equiv$
p-restrict :: *'a adelic-int coset-by-dist-sbgrp* $\Rightarrow$
(*'a prime* $\Rightarrow$ *bool*) $\Rightarrow$ *'a adelic-int coset-by-dist-sbgrp*
p-restrict-adelic-int-quot $\equiv$
p-restrict :: *'a adelic-int-quot* $\Rightarrow$
(*'a prime* $\Rightarrow$ *bool*) $\Rightarrow$ *'a adelic-int-quot*
begin

lift-definition *p-equal-adelic-int-coset* ::
'a::nontrivial-factorial-idom prime $\Rightarrow$
'a adelic-int coset-by-dist-sbgrp $\Rightarrow$
'a adelic-int coset-by-dist-sbgrp $\Rightarrow$ *bool*
is $\lambda p\ x\ y.\ x \simeq_p y \vee ((x - y)^{\circ p}) \geq 1$
unfolding *distinguished-subset-adelic-int-def*
proof$-$
fix p :: *'a prime* **and** $w\ x\ y\ z$ :: *'a adelic-int*
assume $-w + y \in \mathcal{P}_{\forall\, p}$ **and** $-x + z \in \mathcal{P}_{\forall\, p}$
moreover have
$\bigwedge w\ x\ y\ z$ :: *'a adelic-int*.
$y - w \in \mathcal{P}_{\forall\, p} \Longrightarrow$
$z - x \in \mathcal{P}_{\forall\, p} \Longrightarrow$
$w \simeq_p x \vee ((w - x)^{\circ p}) \geq 1 \Longrightarrow$
$((y - z)^{\circ p}) \geq 1 \vee y \simeq_p z$
proof *clarify*
fix $w\ x\ y\ z$ :: *'a adelic-int*
assume *wy* : $y - w \in \mathcal{P}_{\forall\, p}$
and *xz* : $z - x \in \mathcal{P}_{\forall\, p}$
and *equiv* : $w \simeq_p x \vee ((w - x)^{\circ p}) \geq 1$
and *nequiv*: $y \neg\simeq_p z$
have *1*: $y - z = w - x + ((y - w) - (z - x))$ **by** *simp*
from *wy xz* **have** *2*: $(y - w) - (z - x) \in \mathcal{P}_{\forall\, p}$
using *global-p-depth-adelic-int.global-depth-set-minus* **by** *auto*
show $((y - z)^{\circ p}) \geq 1$
proof (
cases $w \simeq_p x$ $(y - w) \simeq_p (z - x)$
rule: *case-split*[*case-product case-split*],
metis nequiv 1 p-equality-adelic-int.conv-0 p-equality-adelic-int.add-0,
metis 1 2 p-equality-adelic-int.conv-0 global-p-depth-adelic-int.depth-add-equiv0-left
global-p-depth-adelic-int.global-depth-setD global-depth-set-adelic-int-def,
metis equiv 1 p-equality-adelic-int.conv-0
global-p-depth-adelic-int.depth-add-equiv0-right
)
case *False-False*
hence *wx*: $w - x \neg\simeq_p 0$
and *wyxz*: $(y - w) - (z - x) \neg\simeq_p 0$
using *p-equality-adelic-int.conv-0*

by (*fast*, *fast*)
with *nequiv equiv False-False*(*1*) **show** *?thesis*
using *1 2 global-p-depth-adelic-int.depth-nonarch p-equality-adelic-int.conv-0 global-p-depth-adelic-int.global-depth-setD*[*of p* ($y - w$) $-$ ($z - x$) *1*]
by *fastforce*
qed
qed
ultimately show
$w \simeq_p x \vee ((w - x)^{\circ p}) \geq 1 \longleftrightarrow$
$y \simeq_p z \vee ((y - z)^{\circ p}) \geq 1$
by (*metis uminus-add-conv-diff symp-lcosetrel distinguished-subset-adelic-int-def*)
qed

definition *p-equal-adelic-int-quot* ::
'a::nontrivial-factorial-idom prime $\Rightarrow$
'a adelic-int-quot $\Rightarrow$ *'a adelic-int-quot* $\Rightarrow$ *bool*
where
p-equal-adelic-int-quot p x y $\equiv$
(*Rep-adelic-int-quot x*) $\simeq_p$ (*Rep-adelic-int-quot y*)

lift-definition *p-restrict-adelic-int-coset* ::
'a::nontrivial-factorial-idom adelic-int coset-by-dist-sbgrp $\Rightarrow$
(*'a prime* $\Rightarrow$ *bool*) $\Rightarrow$ *'a adelic-int coset-by-dist-sbgrp*
is $\lambda x\ P.\ x$ *prestrict P*
by (
simp add: *global-p-depth-adelic-int.global-depth-set-p-restrict*
flip: *global-p-depth-adelic-int.p-restrict-minus*
)

definition *p-restrict-adelic-int-quot* ::
'a::nontrivial-factorial-idom adelic-int-quot $\Rightarrow$
(*'a prime* $\Rightarrow$ *bool*) $\Rightarrow$ *'a adelic-int-quot*
where
p-restrict-adelic-int-quot x P $\equiv$
Abs-adelic-int-quot ((*Rep-adelic-int-quot x*) *prestrict P*)

end

context
fixes *p* :: *'a::nontrivial-factorial-idom prime*
begin

lemma *p-equal-adelic-int-coset-abs-eq0*:
distinguished-quotient-coset $x \simeq_p 0 \longleftrightarrow$
$x \simeq_p 0 \vee (x^{\circ p}) \geq 1$
for *x* :: *'a adelic-int*
unfolding *p-equal-adelic-int-coset.abs-eq zero-coset-by-dist-sbgrp.abs-eq* **by** *simp*

lemma *p-equal-adelic-int-coset-abs-eq1*:

distinguished-quotient-coset $x \simeq_p 1 \longleftrightarrow$
$x \simeq_p 1 \vee ((x - 1)^{\circ p}) \geq 1$
for x :: *'a adelic-int*
unfolding *p-equal-adelic-int-coset.abs-eq one-coset-by-dist-sbgrp.abs-eq* **by** *simp*

lemma *p-equal-adelic-int-quot-abs-eq*:
(*adelic-int-quot* x) $\simeq_p$ (*adelic-int-quot* y) $\longleftrightarrow$
$x \simeq_p y \vee ((x - y)^{\circ p}) \geq 1$
for x y :: *'a adelic-int*
using *Abs-adelic-int-quot-inverse*[*of distinguished-quotient-coset* x]
Abs-adelic-int-quot-inverse[*of distinguished-quotient-coset* y]
p-equal-adelic-int-coset.abs-eq
unfolding *p-equal-adelic-int-quot-def*
by *auto*

lemma *p-equal-adelic-int-quot-abs-eq0*:
adelic-int-quot $x \simeq_p 0 \longleftrightarrow$
$x \simeq_p 0 \vee (x^{\circ p}) \geq 1$
for x :: *'a adelic-int*
using *p-equal-adelic-int-quot-abs-eq* **unfolding** *zero-adelic-int-quot* **by** *fastforce*

lemma *p-equal-adelic-int-quot-abs-eq1*:
adelic-int-quot $x \simeq_p 1 \longleftrightarrow$
$x \simeq_p 1 \vee ((x - 1)^{\circ p}) \geq 1$
for x :: *'a adelic-int*
using *p-equal-adelic-int-quot-abs-eq* **unfolding** *one-adelic-int-quot* **by** *fastforce*

end

lemma *p-restrict-adelic-int-quot-abs-eq*:
(*adelic-int-quot* x) *prestrict* P = *adelic-int-quot* (x *prestrict* P)
for P :: *'a::nontrivial-factorial-idom prime* $\Rightarrow$ *bool* **and** x :: *'a adelic-int*
unfolding *p-restrict-adelic-int-quot-def*
by (*simp add*: *Abs-adelic-int-quot-inverse p-restrict-adelic-int-coset.abs-eq*)

global-interpretation *p-equality-adelic-int-quot*:
p-equality-no-zero-divisors-1
p-equal :: *'a::nontrivial-euclidean-ring-cancel prime* $\Rightarrow$
'a adelic-int-quot $\Rightarrow$ *'a adelic-int-quot* $\Rightarrow$ *bool*
proof

fix p :: *'a prime*

define E :: *'a adelic-int-quot* $\Rightarrow$ *'a adelic-int-quot* $\Rightarrow$ *bool*
where $E \equiv$ *p-equal* p
have *reflp* E
unfolding *E-def*
proof (*intro reflpI*)
fix x :: *'a adelic-int-quot* **show** $x \simeq_p x$

by (*cases x rule*: *adelic-int-quot-cases*, *simp add*: *p-equal-adelic-int-quot-abs-eq*)
qed
moreover have *symp E*
unfolding *E-def*
proof (*intro sympI*)
fix *x y* :: *'a adelic-int-quot*
show $x \simeq_p y \Longrightarrow y \simeq_p x$
by (
cases x y rule: *two-adelic-int-quot-cases*, *simp add*: *p-equal-adelic-int-quot-abs-eq*,
metis p-equality-adelic-int.sym global-p-depth-adelic-int.depth-diff
)
qed
moreover have *transp E*
unfolding *E-def*
proof (*intro transpI*)
fix *x y z* :: *'a adelic-int-quot*
assume *xy*: $x \simeq_p y$ **and** *yz*: $y \simeq_p z$
show $x \simeq_p z$
proof (*cases x y z rule*: *three-adelic-int-quot-cases*)
case (*adelic-int-quot-adelic-int-quot-adelic-int-quot a b c*)
have $a \simeq_p c \vee ((a - c)^{\circ p}) \geq 1$
proof (*rule iffD1*, *rule disj-commute*, *intro disjCI*)
assume *ac*: $a \neg\simeq_p c$
with *xy yz adelic-int-quot-adelic-int-quot-adelic-int-quot* **consider**
(*ab*) $a \simeq_p b$ $1 \leq ((b - c)^{\circ p})$ |
(*bc*) $b \simeq_p c$ $1 \leq ((a - b)^{\circ p})$ |
(*neither*) $1 \leq ((a - b)^{\circ p})$ $1 \leq ((b - c)^{\circ p})$
using *p-equal-adelic-int-quot-abs-eq p-equality-adelic-int.trans* **by** *meson*
thus $1 \leq ((a - c)^{\circ p})$
proof *cases*
case *ab* **thus** *?thesis* **using** *adelic-int-diff-depth-gt0-equiv-trans*[*of 0 p a b c*] **by** *simp*
next
case *bc* **thus** *?thesis*
using *adelic-int-diff-depth-gt0-equiv-trans*[*of 0 p c b a*] *p-equality-adelic-int.sym*
global-p-depth-adelic-int.depth-diff[*of p b a*]
global-p-depth-adelic-int.depth-diff[*of p a c*]
by *force*
next
case *neither* **with** *ac* **show** *?thesis*
using *adelic-int-diff-depth-gt0-trans*[*of 0 p a b c*] **by** *simp*
qed
qed
with *adelic-int-quot-adelic-int-quot-adelic-int-quot*(*1*,*3*) **show** *?thesis*
using *p-equal-adelic-int-quot-abs-eq* **by** *auto*
qed
qed
ultimately show *equivp E* **by** (*auto intro*: *equivpI*)

fix $x\ y$ :: $'a$ *adelic-int-quot*

show $(x \simeq_p y) = (x - y \simeq_p 0)$
  by (
    cases x y rule: *two-adelic-int-quot-cases*,
    simp add: *minus-adelic-int-quot p-equal-adelic-int-quot-abs-eq*
        p-equal-adelic-int-quot-abs-eq0,
    metis p-equality-adelic-int.conv-0
)

show $y \simeq_p 0 \Longrightarrow x * y \simeq_p 0$
proof (*cases x y rule*: *two-adelic-int-quot-cases*)
  case (*adelic-int-quot-adelic-int-quot a b*)
  assume $y \simeq_p 0$
  with *adelic-int-quot-adelic-int-quot*(2)
    have $b \simeq_p 0 \vee 1 \leq (b^{\circ p})$
    using *p-equal-adelic-int-quot-abs-eq0*
    by *fast*
  hence $a * b \simeq_p 0 \vee 1 \leq ((a * b)^{\circ p})$
    using *adelic-int-depth*[*of p a*] *global-p-depth-adelic-int.depth-mult-additive*
      p-equality-adelic-int.mult-0-right
    by *fastforce*
  with *adelic-int-quot-adelic-int-quot* **show** *?thesis*
    using *times-adelic-int-quot p-equal-adelic-int-quot-abs-eq0* **by** *metis*
qed

have $(1{::}'a$ *adelic-int-quot*$)\ \neg\simeq_p 0$
  unfolding *p-equal-adelic-int-quot-def*
  by (
    simp only: *zero-adelic-int-quot-rep-eq one-adelic-int-quot-rep-eq*, *transfer*,
    simp, *rule p-equality-adelic-int.one-p-nequal-zero*
)
thus $\exists x{::}'a$ *adelic-int-quot*. $x \neg\simeq_p 0$ **by** *blast*

show
  $x \neg\simeq_p 0 \Longrightarrow y \neg\simeq_p 0 \Longrightarrow$
    $x * y \neg\simeq_p 0$
proof (*cases x y rule*: *two-adelic-int-quot-cases*)
  case (*adelic-int-quot-adelic-int-quot a b*)
  assume x: $x \neg\simeq_p 0$ **and** y: $y \neg\simeq_p 0$
  with *adelic-int-quot-adelic-int-quot*
    have a: $a \neg\simeq_p 0\ (a^{\circ p}) < 1$
    and b: $b \neg\simeq_p 0\ (b^{\circ p}) < 1$
    using *p-equal-adelic-int-quot-abs-eq0*
    by (*fast*, *fastforce*, *fast*, *fastforce*)
  from $a(1)\ b(1)$ **have** $a * b \neg\simeq_p 0$
    using *p-equality-adelic-int.no-zero-divisors* **by** *blast*
  moreover from *this* $a(2)\ b(2)$ **have** $((a * b)^{\circ p}) < 1$
   using *adelic-int-depth global-p-depth-adelic-int.depth-mult-additive* **by** *fastforce*

ultimately have *adelic-int-quot* $(a * b) \neg\simeq_p 0$
using *p-equal-adelic-int-quot-abs-eq0* **by** *fastforce*
with *adelic-int-quot-adelic-int-quot times-adelic-int-quot* **show** $x * y \neg\simeq_p 0$
by *metis*
qed

qed

overloading
globally-p-equal-adelic-int-quot ≡
globally-p-equal ::
'a::nontrivial-euclidean-ring-cancel adelic-int-quot ⇒
'a adelic-int-quot ⇒ *bool*
begin

definition *globally-p-equal-adelic-int-quot* ::
'a::nontrivial-euclidean-ring-cancel adelic-int-quot ⇒
'a adelic-int-quot ⇒ *bool*
where *globally-p-equal-adelic-int-quot-def*[*simp*]:
globally-p-equal-adelic-int-quot = *p-equality-adelic-int-quot.globally-p-equal*

end

6.2.2 Embedding of constants

global-interpretation *global-p-equal-adelic-int-quot*:
global-p-equal-w-consts
p-equal :: *'a::nontrivial-euclidean-ring-cancel prime* ⇒
'a adelic-int-quot ⇒ *'a adelic-int-quot* ⇒ *bool*
p-restrict ::
'a adelic-int-quot ⇒ (*'a prime* ⇒ *bool*) ⇒
'a adelic-int-quot
λf. *adelic-int-quot* (*adelic-int-consts f*)
proof (*unfold-locales*, *fold globally-p-equal-adelic-int-quot-def*)

fix *p* :: *'a prime*
and *x y* :: *'a adelic-int-quot*

show $x \simeq_{\forall p} y \Longrightarrow x = y$
proof (*cases x y rule*: *two-adelic-int-quot-cases*)
case (*adelic-int-quot-adelic-int-quot a b*)
moreover assume *xy*: $x \simeq_{\forall p} y$
ultimately have
$\forall p$::*'a prime*.
$- a + b \neg\simeq_p 0 \longrightarrow ((- a + b)^{\circ p}) \geq 1$
using *p-equality-adelic-int.conv-0*[*of - b a*] *p-equal-adelic-int-quot-abs-eq*[*of - a b*]
p-equality-adelic-int.sym[*of - a b*]
global-p-depth-adelic-int.depth-diff[*of - a b*]

by *fastforce*
with *adelic-int-quot-adelic-int-quot* **show** $x = y$
using *distinguished-quotient-coset-eqI*[*of a b*] *global-p-depth-adelic-int.global-depth-setI*
by *fastforce*
qed

fix P :: *'a prime* $\Rightarrow$ *bool*

show $P\ p \Longrightarrow x\ prestrict\ P \simeq_p x$
by (
cases x rule: *adelic-int-quot-cases*,
simp add: *p-restrict-adelic-int-quot-abs-eq p-equal-adelic-int-quot-abs-eq*
global-p-depth-adelic-int.p-restrict-equiv
)

show $\neg\ P\ p \Longrightarrow x\ prestrict\ P \simeq_p 0$
by (
cases x rule: *adelic-int-quot-cases*,
simp add: *p-restrict-adelic-int-quot-abs-eq p-equal-adelic-int-quot-abs-eq0*,
metis global-p-depth-adelic-int.p-restrict-equiv0
)

fix $f\ g$ :: *'a prime* $\Rightarrow$ *'a*

show
adelic-int-quot (*adelic-int-consts* $(f - g)$) =
adelic-int-quot (*adelic-int-consts* f) $-$ *adelic-int-quot* (*adelic-int-consts* g)
using *global-p-depth-adelic-int.consts-diff minus-adelic-int-quot* **by** *metis*

show
adelic-int-quot (*adelic-int-consts* $(f * g)$) =
adelic-int-quot (*adelic-int-consts* f) $*$ *adelic-int-quot* (*adelic-int-consts* g)
using *global-p-depth-adelic-int.consts-mult times-adelic-int-quot* **by** *metis*

qed (*metis global-p-depth-adelic-int.consts-1 one-adelic-int-quot*)

abbreviation *adelic-int-quot-consts* $\equiv$ *global-p-equal-adelic-int-quot.consts*
abbreviation *adelic-int-quot-const* $\equiv$ *global-p-equal-adelic-int-quot.const*

lemma *p-equal-adelic-int-quot-consts-iff*:
(*adelic-int-quot-consts* f) $\simeq_p$ (*adelic-int-quot-consts* g) $\longleftrightarrow$
$(f\ p)$ *pmod* p = $(g\ p)$ *pmod* p
(**is** $?L \longleftrightarrow ?R$)
proof−
have $((\lambda p.\ fls\text{-}const\ (f\ p)) - (\lambda p.\ fls\text{-}const\ (g\ p)))\ p = fls\text{-}const\ (f\ p - g\ p)$
by (*simp add*: *fls-minus-const*)
have $?L \longleftrightarrow f\ p = g\ p \lor 1 \le int\ (pmultiplicity\ p\ ((f - g)\ p))$
using *p-equal-adelic-int-quot-abs-eq global-p-depth-adelic-int.consts-diff*
global-p-depth-adelic-int.p-equal-func-of-consts

global-p-depth-adelic-int.p-depth-func-of-consts
by *metis*
also have ... ⟷ *?R*
using *Rep-prime-not-unit multiplicity-gt-zero-iff*[*of f p* − *g p*]
mod-eq-dvd-iff[*of f p Rep-prime p g p*]
by *force*
finally show *?thesis* **by** *blast*
qed

lemma *p-equal0-adelic-int-quot-consts-iff*:
(*adelic-int-quot-consts f*) $\simeq_p$ *0* ⟷ (*f p*) *pmod p* = *0*
(**is** *?L* ⟷ *?R*)
proof−
have *?L* ⟷ *f p pmod p* = *0 pmod p*
by (
metis global-p-equal-adelic-int-quot.consts-0 p-equal-adelic-int-quot-consts-iff
zero-fun-apply
)
thus *?thesis* **by** *auto*
qed

lemma *adelic-int-quot-consts-eq-iff*:
adelic-int-quot-consts f = *adelic-int-quot-consts g* ⟷
(∀ *p*. (*f p*) *pmod p* = (*g p*) *pmod p*)
using *global-p-equal-adelic-int-quot.global-eq-iff p-equal-adelic-int-quot-consts-iff*[*of*
- f g]
by *auto*

lemma *adelic-int-quot-consts-eq0-iff*:
(*adelic-int-quot-consts f*) = *0* ⟷
(∀ *p*. (*f p*) *pmod p* = *0*)
using *global-p-equal-adelic-int-quot.global-eq-iff p-equal0-adelic-int-quot-consts-iff*[*of*
- f]
by *auto*

lemmas
p-equal-adelic-int-quot-const-iff =
p-equal-adelic-int-quot-consts-iff[*of - λp. - λp. -*]
and *p-equal0-adelic-int-quot-const-iff* = *p-equal0-adelic-int-quot-consts-iff*[*of - λp.*
-]
and *adelic-int-quot-const-eq-iff* = *adelic-int-quot-consts-eq-iff*[*of λp. - λp. -*]
and *adelic-int-quot-const-eq0-iff* = *adelic-int-quot-consts-eq0-iff*[*of λp. -*]

lemma *adelic-int-quot-UNIV-eq-consts-range*:
(*UNIV*::*'a*::*nontrivial-euclidean-ring-cancel adelic-int-quot set*) =
range (*adelic-int-quot-consts*)
proof (*standard*, *standard*)
fix *x* :: *'a adelic-int-quot*
show *x* ∈ *range adelic-int-quot-consts*

proof (*cases x rule*: *Abs-adelic-int-quot-cases*)
case (*adelic-int-quot-Abs-adelic-int a*)
have *p-adic-prod-int-quot a* $\in$ *range adelic-int-quot-consts*
proof (*cases a*)
case (*abs-p-adic-prod X*)
with *adelic-int-quot-Abs-adelic-int*(*2*)
have *dX*: $\forall$ *p*::*'a prime*. ($X^{\circ p}$) $\geq$ *0*
using *global-p-depth-p-adic-prod.nonpos-global-depth-setD*
p-depth-p-adic-prod.abs-eq[*of - X*]
by *fastforce*
define *f* :: *'a prime* $\Rightarrow$ *'a* **where** *f* $\equiv$ λp. (*X p pmod p*) $$ *0*
have *adelic-int-consts f* $-$ *Abs-adelic-int a* $\in$ $\mathcal{P}_{\forall\, p}$
unfolding *global-depth-set-adelic-int-def*
proof (*intro global-p-depth-adelic-int.global-depth-setI*)
fix *p* :: *'a prime*
assume (*adelic-int-consts f* $-$ *Abs-adelic-int a*) $\neg\simeq_p$ *0*
with *adelic-int-quot-Abs-adelic-int*(*2*) *abs-p-adic-prod*
have *nequiv*: (*fls-const* (*f p*)) $\neg\simeq_p$ (*X p pmod p*)
using *p-equality-adelic-int.conv-0*
global-p-depth-p-adic-prod.global-depth-set-consts p-equal-adelic-int-abs-eq
p-equal-p-adic-prod.abs-eq global-depth-set-p-adic-prod-def p-equal-fls-pseq-def
fls-pmod-equiv p-equality-fls.trans-left
by *metis*
have $\forall$ *k*<*0*. (*X p pmod p*) $$ *k* = *0*
proof *clarify*
fix *k* :: *int* **assume** *k* < *0*
moreover have (*X* $p^{\circ p}$) $\geq$ *0* **using** *dX* **by** *auto*
ultimately show (*X p pmod p*) $$ *k* = *0* **using** *fls-pmod-subdegree*[*of - p*]
by *simp*
qed
moreover have *fls-const* (*f p*) $\neq$ *X p pmod p* **using** *nequiv* **by** *auto*
ultimately have *fls-subdegree* (*fls-const* (*f p*) $-$ *X p pmod p*) > *0*
by (*auto intro*: *fls-subdegree-greaterI simp add*: *f-def*)
moreover from *adelic-int-quot-Abs-adelic-int*(*2*) *abs-p-adic-prod* **have**
((*adelic-int-consts f* $-$ *Abs-adelic-int a*)$^{\circ p}$) =
((*fls-const* (*f p*) $-$ *X p pmod p*)$^{\circ p}$)
using *global-p-depth-p-adic-prod.global-depth-set-consts minus-apply*[*of - X*
p]
p-depth-adelic-int-diff-abs-eq[*of - a p*] *p-depth-p-adic-prod-diff-abs-eq*[*of*
p - X]
p-depth-fls-pseq-def global-depth-set-p-adic-prod-def p-depth-fls.depth-equiv
p-equality-fls.minus-left fls-pmod-equiv
by *metis*
ultimately show ((*adelic-int-consts f* $-$ *Abs-adelic-int a*)$^{\circ p}$) $\geq$ *1*
using *nequiv p-equality-fls.conv-0 p-depth-fls-ge-p-equal-subdegree'* **by**
fastforce
qed
hence
p-adic-prod-int-quot a =

p-adic-prod-int-quot (*abs-p-adic-prod* (λp. *fls-const* (*f p*)))
using *Abs-adelic-int-quot-inject distinguished-quotient-coset-eqI* **by** *force*
thus *adelic-int-quot* (*Abs-adelic-int a*) $\in$ *range adelic-int-quot-consts* **by** *blast*
qed
with *adelic-int-quot-Abs-adelic-int*(*1*) **show** *?thesis* **by** *blast*
qed
qed *simp*

lemma *adelic-int-quot-constsE*:
fixes x :: *'a*::*nontrivial-euclidean-ring-cancel adelic-int-quot*
obtains f **where** x = *adelic-int-quot-consts f*
using *adelic-int-quot-UNIV-eq-consts-range* **by** *blast*

lemma *adelic-int-quot-reduced-constsE*:
fixes x :: *'a*::*nontrivial-euclidean-ring-cancel adelic-int-quot*
obtains f
where x = *adelic-int-quot-consts f*
and $\forall$ *p*::*'a prime*. (*f p*) *pdiv p* = *0*
proof−
obtain g **where** g: x = *adelic-int-quot-consts g*
using *adelic-int-quot-constsE* **by** *blast*
define f :: *'a prime* $\Rightarrow$ *'a* **where** $f \equiv \lambda p$. *g p pmod p*
hence $\forall$ *p*::*'a prime*. (*f p*) *pdiv p* = *0*
using *div-mult-mod-eq* **by** *fastforce*
moreover from *g f-def* **have** x = *adelic-int-quot-consts f*
using *adelic-int-quot-consts-eq-iff* **by** *force*
ultimately show *thesis* **using** *that* **by** *presburger*
qed

lemma *adelic-int-quot-prestrict-zero-depth-abs-eq*:
adelic-int-quot (*x prestrict* (λ*p*::*'a prime*. ($x^{\circ p}$) = *0*)) =
adelic-int-quot x
for x :: *'a*::*nontrivial-euclidean-ring-cancel adelic-int*
proof (*intro global-p-equal-adelic-int-quot.global-imp-eq*, *standard*)
fix p :: *'a prime*
define P :: *'a prime* $\Rightarrow$ *bool* **where** $P \equiv \lambda p$. ($x^{\circ p}$) = *0*
hence
x prestrict P $\neg\simeq_p$ $x \Longrightarrow$
((*x prestrict P* − x)$^{\circ p}$) $\geq$ *1*
using *adelic-int-prestrict-zero-depth* **by** *fast*
thus (*adelic-int-quot* (*x prestrict P*)) $\simeq_p$ (*adelic-int-quot x*)
using *p-equal-adelic-int-quot-abs-eq* **by** *blast*
qed

6.2.3 Division and inverses

global-interpretation *p-equal-div-inv-adelic-int-quot*:
global-p-equal-div-inv
p-equal :: *'a*::*nontrivial-factorial-unique-euclidean-bezout prime* $\Rightarrow$

$'a$ *adelic-int-quot* $\Rightarrow$ $'a$ *adelic-int-quot* $\Rightarrow$ *bool*
p-restrict ::
$'a$ *adelic-int-quot* $\Rightarrow$ ($'a$ *prime* $\Rightarrow$ *bool*) $\Rightarrow$
$'a$ *adelic-int-quot*
proof

define *A* :: $'a$ *adelic-int coset-by-dist-sbgrp* $\Rightarrow$ $'a$ *adelic-int-quot*
where *A* $\equiv$ *Abs-adelic-int-quot*

fix *p* :: $'a$ *prime* **and** *x y* :: $'a$ *adelic-int-quot*

show *inverse* (*inverse x*) = *x*
proof (*cases x rule*: *adelic-int-quot-cases*)
case (*adelic-int-quot a*)
moreover define *P* :: $'a$ *prime* $\Rightarrow$ *bool*
where $P \equiv \lambda p.\ (a^{\circ p}) = 0$
moreover from *this* **have** *adelic-int-quot* (*a prestrict P*) = *adelic-int-quot a*
using *global-p-equal-adelic-int-quot.global-imp-eq p-equal-adelic-int-quot-abs-eq*
adelic-int-prestrict-zero-depth
by *fast*
ultimately show *inverse* (*inverse x*) = *x*
using *inverse-adelic-int-quot adelic-int-inverse-inverse* **by** *metis*
qed

show (*inverse x* $\simeq_p$ *0*) = (*x* $\simeq_p$ *0*)
by (
cases x rule: *adelic-int-quot-cases*,
simp add: *A-def inverse-adelic-int-quot p-equal-adelic-int-quot-abs-eq0*
adelic-int-inverse-depth adelic-int-inverse-equiv0-iff'
)

show *x* $\neg\simeq_p$ *0* $\Longrightarrow$ *x* / *x* $\simeq_p$ *1*
proof (*cases x rule*: *adelic-int-quot-cases*)
case (*adelic-int-quot a*)
moreover assume *x* $\neg\simeq_p$ *0*
ultimately have *a* $\neg\simeq_p$ *0* **and** $a^{\circ p} = 0$
using *p-equal-adelic-int-quot-abs-eq0 adelic-int-depth*[*of p a*] **by** (*blast*, *fast-force*)
hence *a* / *a* $\simeq_p$ *1*
using *adelic-int-divide-self*[*of a*] *global-p-depth-adelic-int.p-restrict-equiv*[*of - p*]
by *presburger*
with *adelic-int-quot* **show** *x* / *x* $\simeq_p$ *1*
using *p-equal-adelic-int-quot-abs-eq1 divide-adelic-int-quot* **by** *metis*
qed

show *inverse* (*x* $*$ *y*) = *inverse x* $*$ *inverse y*
by (
cases x y rule: *two-adelic-int-quot-cases*,

simp add: *A-def inverse-adelic-int-quot times-adelic-int-quot adelic-int-inverse-mult*
)

qed *simp*

6.3 Special case of integer

context
fixes p :: *int prime*
begin

lemma *inj-on-const-prestrict-single-int-prime*:
inj-on (λk. *adelic-int-quot-const k prestrict* ((=) p)) $\{0..Rep\text{-}prime\ p - 1\}$
proof (*standard*, *unfold atLeastAtMost-iff*, *clarify*)
define *pp* :: *int* **and** C :: *int* ⇒ *int adelic-int-quot*
where *pp* ≡ *Rep-prime p*
and $C \equiv \lambda k$. *adelic-int-quot-const k prestrict* ((=) p)
fix j k :: *int*
show
$[\![j \geq 0; j \leq pp - 1; k \geq 0; k \leq pp - 1; C\ j = C\ k]\!]$
$\Longrightarrow j = k$
proof (*induct j k rule*: *linorder-wlog*)
fix j k :: *int*
assume $j : j \geq 0\ j \leq pp - 1$
and $k : k \geq 0\ k \leq pp - 1$
and jk: $j \leq k$ $C\ j = C\ k$
from *pp-def* $j(1)$ $k(2)$ $jk(1)$ **have** $((k - j)^{\circ p}) = 0$
using *p-depth-const*[*of p k* − *j*] *zdvd-imp-le*[*of pp k* − *j*] *not-dvd-imp-multiplicity-0*
by (*cases* $j = k$) *auto*
hence $\neg$ $((k - j)^{\circ p}) \geq 1$ **by** *presburger*
with *C-def* $jk(2)$ **show** $j = k$
using *global-p-equal-adelic-int-quot.p-equal-iff-p-restrict-eq adelic-int-p-depth-const*
global-p-depth-adelic-int.const-diff global-p-depth-adelic-int.const-p-equal[*of*
p]
p-equal-adelic-int-quot-abs-eq[*of p adelic-int-const k adelic-int-const j*]
by *metis*
qed *simp*
qed

lemma *range-prestrict-single-int-prime*:
range (λx. *x prestrict* ((=) p)) =
(λk. *adelic-int-quot-const k prestrict* ((=) p)) ‘ $\{0..Rep\text{-}prime\ p - 1\}$
proof *safe*
define *pp* :: *int* **and** C :: *int* ⇒ *int adelic-int-quot*
where *pp* ≡ *Rep-prime p*
and $C \equiv \lambda k$. *adelic-int-quot-const k prestrict* ((=) p)
fix x :: *int adelic-int-quot*
obtain f **where** f: x = *adelic-int-quot-consts f* (f p) *pdiv* $p = 0$
using *adelic-int-quot-reduced-constsE* **by** *blast*

from *f*(*2*) *pp-def* **have** *f p* ∈ {*0*..*pp* − *1*}
using *div-mult-mod-eq*[*of f p Rep-prime p*] *prime-gt-0-int Rep-prime pos-mod-sign*[*of pp f p*]
pos-mod-bound[*of pp f p*]
by *auto*
with *f*(*1*) **show** *x prestrict* ((=) *p*) ∈ *C* ‘ {*0*..*pp* − *1*}
using *p-equal-adelic-int-quot-consts-iff*
unfolding *C-def*
by (*fastforce intro*: *global-p-equal-adelic-int-quot.p-restrict-eq-p-restrictI*)
qed *simp*

lemma *finite-range-prestrict-single-int-prime*:
finite (*range* (*λx*::*int adelic-int-quot*. *x prestrict* ((=) *p*)))
using *range-prestrict-single-int-prime finite-image-iff inj-on-const-prestrict-single-int-prime*
by *simp*

lemma *card-range-prestrict-single-int-prime*:
card (*range* (*λx*::*int adelic-int-quot*. *x prestrict* ((=) *p*))) = *nat* (*Rep-prime p*)
using *bij-betw-imageI*[*of - - range* (*λx*. *x prestrict* ((=) *p*))] *bij-betw-same-card*
inj-on-const-prestrict-single-int-prime range-prestrict-single-int-prime[*symmetric*]
by *fastforce*

end

end

theory *More-pAdic-Product*

imports *Fin-Field-Product*

begin

7 Compactness of local ring of integers

7.1 Preliminaries

lemma *infinite-vimageE*:
fixes *f* :: *′a* ⇒ *′b*
assumes *infinite* (*UNIV* :: *′a set*) **and** *range f* ⊆ *B* **and** *finite B*
obtains *b* **where** *b* ∈ *B* **and** *infinite* (*f* −‘ {*b*})
proof−
from *assms*(*2*) **have** *f* −‘ *B* = *UNIV* **using** *subsetD rangeI* **by** *blast*
with *assms*(*1*,*3*) **have** ¬ (∀ *b*∈*B*. *finite* (*f* −‘ {*b*}))
using *finite-UN*[*of B λb*. *f* −‘ {*b*}] *vimage-eq-UN*[*of f B*] **by** *argo*
from *this* **obtain** *b* **where** *b* ∈ *B* **and** *infinite* (*f* −‘ {*b*}) **by** *blast*
with *that* **show** *?thesis* **by** *fast*
qed

— This is used to create a subsequence with outputs restricted to a specific image subset.

primrec *subset-subseq* ::
  $(nat \Rightarrow 'a) \Rightarrow 'a\ set \Rightarrow nat \Rightarrow nat$
  where
    $subset\text{-}subseq\ f\ A\ 0 = (LEAST\ k.\ f\ k \in A)$ |
    $subset\text{-}subseq\ f\ A\ (Suc\ n) = (LEAST\ k.\ k > subset\text{-}subseq\ f\ A\ n \wedge f\ k \in A)$

lemma
  assumes $infinite\ (f -` A)$
  shows *subset-subseq-strict-mono*: $strict\text{-}mono\ (subset\text{-}subseq\ f\ A)$
    and *subset-subseq-range* : $f\ (subset\text{-}subseq\ f\ A\ n) \in A$
proof−
  from *assms* **have** $f -` A \neq \{\}$ **by** *auto*
  hence $\exists k.\ f\ k \in A$ **by** *blast*
  hence *ex-first*:
    $\exists!k.\ f\ k \in A \wedge (\forall j.\ f\ j \in A \longrightarrow k \leq j)$
    using *ex-ex1-least-nat-le* **by** *simp*
  define P **where** $P \equiv \lambda N\ k.\ k > N \wedge f\ k \in A$
  have $\bigwedge N.\ (f -` A) \cap \{N<..\} \neq \{\}$
  proof
    fix N **assume** $(f -` A) \cap \{N<..\} = \{\}$
    hence $(f -` A) \subseteq \{..N\}$ **by** *fastforce*
    with *assms* **show** *False* **using** *finite-subset* **by** *fast*
  qed
  with *P-def* **have** $\bigwedge N.\ \exists k.\ P\ N\ k$ **by** *fast*
  hence *ex-next*:
    $\bigwedge N.\ \exists!k.\ P\ N\ k \wedge (\forall j.\ P\ N\ j \longrightarrow k \leq j)$
    using *ex-ex1-least-nat-le* **by** *force*

  from *P-def* **show** $f\ (subset\text{-}subseq\ f\ A\ n) \in A$
    using *Least1I*[*OF ex-first*] *Least1I*[*OF ex-next*] **by** (*cases n*) *auto*

  from *P-def* **have** *step*: $\bigwedge n.\ subset\text{-}subseq\ f\ A\ (Suc\ n) > subset\text{-}subseq\ f\ A\ n$
    using *Least1I*[*OF ex-next*] **by** *auto*

  show $strict\text{-}mono\ (subset\text{-}subseq\ f\ A)$
  proof
    fix $m\ n$ **show** $m < n \Longrightarrow subset\text{-}subseq\ f\ A\ m < subset\text{-}subseq\ f\ A\ n$
    proof (*induct n*)
      case (*Suc n*) **thus** *?case* **using** *step*[*of n*] **by** (*cases n = m*) *simp-all*
    qed *simp*
  qed

qed

7.2 Sequential compactness

7.2.1 Creating a Cauchy subsequence

abbreviation
p-adic-prod-int-convergent-subseq-seq-step p X g n ≡
λk. *p-adic-prod-int-quot* (*p-adic-prod-shift-p-depth p* (− *int n*) ((*X* (*g k*) − *X* (*g 0*))))

primrec *p-adic-prod-int-convergent-subseq-seq* ::
′*a*::*nontrivial-factorial-unique-euclidean-bezout prime* ⇒
(*nat* ⇒ ′*a p-adic-prod*) ⇒ *nat* ⇒ (*nat* ⇒ *nat*)
where *p-adic-prod-int-convergent-subseq-seq p X 0* = *id* |
p-adic-prod-int-convergent-subseq-seq p X (*Suc n*) =
p-adic-prod-int-convergent-subseq-seq p X n ∘
subset-subseq
(
p-adic-prod-int-convergent-subseq-seq-step p X (
p-adic-prod-int-convergent-subseq-seq p X n
) *n*
)
{*SOME z*.
infinite (
p-adic-prod-int-convergent-subseq-seq-step p X (
p-adic-prod-int-convergent-subseq-seq p X n
) *n*
−‘ {*z*}
)
}

abbreviation
p-adic-prod-int-convergent-subseq p X n ≡
p-adic-prod-int-convergent-subseq-seq p X n n

lemma *restricted-X-infinite-quot-vimage*:
∃ *z*. *infinite* ((λk. *p-adic-prod-int-quot* (*X k*)) −‘ {*z*})
if *fin-p-quot* : *finite* (*range* (λx::′*a adelic-int-quot*. *x prestrict* ((=) *p*)))
and *range-X* : *range X* ⊆ $\mathcal{O}_{\forall\, p}$
and *restricted-X*: ∀ *n*. *X n* = (*X n*) *prestrict* ((=) *p*)
for *p* :: ′*a*::*nontrivial-factorial-idom prime* **and** *X* :: *nat* ⇒ ′*a p-adic-prod*
proof−
define *resp* **where** *resp* ≡ λx::′*a adelic-int-quot*. *x prestrict* ((=) *p*)
define *X-quot* **where** *X-quot* ≡ λk. *p-adic-prod-int-quot* (*X k*)
have *range X-quot* ⊆ *range resp*
proof *safe*
fix *k*
from *range-X restricted-X X-quot-def resp-def* **have**
X-quot k = *resp* (*adelic-int-quot* (*Abs-adelic-int* (*X k*)))
using *p-restrict-adelic-int-abs-eq p-restrict-adelic-int-quot-abs-eq rangeI subsetD* **by** *metis*

thus *X-quot k* $\in$ *range resp* **by** *simp*
qed
with *fin-p-quot resp-def* **obtain** *b* **where** $b \in$ *range resp* **and** *infinite* (*X-quot* $-$‘ $\{b\}$)
using *infinite-vimageE* **by** *blast*
with *X-quot-def* **show** *?thesis* **by** *fast*
qed

context
fixes *p* :: $'a$::*nontrivial-factorial-unique-euclidean-bezout prime*
and *X* :: *nat* $\Rightarrow$ $'a$ *p-adic-prod*
and *ss* :: *nat* $\Rightarrow$ (*nat* $\Rightarrow$ *nat*)
and *ss-step* :: *nat* $\Rightarrow$ (*nat* $\Rightarrow$ $'a$ *adelic-int-quot*)
and *nth-im* :: *nat* $\Rightarrow$ $'a$ *adelic-int-quot*
and *subss* :: *nat* $\Rightarrow$ (*nat* $\Rightarrow$ *nat*)
defines
ss $\equiv$ *p-adic-prod-int-convergent-subseq-seq p X*
and
ss-step $\equiv$ λn. *p-adic-prod-int-convergent-subseq-seq-step p X* (*ss n*) *n*
and *nth-im* $\equiv$ λn. (*SOME z. infinite* (*ss-step n* $-$‘ $\{z\}$))
and *subss* $\equiv$ λn. *subset-subseq* (*ss-step n*) $\{$*nth-im n*$\}$
assumes *fin-p-quot* : *finite* (*range* (λx::$'a$ *adelic-int-quot. x prestrict* ((=) *p*)))
and *range-X* : *range X* $\subseteq \mathcal{O}_{\forall p}$
and *restricted-X*: $\forall n$. *X n* = (*X n*) *prestrict* ((=) *p*)
begin

lemma *p-adic-prod-int-convergent-subseq-seq-step-infinite-quot-vimage*:
$\exists z$. *infinite* (*ss-step n* $-$‘ $\{z\}$) (**is** *?A*)
and *p-adic-prod-int-convergent-subseq-seq-step-depth*:
$\forall k$. (*X* (*ss n k*)) $\neg\simeq_p$ (*X* (*ss n 0*)) $\longrightarrow$
((*X* (*ss n k*) $-$ *X* (*ss n 0*))$^{\circ p}$) $\geq$ *int n*
(**is** *?B*)
proof$-$
have *?A* $\wedge$ *?B*
proof (*induct n*)
case *0*
from *range-X* **have** *range* (λk. *X k* $-$ *X 0*) $\subseteq \mathcal{O}_{\forall p}$
using *global-p-depth-p-adic-prod.global-depth-set-minus* **by** *fastforce*
moreover from *restricted-X* **have**
$\forall k$. *X k* $-$ *X 0* = (*X k* $-$ *X 0*) *prestrict* (=) *p*
using *global-p-depth-p-adic-prod.p-restrict-minus* **by** *metis*
ultimately
have *A*: $\exists z$. *infinite* ((λk. *p-adic-prod-int-quot* (*X k* $-$ *X 0*)) $-$‘ $\{z\}$)
using *fin-p-quot restricted-X-infinite-quot-vimage*
by *blast*
from *ss-def range-X* **have** *B*:
$\forall k$. (*X* (*ss 0 k*)) $\neg\simeq_p$ (*X* (*ss 0 0*)) $\longrightarrow$
int 0 $\leq$ ((*X* (*ss 0 k*) $-$ *X* (*ss 0 0*))$^{\circ p}$)
using *global-p-depth-p-adic-prod.global-depth-set-minus*

p-equality-p-adic-prod.conv-0 global-p-depth-p-adic-prod.global-depth-setD
by *fastforce*
with *ss-step-def ss-def* **show**
$(\exists z.\ infinite\ (ss\text{-}step\ 0\ -`\ \{z\})) \wedge$
$(\forall k.\ (X\ (ss\ 0\ k))\ \neg\simeq_p\ (X\ (ss\ 0\ 0)) \longrightarrow$
$int\ 0 \leq ((X\ (ss\ 0\ k) - X\ (ss\ 0\ 0))^{\circ p}))$
using *A B* **by** *simp*
next
case *(Suc n)*
hence *inf* : $\exists z.\ infinite\ (ss\text{-}step\ n\ -`\ \{z\})$
and *depth*:
$\bigwedge k.\ (X\ (ss\ n\ k))\ \neg\simeq_p\ (X\ (ss\ n\ 0)) \Longrightarrow$
$int\ n \leq ((X\ (ss\ n\ k) - X\ (ss\ n\ 0))^{\circ p})$
by *auto*
from *nth-im-def* **have** *inf-nth-vimage*: $infinite\ (ss\text{-}step\ n\ -`\ \{nth\text{-}im\ n\})$
using *someI-ex[OF inf]* **by** *blast*
from *subss-def* **have** *ss-step-kth*: $\forall k.\ ss\text{-}step\ n\ (subss\ n\ k) = nth\text{-}im\ n$
using *subset-subseq-range[OF inf-nth-vimage]* **by** *blast*
have
$\bigwedge k.$
$p\text{-}adic\text{-}prod\text{-}shift\text{-}p\text{-}depth\ p\ (-\ int\ n)\ ((X\ (ss\ n\ k) - X\ (ss\ n\ 0))\ prestrict$
$((=)\ p))$
$\in \mathcal{O}_{\forall\ p}$
using *p-adic-prod-depth-embeds.shift-p-depth-p-restrict-global-depth-set-memI*
depth p-equality-p-adic-prod.conv-0
by *fastforce*
with *restricted-X* **have** *depth-set*:
$\bigwedge k.$
$p\text{-}adic\text{-}prod\text{-}shift\text{-}p\text{-}depth\ p\ (-\ int\ n)\ (X\ (ss\ n\ k) - X\ (ss\ n\ 0))$
$\in \mathcal{O}_{\forall\ p}$
using *global-p-depth-p-adic-prod.p-restrict-minus* **by** *metis*

have *B*:
$\forall k.\ (X\ (ss\ (Suc\ n)\ k))\ \neg\simeq_p\ (X\ (ss\ (Suc\ n)\ 0)) \longrightarrow$
$((X\ (ss\ (Suc\ n)\ k) - X\ (ss\ (Suc\ n)\ 0))^{\circ p}) \geq int\ (Suc\ n)$
proof *clarify*
fix *k* **assume** *k*: $(X\ (ss\ (Suc\ n)\ k))\ \neg\simeq_p\ (X\ (ss\ (Suc\ n)\ 0))$
from *ss-def ss-step-def subss-def nth-im-def*
have *X-Suc-n-k*: $X\ (ss\ (Suc\ n)\ k) = X\ (ss\ n\ (subss\ n\ k))$
and *X-Suc-n-0*: $X\ (ss\ (Suc\ n)\ 0) = X\ (ss\ n\ (subss\ n\ 0))$
by *auto*
from *ss-step-def* **have**
$p\text{-}adic\text{-}prod\text{-}shift\text{-}p\text{-}depth\ p\ (-\ int\ n)\ (X\ (ss\ (Suc\ n)\ 0) - X\ (ss\ (Suc\ n)\ k))$
$\in \mathcal{P}_{\forall\ p}$
using *depth-set X-Suc-n-k X-Suc-n-0 ss-step-kth p-adic-prod-int-quot-eq-iff*
p-adic-prod-depth-embeds.shift-p-depth-minus
by *fastforce*
with *k* **show** $((X\ (ss\ (Suc\ n)\ k) - X\ (ss\ (Suc\ n)\ 0))^{\circ p}) \geq int\ (Suc\ n)$
using *p-adic-prod-depth-embeds.shift-p-depth-mem-global-depth-set*

p-equality-p-adic-prod.sym[*of p*] *global-p-depth-p-adic-prod.depth-diff*[*of p*]
p-equality-p-adic-prod.conv-0[*of p X* (*ss* (*Suc n*) *0*)]
by *fastforce*
qed
from *ss-step-def ss-def* **have** *ss-step-Suc-n*:
ss-step (*Suc n*) = (λk.
p-adic-prod-int-quot (
p-adic-prod-shift-p-depth p (− *int* (*Suc n*)) ((*X* (*ss* (*Suc n*) *k*) − *X* (*ss* (*Suc n*) *0*)))
)
)
by *presburger*
moreover have
∃ *z*. *infinite* (
(λk.
p-adic-prod-int-quot (
p-adic-prod-shift-p-depth p (−*int* (*Suc n*)) ((*X* (*ss* (*Suc n*) *k*) − *X* (*ss* (*Suc n*) *0*)))
)
) −‘ {*z*}
)
proof (*intro restricted-X-infinite-quot-vimage*, *rule fin-p-quot*, *safe*)
fix *k*
have
p-adic-prod-shift-p-depth p (− *int* (*Suc n*)) (
(*X* (*ss* (*Suc n*) *k*) − *X* (*ss* (*Suc n*) *0*)) *prestrict* ((=) *p*)
) ∈ $\mathcal{O}_{\forall\, p}$
using *B p-adic-prod-depth-embeds.shift-p-depth-p-restrict-global-depth-set-memI*
p-equality-p-adic-prod.conv-0
by *fastforce*
with *restricted-X* **show**
p-adic-prod-shift-p-depth p (− *int* (*Suc n*)) (*X* (*ss* (*Suc n*) *k*) − *X* (*ss* (*Suc n*) *0*))
∈ $\mathcal{O}_{\forall\, p}$
using *global-p-depth-p-adic-prod.p-restrict-minus* **by** *metis*
from *restricted-X* **show**
p-adic-prod-shift-p-depth p (− *int* (*Suc n*)) (*X* (*ss* (*Suc n*) *k*) − *X* (*ss* (*Suc n*) *0*)) =
p-adic-prod-shift-p-depth p (− *int* (*Suc n*)) (
X (*ss* (*Suc n*) *k*) − *X* (*ss* (*Suc n*) *0*)
) *prestrict* (=) *p*
using *p-adic-prod-depth-embeds.shift-p-depth-p-restrict*
global-p-depth-p-adic-prod.p-restrict-minus
by *metis*
qed
ultimately have *A*: ∃ *z*. *infinite* (*ss-step* (*Suc n*) −‘ {*z*}) **by** *presburger*
show
(∃ *z*. *infinite* (*ss-step* (*Suc n*) −‘ {*z*})) ∧

$(\forall\, k.\ (X\ (ss\ (Suc\ n)\ k))\ \neg\simeq_p\ (X\ (ss\ (Suc\ n)\ 0)) \longrightarrow$
$int\ (Suc\ n) \leq ((X\ (ss\ (Suc\ n)\ k) - X\ (ss\ (Suc\ n)\ 0))^{\circ p}))$
using *A B* **by** *blast*
qed
thus *?A* **and** *?B* **by** *auto*
qed

lemma *p-adic-prod-int-convergent-subset-subseq-strict-mono*:
strict-mono (*subset-subseq* (*ss-step n*) {*nth-im n*})
using *ss-step-def nth-im-def p-adic-prod-int-convergent-subseq-seq-step-infinite-quot-vimage subset-subseq-strict-mono someI-ex*
by *fast*

lemma *p-adic-prod-int-convergent-subseq-strict-mono′*: *strict-mono* (*ss n*)
proof (*induct n*)
case (*Suc n*) **from** *ss-def ss-step-def nth-im-def* **show** *strict-mono* (*ss* (*Suc n*))
using *ss-def ss-step-def nth-im-def strict-monoD*[*OF Suc*]
strict-monoD[*OF p-adic-prod-int-convergent-subset-subseq-strict-mono*]
by (*force intro*: *strict-mono-onI*)
qed (*simp add*: *ss-def strict-mono-id flip*: *id-def*)

lemma *p-adic-prod-int-convergent-subseq-strict-mono*:
strict-mono (*p-adic-prod-int-convergent-subseq p X*)
proof (*rule iffD2*, *rule strict-mono-Suc-iff*, *clarify*)
fix *n* :: *nat*
have *Suc n* $\leq$ *subset-subseq* (*ss-step n*) {*nth-im n*} (*Suc n*)
using *strict-mono-imp-increasing p-adic-prod-int-convergent-subset-subseq-strict-mono*
by *fast*
hence *n* $<$ *subset-subseq* (*ss-step n*) {*nth-im n*} (*Suc n*) **by** *linarith*
with *ss-def ss-step-def nth-im-def*
show *p-adic-prod-int-convergent-subseq p X n* $<$ *p-adic-prod-int-convergent-subseq p X* (*Suc n*)
using *ss-def strict-monoD p-adic-prod-int-convergent-subseq-strict-mono′*
by *auto*
qed

lemma *p-adic-prod-int-convergent-subseq-Cauchy*:
p-adic-prod-p-cauchy p (*X* $\circ$ *p-adic-prod-int-convergent-subseq p X*)
proof (*rule iffD2*, *rule global-p-depth-p-adic-prod.p-consec-cauchy*, *clarify*)
fix *n*
define *C* **where** *C* $\equiv$ *X* $\circ$ *p-adic-prod-int-convergent-subseq p X*
have
global-p-depth-p-adic-prod.p-consec-cauchy-condition p C n (*nat* $(n + 1)$)
unfolding *global-p-depth-p-adic-prod.p-consec-cauchy-condition-def*
proof *clarify*
fix *k* :: *nat*
assume *k*: $k \geq nat\ (n + 1)\ (C\ (Suc\ k))\ \neg\simeq_p\ (C\ k)$
define *D0 D1 D2*
where $D0 \equiv C\ (Suc\ k) - C\ k$

```
      and   D1 ≡ X (ss k (subss k (Suc k))) − X (ss k 0)
      and   D2 ≡ X (ss k k) − X (ss k 0)
    with C-def ss-def ss-step-def subss-def nth-im-def have D0-eq: D0 = D1 − D2
      by (force simp add: algebra-simps)
    from k(2) D0-def have D0-nequiv0: D0 ¬≃p 0
      using p-equality-p-adic-prod.conv-0 by fast
    consider (D1) D1 ≃p 0 | (D2) D2 ≃p 0 |
            (neither) D1 ¬≃p 0 D2 ¬≃p 0
      by blast
    thus (D0°p) > n
    proof cases
      case D1
      moreover from this have D2 ¬≃p 0
        using D0-eq D0-nequiv0 p-equality-p-adic-prod.minus by fastforce
      ultimately show ?thesis
        using k(1) D2-def D0-eq global-p-depth-p-adic-prod.depth-diff-equiv0-left
          p-equality-p-adic-prod.conv-0 p-adic-prod-int-convergent-subseq-seq-step-depth
        by   fastforce
    next
      case D2
      moreover from this have D1 ¬≃p 0
        using D0-eq D0-nequiv0 p-equality-p-adic-prod.minus by fastforce
      ultimately show ?thesis
        using k(1) D1-def D0-eq global-p-depth-p-adic-prod.depth-diff-equiv0-right
          p-equality-p-adic-prod.conv-0 p-adic-prod-int-convergent-subseq-seq-step-depth
        by   fastforce
    next
      case neither
      moreover from this D1-def D2-def
        have  k ≤ min (D1°p) (D2°p)
      using p-equality-p-adic-prod.conv-0 p-adic-prod-int-convergent-subseq-seq-step-depth
        by   fastforce
      ultimately show ?thesis
        using k(1) D0-eq D0-nequiv0 global-p-depth-p-adic-prod.depth-nonarch-diff
by fastforce
    qed
  qed
  thus ∃K. global-p-depth-p-adic-prod.p-consec-cauchy-condition p C n K by blast
qed

lemma p-adic-prod-int-convergent-subseq-limseq:
  x ∈ (λz. z prestrict ((=) p)) ‘ O∀p
  if p-adic-prod-p-open-nbhds-limseq (X ∘ g) x
proof (intro image-eqI global-p-depth-p-adic-prod.global-imp-eq, standard)
  fix q :: 'a prime
  show x ≃q x prestrict ((=) p)
  proof (cases q = p)
    case False
    moreover from that have p-adic-prod-p-limseq q (X ∘ g) x
```

using *global-p-depth-p-adic-prod.globally-limseq-imp-locally-limseq* **by** *fast*
moreover from *restricted-X* **have** $\forall\, n.\ (X \circ g)\ n = (X \circ g)\ n\ prestrict\ (=)\ p$
by *force*
ultimately show $x \simeq_q (x\ prestrict\ ((=)\ p))$
using *global-p-depth-p-adic-prod.p-restrict-equiv0*
global-p-depth-p-adic-prod.p-limseq-p-constant
global-p-depth-p-adic-prod.p-limseq-unique p-equality-p-adic-prod.trans-left[*of q x 0*]
by (*metis* (*full-types*))
qed (*simp add*: *global-p-depth-p-adic-prod.p-restrict-equiv*[*symmetric*])
from *that range-X* **show** $x \in \mathcal{O}_{\forall\, p}$
using *eventually-sequentially*
global-p-depth-p-adic-prod.global-depth-set-p-open-nbhds-LIMSEQ-closed
by *force*
qed

end

7.2.2 Proving sequential compactness

context
fixes *p* :: *'a*::*nontrivial-factorial-unique-euclidean-bezout prime*
assumes *fin-p-quot*: *finite* (*range* (λx::*'a adelic-int-quot. x prestrict* ((=) *p*)))
begin

lemma *p-adic-prod-local-int-ring-seq-compact'*:
$\exists\, g$::*nat*$\Rightarrow$*nat. strict-mono g* $\wedge$ *p-adic-prod-p-cauchy p* ($X \circ g$)
if *range-X*:
range $X \subseteq (\lambda x.\ x\ prestrict\ ((=)\ p))$ ‘ $(\mathcal{O}_{\forall\, p})$
for X :: *nat* $\Rightarrow$ *'a p-adic-prod*
proof−
define *g* **where** $g \equiv$ *p-adic-prod-int-convergent-subseq p X*
from *range-X* **have** *range* $X \subseteq \mathcal{O}_{\forall\, p}$
using *global-p-depth-p-adic-prod.global-depth-set-closed-under-p-restrict-image* **by** *force*
moreover from *range-X* **have** $\forall\, n.\ X\ n = X\ n\ prestrict\ ((=)\ p)$
using *subsetD*
global-p-depth-p-adic-prod.p-restrict-image-restrict[*of*
X - (=) *p* $\mathcal{O}_{\forall\, p}$
]
by *force*
ultimately have *strict-mono g* **and** *p-adic-prod-p-cauchy p* ($X \circ g$)
using *g-def fin-p-quot p-adic-prod-int-convergent-subseq-Cauchy*[*of p X*]
p-adic-prod-int-convergent-subseq-strict-mono
by (*blast*, *presburger*)
thus *?thesis* **by** *blast*
qed

lemma *p-adic-prod-local-int-ring-seq-compact*:

$\exists\, x{\in}(\lambda x.\ x\ prestrict\ ((=)\ p))\ `\ (\mathcal{O}_{\forall\, p}).$
  $\exists\, g{::}nat{\Rightarrow}nat.$
    strict-mono $g \wedge$ *p-adic-prod-p-open-nbhds-limseq* $(X \circ g)\ x$
if *range-X*:
  range $X \subseteq (\lambda x.\ x\ prestrict\ ((=)\ p))\ `\ (\mathcal{O}_{\forall\, p})$
for $X :: nat \Rightarrow\ 'a$ *p-adic-prod*
proof−
from *range-X* **have** *1*: *range* $X \subseteq \mathcal{O}_{\forall\, p}$
  using *global-p-depth-p-adic-prod.global-depth-set-closed-under-p-restrict-image*
by *force*
moreover from *range-X* **have** *2*: $\forall\, n.\ X\ n = X\ n\ prestrict\ ((=)\ p)$
  using *subsetD*
    global-p-depth-p-adic-prod.p-restrict-image-restrict[*of*
      X - $(=)\ p\ \mathcal{O}_{\forall\, p}$
    ]
  by *force*
from *range-X* **obtain** g **where** g: *strict-mono* g *p-adic-prod-p-cauchy* $p\ (X \circ g)$
  using *p-adic-prod-local-int-ring-seq-compact′* **by** *blast*
from $g(2)$ **obtain** x
  where *p-adic-prod-p-open-nbhds-limseq* $(\lambda n.\ (X \circ g)\ n\ prestrict\ (=)\ p)\ x$
  using *p-complete-p-adic-prod.p-cauchyE*
  by *blast*
moreover have $(\lambda n.\ (X \circ g)\ n\ prestrict\ (=)\ p) = X \circ g$ **using** *2* **by** *auto*
ultimately have *p-adic-prod-p-open-nbhds-limseq* $(X \circ g)\ x$ **by** *auto*
moreover from *this fin-p-quot*
  have $x \in (\lambda x.\ x\ prestrict\ ((=)\ p))\ `\ (\mathcal{O}_{\forall\, p})$
  using *1 2 p-adic-prod-int-convergent-subseq-limseq*
  by *blast*
ultimately show *?thesis* **using** $g(1)$ **by** *blast*
qed

lemma *adelic-int-local-int-ring-seq-abs*:
range $X \subseteq (\lambda x.\ x\ prestrict\ ((=)\ p))\ `\ (\mathcal{O}_{\forall\, p})$
if *range-X*: *range* $X \subseteq \mathcal{O}_{\forall\, p}$
and *range-abs-X*:
  range $(\lambda k.$ *Abs-adelic-int* $(X\ k)) \subseteq$ *range* $(\lambda x.\ x\ prestrict\ ((=)\ p))$
for $X :: nat \Rightarrow\ 'a$ *p-adic-prod*
proof (*standard*, *clarify*, *rule image-eqI*)
fix n
from *range-X* **show** $*$: $X\ n \in \mathcal{O}_{\forall\, p}$ **by** *fast*
from *range-abs-X* **obtain** y **where** *Abs-adelic-int* $(X\ n) = y\ prestrict\ ((=)\ p)$
by *blast*
from *this* **obtain** x
  where $x \in \mathcal{O}_{\forall\, p}$
  and *Abs-adelic-int* $(X\ n) =$ *Abs-adelic-int* $(x\ prestrict\ ((=)\ p))$
  using *Abs-adelic-int-cases p-restrict-adelic-int-abs-eq*
  by *metis*
thus $X\ n = X\ n\ prestrict\ ((=)\ p)$
  using $*$ *global-p-depth-p-adic-prod.global-depth-set-p-restrict Abs-adelic-int-inject*

global-p-depth-p-adic-prod.p-restrict-restrict'
by *force*
qed

lemma *adelic-int-local-int-ring-seq-compact'*:
$\exists$ *g*::*nat*$\Rightarrow$*nat*. *strict-mono g* $\wedge$ *adelic-int-p-cauchy p* $(X \circ g)$
if *range-X*:
range X $\subseteq$ $(\lambda x.$ *x prestrict* $((=)\ p))$ ‘ $(\mathcal{O}_{\forall\, p})$
for *X* :: *nat* $\Rightarrow$ *'a adelic-int*
proof (*cases X rule* : *adelic-int-seq-cases*)
case (*Abs-adelic-int F*)
with *range-X*
have *range F* $\subseteq$ $(\lambda x.$ *x prestrict* $((=)\ p))$ ‘ $(\mathcal{O}_{\forall\, p})$
using *adelic-int-local-int-ring-seq-abs*
by *fast*
with *fin-p-quot* **obtain** *g* **where** *g*: *strict-mono g p-adic-prod-p-cauchy p* $(F \circ g)$
using *p-adic-prod-local-int-ring-seq-compact'* **by** *blast*
moreover from *Abs-adelic-int*(*1*)
have $(\lambda n.$ *Abs-adelic-int* $((F \circ g)\ n)) = X \circ g$
by *fastforce*
moreover from *Abs-adelic-int*(*2*)
have *range* $(F \circ g) \subseteq \mathcal{O}_{\forall\, p}$
by *auto*
ultimately have *adelic-int-p-cauchy p* $(X \circ g)$ **using** *adelic-int-p-cauchy-abs-eq*
by *metis*
with *g*(*1*) **show** *?thesis* **by** *blast*
qed

lemma *adelic-int-local-int-ring-seq-compact*:
$\exists$ *x*$\in$*range* $(\lambda x.$ *x prestrict* $((=)\ p))$.
$\exists$ *g*::*nat*$\Rightarrow$*nat*.
strict-mono g $\wedge$ *adelic-int-p-open-nbhds-limseq* $(X \circ g)$ *x*
if *range-X*:
range X $\subseteq$ *range* $(\lambda x.$ *x prestrict* $((=)\ p))$
for *X* :: *nat* $\Rightarrow$ *'a adelic-int*
proof (*cases X rule* : *adelic-int-seq-cases*)
case (*Abs-adelic-int F*)
with *range-X*
have *range F* $\subseteq$ $(\lambda x.$ *x prestrict* $((=)\ p))$ ‘ $(\mathcal{O}_{\forall\, p})$
using *adelic-int-local-int-ring-seq-abs*
by *fast*
with *fin-p-quot* **obtain** *y g*
where *y* : $y \in (\lambda x.$ *x prestrict* $(=)\ p)$ ‘ $\mathcal{O}_{\forall\, p}$
and *g* : *strict-mono g*
and *g-y*: *p-adic-prod-p-open-nbhds-limseq* $(F \circ g)$ *y*
using *p-adic-prod-local-int-ring-seq-compact* **by** *blast*
from *y* **have** $y \in \mathcal{O}_{\forall\, p}$
using *global-p-depth-p-adic-prod.global-depth-set-closed-under-p-restrict-image*
by *auto*

moreover from *Abs-adelic-int(2)*
  have *range* $(F \circ g) \subseteq \mathcal{O}_{\forall\, p}$
  by *force*
ultimately have
  adelic-int-p-open-nbhds-limseq
    $(\lambda k.\ \mathit{Abs\text{-}adelic\text{-}int}\ ((F \circ g)\ k))\ (\mathit{Abs\text{-}adelic\text{-}int}\ y)$
  using *g-y adelic-int-p-open-nbhds-limseq-abs-eq*
  by *blast*
moreover from *Abs-adelic-int(1)*
  have $(\lambda k.\ \mathit{Abs\text{-}adelic\text{-}int}\ ((F \circ g)\ k)) = X \circ g$
  by *fastforce*
ultimately have *lim*: *adelic-int-p-open-nbhds-limseq* $(X \circ g)$ $(\mathit{Abs\text{-}adelic\text{-}int}\ y)$

by *force*
from *y* **obtain** *z* **where** *z*: $z \in \mathcal{O}_{\forall\, p}$ $y = z$ *prestrict* $(=)$ *p* **by** *fast*
hence *Abs-adelic-int y* = *Abs-adelic-int z prestrict* $(=)$ *p*
  using *p-restrict-adelic-int-abs-eq* **by** *fastforce*
with *g* **show** *?thesis* **using** *lim* **by** *blast*

qed

end

lemma *int-adic-prod-local-int-ring-seq-compact*:
  $\exists\, x \in (\lambda x.\ x\ \mathit{prestrict}\ ((=)\ p))\ \text{`}\ (\mathcal{O}_{\forall\, p})$.
    $\exists\, g{::}nat \Rightarrow nat$.
      strict-mono $g \wedge$ *p-adic-prod-p-open-nbhds-limseq* $(X \circ g)\ x$
  if *range* $X \subseteq (\lambda x.\ x\ \mathit{prestrict}\ ((=)\ p))\ \text{`}\ (\mathcal{O}_{\forall\, p})$
  for p :: *int prime*
  and X :: $nat \Rightarrow int$ *p-adic-prod*
  using *that p-adic-prod-local-int-ring-seq-compact finite-range-prestrict-single-int-prime*
  by *blast*

lemma *int-adelic-int-local-int-ring-seq-compact*:
  $\exists\, x \in \mathit{range}\ (\lambda x.\ x\ \mathit{prestrict}\ ((=)\ p))$.
    $\exists\, g{::}nat \Rightarrow nat$.
      strict-mono $g \wedge$ *adelic-int-p-open-nbhds-limseq* $(X \circ g)\ x$
  if *range* $X \subseteq \mathit{range}\ (\lambda x.\ x\ \mathit{prestrict}\ ((=)\ p))$
  for p :: *int prime*
  and X :: $nat \Rightarrow int$ *adelic-int*
  using *that adelic-int-local-int-ring-seq-compact finite-range-prestrict-single-int-prime*
  by *blast*

7.3 Finite-open-cover compactness

function *exclusion-sequence* ::
  $'a\ set \Rightarrow ('a \Rightarrow 'a\ set) \Rightarrow$
    $'a \Rightarrow nat \Rightarrow 'a$
  where *exclusion-sequence* $A\ f\ a\ 0 = a$ |
  exclusion-sequence $A\ f\ a\ (Suc\ n) =$
    $(SOME\ x.\ x \in A \wedge x \notin (\bigcup k \le n.\ f\ (\mathit{exclusion\text{-}sequence}\ A\ f\ a\ k)))$

```
  by pat-completeness auto
termination by (relation measure(λ(A,f,a,n). n)) auto

lemma
  assumes ¬ A ⊆ (⋃k≤n. f (exclusion-sequence A f a k))
  shows exclusion-sequence-mem: exclusion-sequence A f a (Suc n) ∈ A
    and exclusion-sequence-excludes:
      exclusion-sequence A f a (Suc n) ∉ (⋃k≤n. f (exclusion-sequence A f a k))
proof−
  from assms have ∗:
    ∃x. x ∈ A ∧ x ∉ (⋃k≤n. f (exclusion-sequence A f a k))
    by   blast
  show  exclusion-sequence A f a (Suc n) ∈ A
    and
    exclusion-sequence A f a (Suc n) ∉ (⋃k≤n. f (exclusion-sequence A f a k))
    using someI-ex[OF ∗] by auto
qed

context
  fixes p      :: 'a::nontrivial-factorial-unique-euclidean-bezout prime
  assumes fin-p-quot: finite (range (λx::'a adelic-int-quot. x prestrict ((=) p)))
begin

lemma p-adic-prod-local-int-ring-finite-cover:
  ∃C. finite C ∧
    C ⊆ (λx. x prestrict ((=) p)) ‘ 𝒪∀ p ∧
    (λx. x prestrict ((=) p)) ‘ 𝒪∀ p ⊆
      (⋃c∈C. p-adic-prod-p-open-nbhd p n c)
  for C :: 'a set
proof−
  define p-ring :: 'a p-adic-prod set
    where p-ring ≡ (λx. x prestrict ((=) p)) ‘ 𝒪∀ p
  have
    ¬ (∀C⊆p-ring.
      p-ring ⊆ (⋃c∈C. p-adic-prod-p-open-nbhd p n c) ⟶
      infinite C
    )
  proof (safe)
    assume n:
      ∀C⊆p-ring.
        p-ring ⊆ ⋃ (p-adic-prod-p-open-nbhd p n ‘ C) ⟶ infinite C
    define X where
      X ≡
      λk. exclusion-sequence p-ring (p-adic-prod-p-open-nbhd p n) (0::'a p-adic-prod)
k
    have partial-dn-cover:
      ∀k. X ‘ {..k} ⊆ p-ring ⟶
        ¬ p-ring ⊆ (⋃j≤k. p-adic-prod-p-open-nbhd p n (X j))
    proof clarify
```

fix *k*
assume X ‘ $\{..k\} \subseteq$ *p-ring*
and *p-ring* $\subseteq$ ($\bigcup j \leq k$. *p-adic-prod-p-open-nbhd p n* (X j))
with *n* **have** *infinite* (X ‘ $\{..k\}$) **by** *auto*
thus *False* **by** *blast*
qed
moreover have *partial-range-X*: $\forall k$. X ‘ $\{..k\} \subseteq$ *p-ring*
proof
fix *k* **show** X ‘ $\{..k\} \subseteq$ *p-ring*
proof (*induct k*)
case *0* **from** *X-def p-ring-def* **show** *?case*
using *global-p-depth-p-adic-prod.global-depth-set-0*
global-p-depth-p-adic-prod.p-restrict-zero
by *fastforce*
next
case (*Suc k*)
hence $\neg$ *p-ring* $\subseteq$ ($\bigcup j \leq k$. *p-adic-prod-p-open-nbhd p n* (X j))
using *partial-dn-cover* **by** *blast*
with *X-def* **have** X (*Suc k*) $\in$ *p-ring* **using** *exclusion-sequence-mem* **by** *force*
moreover have $\{..$*Suc k*$\}$ = *insert* (*Suc k*) $\{..k\}$ **by** *auto*
ultimately show *?case* **using** *Suc* **by** *auto*
qed
qed
ultimately
have *dn-cover*:
$\forall k$. $\neg$ *p-ring* $\subseteq$ ($\bigcup j \leq k$. *p-adic-prod-p-open-nbhd p n* (X j))
and *range-X*: *range X* $\subseteq$ *p-ring*
by (*fastforce*, *fast*)
from *p-ring-def fin-p-quot* **obtain** *g*
where *g*: *strict-mono g p-adic-prod-p-cauchy p* ($X \circ g$)
using *range-X p-adic-prod-local-int-ring-seq-compact′*
by *metis*
from *this* **obtain** *K* **where** *K*: *p-adic-prod-p-cauchy-condition p* ($X \circ g$) *n K* **by** *fast*
have X (g (*Suc K*)) $\notin$ *p-adic-prod-p-open-nbhd p n* (X (g K))
proof
assume X (g (*Suc K*)) $\in$ *p-adic-prod-p-open-nbhd p n* (X (g K))
moreover from $g(1)$ **have** *g-K*: g K < g (*Suc K*) **using** *strict-monoD* **by** *blast*
ultimately have
X (g (*Suc K*)) $\in$ ($\bigcup j \leq g$ (*Suc K*) − *1*. *p-adic-prod-p-open-nbhd p n* (X j))
by *force*
with *X-def* **show** *False*
using *g-K dn-cover*
exclusion-sequence-excludes[*of p-ring p-adic-prod-p-open-nbhd p n 0 g* (*Suc K*) − *1*]
by *simp*
qed

hence $(X\ (g\ (Suc\ K)))\ \neg{\simeq}_p\ (X\ (g\ K))$
and $((X\ (g\ (Suc\ K)) - X\ (g\ K))^{\circ p}) < n$
using *global-p-depth-p-adic-prod.p-open-nbhd-eq-circle*
by (*blast*, *fastforce*)
with K **show** *False*
using *global-p-depth-p-adic-prod.p-cauchy-conditionD*[*of p X* $\circ$ *g n K Suc K*
K] **by** *auto*
qed
thus *?thesis* **using** *p-ring-def* **by** *fast*
qed

lemma *p-adic-prod-local-int-ring-lebesgue-number*:
$\exists\, d.\ \forall\, x{\in}(\lambda x.\ x\ prestrict\ ((=)\ p))\ `\ \mathcal{O}_{\forall\, p}.$
$\exists\, A{\in}\mathcal{A}.$ *p-adic-prod-p-open-nbhd p d x* $\subseteq A$
if *cover*:
$(\lambda x.\ x\ prestrict\ ((=)\ p))\ `\ \mathcal{O}_{\forall\, p} \subseteq \bigcup\ \mathcal{A}$
and *by-opens*:
$\forall\, A{\in}\mathcal{A}.$ *generate-topology* (*p-adic-prod-local-p-open-nbhds p*) A
proof−
define *p-ring* :: *'a p-adic-prod set*
where *p-ring* $\equiv (\lambda x.\ x\ prestrict\ ((=)\ p))\ `\ \mathcal{O}_{\forall\, p}$
have
$\neg\ (\forall\, d.\ \exists\, x{\in}p\text{-}ring.\ \forall\, A{\in}\mathcal{A}.$
$\neg$ *p-adic-prod-p-open-nbhd p d x* $\subseteq A)$
proof
assume $*$:
$\forall\, d.\ \exists\, x{\in}p\text{-}ring.\ \forall\, A{\in}\mathcal{A}.$
$\neg$ *p-adic-prod-p-open-nbhd p d x* $\subseteq A$
define X **where**
$X \equiv \lambda n.$ *SOME* $x.$
$x \in$ *p-ring* $\wedge$
$(\forall\, A{\in}\mathcal{A}.\ \neg$ *p-adic-prod-p-open-nbhd p* (*int n*) $x \subseteq A)$
have *range-X*: *range* $X \subseteq$ *p-ring*
proof *safe*
fix n
from $*$ **have** *ex-x*:
$\exists\, x.\ x \in$ *p-ring* $\wedge$
$(\forall\, A{\in}\mathcal{A}.\ \neg$ *p-adic-prod-p-open-nbhd p* (*int n*) $x \subseteq A)$
by *force*
from *X-def* **show** $X\ n \in$ *p-ring* **using** *someI-ex*[*OF ex-x*] **by** *fast*
qed
with *fin-p-quot* **obtain** *a g*
where *g*: *strict-mono g*
and *a*: *p-adic-prod-p-open-nbhds-limseq* $(X \circ g)$ *a*
using *p-adic-prod-local-int-ring-seq-compact*
unfolding *X-def p-ring-def*
by *blast*
from *range-X p-ring-def* **have** *range* $X \subseteq \mathcal{O}_{\forall\, p}$
using *global-p-depth-p-adic-prod.global-depth-set-closed-under-p-restrict-image*

by *force*
  moreover from *range-X p-ring-def* **have** $\forall n.\ X\ n = X\ n$ *prestrict* $((=)\ p)$
    using *subsetD*
      global-p-depth-p-adic-prod.p-restrict-image-restrict[*of*
        X - $(=)\ p\ \mathcal{O}_{\forall\ p}$
      ]
    by *force*
  ultimately have $a \in$ *p-ring*
   using *p-ring-def fin-p-quot a p-adic-prod-int-convergent-subseq-limseq* **by** *blast*
  with *cover p-ring-def* **obtain** A **where** A: $A \in \mathcal{A}\ a \in A$ **by** *blast*
  with *by-opens* **obtain** n **where** n: *p-adic-prod-p-open-nbhd* $p\ n\ a \subseteq A$
    using *global-p-depth-p-adic-prod.p-open-nbhds-open-subopen* **by** *blast*
  from *a* **have** *p-adic-prod-p-limseq* $p\ (X \circ g)\ a$
    using *global-p-depth-p-adic-prod.globally-limseq-imp-locally-limseq* **by** *fast*
  from *this* **obtain** K **where** K: *p-adic-prod-p-limseq-condition* $p\ (X \circ g)\ a\ n\ K$
by *presburger*
  define k **where** $k \equiv max\ (nat\ n)\ K$
  from $*$ **have** *ex-x*:
    $\exists x.\ x \in$ *p-ring* $\wedge$
      $(\forall A{\in}\mathcal{A}.\ \neg$ *p-adic-prod-p-open-nbhd* $p\ (int\ (g\ k))\ x \subseteq A)$
    by *force*
  from *X-def A*(*1*)
    have *X-g-m*: $\neg$ *p-adic-prod-p-open-nbhd* $p\ (int\ (g\ k))\ (X\ (g\ k)) \subseteq A$
    using *someI-ex*[*OF ex-x*]
    by *fast*
  moreover from *g k-def* **have** *n-g-k*: $n \le int\ (g\ k)$
    using *strict-mono-imp-increasing*[*of g k*] **by** *linarith*
  ultimately have $X\ (g\ k)\ \neg\simeq_p\ a$
    using *n X-g-m global-p-depth-p-adic-prod.p-open-nbhd-eq-circle*
      global-p-depth-p-adic-prod.p-open-nbhd-circle-multicentre
      global-p-depth-p-adic-prod.p-open-nbhd-antimono
    by *blast*
  with *K k-def* **have** $X\ (g\ k) \in$ *p-adic-prod-p-open-nbhd* $p\ n\ a$
    using *global-p-depth-p-adic-prod.p-limseq-conditionD*[*of p - a n K k*]
      global-p-depth-p-adic-prod.p-open-nbhd-eq-circle
    by *force*
  with *n* **show** *False*
    using *n-g-k X-g-m global-p-depth-p-adic-prod.p-open-nbhd-circle-multicentre*
      global-p-depth-p-adic-prod.p-open-nbhd-antimono
    by *fast*
 qed
 thus *?thesis* **using** *p-ring-def* **by** *force*
qed

lemma *p-adic-prod-local-int-ring-compact*:
 $\exists \mathcal{B}{\subseteq}\mathcal{A}.$ *finite* $\mathcal{B}\ \wedge$
  $(\lambda x.\ x$ *prestrict* $((=)\ p))$ ‘ $\mathcal{O}_{\forall\ p} \subseteq \bigcup\ \mathcal{B}$
 if *cover*:
  $(\lambda x.\ x$ *prestrict* $((=)\ p))$ ‘ $\mathcal{O}_{\forall\ p} \subseteq \bigcup\ \mathcal{A}$

and *by-opens*:
$\forall A{\in}\mathcal{A}$. *generate-topology* (*p-adic-prod-local-p-open-nbhds p*) *A*
proof−
define *p-ring* :: *'a p-adic-prod set*
where *p-ring* $\equiv$ (λx. *x prestrict* ((=) *p*)) ' $\mathcal{O}_{\forall\, p}$
with *cover by-opens* **obtain** *d* **where** *d*:
$\forall x{\in}$*p-ring*. $\exists A{\in}\mathcal{A}$. *p-adic-prod-p-open-nbhd p d x* $\subseteq$ *A*
using *p-adic-prod-local-int-ring-lebesgue-number* **by** *presburger*
from *p-ring-def* **obtain** *C* **where** *C*:
finite C C $\subseteq$ *p-ring*
p-ring $\subseteq$ ($\bigcup c{\in}C$. *p-adic-prod-p-open-nbhd p d c*)
using *p-adic-prod-local-int-ring-finite-cover* **by** *metis*
define *f* **where**
f $\equiv$ λc. *SOME A*. *A* $\in \mathcal{A}$ $\wedge$ *p-adic-prod-p-open-nbhd p d c* $\subseteq$ *A*
define $\mathcal{B}$ **where** $\mathcal{B} \equiv f$ ' *C*
have $\mathcal{B} \subseteq \mathcal{A}$
unfolding $\mathcal{B}$*-def*
proof *safe*
fix *c* **assume** *c* $\in$ *C*
with *d C*(*2*)
have *ex-A*: $\exists A$. *A* $\in \mathcal{A}$ $\wedge$ *p-adic-prod-p-open-nbhd p d c* $\subseteq$ *A*
by *blast*
from *f-def* **show** *f c* $\in \mathcal{A}$ **using** *someI-ex*[*OF ex-A*] **by** *blast*
qed
moreover from $\mathcal{B}$*-def C*(*1*) **have** *finite* $\mathcal{B}$ **by** *simp*
moreover have *p-ring* $\subseteq \bigcup\, \mathcal{B}$
proof
fix *x* **assume** *x* $\in$ *p-ring*
with *C*(*3*) **obtain** *c* **where** *c*: *c* $\in$ *C x* $\in$ *p-adic-prod-p-open-nbhd p d c* **by** *auto*
from *d C*(*2*) *c*(*1*)
have *ex-A*: $\exists A$. *A* $\in \mathcal{A}$ $\wedge$ *p-adic-prod-p-open-nbhd p d c* $\subseteq$ *A*
by *blast*
from *f-def* **have** *p-adic-prod-p-open-nbhd p d c* $\subseteq$ *f c* **using** *someI-ex*[*OF ex-A*] **by** *auto*
with $\mathcal{B}$*-def c* **show** *x* $\in \bigcup\, \mathcal{B}$ **by** *auto*
qed
ultimately show *?thesis* **using** *p-ring-def* **by** *blast*
qed

lemma *p-adic-prod-local-scaled-int-ring-compact*:
$\exists \mathcal{B}{\subseteq}\mathcal{A}$. *finite* $\mathcal{B}$ $\wedge$
(λx. *x prestrict* ((=) *p*)) ' ($\mathcal{P}_{\forall\, p}{}^{n}$)
$\subseteq \bigcup\, \mathcal{B}$
if *cover*:
(λx. *x prestrict* ((=) *p*)) ' ($\mathcal{P}_{\forall\, p}{}^{n}$)
$\subseteq \bigcup\, \mathcal{A}$
and *by-opens*:
$\forall A{\in}\mathcal{A}$. *generate-topology* (*p-adic-prod-local-p-open-nbhds p*) *A*

for $\mathcal{A}$ :: *'a p-adic-prod set set*
proof−
define *p-ring p-depth-ring* :: *'a p-adic-prod set*
where *p-ring* ≡ (λ*x*. *x prestrict* ((=) *p*)) ‘ $\mathcal{O}_{\forall\, p}$
and
p-depth-ring ≡
(λ*x*. *x prestrict* ((=) *p*)) ‘ ($\mathcal{P}_{\forall\, p}{}^{n}$)
from *p-ring-def p-depth-ring-def*
have *drop*: *p-adic-prod-shift-p-depth p n* ‘ *p-ring* = *p-depth-ring*
and *lift*: *p-adic-prod-shift-p-depth p* (−*n*) ‘ *p-depth-ring* = *p-ring*
by (*simp-all add*: *p-adic-prod-depth-embeds.shift-p-depth-p-restrict-global-depth-set-image*)
define $\mathcal{A}'$ **where** $\mathcal{A}'$ ≡ (‘) (*p-adic-prod-shift-p-depth p* (−*n*)) ‘ $\mathcal{A}$
from *cover p-depth-ring-def* $\mathcal{A}'$*-def* **have** *p-ring* ⊆ ⋃ $\mathcal{A}'$ **using** *lift* **by** *blast*
moreover from *by-opens* **have** *by-opens'*:
∀ *A'*∈$\mathcal{A}'$. *generate-topology* (*p-adic-prod-local-p-open-nbhds p*) *A'*
using *p-adic-prod-depth-embeds.shift-p-depth-p-open-set* **unfolding** $\mathcal{A}'$*-def* **by** *fast*
ultimately obtain $\mathcal{B}'$
where $\mathcal{B}'$: $\mathcal{B}'$ ⊆ $\mathcal{A}'$ *finite* $\mathcal{B}'$ *p-ring* ⊆ ⋃ $\mathcal{B}'$
using *fin-p-quot p-ring-def p-adic-prod-local-int-ring-compact*
by *force*
define $\mathcal{B}$ **where** $\mathcal{B}$ ≡ (‘) (*p-adic-prod-shift-p-depth p n*) ‘ $\mathcal{B}'$
have $\mathcal{B}$ ⊆ $\mathcal{A}$
unfolding $\mathcal{B}$*-def*
proof *clarify*
fix *B'* **assume** *B'* ∈ $\mathcal{B}'$
with $\mathcal{B}'$(*1*) $\mathcal{A}'$*-def* **obtain** *A*
where *A* ∈ $\mathcal{A}$ **and** *B'* = *p-adic-prod-shift-p-depth p* (−*n*) ‘ *A*
by *blast*
thus *p-adic-prod-shift-p-depth p n* ‘ *B'* ∈ $\mathcal{A}$
using *p-adic-prod-depth-embeds.shift-shift-p-depth-image*[*of p n* −*n*] **by** *simp*
qed
moreover from $\mathcal{B}$*-def* $\mathcal{B}'$(*2*) **have** *finite* $\mathcal{B}$ **by** *blast*
moreover from $\mathcal{B}$*-def* $\mathcal{B}'$(*3*) **have** *p-depth-ring* ⊆ ⋃ $\mathcal{B}$ **using** *drop* **by** *blast*
ultimately show *?thesis* **using** *p-depth-ring-def* **by** *blast*
qed

lemma *adelic-int-local-int-ring-compact*:
∃$\mathcal{B}$⊆$\mathcal{A}$. *finite* $\mathcal{B}$ ∧
range (λ*x*. *x prestrict* ((=) *p*)) ⊆ ⋃ $\mathcal{B}$
if *fin-p-quot*: *finite* (*range* (λ*x*::*'a adelic-int-quot*. *x prestrict* ((=) *p*)))
and *cover*: *range* (λ*x*. *x prestrict* ((=) *p*)) ⊆ ⋃ $\mathcal{A}$
and *by-opens*:
∀ *A*∈$\mathcal{A}$. *generate-topology* (*adelic-int-local-p-open-nbhds p*) *A*
for $\mathcal{A}$ :: *'a adelic-int set set*
proof−
define *f f'* :: *'a p-adic-prod* ⇒ *'a p-adic-prod*
where *f* ≡ λ*x*. *x prestrict* ((=) *p*)
and *f'* ≡ λ*x*. *x prestrict* ((≠) *p*)

define *f-f'* **where** $f\text{-}f' \equiv \lambda A.\ f\ \text{‘}\ \mathit{Rep\text{-}adelic\text{-}int}\ \text{‘}\ A + \mathit{range}\ f'$
define *p-ring* :: *'a p-adic-prod set* **where** $p\text{-}ring \equiv f\ \text{‘}\ \mathcal{O}_{\forall\ p}$
define $\mathcal{A}'$ **where** $\mathcal{A}' \equiv f\text{-}f'\ \text{‘}\ \mathcal{A}$
from *cover f-def f'-def f-f'-def p-ring-def* $\mathcal{A}'$*-def* **have** $p\text{-}ring \subseteq \bigcup\ \mathcal{A}'$
  using *adelic-int-local-depth-ring-lift-cover*[*of 0*] **by** *fast*
moreover from *by-opens f-def f'-def f-f'-def* $\mathcal{A}'$*-def* **have**
  $\forall\ A' \in \mathcal{A}'.$ *generate-topology* (*p-adic-prod-local-p-open-nbhds p*) A'
  using *adelic-int-lift-local-p-open* **by** *fast*
ultimately obtain $\mathcal{B}'$
  where $\mathcal{B}'$: $\mathcal{B}' \subseteq \mathcal{A}'$ *finite* $\mathcal{B}'$ $p\text{-}ring \subseteq \bigcup\ \mathcal{B}'$
  using *fin-p-quot f-def p-ring-def p-adic-prod-local-int-ring-compact*
  by *force*
define *g* **where** $g \equiv \lambda B'.\ \mathit{SOME}\ B.\ B \in \mathcal{A} \wedge B' = f\text{-}f'\ B$
define $\mathcal{B}$ **where** $\mathcal{B} \equiv g\ \text{‘}\ \mathcal{B}'$
from $\mathcal{B}$*-def* $\mathcal{B}'(2)$ **have** *finite* $\mathcal{B}$ **by** *fast*
moreover have *subcover*: $\mathcal{B} \subseteq \mathcal{A}$
  unfolding $\mathcal{B}$*-def*
proof *clarify*
  fix B' **assume** $B' \in \mathcal{B}'$
  with $\mathcal{A}'$*-def* $\mathcal{B}'(1)$ **have** *ex-B*: $\exists\ B.\ B \in \mathcal{A} \wedge B' = f\text{-}f'\ B$ **by** *blast*
  from *g-def* **show** $g\ B' \in \mathcal{A}$ **using** *someI-ex*[*OF ex-B*] **by** *fastforce*
qed
moreover have *range* ($\lambda x.\ x$ *prestrict* $((=)\ p)) \subseteq \bigcup\ \mathcal{B}$
proof *clarify*
  fix *x* **show** *x prestrict* $(=)\ p \in \bigcup\ \mathcal{B}$
  proof (*cases x*)
    case (*Abs-adelic-int a*)
    from *p-ring-def Abs-adelic-int*(*2*) $\mathcal{B}'(3)$ **have** $f\ a \in \bigcup\ \mathcal{B}'$ **by** *auto*
    from *this* **obtain** B' **where** B': $B' \in \mathcal{B}'$ $f\ a \in B'$ **by** *fast*
    from $\mathcal{A}'$*-def* $\mathcal{B}'(1)$ $B'(1)$ **have** *ex-B*: $\exists\ B.\ B \in \mathcal{A} \wedge B' = f\text{-}f'\ B$ **by** *auto*
    from $B'(1)$ $\mathcal{B}$*-def* **have** *g-B'*: $g\ B' \in \mathcal{B}$ **by** *auto*
    from *g-def* $B'(2)$ **have** $f\ a \in f\text{-}f'\ (g\ B')$ **using** *someI-ex*[*OF ex-B*] **by** *simp*
    with *f-f'-def* **obtain** *b c* **where** *b*: $b \in g\ B'$ **and** *bc*: $f\ a = f$ (*Rep-adelic-int*
b) + $f'\ c$
      using *set-plus-elim* **by** *blast*
    have $f'\ c = 0$
    proof (*intro global-p-depth-p-adic-prod.global-imp-eq*, *standard*)
      fix *q* :: *'a prime* **show** $f'\ c \simeq_q 0$
      proof (*cases* $p = q$)
        case *True* **with** *f'-def* **show** *?thesis*
          using *global-p-depth-p-adic-prod.p-restrict-equiv0* **by** *fast*
      next
        case *False*
        moreover from *f-def bc* **have** $f'\ c = f\ (a -$ *Rep-adelic-int b*)
          by (*simp add*: *global-p-depth-p-adic-prod.p-restrict-minus*)
        ultimately show *?thesis*
          using *f-def global-p-depth-p-adic-prod.p-restrict-equiv0* **by** *auto*
      qed
    qed

with *f-def Abs-adelic-int bc* **have**
  x prestrict (=) *p* = *b prestrict* ((=) *p*)
  using *p-restrict-adelic-int-abs-eq p-restrict-adelic-int-abs-eq*
    Rep-adelic-int[*of b*] *Rep-adelic-int-inverse*[*of b*]
  by *fastforce*
moreover from *by-opens b*(*1*) **have** *b prestrict* ((=) *p*) ∈ *g B′*
  using *subcover g-B′ global-p-depth-adelic-int.p-restrict-p-open-set-mem-iff*
by *blast*
ultimately show *?thesis* **using** *g-B′* **by** *auto*
qed
qed
ultimately show *?thesis* **by** *blast*
qed

lemma *adelic-int-local-scaled-int-ring-compact*:
  $\exists \mathcal{B} \subseteq \mathcal{A}$. *finite* $\mathcal{B}$ ∧
    (λx. *x prestrict* ((=) *p*)) ‘ ($\mathcal{P}_{\forall\, p}{}^{n}$)
      $\subseteq \bigcup \mathcal{B}$
  if *fin-p-quot*: *finite* (*range* (λx::′*a adelic-int-quot*. *x prestrict* ((=) *p*)))
  and *cover*:
    (λx. *x prestrict* ((=) *p*)) ‘ ($\mathcal{P}_{\forall\, p}{}^{n}$)
      $\subseteq \bigcup \mathcal{A}$
  and *by-opens*:
    $\forall A \in \mathcal{A}$. *generate-topology* (*adelic-int-local-p-open-nbhds p*) *A*
  for $\mathcal{A}$ :: ′*a adelic-int set set*
proof−
  show *?thesis*
  proof (*cases n* ≤ *0*)
    case *True*
    hence
      (λx::′*a adelic-int*. *x prestrict* ((=) *p*)) ‘ ($\mathcal{P}_{\forall\, p}{}^{n}$) =
        range (λx. *x prestrict* ((=) *p*))
      using *nonpos-global-depth-set-adelic-int* **by** *auto*
    with *fin-p-quot cover by-opens* **show** *?thesis*
      using *adelic-int-local-int-ring-compact* **by** *fastforce*
  next
    case *False*
    define *f f′* :: ′*a p-adic-prod* ⇒ ′*a p-adic-prod*
      where *f* ≡ λx. *x prestrict* ((=) *p*)
      and *f′* ≡ λx. *x prestrict* ((≠) *p*)
    define *f-f′* **where** *f-f′* ≡ λA. *f* ‘ *Rep-adelic-int* ‘ *A* + *range f′*
    define *p-ring p-depth-ring* :: ′*a p-adic-prod set*
      where *p-ring* ≡ *f* ‘ $\mathcal{O}_{\forall\, p}$
      and *p-depth-ring* ≡ *f* ‘ ($\mathcal{P}_{\forall\, p}{}^{n}$)
    define $\mathcal{A}'$ **where** $\mathcal{A}' \equiv$ *f-f′* ‘ $\mathcal{A}$
    from *False cover f-def f′-def f-f′-def p-depth-ring-def* $\mathcal{A}'$*-def*
      have *p-depth-ring* $\subseteq \bigcup \mathcal{A}'$
      using *adelic-int-local-depth-ring-lift-cover*[*of n p* $\mathcal{A}$]
      by *auto*

moreover from *by-opens f-def f′-def f-f′-def* $\mathcal{A}'$*-def* **have**
$\forall A' \in \mathcal{A}'$. *generate-topology* (*p-adic-prod-local-p-open-nbhds p*) A'
using *adelic-int-lift-local-p-open* **by** *fast*
ultimately obtain $\mathcal{B}'$ **where** $\mathcal{B}'$:
$\mathcal{B}' \subseteq \mathcal{A}'$ *finite* $\mathcal{B}'$ *p-depth-ring* $\subseteq \bigcup \mathcal{B}'$
using *fin-p-quot f-def p-depth-ring-def p-adic-prod-local-scaled-int-ring-compact*[*of n*]
by *force*
define g **where** $g \equiv \lambda B'.$ *SOME B*. $B \in \mathcal{A} \wedge B' =$ *f-f′* B
define $\mathcal{B}$ **where** $\mathcal{B} \equiv g$ ‘ $\mathcal{B}'$
from $\mathcal{B}$*-def* $\mathcal{B}'(2)$ **have** *finite* $\mathcal{B}$ **by** *fast*
moreover have *subcover*: $\mathcal{B} \subseteq \mathcal{A}$
unfolding $\mathcal{B}$*-def*
proof *clarify*
fix B' **assume** $B' \in \mathcal{B}'$
with $\mathcal{A}'$*-def* $\mathcal{B}'(1)$ **have** *ex-B*: $\exists B.\ B \in \mathcal{A} \wedge B' =$ *f-f′* B **by** *blast*
from *g-def* **show** $g\ B' \in \mathcal{A}$ **using** *someI-ex*[*OF ex-B*] **by** *fastforce*
qed
moreover have
($\lambda x.$ *x prestrict* ((=) *p*)) ‘ ($\mathcal{P}_{\forall\, p}{}^{n}$) $\subseteq$
$\bigcup \mathcal{B}$
proof *clarify*
fix *x* :: *′a adelic-int* **assume** *x*: $x \in \mathcal{P}_{\forall\, p}{}^{n}$
show *x prestrict* (=) $p \in \bigcup \mathcal{B}$
proof (*cases x*)
case (*Abs-adelic-int a*)
hence *a* = *Rep-adelic-int x* **using** *Abs-adelic-int-inverse* **by** *fastforce*
with *False x* **have** $a \in \mathcal{P}_{\forall\, p}{}^{n}$
using *lift-nonneg-global-depth-set-adelic-int*[*of n*] **by** *auto*
with *p-depth-ring-def* $\mathcal{B}'(3)$ **have** $f\ a \in \bigcup \mathcal{B}'$
using *nonneg-global-depth-set-adelic-int-eq-projection*[*of n*] **by** *fast*
from *this* **obtain** B' **where** B': $B' \in \mathcal{B}'$ $f\ a \in B'$ **by** *fast*
from $\mathcal{A}'$*-def* $\mathcal{B}'(1)$ $B'(1)$ **have** *ex-B*: $\exists B.\ B \in \mathcal{A} \wedge B' =$ *f-f′* B
by *auto*
from $B'(1)$ $\mathcal{B}$*-def* **have** *g-B′*: $g\ B' \in \mathcal{B}$ **by** *auto*
from *g-def* $B'(2)$ **have** $f\ a \in$ *f-f′* ($g\ B'$) **using** *someI-ex*[*OF ex-B*] **by** *simp*
with *f-f′-def* **obtain** *b c* **where** *b*: $b \in g\ B'$ **and** *bc*: $f\ a = f$ (*Rep-adelic-int b*) + $f'\ c$
using *set-plus-elim* **by** *blast*
have $f'\ c = 0$
proof (*intro global-p-depth-p-adic-prod.global-imp-eq, standard*)
fix *q* :: *′a prime* **show** $f'\ c \simeq_q 0$
proof (*cases p = q*)
case *True* **with** *f′-def* **show** *?thesis*
using *global-p-depth-p-adic-prod.p-restrict-equiv0* **by** *fast*
next
case *False*
moreover from *f-def bc* **have** $f'\ c = f$ ($a -$ *Rep-adelic-int b*)
by (*simp add*: *global-p-depth-p-adic-prod.p-restrict-minus*)

ultimately show *?thesis*
using *f-def global-p-depth-p-adic-prod.p-restrict-equiv0* **by** *auto*
qed
qed
with *f-def Abs-adelic-int bc* **have**
x prestrict (=) *p* = *b prestrict* ((=) *p*)
using *p-restrict-adelic-int-abs-eq p-restrict-adelic-int-abs-eq*
Rep-adelic-int[*of b*] *Rep-adelic-int-inverse*[*of b*]
by *fastforce*
moreover from *by-opens b*(*1*) **have** *b prestrict* ((=) *p*) ∈ *g B′*
using *subcover g-B′ global-p-depth-adelic-int.p-restrict-p-open-set-mem-iff*
by *blast*
ultimately show *?thesis* **using** *g-B′* **by** *auto*
qed
qed
ultimately show *?thesis* **by** *blast*
qed
qed

end

lemma *int-adic-prod-local-int-ring-compact*:
$\exists\,\mathcal{B}\subseteq\mathcal{A}$. *finite* $\mathcal{B}$ ∧
(λx. *x prestrict* ((=) *p*)) ‘ $\mathcal{O}_{\forall\, p}$ ⊆ $\bigcup$ $\mathcal{B}$
if (λx. *x prestrict* ((=) *p*)) ‘ $\mathcal{O}_{\forall\, p}$ ⊆ $\bigcup$ $\mathcal{A}$
and ∀ *A*∈$\mathcal{A}$. *generate-topology* (*p-adic-prod-local-p-open-nbhds p*) *A*
for *p* :: *int prime*
and $\mathcal{A}$ :: *int p-adic-prod set set*
using *that p-adic-prod-local-int-ring-compact*[*of p* $\mathcal{A}$] *finite-range-prestrict-single-int-prime*
by *presburger*

lemma *int-adic-prod-local-scaled-int-ring-compact*:
$\exists\,\mathcal{B}\subseteq\mathcal{A}$. *finite* $\mathcal{B}$ ∧
(λx. *x prestrict* ((=) *p*)) ‘ ($\mathcal{P}_{\forall\, p}{}^{n}$)
⊆ $\bigcup$ $\mathcal{B}$
if
(λx. *x prestrict* ((=) *p*)) ‘ ($\mathcal{P}_{\forall\, p}{}^{n}$)
⊆ $\bigcup$ $\mathcal{A}$
and ∀ *A*∈$\mathcal{A}$. *generate-topology* (*p-adic-prod-local-p-open-nbhds p*) *A*
for *p* :: *int prime*
and $\mathcal{A}$ :: *int p-adic-prod set set*
using *that p-adic-prod-local-scaled-int-ring-compact*[*of p n* $\mathcal{A}$]
finite-range-prestrict-single-int-prime
by *presburger*

lemma *int-adelic-int-local-int-ring-compact*:
$\exists\,\mathcal{B}\subseteq\mathcal{A}$. *finite* $\mathcal{B}$ ∧
range (λx. *x prestrict* ((=) *p*)) ⊆ $\bigcup$ $\mathcal{B}$
if *range* (λx. *x prestrict* ((=) *p*)) ⊆ $\bigcup$ $\mathcal{A}$

and $\forall A \in \mathcal{A}$. *generate-topology* (*adelic-int-local-p-open-nbhds* p) A
for p :: *int prime*
and $\mathcal{A}$ :: *int adelic-int set set*
using *that adelic-int-local-int-ring-compact*[*of* p $\mathcal{A}$] *finite-range-prestrict-single-int-prime*
by *presburger*

lemma *int-adelic-int-local-scaled-int-ring-compact*:
$\exists \mathcal{B} \subseteq \mathcal{A}$. *finite* $\mathcal{B}$ $\wedge$
(λx. x *prestrict* ((=) p)) ‘ ($\mathcal{P}_{\forall p}{}^{n}$)
$\subseteq \bigcup \mathcal{B}$
if
(λx. x *prestrict* ((=) p)) ‘ ($\mathcal{P}_{\forall p}{}^{n}$)
$\subseteq \bigcup \mathcal{A}$
and $\forall A \in \mathcal{A}$. *generate-topology* (*adelic-int-local-p-open-nbhds* p) A
for p :: *int prime*
and $\mathcal{A}$ :: *int adelic-int set set*
using *that adelic-int-local-scaled-int-ring-compact*[*of* p n $\mathcal{A}$]
finite-range-prestrict-single-int-prime
by *presburger*

end

References

[1] J. W. S. Cassels. *Local Fields*, volume 3 of *London Mathematical Society Student Texts*. Cambridge University Press, 1986.

[2] D. S. Dummit and R. M. Foote. *Abstract Algebra*. Wiley, 3rd edition, 2004.

[3] J. R. Munkres. *Topology*. Prentice Hall, 2nd edition, 1975.