p-adic Hensel's Lemma

Aaron Crighton

March 17, 2025

Contents

Abstract

We formalize the ring of p-adic integers within the framework of the HOL-Algebra library. The carrier of the ring $\mathbb{Z}_p$ is formalized as the inverse limit of the residue rings $\mathbb{Z}/p^n\mathbb{Z}$ for a fixed prime p. We define a locale for reasoning about $\mathbb{Z}_p$ for a fixed prime p, and define an integer-valued valuation, as well as an extended-integer valued valuation on $\mathbb{Z}_p$ (where $0 \in \mathbb{Z}_p$ is the unique ring element mapped to ∞). Basic topological facts about the p-adic integers are formalized, including the completeness and sequential compactness of $\mathbb{Z}_p$. Taylor expansions of polynomials over a commutative ring are defined, culminating in the formalization of Hensel's Lemma based on a proof due to Keith Conrad [1].

theory *Function-Ring*
imports *HOL−Algebra.Ring HOL−Library.FuncSet HOL−Algebra.Module*
begin

This theory formalizes basic facts about the ring of extensional functions from a fixed set to a fixed ring. This will be useful for providing a generic framework for various constructions related to the p-adics such as polynomial evaluation and sequences. The rings of semialgebraic functions will be defined as subrings of these function rings, which will be necessary for the proof of p-adic quantifier elimination.

1 The Ring of Extensional Functions from a Fixed Base Set to a Fixed Base Ring

1.1 Basic Operations on Extensional Functions

definition *function-mult*:: *'c set* ⇒ (*'a*, *'b*) *ring-scheme* ⇒ (*'c* ⇒ *'a*) ⇒ (*'c* ⇒ *'a*) ⇒ (*'c* ⇒ *'a*) **where**
function-mult S R f g = (λ*x* ∈ *S*. (*f x*) ⊗$_R$ (*g x*))

abbreviation(*input*) *ring-function-mult*:: (*'a*, *'b*) *ring-scheme* ⇒ (*'a* ⇒ *'a*) ⇒ (*'a* ⇒ *'a*) ⇒ (*'a* ⇒ *'a*) **where**
ring-function-mult R f g ≡ *function-mult* (*carrier R*) *R f g*

definition *function-add*:: *'c set* ⇒ (*'a*, *'b*) *ring-scheme* ⇒ (*'c* ⇒ *'a*) ⇒ (*'c* ⇒ *'a*) ⇒ (*'c* ⇒ *'a*) **where**
function-add S R f g = (λ*x* ∈ *S*. (*f x*) ⊕$_R$ (*g x*))

abbreviation(*input*) *ring-function-add*:: (*'a*, *'b*) *ring-scheme* ⇒ (*'a* ⇒ *'a*) ⇒ (*'a* ⇒ *'a*) ⇒ (*'a* ⇒ *'a*) **where**
ring-function-add R f g ≡ *function-add* (*carrier R*) *R f g*

definition *function-one*:: *'c set* ⇒ (*'a*, *'b*) *ring-scheme* ⇒ (*'c* ⇒ *'a*) **where**
function-one S R = (λ*x* ∈ *S*. **1**$_R$)

abbreviation(*input*) *ring-function-one* :: (*'a*, *'b*) *ring-scheme* ⇒ (*'a* ⇒ *'a*) **where**
ring-function-one R ≡ *function-one* (*carrier R*) *R*

definition *function-zero*:: *'c set* ⇒ (*'a*, *'b*) *ring-scheme* ⇒ (*'c* ⇒ *'a*) **where**
function-zero S R = (λ*x* ∈ *S*. **0**$_R$)

abbreviation(*input*) *ring-function-zero* :: (*'a*, *'b*) *ring-scheme* ⇒ (*'a* ⇒ *'a*) **where**
ring-function-zero R ≡ *function-zero* (*carrier R*) *R*

definition *function-uminus*:: *'c set* ⇒ (*'a*, *'b*) *ring-scheme* ⇒ (*'c* ⇒ *'a*) ⇒ (*'c* ⇒ *'a*) **where**
function-uminus S R a = (λ *x* ∈ *S*. ⊖$_R$ (*a x*))

definition *ring-function-uminus*:: (*'a*, *'b*) *ring-scheme* ⇒ (*'a* ⇒ *'a*) ⇒ (*'a* ⇒ *'a*) **where**
ring-function-uminus R a = *function-uminus* (*carrier R*) *R a*

definition *function-scalar-mult*:: *'c set* ⇒ (*'a*, *'b*) *ring-scheme* ⇒ *'a* ⇒ (*'c* ⇒ *'a*) ⇒ (*'c* ⇒ *'a*) **where**
function-scalar-mult S R a f = (λ *x* ∈ *S*. *a* ⊗$_R$ (*f x*))

1.2 Defining the Ring of Extensional Functions

definition *function-ring*:: *'c set* ⇒ (*'a*, *'b*) *ring-scheme* ⇒ (*'a*, *'c* ⇒ *'a*) *module* **where**

function-ring S R = ⦇
  carrier = *extensional-funcset S* (*carrier R*),
  Group.monoid.mult = (*function-mult S R*),
  one = (*function-one S R*),
  zero = (*function-zero S R*),
  add = (*function-add S R*),
  smult = *function-scalar-mult S R* ⦈

The following locale consists of a struct R, and a distinguished set S which is meant to serve as the domain for a ring of functions $S \to carrier R$.

locale *struct-functions* =
  fixes *R* ::('*a*, '*b*) *partial-object-scheme* (**structure**)
    and *S* :: '*c set*

The following are locales which fix a ring R (which may be commutative, a domain, or a field) and a function ring F of extensional functions from a fixed set S to $carrier R$

locale *ring-functions* = *struct-functions* + *R?*: *ring R* +
  fixes *F* (**structure**)
  defines *F-def*: *F* ≡ *function-ring S R*

locale *cring-functions* = *ring-functions* + *R?*: *cring R*

locale *domain-functions* = *ring-functions* + *R?*: *domain R*

locale *field-functions* = *ring-functions* + *R?*: *field R*

sublocale *cring-functions* < *ring-functions*
  apply (*simp add*: *ring-functions-axioms*)
  by (*simp add*: *F-def*)

sublocale *domain-functions* < *ring-functions*
  apply (*simp add*: *ring-functions-axioms*)
  by (*simp add*: *F-def*)

sublocale *domain-functions* < *cring-functions*
  apply (*simp add*: *cring-functions-def is-cring ring-functions-axioms*)
  by (*simp add*: *F-def*)

sublocale *field-functions* < *domain-functions*
  apply (*simp add*: *domain-axioms domain-functions-def ring-functions-axioms*)
  by (*simp add*: *F-def*)

sublocale *field-functions* < *ring-functions*
  apply (*simp add*: *ring-functions-axioms*)
  by (*simp add*: *F-def*)

sublocale *field-functions* < *cring-functions*
  apply (*simp add*: *cring-functions-axioms*)

by (*simp add*: *F-def*)

abbreviation(*input*) *ring-function-ring*:: ('*a*, '*b*) *ring-scheme* ⇒ ('*a*, '*a* ⇒ '*a*) *module* (‹*Fun*›) **where**
ring-function-ring R ≡ *function-ring* (*carrier R*) *R*

1.3 Algebraic Properties of the Basic Operations

1.3.1 Basic Carrier Facts

lemma(**in** *ring-functions*) *function-ring-defs*:
carrier F = *extensional-funcset S* (*carrier R*)
$(\otimes_F)$ = (*function-mult S R*)
$(\oplus_F)$ = (*function-add S R*)
$\mathbf{1}_F$ = *function-one S R*
$\mathbf{0}_F$ = *function-zero S R*
$(\odot_F)$ = *function-scalar-mult S R*
unfolding *F-def*
by (*auto simp add*: *function-ring-def*)

lemma(**in** *ring-functions*) *function-ring-car-memE*:
assumes *a* ∈ *carrier F*
shows *a* ∈ *extensional S*
a ∈ *S* → *carrier R*
using *assms function-ring-defs* **apply** *auto*[*1*]
using *assms function-ring-defs PiE-iff* **apply** *blast*
using *assms function-ring-defs*(*1*) **by** *fastforce*

lemma(**in** *ring-functions*) *function-ring-car-closed*:
assumes *a* ∈ *S*
assumes *f* ∈ *carrier F*
shows *f a* ∈ *carrier R*
using *assms* **unfolding** *function-ring-def F-def* **by** *auto*

lemma(**in** *ring-functions*) *function-ring-not-car*:
assumes *a* ∉ *S*
assumes *f* ∈ *carrier F*
shows *f a* = *undefined*
using *assms* **unfolding** *function-ring-def F-def* **by** *auto*

lemma(**in** *ring-functions*) *function-ring-car-eqI*:
assumes *f* ∈ *carrier F*
assumes *g* ∈ *carrier F*
assumes ⋀*a*. *a* ∈ *S* ⟹ *f a* = *g a*
shows *f* = *g*
using *assms*(*1*) *assms*(*2*) *assms*(*3*) *extensionalityI function-ring-car-memE*(*1*)
by *blast*

lemma(**in** *ring-functions*) *function-ring-car-memI*:
assumes ⋀*a*. *a* ∈ *S* ⟹ *f a* ∈ *carrier R*

assumes $\bigwedge a.\ a \notin S \Longrightarrow f\ a = undefined$
shows $f \in carrier\ F$
using *function-ring-defs assms*
unfolding *extensional-funcset-def*
by (*simp add*: $\langle \bigwedge a.\ a \in S \Longrightarrow f\ a \in carrier\ R \rangle$ *extensional-def*)

lemma(**in** *ring*) *function-ring-car-memI*:
assumes $\bigwedge a.\ a \in S \Longrightarrow f\ a \in carrier\ R$
assumes $\bigwedge a.\ a \notin S \Longrightarrow f\ a = undefined$
shows $f \in carrier\ (function\text{-}ring\ S\ R)$
by (*simp add*: *assms*(*1*) *assms*(*2*) *local.ring-axioms ring-functions.function-ring-car-memI ring-functions.intro*)

1.3.2 Basic Multiplication Facts

lemma(**in** *ring-functions*) *function-mult-eval-car*:
assumes $a \in S$
assumes $f \in carrier\ F$
assumes $g \in carrier\ F$
shows $(f \otimes_F g)\ a = (f\ a) \otimes (g\ a)$
using *assms function-ring-defs*
unfolding *function-mult-def*
by *simp*

lemma(**in** *ring-functions*) *function-mult-eval-closed*:
assumes $a \in S$
assumes $f \in carrier\ F$
assumes $g \in carrier\ F$
shows $(f \otimes_F g)\ a \in carrier\ R$
using *assms function-mult-eval-car*
using *F-def ring-functions.function-ring-car-closed ring-functions-axioms* **by** *fast-force*

lemma(**in** *ring-functions*) *fun-mult-closed*:
assumes $f \in carrier\ F$
assumes $g \in carrier\ F$
shows $f \otimes_F g \in carrier\ F$
apply(*rule function-ring-car-memI*)
apply (*simp add*: *assms*(*1*) *assms*(*2*) *function-mult-eval-closed*)
by (*simp add*: *function-mult-def function-ring-defs*(*2*))

lemma(**in** *ring-functions*) *fun-mult-eval-assoc*:
assumes $x \in carrier\ F$
assumes $y \in carrier\ F$
assumes $z \in carrier\ F$
assumes $a \in S$
shows $(x \otimes_F y \otimes_F z)\ a = (x \otimes_F (y \otimes_F z))\ a$
proof−
have *0*: $(x \otimes_F y \otimes_F z)\ a = (x\ a) \otimes (y\ a) \otimes (z\ a)$

by (*simp add*: *assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *fun-mult-closed function-mult-eval-car*)
have *1*: $(x \otimes_F (y \otimes_F z))\ a = (x\ a) \otimes ((y\ a) \otimes (z\ a))$
by (*simp add*: *assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *fun-mult-closed function-mult-eval-car*)
have *2*:$(x \otimes_F (y \otimes_F z))\ a = (x\ a) \otimes (y\ a) \otimes (z\ a)$
using *1 assms*
by (*simp add*: *function-ring-car-closed m-assoc*)
show *?thesis*
using *0 2* **by** *auto*
qed

lemma(**in** *ring-functions*) *fun-mult-assoc*:
assumes $x \in$ *carrier F*
assumes $y \in$ *carrier F*
assumes $z \in$ *carrier F*
shows $(x \otimes_F y \otimes_F z) = (x \otimes_F (y \otimes_F z))$
using *fun-mult-eval-assoc*[*of x*]
by (*simp add*: *assms*(*1*) *assms*(*2*) *assms*(*3*) *fun-mult-closed function-ring-car-eqI*)

1.3.3 Basic Addition Facts

lemma(**in** *ring-functions*) *fun-add-eval-car*:
assumes $a \in S$
assumes $f \in$ *carrier F*
assumes $g \in$ *carrier F*
shows $(f \oplus_F g)\ a = (f\ a) \oplus (g\ a)$
by (*simp add*: *assms*(*1*) *function-add-def function-ring-defs*(*3*))

lemma(**in** *ring-functions*) *fun-add-eval-closed*:
assumes $a \in S$
assumes $f \in$ *carrier F*
assumes $g \in$ *carrier F*
shows $(f \oplus_F g)\ a \in$ *carrier R*
using *assms* **unfolding** *F-def*
using *F-def fun-add-eval-car function-ring-car-closed*
by *auto*

lemma(**in** *ring-functions*) *fun-add-closed*:
assumes $f \in$ *carrier F*
assumes $g \in$ *carrier F*
shows $f \oplus_F g \in$ *carrier F*
apply(*rule function-ring-car-memI*)
using *assms* **unfolding** *F-def*
using *F-def fun-add-eval-closed* **apply** *blast*
by (*simp add*: *function-add-def function-ring-def*)

lemma(**in** *ring-functions*) *fun-add-eval-assoc*:
assumes $x \in$ *carrier F*

assumes $y \in carrier\ F$
assumes $z \in carrier\ F$
assumes $a \in S$
shows $(x \oplus_F y \oplus_F z)\ a = (x \oplus_F (y \oplus_F z))\ a$
proof−
have *0*: $(x \oplus_F y \oplus_F z)\ a = (x\ a) \oplus (y\ a) \oplus (z\ a)$
by (*simp add*: *assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *fun-add-closed fun-add-eval-car*)
have *1*: $(x \oplus_F (y \oplus_F z))\ a = (x\ a) \oplus ((y\ a) \oplus (z\ a))$
by (*simp add*: *assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *fun-add-closed fun-add-eval-car*)
have *2*:$(x \oplus_F (y \oplus_F z))\ a = (x\ a) \oplus (y\ a) \oplus (z\ a)$
using *1 assms*
by (*simp add*: *add.m-assoc function-ring-car-closed*)
show *?thesis*
using *0 2* **by** *auto*
qed

lemma(**in** *ring-functions*) *fun-add-assoc*:
assumes $x \in carrier\ F$
assumes $y \in carrier\ F$
assumes $z \in carrier\ F$
shows $x \oplus_F y \oplus_F z = x \oplus_F (y \oplus_F z)$
apply(*rule function-ring-car-eqI*)
using *assms* **apply** (*simp add*: *fun-add-closed*)
apply (*simp add*: *assms*(*1*) *assms*(*2*) *assms*(*3*) *fun-add-closed*)
by (*simp add*: *assms*(*1*) *assms*(*2*) *assms*(*3*) *fun-add-eval-assoc*)

lemma(**in** *ring-functions*) *fun-add-eval-comm*:
assumes $a \in S$
assumes $x \in carrier\ F$
assumes $y \in carrier\ F$
shows $(x \oplus_F y)\ a = (y \oplus_F x)\ a$
by (*metis F-def assms*(*1*) *assms*(*2*) *assms*(*3*) *fun-add-eval-car ring.ring-simprules*(*10*)
ring-functions.function-ring-car-closed ring-functions-axioms ring-functions-def)

lemma(**in** *ring-functions*) *fun-add-comm*:
assumes $x \in carrier\ F$
assumes $y \in carrier\ F$
shows $x \oplus_F y = y \oplus_F x$
using *fun-add-eval-comm assms*
by (*metis* (*no-types*, *opaque-lifting*) *fun-add-closed function-ring-car-eqI*)

1.3.4 Basic Facts About the Multiplicative Unit

lemma(**in** *ring-functions*) *function-one-eval*:
assumes $a \in S$
shows $\mathbf{1}_F\ a = \mathbf{1}$
using *assms function-ring-defs* **unfolding** *function-one-def*
by *simp*

lemma(**in** *ring-functions*) *function-one-closed*:
$\mathbf{1}_F \in$*carrier F*
apply(*rule function-ring-car-memI*)
using *function-ring-defs*
using *function-one-eval* **apply** *auto*[*1*]
by (*simp add*: *function-one-def function-ring-defs*(*4*))

lemma(**in** *ring-functions*) *function-times-one-l*:
assumes $a \in$ *carrier F*
shows $\mathbf{1}_F \otimes_F a = a$
proof(*rule function-ring-car-eqI*)
show $\mathbf{1}_F \otimes_F a \in$ *carrier F*
using *assms fun-mult-closed function-one-closed*
by *blast*
show $a \in$ *carrier F*
using *assms* **by** *simp*
show $\bigwedge c.\ c \in S \Longrightarrow (\mathbf{1}_F \otimes_F a)\ c = a\ c$
by (*simp add*: *assms function-mult-eval-car function-one-eval function-one-closed function-ring-car-closed*)
qed

lemma(**in** *ring-functions*) *function-times-one-r*:
assumes $a \in$ *carrier F*
shows $a \otimes_F \mathbf{1}_F = a$
proof(*rule function-ring-car-eqI*)
show $a \otimes_F \mathbf{1}_F \in$ *carrier F*
using *assms fun-mult-closed function-one-closed*
by *blast*
show $a \in$ *carrier F*
using *assms* **by** *simp*
show $\bigwedge c.\ c \in S \Longrightarrow (a \otimes_F \mathbf{1}_F)\ c = a\ c$
using *assms*
by (*simp add*: *function-mult-eval-car function-one-eval function-one-closed function-ring-car-closed*)
qed

1.3.5 Basic Facts About the Additive Unit

lemma(**in** *ring-functions*) *function-zero-eval*:
assumes $a \in S$
shows $\mathbf{0}_F\ a = \mathbf{0}$
using *assms function-ring-defs*
unfolding *function-zero-def*
by *simp*

lemma(**in** *ring-functions*) *function-zero-closed*:
$\mathbf{0}_F \in$*carrier F*
apply(*rule function-ring-car-memI*)
apply (*simp add*: *function-zero-eval*)

by (*simp add*: *function-ring-defs*(5) *function-zero-def*)

lemma(**in** *ring-functions*) *fun-add-zeroL*:
assumes $a \in$ *carrier F*
shows $\mathbf{0}_F \oplus_F a = a$
proof(*rule function-ring-car-eqI*)
show $\mathbf{0}_F \oplus_F a \in$ *carrier F*
using *assms fun-add-closed function-zero-closed*
by *blast*
show $a \in$ *carrier F*
using *assms* **by** *simp*
show $\bigwedge c.\ c \in S \Longrightarrow (\mathbf{0}_F \oplus_F a)\ c = a\ c$
using *assms F-def fun-add-eval-car function-zero-closed*
ring-functions.function-zero-eval ring-functions-axioms
by (*simp add*: *ring-functions.function-zero-eval function-ring-car-closed*)
qed

lemma(**in** *ring-functions*) *fun-add-zeroR*:
assumes $a \in$ *carrier F*
shows $a \oplus_F \mathbf{0}_F = a$
using *assms fun-add-comm fun-add-zeroL*
by (*simp add*: *function-zero-closed*)

1.3.6 Distributive Laws

lemma(**in** *ring-functions*) *function-mult-r-distr*:
assumes $x \in$ *carrier F*
assumes $y \in$ *carrier F*
assumes $z \in$ *carrier F*
shows $(x \oplus_F y) \otimes_F z = x \otimes_F z \oplus_F y \otimes_F z$
proof(*rule function-ring-car-eqI*)
show $(x \oplus_F y) \otimes_F z \in$ *carrier F*
by (*simp add*: *assms*(1) *assms*(2) *assms*(3) *fun-add-closed fun-mult-closed*)
show $x \otimes_F z \oplus_F y \otimes_F z \in$ *carrier F*
by (*simp add*: *assms*(1) *assms*(2) *assms*(3) *fun-add-closed fun-mult-closed*)
show $\bigwedge a.\ a \in S \Longrightarrow ((x \oplus_F y) \otimes_F z)\ a = (x \otimes_F z \oplus_F y \otimes_F z)\ a$
proof−
fix *a*
assume *A*: $a \in S$
show $((x \oplus_F y) \otimes_F z)\ a = (x \otimes_F z \oplus_F y \otimes_F z)\ a$
using *A assms fun-add-eval-car*[*of a x y*] *fun-add-eval-car*[*of a x* $\otimes_F$*z y* $\otimes_F$ *z*]
function-mult-eval-car[*of a x* $\oplus_F$ *y z*] *semiring-simprules*(10)
F-def
by (*smt* (*verit*) *fun-add-closed function-mult-eval-car function-ring-car-closed*
ring-functions.fun-mult-closed ring-functions-axioms)
qed
qed

lemma(**in** *ring-functions*) *function-mult-l-distr*:
assumes $x \in$ *carrier* F
assumes $y \in$ *carrier* F
assumes $z \in$ *carrier* F
shows $z \otimes_F (x \oplus_F y) = z \otimes_F x \oplus_F z \otimes_F y$
proof(*rule function-ring-car-eqI*)
show $z \otimes_F (x \oplus_F y) \in$ *carrier* F
by (*simp add*: *assms*(*1*) *assms*(*2*) *assms*(*3*) *fun-add-closed fun-mult-closed*)
show $z \otimes_F x \oplus_F z \otimes_F y \in$ *carrier* F
by (*simp add*: *assms*(*1*) *assms*(*2*) *assms*(*3*) *fun-add-closed fun-mult-closed*)
show $\bigwedge a.\ a \in S \Longrightarrow (z \otimes_F (x \oplus_F y))\ a = (z \otimes_F x \oplus_F z \otimes_F y)\ a$
proof−
fix a
assume A: $a \in S$
show $(z \otimes_F (x \oplus_F y))\ a = (z \otimes_F x \oplus_F z \otimes_F y)\ a$
using A *assms function-ring-defs fun-add-closed fun-mult-closed*
function-mult-eval-car[*of a z x* $\oplus_F$ *y*]
function-mult-eval-car[*of a z x*]
function-mult-eval-car[*of a z y*]
fun-add-eval-car[*of a x y*]
semiring-simprules(*13*)
fun-add-eval-car function-ring-car-closed **by** *auto*
qed
qed

1.3.7 Additive Inverses

lemma(**in** *ring-functions*) *function-uminus-closed*:
assumes $f \in$ *carrier* F
shows *function-uminus* S R $f \in$ *carrier* F
proof(*rule function-ring-car-memI*)
show $\bigwedge a.\ a \in S \Longrightarrow$ *function-uminus* S R f $a \in$ *carrier* R
using *assms function-ring-car-closed*[*of - f*] **unfolding** *function-uminus-def*
by *simp*
show $\bigwedge a.\ a \notin S \Longrightarrow$ *function-uminus* S R f $a =$ *undefined*
by (*simp add*: *function-uminus-def*)
qed

lemma(**in** *ring-functions*) *function-uminus-eval*:
assumes $a \in S$
assumes $f \in$ *carrier* F
shows (*function-uminus* S R f) $a = \ominus (f\ a)$
using *assms* **unfolding** *function-uminus-def*
by *simp*

lemma(**in** *ring-functions*) *function-uminus-add-r*:
assumes $a \in S$
assumes $f \in$ *carrier* F

shows $f \oplus_F$ *function-uminus* S R $f = \mathbf{0}_F$
apply(*rule function-ring-car-eqI*)
using *assms fun-add-closed function-uminus-closed* **apply** *blast*
unfolding *F-def* **using** *F-def function-zero-closed* **apply** *blast*
using *F-def assms*(*2*) *fun-add-eval-car function-ring-car-closed function-uminus-closed*

function-uminus-eval function-zero-eval r-neg **by** *auto*

lemma(**in** *ring-functions*) *function-uminus-add-l*:
assumes $a \in S$
assumes $f \in$ *carrier* F
shows *function-uminus* S R $f \oplus_F f = \mathbf{0}_F$
using *assms*(*1*) *assms*(*2*) *fun-add-comm function-uminus-add-r function-uminus-closed*
by *auto*

1.3.8 Scalar Multiplication

lemma(**in** *ring-functions*) *function-smult-eval*:
assumes $a \in$ *carrier* R
assumes $f \in$ *carrier* F
assumes $b \in S$
shows $(a \odot_F f)\ b = a \otimes (f\ b)$
using *function-ring-defs*(*6*) **unfolding** *function-scalar-mult-def*
by(*simp add*: *assms*)

lemma(**in** *ring-functions*) *function-smult-closed*:
assumes $a \in$ *carrier* R
assumes $f \in$ *carrier* F
shows $a \odot_F f \in$ *carrier* F
apply(*rule function-ring-car-memI*)
using *function-smult-eval assms*
apply (*simp add*: *function-ring-car-closed*)
using *function-scalar-mult-def F-def*
by (*metis function-ring-defs*(*6*) *restrict-apply*)

lemma(**in** *ring-functions*) *function-smult-assoc1*:
assumes $a \in$ *carrier* R
assumes $b \in$ *carrier* R
assumes $f \in$ *carrier* F
shows $b \odot_F (a \odot_F f) = (b \otimes a) \odot_F f$
apply(*rule function-ring-car-eqI*)
using *assms function-smult-closed* **apply** *simp*
using *assms function-smult-closed* **apply** *simp*
by (*metis F-def assms*(*1*) *assms*(*2*) *assms*(*3*) *function-mult-eval-closed function-one-closed*
function-smult-eval function-times-one-r m-assoc m-closed ring-functions.function-smult-closed ring-functions-axioms)

lemma(**in** *ring-functions*) *function-smult-assoc2*:

assumes $a \in$ *carrier* R
assumes $f \in$ *carrier* F
assumes $g \in$ *carrier* F
shows $(a \odot_F f) \otimes_F g = a \odot_F (f \otimes_F g)$
apply(*rule function-ring-car-eqI*)
using *assms function-smult-closed* **apply** (*simp add*: *fun-mult-closed*)
apply (*simp add*: *assms*(*1*) *assms*(*2*) *assms*(*3*) *fun-mult-closed function-smult-closed*)
by (*metis* (*full-types*) *F-def assms*(*1*) *assms*(*2*) *assms*(*3*) *fun-mult-closed function-mult-eval-car function-smult-closed function-smult-eval m-assoc ring-functions.function-ring-car-cl ring-functions-axioms*)

lemma(**in** *ring-functions*) *function-smult-one*:
assumes $f \in$ *carrier* F
shows $\mathbf{1} \odot_F f = f$
apply(*rule function-ring-car-eqI*)
apply (*simp add*: *assms function-smult-closed*)
apply (*simp add*: *assms*)
by (*simp add*: *assms function-ring-car-closed function-smult-eval*)

lemma(**in** *ring-functions*) *function-smult-l-distr*:
[| $a \in$ *carrier* R; $b \in$ *carrier* R; $x \in$ *carrier* F |] ==>
$(a \oplus b) \odot_F x = a \odot_F x \oplus_F b \odot_F x$
apply(*rule function-ring-car-eqI*)
apply (*simp add*: *function-smult-closed*)
apply (*simp add*: *fun-add-closed function-smult-closed*)
using *function-smult-eval*
by (*simp add*: *fun-add-eval-car function-ring-car-closed function-smult-closed l-distr*)

lemma(**in** *ring-functions*) *function-smult-r-distr*:
[| $a \in$ *carrier* R; $x \in$ *carrier* F; $y \in$ *carrier* F |] ==>
$a \odot_F (x \oplus_F y) = a \odot_F x \oplus_F a \odot_F y$
apply(*rule function-ring-car-eqI*)
apply (*simp add*: *fun-add-closed function-smult-closed*)
apply (*simp add*: *fun-add-closed function-smult-closed*)
by (*simp add*: *fun-add-closed fun-add-eval-car function-ring-car-closed function-smult-closed function-smult-eval r-distr*)

1.3.9 The Ring of Functions Forms an Algebra

lemma(**in** *ring-functions*) *function-ring-is-abelian-group*:
abelian-group F
apply(*rule abelian-groupI*)
apply (*simp add*: *fun-add-closed*)
apply (*simp add*: *function-zero-closed*)
using *fun-add-assoc* **apply** *simp*
apply (*simp add*: *fun-add-comm*)
apply (*simp add*: *fun-add-comm fun-add-zeroR function-zero-closed*)
using *fun-add-zeroL function-ring-car-eqI function-uminus-add-l*

function-uminus-closed function-zero-closed **by** *blast*

lemma(**in** *ring-functions*) *function-ring-is-monoid*:
monoid F
apply(*rule monoidI*)
apply (*simp add*: *fun-mult-closed*)
apply (*simp add*: *function-one-closed*)
apply (*simp add*: *fun-mult-assoc*)
apply (*simp add*: *function-times-one-l*)
by (*simp add*: *function-times-one-r*)

lemma(**in** *ring-functions*) *function-ring-is-ring*:
ring F
apply(*rule ringI*)
apply (*simp add*: *function-ring-is-abelian-group*)
apply (*simp add*: *function-ring-is-monoid*)
apply (*simp add*: *function-mult-r-distr*)
by (*simp add*: *function-mult-l-distr*)

sublocale *ring-functions* < *F?*: *ring F*
by (*rule function-ring-is-ring*)

lemma(**in** *cring-functions*) *function-mult-comm*:
assumes $x \in$ *carrier F*
assumes $y \in$ *carrier F*
shows $x \otimes_F y = y \otimes_F x$
apply(*rule function-ring-car-eqI*)
apply (*simp add*: *assms*(*1*) *assms*(*2*) *fun-mult-closed*)
apply (*simp add*: *assms*(*1*) *assms*(*2*) *fun-mult-closed*)
by (*simp add*: *assms*(*1*) *assms*(*2*) *function-mult-eval-car function-ring-car-closed m-comm*)

lemma(**in** *cring-functions*) *function-ring-is-comm-monoid*:
comm-monoid F
apply(*rule comm-monoidI*)
using *fun-mult-assoc function-one-closed*
apply (*simp add*: *fun-mult-closed*)
apply (*simp add*: *function-one-closed*)
apply (*simp add*: *fun-mult-assoc*)
apply (*simp add*: *function-times-one-l*)
by (*simp add*: *function-mult-comm*)

lemma(**in** *cring-functions*) *function-ring-is-cring*:
cring F
apply(*rule cringI*)
apply (*simp add*: *function-ring-is-abelian-group*)
apply (*simp add*: *function-ring-is-comm-monoid*)
by (*simp add*: *function-mult-r-distr*)

lemma(**in** *cring-functions*) *function-ring-is-algebra*:
algebra R F
apply(*rule algebraI*)
apply (*simp add*: *is-cring*)
apply (*simp add*: *function-ring-is-cring*)
using *function-smult-closed* **apply** *blast*
apply (*simp add*: *function-smult-l-distr*)
apply (*simp add*: *function-smult-r-distr*)
apply (*simp add*: *function-smult-assoc1*)
apply (*simp add*: *function-smult-one*)
by (*simp add*: *function-smult-assoc2*)

lemma(**in** *ring-functions*) *function-uminus*:
assumes $f \in carrier\ F$
shows $\ominus_F f = (function\text{-}uminus\ S\ R)\ f$
using *assms a-inv-def*[*of F*]
by (*metis F-def abelian-group.a-group abelian-group.r-neg function-uminus-add-r*
function-uminus-closed group.inv-closed partial-object.select-convs(*1*) *ring.ring-simprules*(*18*)
ring-functions.function-ring-car-eqI ring-functions.function-ring-is-abelian-group ring-functions.function-ring-i
ring-functions-axioms)

lemma(**in** *ring-functions*) *function-uminus-eval'*:
assumes $f \in carrier\ F$
assumes $a \in S$
shows $(\ominus_F f)\ a = (function\text{-}uminus\ S\ R)\ f\ a$
using *assms*
by (*simp add*: *function-uminus*)

lemma(**in** *ring-functions*) *function-uminus-eval''*:
assumes $f \in carrier\ F$
assumes $a \in S$
shows $(\ominus_F f)\ a = \ominus\ (f\ a)$
using *assms*(*1*) *assms*(*2*) *function-uminus*
by (*simp add*: *function-uminus-eval*)

sublocale *cring-functions* $<$ *F*?: *algebra R F*
using *function-ring-is-algebra* **by** *auto*

1.4 Constant Functions

definition *constant-function* **where**
$constant\text{-}function\ S\ a = (\lambda x \in S.\ a)$

abbreviation(**in** *ring-functions*)(*input*) *const* **where**
$const \equiv constant\text{-}function\ S$

lemma(**in** *ring-functions*) *constant-function-closed*:
assumes $a \in carrier\ R$
shows $const\ a \in carrier\ F$

apply(*rule function-ring-car-memI*)
unfolding *constant-function-def*
apply (*simp add*: *assms*)
by *simp*

lemma(**in** *ring-functions*) *constant-functionE*:
assumes $a \in carrier\ R$
assumes $b \in S$
shows $const\ a\ b = a$
by (*simp add*: *assms*(*2*) *constant-function-def*)

lemma(**in** *ring-functions*) *constant-function-add*:
assumes $a \in carrier\ R$
assumes $b \in carrier\ R$
shows $const\ (a \oplus_R b) = (const\ a) \oplus_F (const\ b)$
apply(*rule function-ring-car-eqI*)
apply (*simp add*: *constant-function-closed assms*(*1*) *assms*(*2*))
using *assms*(*1*) *constant-function-closed assms*(*2*) *fun-add-closed* **apply** *auto*[*1*]
by (*simp add*: *assms*(*1*) *assms*(*2*) *constant-function-closed constant-functionE fun-add-eval-car*)

lemma(**in** *ring-functions*) *constant-function-mult*:
assumes $a \in carrier\ R$
assumes $b \in carrier\ R$
shows $const\ (a \otimes_R b) = (const\ a) \otimes_F (const\ b)$
apply(*rule function-ring-car-eqI*)
apply (*simp add*: *constant-function-closed assms*(*1*) *assms*(*2*))
using *assms*(*1*) *constant-function-closed assms*(*2*) *fun-mult-closed* **apply** *auto*[*1*]
by (*simp add*: *constant-function-closed assms*(*1*) *assms*(*2*) *constant-functionE function-mult-eval-car*)

lemma(**in** *ring-functions*) *constant-function-minus*:
assumes $a \in carrier\ R$
shows $\ominus_F(const\ a) = (const\ (\ominus_R\ a))$
apply(*rule function-ring-car-eqI*)
apply (*simp add*: *constant-function-closed assms local.function-uminus*)
apply (*simp add*: *constant-function-closed assms function-uminus-closed*)
apply (*simp add*: *constant-function-closed assms*)
by (*simp add*: *constant-function-closed assms constant-functionE function-uminus-eval''*)

lemma(**in** *ring-functions*) *function-one-is-constant*:
$const\ \mathbf{1} = \mathbf{1}_F$
unfolding *F-def*
apply(*rule function-ring-car-eqI*)
apply (*simp add*: *constant-function-closed*)
using *F-def function-one-closed* **apply** *auto*[*1*]
using *F-def constant-functionE function-one-eval* **by** *auto*

lemma(**in** *ring-functions*) *function-zero-is-constant*:
const $\mathbf{0} = \mathbf{0}_F$
apply(*rule function-ring-car-eqI*)
apply (*simp add*: *constant-function-closed*)
using *F-def function-zero-closed* **apply** *auto*[*1*]
using *F-def constant-functionE function-zero-eval* **by** *auto*

1.5 Special Examples of Functions Rings

1.5.1 Functions from the Carrier of a Ring to Itself

locale *U-function-ring* = *ring*

locale *U-function-cring* = *U-function-ring* + *cring*

sublocale *U-function-ring* < *S?*: *struct-functions R carrier R*
done

sublocale *U-function-ring* < *FunR?*: *ring-functions R carrier R Fun R*
apply (*simp add*: *local.ring-axioms ring-functions.intro*)
by *simp*

sublocale *U-function-cring* < *FunR?*: *cring-functions R carrier R Fun R*
apply (*simp add*: *cring-functions-def is-cring ring-functions-axioms*)
by *simp*

abbreviation(**in** *U-function-ring*)(*input*) *ring-compose* :: ('*a* ⇒ '*a*) ⇒ ('*a* ⇒ '*a*)
⇒ ('*a* ⇒ '*a*) **where**
ring-compose ≡ *compose* (*carrier R*)

lemma(**in** *U-function-ring*) *ring-function-ring-comp*:
assumes *f* ∈ *carrier* (*Fun R*)
assumes *g* ∈ *carrier* (*Fun R*)
shows *ring-compose f g* ∈ *carrier* (*Fun R*)
apply(*rule function-ring-car-memI*)
apply (*simp add*: *assms*(*1*) *assms*(*2*) *compose-eq*)
apply (*simp add*: *assms*(*1*) *assms*(*2*) *function-ring-car-closed*)
by (*meson compose-extensional extensional-arb*)

abbreviation(**in** *U-function-ring*)(*input*) *ring-const* (‹𝔠1›) **where**
ring-const ≡ *constant-function* (*carrier R*)

lemma(**in** *ring-functions*) *function-nat-pow-eval*:
assumes *f* ∈ *carrier F*
assumes *s* ∈ *S*
shows (*f*[^]$_F$(*n*::*nat*)) *s* = (*f s*)[^]*n*
apply(*induction n*)
using *assms*(*2*) *function-one-eval* **apply** *auto*[*1*]
by (*simp add*: *assms*(*1*) *assms*(*2*) *function-mult-eval-car function-ring-is-monoid*

monoid.nat-pow-closed)

context *U-function-ring*
begin

definition *a-translate* :: *'a* ⇒ *'a* ⇒ *'a* **where**
a-translate = (λ *r* ∈ *carrier R*. *restrict* ((*add R*) *r*) (*carrier R*))

definition *m-translate* :: *'a* ⇒ *'a* ⇒ *'a* **where**
m-translate = (λ *r* ∈ *carrier R*. *restrict* ((*mult R*) *r*) (*carrier R*))

definition *nat-power* :: *nat* ⇒ *'a* ⇒ *'a* **where**
nat-power = (λ(*n*::*nat*). *restrict* (λ*a*. $a[\widehat{}]_R n$) (*carrier R*))

Restricted operations are in Fs

lemma *a-translate-functions*:
assumes *c* ∈ *carrier R*
shows *a-translate c* ∈ *carrier* (*Fun R*)
apply(*rule function-ring-car-memI*)
using *assms a-translate-def*
apply *simp*
using *assms a-translate-def*
by *simp*

lemma *m-translate-functions*:
assumes *c* ∈ *carrier R*
shows *m-translate c* ∈ *carrier* (*Fun R*)
apply(*rule function-ring-car-memI*)
using *assms m-translate-def*
apply *simp*
using *assms m-translate-def*
by *simp*

lemma *nat-power-functions*:
shows *nat-power n* ∈ *carrier* (*Fun R*)
apply(*rule function-ring-car-memI*)
using *nat-power-def*
apply *simp*
by (*simp add*: *nat-power-def*)

Restricted operations simps

lemma *a-translate-eq*:
assumes *c* ∈ *carrier R*
assumes *a* ∈ *carrier R*
shows *a-translate c a* = *c* ⊕ *a*
by (*simp add*: *a-translate-def assms*(*1*) *assms*(*2*))

lemma *a-translate-eq'*:

assumes $c \in$ *carrier* R
assumes $a \notin$ *carrier* R
shows *a-translate* c a = *undefined*
by (*meson a-translate-functions assms*(*1*) *assms*(*2*) *function-ring-not-car*)

lemma *a-translate-eq''*:
assumes $c \notin$ *carrier* R
shows *a-translate* c = *undefined*
by (*simp add*: *a-translate-def assms*)

lemma *m-translate-eq*:
assumes $c \in$ *carrier* R
assumes $a \in$ *carrier* R
shows *m-translate* c $a = c \otimes a$
by (*simp add*: *m-translate-def assms*(*1*) *assms*(*2*))

lemma *m-translate-eq'*:
assumes $c \in$ *carrier* R
assumes $a \notin$ *carrier* R
shows *m-translate* c a = *undefined*
by (*meson m-translate-functions assms*(*1*) *assms*(*2*) *function-ring-not-car*)

lemma *m-translate-eq''*:
assumes $c \notin$ *carrier* R
shows *m-translate* c = *undefined*
by (*simp add*: *m-translate-def assms*)

lemma *nat-power-eq*:
assumes $a \in$ *carrier* R
shows *nat-power* n $a = a[\widehat{}]_R$ n
by (*simp add*: *assms nat-power-def*)

lemma *nat-power-eq'*:
assumes $a \notin$ *carrier* R
shows *nat-power* n a = *undefined*
by (*simp add*: *assms nat-power-def*)

Constant ring_function properties

lemma *constant-function-eq*:
assumes $a \in$ *carrier* R
assumes $b \in$ *carrier* R
shows $\mathfrak{c}_a$ $b = a$
using *assms*

by (*simp add*: *constant-functionE*)

lemma *constant-function-eq'*:
assumes $a \in$ *carrier* R
assumes $b \notin$ *carrier* R

shows $\mathfrak{c}_a\ b = \mathit{undefined}$
by (*simp add*: *constant-function-closed assms*(*1*) *assms*(*2*) *function-ring-not-car*)

Compound expressions from algebraic operations

end

definition *monomial-function* **where**
monomial-function R c (*n*::*nat*) = (λ *x* ∈ *carrier R*. *c* $\otimes_R$ (*x*[^]$_R$*n*))

context *U-function-ring*
begin

abbreviation *monomial* **where**
monomial ≡ *monomial-function R*

lemma *monomial-functions*:
assumes *c* ∈ *carrier R*
shows *monomial c n* ∈ *carrier* (*Fun R*)
apply(*rule function-ring-car-memI*)
unfolding *monomial-function-def*
apply (*simp add*: *assms*)
by *simp*

definition *ring-id* **where**
ring-id ≡ *restrict* (λ*x*. *x*) (*carrier R*)

lemma *ring-id-closed*[*simp*]:
ring-id ∈ *carrier* (*Fun R*)
by (*simp add*: *function-ring-car-memI ring-id-def*)

lemma *ring-id-eval*:
assumes *a* ∈ *carrier R*
shows *ring-id a* = *a*
using *assms* **unfolding** *ring-id-def*
by *simp*

lemma *constant-a-trans*:
assumes *a* ∈*carrier R*
shows *m-translate a* = $\mathfrak{c}_a$ $\otimes_{Fun\ R}$ *ring-id*
proof(*rule function-ring-car-eqI*)
show *m-translate a* ∈ *carrier* (*Fun R*)
using *assms*
using *m-translate-functions* **by** *blast*
show $\mathfrak{c}_a$ $\otimes_{Fun\ R}$ *ring-id* ∈ *carrier* (*Fun R*)
unfolding *ring-id-def*
using *assms ring-id-closed ring-id-def*
by (*simp add*: *constant-function-closed fun-mult-closed*)
show $\bigwedge x.\ x \in carrier\ R \Longrightarrow$ *m-translate a x* = ($\mathfrak{c}_a$ $\otimes_{Fun\ R}$ *ring-id*) *x*
by (*simp add*: *constant-function-closed assms constant-function-eq function-mult-eval-car*

m-translate-eq ring-id-eval)
qed

polynomials in one variable

fun *polynomial* :: *'a list* ⇒ (*'a* ⇒ *'a*) **where**
polynomial [] = $\mathbf{0}_{Fun\ R}$ |
polynomial (*a*#*as*) = (λ*x* ∈ *carrier R*. *a* ⊕ *x* ⊗ (*polynomial as x*))

lemma *polynomial-induct-lemma*:
assumes *f* ∈ *carrier* (*Fun R*)
assumes *a* ∈ *carrier R*
shows (λ*x* ∈ *carrier R*. *a* ⊕ *x* ⊗ (*f x*)) ∈ *carrier* (*Fun R*)
proof(*rule function-ring-car-memI*)
show ⋀*aa*. *aa* ∈ *carrier R* ⟹ (λ*x*∈*carrier R*. *a* ⊕ *x* ⊗ *f x*) *aa* ∈ *carrier R*
proof− **fix** *y* **assume** *A*: *y* ∈ *carrier R*
have *a* ⊕ *y* ⊗ *f y* ∈ *carrier R*
using *A assms*(*1*) *assms*(*2*) *function-ring-car-closed* **by** *blast*
thus (λ*x*∈*carrier R*. *a* ⊕ *x* ⊗ *f x*) *y* ∈ *carrier R*
using *A* **by** *auto*
qed
show ⋀*aa*. *aa* ∉ *carrier R* ⟹ (λ*x*∈*carrier R*. *a* ⊕ *x* ⊗ *f x*) *aa* = *undefined*
by *auto*
qed

lemma *polynomial-function*:
shows *set as* ⊆ *carrier R* ⟹ *polynomial as* ∈ *carrier* (*Fun R*)
proof(*induction as*)
case *Nil*
then show *?case*
by (*simp add*: *function-zero-closed*)
next
case (*Cons a as*)
then show *polynomial* (*a* # *as*) ∈ *carrier* (*function-ring* (*carrier R*) *R*)
using *polynomial.simps*(*2*)[*of a as*] *polynomial-induct-lemma*[*of polynomial as a*]
by *simp*
qed

lemma *polynomial-constant*:
assumes *a* ∈ *carrier R*
shows *polynomial* [*a*] = $\mathfrak{c}_a$
apply(*rule function-ring-car-eqI*)
using *assms polynomial-function*
apply (*metis* (*full-types*) *list.distinct*(*1*) *list.set-cases set-ConsD subset-code*(*1*))
apply (*simp add*: *constant-function-closed assms*)
using *polynomial.simps*(*2*)[*of a* []] *polynomial.simps*(*1*) *assms*
by (*simp add*: *constant-function-eq function-zero-eval*)

end

1.5.2 Sequences Indexed by the Natural Numbers

definition *nat-seqs* ($\langle$-$^{\omega}\rangle$)**where**
nat-seqs R ≡ *function-ring* (*UNIV*::*nat set*) *R*

abbreviation(*input*) *closed-seqs* **where**
closed-seqs R ≡ *carrier* (R^{ω})

lemma *closed-seqs-memI*:
assumes ⋀*k*. *s k* ∈ *carrier R*
shows *s* ∈ *closed-seqs R*
unfolding *nat-seqs-def function-ring-def*
by (*simp add*: *PiE-UNIV-domain assms*)

lemma *closed-seqs-memE*:
assumes *s* ∈ *closed-seqs R*
shows *s k* ∈ *carrier R*
using *assms* **unfolding** *nat-seqs-def function-ring-def*
by (*simp add*: *PiE-iff*)

definition *is-constant-fun* **where**
is-constant-fun R f = (∃ *x* ∈ *carrier R*. *f* = *constant-function* (*carrier R*) *R x*)

definition *is-constant-seq* **where**
is-constant-seq R s = (∃ *x* ∈ *carrier R*. *s* = *constant-function* (*UNIV*::*nat set*) *x*)

lemma *is-constant-seqI*:
fixes *a*
assumes *s* ∈ *closed-seqs R*
assumes ⋀*k*. *s k* = *a*
shows *is-constant-seq R s*
unfolding *is-constant-seq-def constant-function-def*
by (*metis assms*(*1*) *assms*(*2*) *closed-seqs-memE restrict-UNIV restrict-ext*)

lemma *is-constant-seqE*:
assumes *is-constant-seq R s*
assumes *s k* = *a*
shows *s n* = *a*
using *assms* **unfolding** *is-constant-seq-def*
by (*metis constant-function-def restrict-UNIV*)

lemma *is-constant-seq-imp-closed*:
assumes *is-constant-seq R s*
shows *s* ∈ *closed-seqs R*
apply(*rule closed-seqs-memI*)
using *assms* **unfolding** *is-constant-seq-def constant-function-def*
by *auto*

context *U-function-ring*
begin

Sequence sums and products are closed

lemma *seq-plus-closed*:
assumes $s \in$ *closed-seqs R*
assumes $s' \in$ *closed-seqs R*
shows $s \oplus_{R^\omega} s' \in$ *closed-seqs R*
by (*metis assms*(*1*) *assms*(*2*) *nat-seqs-def ring-functions.fun-add-closed ring-functions-axioms*)

lemma *seq-mult-closed*:
assumes $s \in$ *closed-seqs R*
assumes $s' \in$ *closed-seqs R*
shows $s \otimes_{R^\omega} s' \in$ *closed-seqs R*
apply(*rule closed-seqs-memI*)
by (*metis assms*(*1*) *assms*(*2*) *closed-seqs-memE nat-seqs-def ring-functions.fun-mult-closed ring-functions-axioms*)

lemma *constant-function-comp-is-closed-seq*:
assumes $a \in$ *carrier R*
assumes $s \in$ *closed-seqs R*
shows (*const a* $\circ$ *s*) $\in$ *closed-seqs R*
by (*simp add*: *constant-functionE assms*(*1*) *assms*(*2*) *closed-seqs-memE closed-seqs-memI*)

lemma *constant-function-comp-is-constant-seq*:
assumes $a \in$ *carrier R*
assumes $s \in$ *closed-seqs R*
shows *is-constant-seq R* ((*const a*) $\circ$ *s*)
apply(*rule is-constant-seqI*[*of - - a*])
apply (*simp add*: *assms*(*1*) *assms*(*2*) *constant-function-comp-is-closed-seq*)
using *assms*(*1*) *assms*(*2*) *closed-seqs-memE*
by (*simp add*: *closed-seqs-memE constant-functionE*)

lemma *function-comp-is-closed-seq*:
assumes $s \in$ *closed-seqs R*
assumes $f \in$ *carrier* (*Fun R*)
shows $f \circ s \in$ *closed-seqs R*
apply(*rule closed-seqs-memI*)
using *assms*(*1*) *assms*(*2*) *closed-seqs-memE*
by (*metis comp-apply fun-add-eval-closed fun-add-zeroR function-zero-closed*)

lemma *function-sum-comp-is-seq-sum*:
assumes $s \in$ *closed-seqs R*
assumes $f \in$ *carrier* (*Fun R*)
assumes $g \in$ *carrier* (*Fun R*)
shows ($f \oplus_{Fun\ R} g$) $\circ$ $s = (f \circ s) \oplus_{R^\omega} (g \circ s)$
apply(*rule ring-functions.function-ring-car-eqI*[*of R - UNIV* :: *nat set*])
apply (*simp add*: *ring-functions-axioms*)

using *function-comp-is-closed-seq*
apply (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *fun-add-closed nat-seqs-def*)
apply (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *function-comp-is-closed-seq nat-seqs-def seq-plus-closed*)
by (*smt* (*verit*) *UNIV-eq-I assms*(*1*) *assms*(*2*) *assms*(*3*) *closed-seqs-memE comp-apply function-comp-is-closed-seq nat-seqs-def ring-functions.fun-add-eval-car ring-functions-axioms*)

lemma *function-mult-comp-is-seq-mult*:
assumes $s \in$ *closed-seqs R*
assumes $f \in$ *carrier* (*Fun R*)
assumes $g \in$ *carrier* (*Fun R*)
shows $(f \otimes_{Fun\ R} g) \circ s = (f \circ s) \otimes_{R^\omega} (g \circ s)$
apply(*rule ring-functions.function-ring-car-eqI*[*of R - UNIV* :: *nat set*])
apply (*simp add*: *ring-functions-axioms*)
using *function-comp-is-closed-seq*
apply (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *fun-mult-closed nat-seqs-def*)
apply (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *function-comp-is-closed-seq nat-seqs-def seq-mult-closed*)
by (*metis* (*no-types*, *lifting*) *assms*(*1*) *assms*(*2*) *assms*(*3*) *comp-apply function-comp-is-closed-seq nat-seqs-def ring-functions.function-mult-eval-car ring-functions.function-ring-car-closed ring-functions-axioms*)

lemma *seq-plus-simp*:
assumes $s \in$ *closed-seqs R*
assumes $t \in$ *closed-seqs R*
shows $(s \oplus_{R^\omega} t)\ k = s\ k \oplus t\ k$
using *assms* **unfolding** *nat-seqs-def*
by (*simp add*: *ring-functions.fun-add-eval-car ring-functions-axioms*)

lemma *seq-mult-simp*:
assumes $s \in$ *closed-seqs R*
assumes $t \in$ *closed-seqs R*
shows $(s \otimes_{R^\omega} t)\ k = s\ k \otimes t\ k$
using *assms* **unfolding** *nat-seqs-def*
by (*simp add*: *ring-functions.function-mult-eval-car ring-functions-axioms*)

lemma *seq-one-simp*:
$\mathbf{1}_{R^\omega}\ k = \mathbf{1}$
by (*simp add*: *nat-seqs-def ring-functions.function-one-eval ring-functions-axioms*)

lemma *seq-zero-simp*:
$\mathbf{0}_{R^\omega}\ k = \mathbf{0}$
by (*simp add*: *nat-seqs-def ring-functions.function-zero-eval ring-functions-axioms*)

lemma(**in** *U-function-ring*) *ring-id-seq-comp*:
assumes $s \in$ *closed-seqs R*
shows *ring-id* $\circ\ s = s$
apply(*rule ring-functions.function-ring-car-eqI*[*of R - UNIV*::*nat set*])
using *ring-functions-axioms* **apply** *auto*[*1*]

apply (*metis assms function-comp-is-closed-seq nat-seqs-def ring-id-closed*)
apply (*metis assms nat-seqs-def*)
by (*simp add*: *assms closed-seqs-memE ring-id-eval*)

lemma(**in** *U-function-ring*) *ring-seq-smult-closed*:
assumes $s \in$ *closed-seqs R*
assumes $a \in$ *carrier R*
shows $a \odot_{R^\omega} s \in$ *closed-seqs R*
apply(*rule closed-seqs-memI*)
by (*metis assms*(*1*) *assms*(*2*) *closed-seqs-memE nat-seqs-def ring-functions.function-smult-closed ring-functions-axioms*)

lemma(**in** *U-function-ring*) *ring-seq-smult-eval*:
assumes $s \in$ *closed-seqs R*
assumes $a \in$ *carrier R*
shows $(a \odot_{R^\omega} s)\ k = a \otimes (s\ k)$
by (*metis UNIV-I assms*(*1*) *assms*(*2*) *nat-seqs-def ring-functions.function-smult-eval ring-functions-axioms*)

lemma(**in** *U-function-ring*) *ring-seq-smult-comp-assoc*:
assumes $s \in$ *closed-seqs R*
assumes $f \in$ *carrier* (*Fun R*)
assumes $a \in$ *carrier R*
shows $((a \odot_{Fun\ R} f) \circ s) = a \odot_{R^\omega} (f \circ s)$
apply(*rule ext*)
using *function-smult-eval*[*of a f*] *ring-seq-smult-eval*[*of f* $\circ$ *s a*]
by (*simp add*: *assms*(*1*) *assms*(*2*) *assms*(*3*) *closed-seqs-memE function-comp-is-closed-seq*)

end

2 Extensional Maps Between the Carriers of two Structures

definition *struct-maps* :: ('*a*, '*c*) *partial-object-scheme* ⇒ ('*b*, '*d*) *partial-object-scheme*

⇒ ('*a* ⇒ '*b*) *set* **where**
struct-maps T S = {*f*. (*f* ∈ (*carrier T*) → (*carrier S*)) ∧ *f* = *restrict f* (*carrier T*) }

definition *to-struct-map* **where**
to-struct-map T f = *restrict f* (*carrier T*)

lemma *to-struct-map-closed*:
assumes *f* ∈ (*carrier T*) → (*carrier S*)
shows *to-struct-map T f* ∈ (*struct-maps T S*)
by (*smt* (*verit*) *PiE-restrict Pi-iff assms mem-Collect-eq restrict-PiE struct-maps-def to-struct-map-def*)

lemma *struct-maps-memI*:
assumes $\bigwedge x.\ x \in carrier\ T \Longrightarrow f\ x \in carrier\ S$
assumes $\bigwedge x.\ x \notin carrier\ T \Longrightarrow f\ x = undefined$
shows $f \in struct\text{-}maps\ T\ S$
proof−
have *0*: $(f \in (carrier\ T) \to (carrier\ S))$
using *assms*
by *blast*
have *1*: $f = restrict\ f\ (carrier\ T)$
using *assms*
by (*simp add*: *extensional-def extensional-restrict*)
show *?thesis*
using *0 1*
unfolding *struct-maps-def*
by *blast*
qed

lemma *struct-maps-memE*:
assumes $f \in struct\text{-}maps\ T\ S$
shows $\bigwedge x.\ x \in carrier\ T \Longrightarrow f\ x \in carrier\ S$
$\bigwedge x.\ x \notin carrier\ T \Longrightarrow f\ x = undefined$
using *assms* **unfolding** *struct-maps-def*
apply *blast*
using *assms* **unfolding** *struct-maps-def*
by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq restrict-apply*)

An abbreviation for restricted composition of function of functions. This is necessary for the composition of two struct maps to again be a struct map.

abbreviation(*input*) *rcomp*
where $rcomp \equiv FuncSet.compose$

lemma *struct-map-comp*:
assumes $g \in (struct\text{-}maps\ T\ S)$
assumes $f \in (struct\text{-}maps\ S\ U)$
shows $rcomp\ (carrier\ T)\ f\ g \in (struct\text{-}maps\ T\ U)$
proof(*rule struct-maps-memI*)
show $\bigwedge x.\ x \in carrier\ T \Longrightarrow rcomp\ (carrier\ T)\ f\ g\ x \in carrier\ U$
using *assms struct-maps-memE*(*1*)
by (*metis compose-eq*)
show $\bigwedge x.\ x \notin carrier\ T \Longrightarrow rcomp\ (carrier\ T)\ f\ g\ x = undefined$
by (*meson compose-extensional extensional-arb*)
qed

lemma *r-comp-is-compose*:
assumes $g \in (struct\text{-}maps\ T\ S)$
assumes $f \in (struct\text{-}maps\ S\ U)$
assumes $a \in (carrier\ T)$
shows $(rcomp\ (carrier\ T)\ f\ g)\ a = (f \circ g)\ a$
by (*simp add*: *FuncSet.compose-def assms*(*3*))

lemma *r-comp-not-in-car*:
assumes $g \in (\mathit{struct\text{-}maps}\ T\ S)$
assumes $f \in (\mathit{struct\text{-}maps}\ S\ U)$
assumes $a \notin (\mathit{carrier}\ T)$
shows $(\mathit{rcomp}\ (\mathit{carrier}\ T)\ f\ g)\ a = \mathit{undefined}$
by (*simp add*: *FuncSet.compose-def assms*(*3*))

The reverse composition of two struct maps:

definition *pullback* ::
$('a, 'd)\ \mathit{partial\text{-}object\text{-}scheme} \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'c)$ **where**
$\mathit{pullback}\ T\ f\ g = \mathit{rcomp}\ (\mathit{carrier}\ T)\ g\ f$

lemma *pullback-closed*:
assumes $f \in (\mathit{struct\text{-}maps}\ T\ S)$
assumes $g \in (\mathit{struct\text{-}maps}\ S\ U)$
shows $\mathit{pullback}\ T\ f\ g \in (\mathit{struct\text{-}maps}\ T\ U)$
by (*metis assms*(*1*) *assms*(*2*) *pullback-def struct-map-comp*)

Composition of struct maps which takes the structure itself rather than the carrier as a parameter:

definition *pushforward* ::
$('a, 'd)\ \mathit{partial\text{-}object\text{-}scheme} \Rightarrow ('b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'c)$ **where**
$\mathit{pushforward}\ T\ f\ g \equiv \mathit{rcomp}\ (\mathit{carrier}\ T)\ f\ g$

lemma *pushforward-closed*:
assumes $g \in (\mathit{struct\text{-}maps}\ T\ S)$
assumes $f \in (\mathit{struct\text{-}maps}\ S\ U)$
shows $\mathit{pushforward}\ T\ f\ g \in (\mathit{struct\text{-}maps}\ T\ U)$
using *assms*(*1*) *assms*(*2*) *struct-map-comp*
by (*metis pushforward-def*)

end
theory *Cring-Poly*
imports *HOL−Algebra.UnivPoly HOL−Algebra.Subrings Function-Ring*
begin

This theory extends the material in *HOL−Algebra.UnivPoly*. The main additions are material on Taylor expansions of polynomials and polynomial derivatives, and various applications of the universal property of polynomial evaluation. These include construing polynomials as functions from the base ring to itself, composing one polynomial with another, and extending homomorphisms between rings to homomoprhisms of their polynomial rings. These formalizations are necessary components of the proof of Hensel's lemma for p-adic integers, and for the proof of p-adic quantifier elimination.

lemma(**in** *ring*) *ring-hom-finsum*:

assumes $h \in$ *ring-hom* R S
assumes *ring* S
assumes *finite* I
assumes $F \in I \to$ *carrier* R
shows h (*finsum* R F I) = *finsum* S ($h \circ F$) I
proof−
have *I*: ($h \in$ *ring-hom* R $S \wedge F \in I \to$ *carrier* R) $\longrightarrow$ h (*finsum* R F I) = *finsum* S ($h \circ F$) I
apply(*rule finite-induct*, *rule assms*)
using *assms ring-hom-zero*[*of h R S*]
apply (*metis abelian-group-def abelian-monoid.finsum-empty ring-axioms ring-def*)
proof(*rule*)
fix *A a*
assume *A*: *finite A a* $\notin$ *A h* $\in$ *ring-hom R S* $\wedge$ *F* $\in$ *A* $\to$ *carrier R* $\longrightarrow$
h (*finsum R F A*) = *finsum S* (*h* $\circ$ *F*) *A h* $\in$ *ring-hom R S* $\wedge$ *F* $\in$ *insert a A* $\to$ *carrier R*
have *0*: *h* $\in$ *ring-hom R S* $\wedge$ *F* $\in$ *A* $\to$ *carrier R*
using *A* **by** *auto*
have *1*: *h* (*finsum R F A*) = *finsum S* (*h* $\circ$ *F*) *A*
using *A 0* **by** *auto*
have *2*: *abelian-monoid S*
using *assms ring-def abelian-group-def* **by** *auto*
have *3*: *h* (*F a* $\oplus$ *finsum R F A*) = *h* (*F a*) $\oplus_S$ (*finsum S* (*h* $\circ$ *F*) *A*)
using *ring-hom-add assms finsum-closed 1 A*(*4*) **by** *fastforce*
have *4*: *finsum R F* (*insert a A*) = *F a* $\oplus$ *finsum R F A*
using *finsum-insert*[*of A a F*] *A assms* **by** *auto*
have *5*: *finsum S* (*h* $\circ$ *F*) (*insert a A*) = (*h* $\circ$ *F*) *a* $\oplus_S$ *finsum S* (*h* $\circ$ *F*) *A*
apply(*rule abelian-monoid.finsum-insert*[*of S A a h* $\circ$ *F*])
apply (*simp add*: *2*)
apply(*rule A*)
apply(*rule A*)
using *ring-hom-closed A 0* **apply** *fastforce*
using *A ring-hom-closed* **by** *auto*
show *h* (*finsum R F* (*insert a A*)) =
finsum S (*h* $\circ$ *F*) (*insert a A*)
unfolding *4 5 3* **by** *auto*
qed
thus *?thesis* **using** *assms* **by** *blast*
qed

lemma(**in** *ring*) *ring-hom-a-inv*:
assumes *ring S*
assumes *h* $\in$ *ring-hom R S*
assumes *b* $\in$ *carrier R*
shows *h* ($\ominus$ *b*) = $\ominus_S$ *h b*
proof−
have *h b* $\oplus_S$ *h* ($\ominus$ *b*) = $\mathbf{0}_S$
by (*metis* (*no-types*, *opaque-lifting*) *abelian-group.a-inv-closed assms*(*1*) *assms*(*2*) *assms*(*3*)

is-abelian-group local.ring-axioms r-neg ring-hom-add ring-hom-zero)
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *abelian-group.minus-equality add.inv-closed assms*(*1*)

assms(*2*) *assms*(*3*) *ring.is-abelian-group ring.ring-simprules*(*10*) *ring-hom-closed*)
qed

lemma(**in** *ring*) *ring-hom-minus*:
assumes *ring S*
assumes $h \in$ *ring-hom R S*
assumes $a \in$ *carrier R*
assumes $b \in$ *carrier R*
shows $h\ (a \ominus b) = h\ a \ominus_S h\ b$
using *assms ring-hom-add*[*of h R S a* $\ominus_R$ *b*]
unfolding *a-minus-def*
using *ring-hom-a-inv*[*of S h b*] **by** *auto*

lemma *ring-hom-nat-pow*:
assumes *ring R*
assumes *ring S*
assumes $h \in$ *ring-hom R S*
assumes $a \in$ *carrier R*
shows $h\ (a[\widehat{}]_R(n::nat)) = (h\ a)[\widehat{}]_S(n::nat)$
using *assms* **by** (*simp add*: *ring-hom-ring.hom-nat-pow ring-hom-ringI2*)

lemma (**in** *ring*) *Units-not-right-zero-divisor*:
assumes $a \in$ *Units R*
assumes $b \in$ *carrier R*
assumes $a \otimes b = \mathbf{0}$
shows $b = \mathbf{0}$
proof−
have *inv a* $\otimes$ *a* $\otimes$ *b* $= \mathbf{0}$
using *assms Units-closed Units-inv-closed r-null m-assoc*[*of inv a a b*] **by** *presburger*
thus *?thesis* **using** *assms*
by (*metis Units-l-inv l-one*)
qed

lemma (**in** *ring*) *Units-not-left-zero-divisor*:
assumes $a \in$ *Units R*
assumes $b \in$ *carrier R*
assumes $b \otimes a = \mathbf{0}$
shows $b = \mathbf{0}$
proof−
have $b \otimes (a \otimes$ *inv a*$) = \mathbf{0}$
using *assms Units-closed Units-inv-closed l-null m-assoc*[*of b ainv a*] **by** *presburger*
thus *?thesis* **using** *assms*
by (*metis Units-r-inv r-one*)

qed

lemma (**in** *cring*) *finsum-remove*:
assumes $\bigwedge i.\ i \in Y \Longrightarrow f\ i \in carrier\ R$
assumes *finite* Y
assumes $i \in Y$
shows $finsum\ R\ f\ Y = f\ i \oplus finsum\ R\ f\ (Y - \{i\})$
proof−
have $finsum\ R\ f\ (insert\ i\ (Y - \{i\})) = f\ i \oplus finsum\ R\ f\ (Y - \{i\})$
apply(*rule finsum-insert*)
using *assms* **apply** *blast* **apply** *blast* **using** *assms* **apply** *blast*
using *assms* **by** *blast*
thus *?thesis* **using** *assms*
by (*metis insert-Diff*)
qed

type-synonym *degree* = *nat*

The composition of two ring homomorphisms is a ring homomorphism

lemma *ring-hom-compose*:
assumes *ring* R
assumes *ring* S
assumes *ring* T
assumes $h \in ring\text{-}hom\ R\ S$
assumes $g \in ring\text{-}hom\ S\ T$
assumes $\bigwedge c.\ c \in carrier\ R \Longrightarrow f\ c = g\ (h\ c)$
shows $f \in\ ring\text{-}hom\ R\ T$
proof(*rule ring-hom-memI*)
show $\bigwedge x.\ x \in carrier\ R \Longrightarrow f\ x \in carrier\ T$
using *assms* **by** (*metis ring-hom-closed*)
show $\bigwedge x\ y.\ x \in carrier\ R \Longrightarrow y \in carrier\ R \Longrightarrow f\ (x \otimes_R y) = f\ x \otimes_T f\ y$
proof−
fix $x\ y$
assume A: $x \in\ carrier\ R\ y \in\ carrier\ R$
show $f\ (x \otimes_R y) = f\ x \otimes_T f\ y$
proof−
have $f\ (x \otimes_R y) = g\ (h\ (x \otimes_R y))$
by (*simp add*: $A(1)$ $A(2)$ $assms(1)$ $assms(6)$ *ring.ring-simprules*(5))
then have $f\ (x \otimes_R y) = g\ ((h\ x) \otimes_S (h\ y))$
using $A(1)$ $A(2)$ $assms(4)$ *ring-hom-mult* **by** *fastforce*
then have $f\ (x \otimes_R y) = g\ (h\ x) \otimes_T g\ (h\ y)$
using $A(1)$ $A(2)$ $assms(4)$ $assms(5)$ *ring-hom-closed ring-hom-mult* **by** *fastforce*
then show *?thesis*
by (*simp add*: $A(1)$ $A(2)$ $assms(6)$)
qed
qed
show $\bigwedge x\ y.\ x \in carrier\ R \Longrightarrow y \in carrier\ R \Longrightarrow f\ (x \oplus_R y) = f\ x \oplus_T f\ y$

proof−
fix x y
assume A: $x \in$ *carrier* R $y \in$ *carrier* R
show $f\ (x \oplus_R y) = f\ x \oplus_T f\ y$
proof−
have $f\ (x \oplus_R y) = g\ (h\ (x \oplus_R y))$
by (*simp add*: $A(1)$ $A(2)$ *assms*(1) *assms*(6) *ring.ring-simprules*(1))
then have $f\ (x \oplus_R y) = g\ ((h\ x) \oplus_S (h\ y))$
using $A(1)$ $A(2)$ *assms*(4) *ring-hom-add* **by** *fastforce*
then have $f\ (x \oplus_R y) = g\ (h\ x) \oplus_T g\ (h\ y)$
by (*metis* (*no-types*, *opaque-lifting*) $A(1)$ $A(2)$ *assms*(4) *assms*(5) *ring-hom-add* *ring-hom-closed*)
then show *?thesis*
by (*simp add*: $A(1)$ $A(2)$ *assms*(6))
qed
qed
show $f\ \mathbf{1}_R = \mathbf{1}_T$
by (*metis assms*(1) *assms*(4) *assms*(5) *assms*(6) *ring.ring-simprules*(6) *ring-hom-one*)
qed

3 Basic Notions about Polynomials

context *UP-ring*
begin

rings are closed under monomial terms

lemma *monom-term-car*:
assumes $c \in$ *carrier* R
assumes $x \in$ *carrier* R
shows $c \otimes x[\widehat{}](n::nat) \in$ *carrier* R
using *assms monoid.nat-pow-closed*
by *blast*

Univariate polynomial ring over R

lemma *P-is-UP-ring*:
UP-ring R
by (*simp add*: *UP-ring-axioms*)

Degree function

abbreviation(*input*) *degree* **where**
degree $f \equiv$ *deg* R f

lemma *UP-car-memI*:
assumes $\bigwedge n.\ n > k \Longrightarrow p\ n = \mathbf{0}$
assumes $\bigwedge n.\ p\ n \in$ *carrier* R
shows $p \in$ *carrier* P
proof−
have *bound* $\mathbf{0}$ k p

by (*simp add*: *assms*(*1*) *bound.intro*)
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *P-def UP-def assms*(*2*) *mem-upI partial-object.select-convs*(*1*))
qed

lemma(**in** *UP-cring*) *UP-car-memI'*:
assumes $\bigwedge x.\ g\ x \in carrier\ R$
assumes $\bigwedge x.\ x > k \Longrightarrow g\ x = \mathbf{0}$
shows $g \in carrier\ (UP\ R)$
proof−
have *bound* **0** *k g*
using *assms* **unfolding** *bound-def* **by** *blast*
then show *?thesis*
using *P-def UP-car-memI assms*(*1*) **by** *blast*
qed

lemma(**in** *UP-cring*) *UP-car-memE*:
assumes $g \in carrier\ (UP\ R)$
shows $\bigwedge x.\ g\ x \in carrier\ R$
$\bigwedge x.\ x > (deg\ R\ g) \Longrightarrow g\ x = \mathbf{0}$
using *P-def assms UP-def*[*of R*] **apply** (*simp add*: *mem-upD*)
using *assms UP-def*[*of R*] *up-def*[*of R*]
by (*smt* (*verit*, *del-insts*) *UP-ring.deg-aboveD is-UP-ring partial-object.select-convs*(*1*)
restrict-apply up-ring.select-convs(*2*))

end

3.1 Lemmas About Coefficients

context *UP-ring*
begin

The goal here is to reduce dependence on the function coeff from Univ_Poly, in favour of using a polynomial itself as its coefficient function.

lemma *coeff-simp*:
assumes $f \in carrier\ P$
shows $coeff\ (UP\ R)\ \ f = f$
proof fix *x* **show** $coeff\ (UP\ R)\ f\ x = f\ x$
using *assms P-def UP-def*[*of R*] **by** *auto*
qed

Coefficients are in R

lemma *cfs-closed*:
assumes $f \in carrier\ P$
shows $f\ n \in carrier\ R$
using *assms coeff-simp*[*of f*] *P-def coeff-closed*
by *fastforce*

lemma *cfs-monom*:

$a \in carrier\ R \Longrightarrow (monom\ P\ a\ m)\ n = (if\ m{=}n\ then\ a\ else\ \mathbf{0})$
using *coeff-simp P-def coeff-monom monom-closed* **by** *auto*

lemma *cfs-zero* [*simp*]: $\mathbf{0}_P\ n = \mathbf{0}$
using *P-def UP-zero-closed coeff-simp coeff-zero* **by** *auto*

lemma *cfs-one* [*simp*]: $\mathbf{1}_P\ n = (if\ n{=}0\ then\ \mathbf{1}\ else\ \mathbf{0})$
by (*metis P-def R.one-closed UP-ring.cfs-monom UP-ring-axioms monom-one*)

lemma *cfs-smult* [*simp*]:
$[\!|\ a \in carrier\ R;\ p \in carrier\ P\ |\!] \Longrightarrow (a \odot_P p)\ n = a \otimes p\ n$
using *P-def UP-ring.coeff-simp UP-ring-axioms UP-smult-closed coeff-smult* **by** *fastforce*

lemma *cfs-add* [*simp*]:
$[\!|\ p \in carrier\ P;\ q \in carrier\ P\ |\!] \Longrightarrow (p \oplus_P q)\ n = p\ n \oplus q\ n$
by (*metis P.add.m-closed P-def UP-ring.coeff-add UP-ring.coeff-simp UP-ring-axioms*)

lemma *cfs-a-inv* [*simp*]:
assumes *R*: $p \in carrier\ P$
shows $(\ominus_P p)\ n = \ominus (p\ n)$
using *P.add.inv-closed P-def UP-ring.coeff-a-inv UP-ring.coeff-simp UP-ring-axioms assms*
by *fastforce*

lemma *cfs-minus* [*simp*]:
$[\!|\ p \in carrier\ P;\ q \in carrier\ P\ |\!] \Longrightarrow (p \ominus_P q)\ n = p\ n \ominus q\ n$
using *P.minus-closed P-def coeff-minus coeff-simp* **by** *auto*

lemma *cfs-monom-mult-r*:
assumes $p \in carrier\ P$
assumes $a \in carrier\ R$
shows $(monom\ P\ a\ n \otimes_P p)\ (k + n) = a \otimes p\ k$
using *coeff-monom-mult assms P.m-closed P-def coeff-simp monom-closed* **by** *auto*

lemma(**in** *UP-cring*) *cfs-monom-mult-l*:
assumes $p \in carrier\ P$
assumes $a \in carrier\ R$
shows $(p \otimes_P monom\ P\ a\ n)\ (k + n) = a \otimes p\ k$
using *UP-m-comm assms*(*1*) *assms*(*2*) *cfs-monom-mult-r* **by** *auto*

lemma(**in** *UP-cring*) *cfs-monom-mult-l'*:
assumes $f \in carrier\ P$
assumes $a \in carrier\ R$
assumes $m \geq n$
shows $(f \otimes_P (monom\ P\ a\ n))\ m = a \otimes (f\ (m - n))$
using *cfs-monom-mult-l*[*of f a n m*−*n*] *assms*
by *simp*

lemma(**in** *UP-cring*) *cfs-monom-mult-r'*:
assumes $f \in$ *carrier* P
assumes $a \in$ *carrier* R
assumes $m \geq n$
shows $((monom\ P\ a\ n) \otimes_P f)\ m = a \otimes (f\ (m - n))$
using *cfs-monom-mult-r*[*of f a n m*−*n*] *assms*
by *simp*
end

3.2 Degree Bound Lemmas

context *UP-ring*
begin

lemma *bound-deg-sum*:
assumes $f \in$ *carrier* P
assumes $g \in$ *carrier* P
assumes *degree* $f \leq n$
assumes *degree* $g \leq n$
shows *degree* $(f \oplus_P g) \leq n$
using *P-def UP-ring-axioms assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*)
by (*meson deg-add max.boundedI order-trans*)

lemma *bound-deg-sum'*:
assumes $f \in$ *carrier* P
assumes $g \in$ *carrier* P
assumes *degree* $f < n$
assumes *degree* $g < n$
shows *degree* $(f \oplus_P g) < n$
using *P-def UP-ring-axioms assms*(*1*) *assms*(*2*)
assms(*3*) *assms*(*4*)
by (*metis bound-deg-sum le-neq-implies-less less-imp-le-nat not-less*)

lemma *equal-deg-sum*:
assumes $f \in$ *carrier* P
assumes $g \in$ *carrier* P
assumes *degree* $f < n$
assumes *degree* $g = n$
shows *degree* $(f \oplus_P g) = n$
proof−
have *0*: *degree* $(f \oplus_P g) \leq n$
using *assms bound-deg-sum*
P-def UP-ring-axioms **by** *auto*
show *degree* $(f \oplus_P g) = n$
proof(*rule ccontr*)
assume *degree* $(f \oplus_P g) \neq n$
then have *1*: *degree* $(f \oplus_P g) < n$
using *0* **by** *auto*

```
    have 2: degree (⊖_P f) < n
      using assms by simp
    have 3: g = (f ⊕_P g) ⊕_P (⊖_P f)
      using assms
      by (simp add: P.add.m-comm P.r-neg1)
    then show False using 1 2 3 assms
      by (metis UP-a-closed UP-a-inv-closed deg-add leD le-max-iff-disj)
  qed
qed

lemma equal-deg-sum':
  assumes f ∈ carrier P
  assumes g ∈ carrier P
  assumes degree g < n
  assumes degree f = n
  shows degree (f ⊕_P g) = n
 using P-def UP-a-comm UP-ring.equal-deg-sum UP-ring-axioms assms(1) assms(2)
assms(3) assms(4)
  by fastforce

lemma degree-of-sum-diff-degree:
  assumes p ∈ carrier P
  assumes q ∈ carrier P
  assumes degree q < degree p
  shows degree (p ⊕_P q) = degree p
  by(rule equal-deg-sum', auto simp: assms)

lemma degree-of-difference-diff-degree:
  assumes p ∈ carrier P
  assumes q ∈ carrier P
  assumes degree q < degree p
  shows degree (p ⊖_P q) = degree p
proof−
  have A: (p ⊖_P q) = p ⊕_P (⊖_P q)
    by (simp add: P.minus-eq)
  have degree (⊖_P q) = degree q
    by (simp add: assms(2))
  then show ?thesis
    using assms A
    by (simp add: degree-of-sum-diff-degree)
qed

lemma (in UP-ring) deg-diff-by-const:
  assumes g ∈ carrier (UP R)
  assumes a ∈ carrier R
  assumes h = g ⊕_UP R up-ring.monom (UP R) a 0
  shows deg R g = deg R h
  unfolding assms using assms
 by (metis P-def UP-ring.bound-deg-sum UP-ring.deg-monom-le UP-ring.monom-closed
```

UP-ring-axioms degree-of-sum-diff-degree gr-zeroI not-less)

lemma (**in** *UP-ring*) *deg-diff-by-const'*:
assumes $g \in carrier\ (UP\ R)$
assumes $a \in carrier\ R$
assumes $h = g \ominus_{UP\ R}\ up\text{-}ring.monom\ (UP\ R)\ a\ 0$
shows $deg\ R\ g = deg\ R\ h$
apply(*rule deg-diff-by-const*[*of* - $\ominus$ *a*])
using *assms* **apply** *blast*
using *assms* **apply** *blast*
by (*metis P.minus-eq P-def assms*(*2*) *assms*(*3*) *monom-a-inv*)

lemma(**in** *UP-ring*) *deg-gtE*:
assumes $p \in carrier\ P$
assumes $i > deg\ R\ p$
shows $p\ i = \mathbf{0}$
using *assms P-def coeff-simp deg-aboveD* **by** *metis*
end

3.3 Leading Term Function

definition *leading-term* **where**
$leading\text{-}term\ R\ f = monom\ (UP\ R)\ (f\ (deg\ R\ f))\ (deg\ R\ f)$

context *UP-ring*
begin

abbreviation(*input*) *ltrm* **where**
$ltrm\ f \equiv monom\ P\ (f\ (deg\ R\ f))\ (deg\ R\ f)$

leading term is a polynomial

lemma *ltrm-closed*:
assumes $f \in carrier\ P$
shows $ltrm\ f \in carrier\ P$
using *assms*
by (*simp add*: *cfs-closed*)

Simplified coefficient function description for leading term

lemma *ltrm-coeff*:
assumes $f \in carrier\ P$
shows $coeff\ P\ (ltrm\ f)\ n\ = (if\ (n = degree\ f)\ then\ (f\ (degree\ f))\ else\ \mathbf{0})$
using *assms*
by (*simp add*: *cfs-closed*)

lemma *ltrm-cfs*:
assumes $f \in carrier\ P$
shows $(ltrm\ f)\ n\ = (if\ (n = degree\ f)\ then\ (f\ (degree\ f))\ else\ \mathbf{0})$
using *assms*
by (*simp add*: *cfs-closed cfs-monom*)

lemma *ltrm-cfs-above-deg*:
assumes $f \in$ *carrier* P
assumes $n >$ *degree* f
shows *ltrm* $f\ n = \mathbf{0}$
using *assms*
by (*simp add*: *ltrm-cfs*)

The leading term of f has the same degree as f

lemma *deg-ltrm*:
assumes $f \in$ *carrier* P
shows *degree* (*ltrm* f) = *degree* f
using *assms*
by (*metis P-def UP-ring.lcoeff-nonzero-deg UP-ring-axioms cfs-closed coeff-simp deg-const deg-monom*)

Subtracting the leading term yields a drop in degree

lemma *minus-ltrm-degree-drop*:
assumes $f \in$ *carrier* P
assumes *degree* f = *Suc* n
shows *degree* ($f \ominus_P$ (*ltrm* f)) $\leq n$
proof(*rule UP-ring.deg-aboveI*)
show *C0*: *UP-ring* R
by (*simp add*: *UP-ring-axioms*)
show *C1*: $f \ominus_P$ *ltrm* $f \in$ *carrier* (*UP* R)
using *assms ltrm-closed P.minus-closed P-def*
by *blast*
show *C2*: $\bigwedge m.\ n < m \Longrightarrow$ *coeff* (*UP* R) ($f \ominus_P$ *ltrm* f) $m = \mathbf{0}$
proof−
fix m
assume *A*: $n<m$
show *coeff* (*UP* R) ($f \ominus_P$ *ltrm* f) $m = \mathbf{0}$
proof(*cases* m = *Suc* n)
case *True*
have *B*: $f\ m \in$ *carrier* R
using *UP.coeff-closed P-def assms*(*1*) *cfs-closed* **by** *blast*
have m = *degree* f
using *True* **by** (*simp add*: *assms*(*2*))
then have $f\ m$ = (*ltrm* f) m
using *ltrm-cfs assms*(*1*) **by** *auto*
then have ($f\ m$) $\ominus_R$(*ltrm* f) $m = \mathbf{0}$
using *B UP-ring-def P-is-UP-ring*
B R.add.r-inv R.is-abelian-group abelian-group.minus-eq **by** *fastforce*
then have ($f \ominus_{UP\ R}$ *ltrm* f) $m = \mathbf{0}$
by (*metis C1 ltrm-closed P-def assms*(*1*) *coeff-minus coeff-simp*)
then show *?thesis*
using *C1 P-def UP-ring.coeff-simp UP-ring-axioms* **by** *fastforce*
next
case *False*

have *D0*: $m >$ *degree f* **using** *False*
using *A assms(2)* **by** *linarith*
have *B*: *f m* $\in$ *carrier R*
using *UP.coeff-closed P-def assms(1) cfs-closed*
by *blast*
have *f m* = (*ltrm f*) *m*
using *D0 ltrm-cfs-above-deg P-def assms(1) coeff-simp deg-aboveD*
by *auto*
then show *?thesis*
by (*metis B ltrm-closed P-def R.r-neg UP-ring.coeff-simp UP-ring-axioms a-minus-def assms(1) coeff-minus*)
qed
qed
qed

lemma *ltrm-decomp*:
assumes *f* $\in$ *carrier P*
assumes *degree f* >(*0*::*nat*)
obtains *g* **where** *g* $\in$ *carrier P* $\wedge$ *f* = *g* $\oplus_P$ (*ltrm f*) $\wedge$ *degree g* < *degree f*
proof−
have *0*: *f* $\ominus_P$ (*ltrm f*) $\in$ *carrier P*
using *ltrm-closed assms(1)* **by** *blast*
have *1*: *f* = (*f* $\ominus_P$ (*ltrm f*)) $\oplus_P$ (*ltrm f*)
using *assms*
by (*metis 0 ltrm-closed P.add.inv-solve-right P.minus-eq*)
show *?thesis* **using** *assms 0 1 minus-ltrm-degree-drop*[*of f*]
by (*metis ltrm-closed Suc-diff-1 Suc-n-not-le-n deg-ltrm equal-deg-sum′ linorder-neqE-nat that*)
qed

leading term of a sum

lemma *coeff-of-sum-diff-degree0*:
assumes *p* $\in$ *carrier P*
assumes *q* $\in$ *carrier P*
assumes *degree q* < *n*
shows (*p* $\oplus_P$ *q*) *n* = *p n*
using *assms P-def UP-ring.deg-aboveD UP-ring-axioms cfs-add coeff-simp cfs-closed deg-aboveD*
by *auto*

lemma *coeff-of-sum-diff-degree1*:
assumes *p* $\in$ *carrier P*
assumes *q* $\in$ *carrier P*
assumes *degree q* < *degree p*
shows (*p* $\oplus_P$ *q*) (*degree p*) = *p* (*degree p*)
using *assms(1) assms(2) assms(3) coeff-of-sum-diff-degree0* **by** *blast*

lemma *ltrm-of-sum-diff-degree*:
assumes $p \in$ *carrier* P
assumes $q \in$ *carrier* P
assumes *degree* $p >$ *degree* q
shows *ltrm* $(p \oplus_P q) =$ *ltrm* p
unfolding *leading-term-def*
using *assms*(*1*) *assms*(*2*) *assms*(*3*) *coeff-of-sum-diff-degree1 degree-of-sum-diff-degree*

by *presburger*

leading term of a monomial

lemma *ltrm-monom*:
assumes $a \in$ *carrier* R
assumes $f =$ *monom* P a n
shows *ltrm* $f = f$
unfolding *leading-term-def*
by (*metis P-def UP-ring.cfs-monom UP-ring.monom-zero UP-ring-axioms assms*(*1*) *assms*(*2*) *deg-monom*)

lemma *ltrm-monom-simp*:
assumes $a \in$ *carrier* R
shows *ltrm* (*monom* P a n) = *monom* P a n
using *assms ltrm-monom* **by** *auto*

lemma *ltrm-inv-simp*[*simp*]:
assumes $f \in$ *carrier* P
shows *ltrm* (*ltrm* f) = *ltrm* f
by (*metis assms deg-ltrm ltrm-cfs*)

lemma *ltrm-deg-0*:
assumes $p \in$ *carrier* P
assumes *degree* $p = 0$
shows *ltrm* $p = p$
using *ltrm-monom assms P-def UP-ring.deg-zero-impl-monom UP-ring-axioms coeff-simp*
by *fastforce*

lemma *ltrm-prod-ltrm*:
assumes $p \in$ *carrier* P
assumes $q \in$ *carrier* P
shows *ltrm* ((*ltrm* p) $\otimes_P$ (*ltrm* q)) = (*ltrm* p) $\otimes_P$ (*ltrm* q)
using *ltrm-monom R.m-closed assms*(*1*) *assms*(*2*) *cfs-closed monom-mult*
by *metis*

lead coefficient function

abbreviation(*input*) *lcf* **where**
lcf $p \equiv p$ (*deg* R p)

lemma(**in** *UP-ring*) *lcf-ltrm*:

ltrm p = monom P (lcf p) (degree p)
by *auto*

lemma *lcf-closed*:
assumes *f ∈ carrier P*
shows *lcf f ∈ carrier R*
by (*simp add*: *assms cfs-closed*)

lemma(**in** *UP-cring*) *lcf-monom*:
assumes *a ∈ carrier R*
shows *lcf (monom P a n) = a lcf (monom (UP R) a n) = a*
using *assms deg-monom cfs-monom* **apply** *fastforce*
by (*metis UP-ring.cfs-monom UP-ring.deg-monom UP-ring-axioms assms*)

end

Function which truncates a polynomial by removing the leading term

definition *truncate* **where**
truncate R f = f $\ominus_{(UP\ R)}$ (*leading-term R f*)

context *UP-ring*
begin

abbreviation(*input*) *trunc* **where**
trunc ≡ truncate R

lemma *trunc-closed*:
assumes *f ∈ carrier P*
shows *trunc f ∈ carrier P*
using *assms* **unfolding** *truncate-def*
by (*metis ltrm-closed P-def UP-ring.UP-ring UP-ring-axioms leading-term-def ring.ring-simprules*(*4*))

lemma *trunc-simps*:
assumes *f ∈ carrier P*
shows *f = (trunc f)* $\oplus_P$ *(ltrm f)*
f $\ominus_P$ *(trunc f) = ltrm f*
apply (*metis ltrm-closed P.add.inv-solve-right P.minus-closed P-def a-minus-def assms Cring-Poly.truncate-def leading-term-def*)
using *trunc-closed*[*of f*] *ltrm-closed*[*of f*] *P-def P.add.inv-solve-right*[*of ltrm f f trunc f*]
assms **unfolding** *UP-cring-def*
by (*metis P.add.inv-closed P.add.m-lcomm P.add.r-inv-ex P.minus-eq P.minus-minus P.r-neg2 P.r-zero Cring-Poly.truncate-def leading-term-def*)

lemma *trunc-zero*:
assumes *f ∈ carrier P*
assumes *degree f = 0*

shows *trunc f* = $\mathbf{0}_P$
unfolding *truncate-def*
using *assms ltrm-deg-0*[*of f*]
by (*metis P.r-neg P-def a-minus-def leading-term-def*)

lemma *trunc-degree*:
assumes *f* ∈ *carrier P*
assumes *degree f* > *0*
shows *degree* (*trunc f*) < *degree f*
unfolding *truncate-def* **using** *assms*
by (*metis ltrm-closed ltrm-decomp P.add.right-cancel Cring-Poly.truncate-def trunc-closed trunc-simps*(*1*))

The coefficients of trunc agree with f for small degree

lemma *trunc-cfs*:
assumes *p* ∈ *carrier P*
assumes *n* < *degree p*
shows (*trunc p*) *n* = *p n*
using *P-def assms*(*1*) *assms*(*2*) **unfolding** *truncate-def*
by (*smt* (*verit*) *ltrm-closed ltrm-cfs R.minus-zero R.ring-axioms UP-ring.cfs-minus UP-ring-axioms a-minus-def cfs-closed leading-term-def nat-neq-iff ring.ring-simprules*(*15*))

monomial predicate

definition *is-UP-monom* **where**
is-UP-monom = (λ*f*. *f* ∈ *carrier* (*UP R*) ∧ *f* = *ltrm f*)

lemma *is-UP-monomI*:
assumes *a* ∈ *carrier R*
assumes *p* = *monom P a n*
shows *is-UP-monom p*
using *assms*(*1*) *assms*(*2*) *is-UP-monom-def ltrm-monom P-def monom-closed*
by *auto*

lemma *is-UP-monomI'*:
assumes *f* ∈ *carrier* (*UP R*)
assumes *f* = *ltrm f*
shows *is-UP-monom f*
using *assms P-def* **unfolding** *is-UP-monom-def* **by** *blast*

lemma *monom-is-UP-monom*:
assumes *a* ∈ *carrier R*
shows *is-UP-monom* (*monom P a n*) *is-UP-monom* (*monom* (*UP R*) *a n*)
using *assms P-def ltrm-monom-simp monom-closed*
unfolding *is-UP-monom-def*
by *auto*

lemma *is-UP-monomE*:
assumes *is-UP-monom f*
shows *f* ∈ *carrier P f* = *monom P* (*lcf f*) (*degree f*) *f* = *monom* (*UP R*) (*lcf*

f) (*degree f*)
using *assms* **unfolding** *is-UP-monom-def*
by(*auto simp*: *P-def*)

lemma *ltrm-is-UP-monom*:
assumes $p \in$ *carrier P*
shows *is-UP-monom* (*ltrm p*)
using *assms*
by (*simp add*: *cfs-closed monom-is-UP-monom*(*1*))

lemma *is-UP-monom-mult*:
assumes *is-UP-monom p*
assumes *is-UP-monom q*
shows *is-UP-monom* ($p \otimes_P q$)
apply(*rule is-UP-monomI'*)
using *assms is-UP-monomE P-def UP-mult-closed*
apply *simp*
using *assms is-UP-monomE*[*of p*] *is-UP-monomE*[*of q*]
P-def monom-mult
by (*metis lcf-closed ltrm-monom R.m-closed*)
end

3.4 Properties of Leading Terms and Leading Coefficients in Commutative Rings and Domains

context *UP-cring*
begin

lemma *cring-deg-mult*:
assumes $q \in$ *carrier P*
assumes $p \in$ *carrier P*
assumes *lcf q* $\otimes$ *lcf p* $\neq \mathbf{0}$
shows *degree* ($q \otimes_P p$) = *degree p* + *degree q*
proof−
have $q \otimes_P p$ = (*trunc q* $\oplus_P$ *ltrm q*) $\otimes_P$ (*trunc p* $\oplus_P$ *ltrm p*)
using *assms*(*1*) *assms*(*2*) *trunc-simps*(*1*) **by** *auto*
then have $q \otimes_P p$ = (*trunc q* $\oplus_P$ *ltrm q*) $\otimes_P$ (*trunc p* $\oplus_P$ *ltrm p*)
by *linarith*
then have *0*: $q \otimes_P p$ = (*trunc q* $\otimes_P$ (*trunc p* $\oplus_P$ *ltrm p*)) $\oplus_P$ (*ltrm q* $\otimes_P$ (*trunc p* $\oplus_P$ *ltrm p*))
by (*simp add*: *P.l-distr assms*(*1*) *assms*(*2*) *ltrm-closed trunc-closed*)
have *1*: (*trunc q* $\otimes_P$ (*trunc p* $\oplus_P$ *ltrm p*)) (*degree p* + *degree q*) = $\mathbf{0}$
proof(*cases degree q* = *0*)
case *True*
then show *?thesis*
using *assms*(*1*) *assms*(*2*) *trunc-simps*(*1*) *trunc-zero* **by** *auto*
next
case *False*
have *degree* ((*trunc q*) $\otimes_P$ *p*) $\leq$ *degree* (*trunc q*) + *degree p*

using *assms trunc-simps*[*of q*] *deg-mult-ring*[*of trunc q p*] *trunc-closed*
by *blast*
then have *degree* (*trunc q* $\otimes_P$ (*trunc p* $\oplus_P$ *ltrm p*)) < *degree q* + *degree p*
using *False assms*(*1*) *assms*(*2*) *trunc-degree trunc-simps*(*1*) **by** *fastforce*
then show *?thesis*
by (*metis P-def UP-mult-closed UP-ring.coeff-simp UP-ring-axioms*
add.commute assms(*1*) *assms*(*2*) *deg-belowI not-less trunc-closed trunc-simps*(*1*))

qed
have *2*: (*q* $\otimes_P$ *p*) (*degree p* + *degree q*) =
(*ltrm q* $\otimes_P$ (*trunc p* $\oplus_P$ *ltrm p*)) (*degree p* + *degree q*)
using *0 1 assms cfs-closed trunc-closed* **by** *auto*
have *3*: (*q* $\otimes_P$ *p*) (*degree p* + *degree q*) =
(*ltrm q* $\otimes_P$ *trunc p*) (*degree p* + *degree q*) $\oplus$ (*ltrm q* $\otimes_P$ *ltrm p*)
(*degree p* + *degree q*)
by (*simp add*: *2 ltrm-closed UP-r-distr assms*(*1*) *assms*(*2*) *trunc-closed*)
have *4*: (*ltrm q* $\otimes_P$ *trunc p*) (*degree p* + *degree q*) = **0**
proof(*cases degree p* = *0*)
case *True*
then show *?thesis*
using *2 3 assms*(*1*) *assms*(*2*) *cfs-closed ltrm-closed trunc-zero* **by** *auto*
next
case *False*
have *degree* (*ltrm q* $\otimes_P$ *trunc p*) $\leq$ *degree* (*ltrm q*) + *degree* (*trunc p*)
using *assms trunc-simps deg-mult-ring ltrm-closed trunc-closed* **by** *presburger*

then have *degree* (*ltrm q* $\otimes_P$ *trunc p*) < *degree q* + *degree p*
using *False assms*(*1*) *assms*(*2*) *trunc-degree trunc-simps*(*1*) *deg-ltrm* **by**
fastforce
then show *?thesis*
by (*metis ltrm-closed P-def UP-mult-closed UP-ring.coeff-simp UP-ring-axioms*
add.commute assms(*1*) *assms*(*2*) *deg-belowI not-less trunc-closed*)
qed
have *5*: (*q* $\otimes_P$ *p*) (*degree p* + *degree q*) = (*ltrm q* $\otimes_P$ *ltrm p*) (*degree p* + *degree*
q)
by (*simp add*: *3 4 assms*(*1*) *assms*(*2*) *cfs-closed*)
have *6*: *ltrm q* $\otimes_P$ *ltrm p* = *monom P* (*lcf q* $\otimes$ *lcf p*) (*degree p* + *degree q*)
unfolding *leading-term-def*
by (*metis P-def UP-ring.monom-mult UP-ring-axioms add.commute assms*(*1*)
assms(*2*) *cfs-closed*)
have *7*: (*ltrm q* $\otimes_P$ *ltrm p*) (*degree p* + *degree q*) $\neq$**0**
using *5 6 assms*
by (*metis R.m-closed cfs-closed cfs-monom*)
have *8*: *degree* (*q* $\otimes_P$ *p*) $\geq$*degree p* + *degree q*
using *5 6 7 P-def UP-mult-closed assms*(*1*) *assms*(*2*)
by (*simp add*: *UP-ring.coeff-simp UP-ring-axioms deg-belowI*)
then show *?thesis*
using *assms*(*1*) *assms*(*2*) *deg-mult-ring* **by** *fastforce*
qed

leading term is multiplicative

lemma *ltrm-of-sum-diff-deg*:
assumes $q \in$ *carrier* P
assumes $a \in$ *carrier* R
assumes $a \neq \mathbf{0}$
assumes *degree* $q < n$
assumes $p = q \oplus_P$ (*monom* P a n)
shows *ltrm* $p =$ (*monom* P a n)
proof−
have *0*: *degree* (*monom* P a n) $= n$
by (*simp add*: *assms*(*2*) *assms*(*3*))
have *1*: (*monom* P a n) $\in$ *carrier* P
using *assms*(*2*) **by** *auto*
have *2*: *ltrm* ((*monom* P a n) $\oplus_P$ q) = *ltrm* (*monom* P a n)
using *assms ltrm-of-sum-diff-degree*[*of* (*monom* P a n) q] *1* *0* **by** *linarith*
then show *?thesis*
using *UP-a-comm assms*(*1*) *assms*(*2*) *assms*(*5*) *ltrm-monom* **by** *auto*
qed

lemma(**in** *UP-cring*) *ltrm-smult-cring*:
assumes $p \in$ *carrier* P
assumes $a \in$ *carrier* R
assumes *lcf* $p \otimes a \neq \mathbf{0}$
shows *ltrm* ($a \odot_P p$) = $a \odot_P$(*ltrm* p)
using *assms*
by (*smt* (*verit*) *lcf-monom*(*1*) *P-def R.m-closed R.m-comm cfs-closed cfs-smult coeff-simp*
cring-deg-mult deg-monom deg-ltrm monom-closed monom-mult-is-smult monom-mult-smult)

lemma(**in** *UP-cring*) *deg-zero-ltrm-smult-cring*:
assumes $p \in$ *carrier* P
assumes $a \in$ *carrier* R
assumes *degree* $p = 0$
shows *ltrm* ($a \odot_P p$) = $a \odot_P$(*ltrm* p)
by (*metis ltrm-deg-0 assms*(*1*) *assms*(*2*) *assms*(*3*) *deg-smult-decr le-0-eq module.smult-closed module-axioms*)

lemma(**in** *UP-domain*) *ltrm-smult*:
assumes $p \in$ *carrier* P
assumes $a \in$ *carrier* R
shows *ltrm* ($a \odot_P p$) = $a \odot_P$(*ltrm* p)
by (*metis lcf-closed ltrm-closed ltrm-smult-cring P-def R.integral-iff UP-ring.deg-ltrm*

UP-ring-axioms UP-smult-zero assms(*1*) *assms*(*2*) *cfs-zero deg-nzero-nzero deg-zero-ltrm-smult-cring monom-zero*)

lemma(**in** *UP-cring*) *cring-ltrm-mult*:
assumes $p \in$ *carrier* P

assumes $q \in carrier\ P$
assumes $lcf\ p \otimes lcf\ q \neq \mathbf{0}$
shows $ltrm\ (p \otimes_P q) = (ltrm\ p) \otimes_P (ltrm\ q)$
proof(*cases* $degree\ p = 0 \vee degree\ q = 0$)
case *True*
then show *?thesis*
by (*smt* (*verit*) *ltrm-closed ltrm-deg-0 ltrm-smult-cring R.m-comm UP-m-comm assms*(1) *assms*(2) *assms*(3) *cfs-closed monom-mult-is-smult*)
next
case *False*
obtain *q0* **where** *q0-def*: $q0 = trunc\ q$
by *simp*
obtain *p0* **where** *p0-def*: $p0 = trunc\ p$
by *simp*
have *Pq*: $degree\ q0 < degree\ q$
using *False P-def assms*(2) *q0-def trunc-degree* **by** *blast*
have *Pp*: $degree\ p0 < degree\ p$
using *False P-def assms*(1) *p0-def trunc-degree* **by** *blast*
have $p \otimes_P q = (p0 \oplus_P ltrm(p)) \otimes_P (q0 \oplus_P ltrm(q))$
using *assms*(1) *assms*(2) *p0-def q0-def trunc-simps*(1) **by** *auto*
then have *P0*: $p \otimes_P q = ((p0 \oplus_P ltrm(p)) \otimes_P q0) \oplus_P ((p0 \oplus_P ltrm(p)) \otimes_P ltrm(q))$
by (*simp add*: *P.r-distr assms*(1) *assms*(2) *ltrm-closed p0-def q0-def trunc-closed*)
have *P1*: $degree\ ((p0 \oplus_P ltrm(p)) \otimes_P q0) < degree\ ((p0 \oplus_P ltrm(p)) \otimes_P ltrm(q))$
proof−
have *LHS*: $degree\ ((p0 \oplus_P ltrm(p)) \otimes_P q0) \leq degree\ p + degree\ q0$
proof(*cases* $q0 = \mathbf{0}_P$)
case *True*
then show *?thesis*
using *assms*(1) *p0-def trunc-simps*(1) **by** *auto*
next
case *False*
then show *?thesis*
using *assms*(1) *assms*(2) *deg-mult-ring p0-def q0-def trunc-simps*(1) *trunc-closed* **by** *auto*
qed
have *RHS*: $degree\ ((p0 \oplus_P ltrm(p)) \otimes_P ltrm(q)) = degree\ p + degree\ q$
using *assms*(1) *assms*(2) *deg-mult-ring ltrm-closed p0-def trunc-simps*(1)
by (*smt* (*verit*) *P-def UP-cring.lcf-monom*(1) *UP-cring.cring-deg-mult UP-cring-axioms add.commute assms*(3) *cfs-closed deg-ltrm*)
then show *?thesis*
using *RHS LHS Pq*
by *linarith*
qed
then have *P2*: $ltrm\ (p \otimes_P q) = ltrm\ ((p0 \oplus_P ltrm(p)) \otimes_P ltrm(q))$
using *P0 P1*
by (*metis* (*no-types*, *lifting*) *ltrm-closed ltrm-of-sum-diff-degree P.add.m-comm UP-mult-closed assms*(1) *assms*(2) *p0-def q0-def trunc-closed trunc-simps*(1))

```
    have P3:  ltrm ((p0 ⊕_P ltrm(p))⊗_P ltrm(q)) = ltrm p ⊗_P ltrm q
    proof−
    have Q0: ((p0 ⊕_P ltrm(p))⊗_P ltrm(q)) = (p0 ⊗_P ltrm(q)) ⊕_P (ltrm(p))⊗_P ltrm(q)
       by (simp add: P.l-distr assms(1) assms(2) ltrm-closed p0-def trunc-closed)
     have Q1: degree ((p0 ⊗_P ltrm(q)) ) < degree ((ltrm(p))⊗_P ltrm(q))
     proof(cases p0 = 0_P)
       case True
       then show ?thesis
         using P1 assms(1) assms(2)  ltrm-closed by auto
     next
       case F: False
       then show ?thesis
         proof−
           have LHS: degree ((p0 ⊗_P ltrm(q))) < degree p + degree q
             using False F  Pp assms(1) assms(2) deg-nzero-nzero
                deg-ltrm ltrm-closed p0-def trunc-closed
             by (smt (verit) add-le-cancel-right deg-mult-ring le-trans not-less)
           have RHS: degree ((ltrm(p))⊗_P ltrm(q)) = degree p + degree q
             using cring-deg-mult[of ltrm p ltrm q] assms
             by (simp add: ltrm-closed ltrm-cfs deg-ltrm)
           then show ?thesis using LHS RHS by auto
       qed
     qed
     have Q2: ltrm ((p0 ⊕_P ltrm(p))⊗_P ltrm(q)) = ltrm ((ltrm(p))⊗_P ltrm(q))
       using Q0 Q1
     by (metis (no-types, lifting) ltrm-closed ltrm-of-sum-diff-degree P.add.m-comm

          UP-mult-closed assms(1) assms(2) p0-def trunc-closed)
     show ?thesis using ltrm-prod-ltrm Q0 Q1 Q2
       by (simp add: assms(1) assms(2))
   qed
   then show ?thesis
     by (simp add: P2)
qed

lemma(in UP-domain) ltrm-mult:
  assumes p ∈ carrier P
  assumes q ∈ carrier P
  shows ltrm (p ⊗_P q) = (ltrm p) ⊗_P (ltrm q)
  using cring-ltrm-mult assms
   by (smt (verit) ltrm-closed ltrm-deg-0 cfs-closed deg-nzero-nzero deg-ltrm lo-
cal.integral-iff monom-mult monom-zero)

lemma lcf-deg-0:
  assumes degree p = 0
  assumes p ∈ carrier P
  assumes q ∈ carrier P
  shows (p ⊗_P q) = (lcf p)⊙_P q
  using P-def assms(1) assms(2) assms(3)
```

by (*metis ltrm-deg-0 cfs-closed monom-mult-is-smult*)

leading term powers

lemma (**in** *domain*) *nonzero-pow-nonzero*:
assumes $a \in$ *carrier* R
assumes $a \neq \mathbf{0}$
shows $a[\hat{}](n::nat) \neq \mathbf{0}$
proof(*induction n*)
case *0*
then show *?case*
by *auto*
next
case (*Suc n*)
fix $n::nat$
assume *IH*: $a[\hat{}]\ n \neq \mathbf{0}$
show $a[\hat{}]\ (Suc\ n) \neq \mathbf{0}$
proof−
have $a[\hat{}]\ (Suc\ n) = a[\hat{}]\ n \otimes a$
by *simp*
then show *?thesis* **using** *assms IH*
using *IH assms*(*1*) *assms*(*2*) *local.integral* **by** *auto*
qed
qed

lemma (**in** *UP-cring*) *cring-monom-degree*:
assumes $a \in$ (*carrier* R)
assumes $p = monom\ P\ a\ m$
assumes $a[\hat{}]n \neq \mathbf{0}$
shows $degree\ (p[\hat{}]_P\ n) = n*m$
by (*simp add*: *assms*(*1*) *assms*(*2*) *assms*(*3*) *monom-pow*)

lemma (**in** *UP-domain*) *monom-degree*:
assumes $a \neq \mathbf{0}$
assumes $a \in$ (*carrier* R)
assumes $p = monom\ P\ a\ m$
shows $degree\ (p[\hat{}]_P\ n) = n*m$
by (*simp add*: *R.domain-axioms assms*(*1*) *assms*(*2*) *assms*(*3*) *domain.nonzero-pow-nonzero monom-pow*)

lemma(**in** *UP-cring*) *cring-pow-ltrm*:
assumes $p \in$ *carrier* P
assumes *lcf* p $[\hat{}]n \neq \mathbf{0}$
shows $ltrm\ (p[\hat{}]_P(n::nat)) = (ltrm\ p)[\hat{}]_P n$
proof−
have *lcf* p $[\hat{}]n \neq \mathbf{0} \Longrightarrow ltrm\ (p[\hat{}]_P(n::nat)) = (ltrm\ p)[\hat{}]_P n$
proof(*induction n*)
case *0*
then show *?case*
using *P.ring-simprules*(*6*) *P.nat-pow-0 cfs-one deg-one monom-one* **by** *pres-*

burger
next
case (*Suc n*) **fix** *n*::*nat*
assume *IH* : (*lcf p* [^] *n* ≠ **0** ⟹ *ltrm* (*p* [^]$_P$ *n*) = *ltrm p* [^]$_P$ *n*)
assume *A*: *lcf p* [^] *Suc n* ≠ **0**
have *a*: *ltrm* (*p* [^]$_P$ *n*) = *ltrm p* [^]$_P$ *n*
apply(*cases lcf p* [^] *n* = **0**)
using *A lcf-closed assms*(*1*) **apply** *auto*[*1*]
by(*rule IH*)
have *0*: *lcf* (*ltrm* (*p* [^]$_P$ *n*)) = *lcf p* [^] *n*
unfolding *a*
by (*simp add*: *lcf-monom*(*1*) *assms*(*1*) *cfs-closed monom-pow*)
then have *1*: *lcf* (*ltrm* (*p* [^]$_P$ *n*)) ⊗ *lcf p* ≠ **0**
using *assms A R.nat-pow-Suc IH* **by** *metis*
then show *ltrm* (*p* [^]$_P$ *Suc n*) = *ltrm p* [^]$_P$ *Suc n*
using *IH 0 assms*(*1*) *cring-ltrm-mult cfs-closed*
by (*smt* (*verit*) *A lcf-monom*(*1*) *ltrm-closed P.nat-pow-Suc2 P.nat-pow-closed R.nat-pow-Suc2 a*)
qed
then show *?thesis*
using *assms*(*2*) **by** *blast*
qed

lemma(**in** *UP-cring*) *cring-pow-deg*:
assumes *p* ∈ *carrier P*
assumes *lcf p* [^]*n* ≠ **0**
shows *degree* (*p*[^]$_P$(*n*::*nat*)) = *n*∗*degree p*
proof−
have *degree* ((*ltrm p*)[^]$_P$*n*) = *n*∗*degree p*
using *assms*(*1*) *assms*(*2*) *cring-monom-degree lcf-closed lcf-ltrm* **by** *auto*
then show *?thesis*
using *assms cring-pow-ltrm*
by (*metis P.nat-pow-closed P-def UP-ring.deg-ltrm UP-ring-axioms*)
qed

lemma(**in** *UP-cring*) *cring-pow-deg-bound*:
assumes *p* ∈ *carrier P*
shows *degree* (*p*[^]$_P$(*n*::*nat*)) ≤ *n*∗*degree p*
apply(*induction n*)
apply (*metis Group.nat-pow-0 deg-one le-zero-eq mult-is-0*)
using *deg-mult-ring*[*of - p*]
by (*smt* (*verit*) *P.nat-pow-Suc2 P.nat-pow-closed ab-semigroup-add-class.add-ac*(*1*) *assms deg-mult-ring le-iff-add mult-Suc*)

lemma(**in** *UP-cring*) *deg-smult*:
assumes *a* ∈ *carrier R*
assumes *f* ∈ *carrier* (*UP R*)
assumes *a* ⊗ *lcf f* ≠ **0**
shows *deg R* (*a* ⊙$_{UP\ R}$ *f*) = *deg R f*

using *assms P-def cfs-smult deg-eqI deg-smult-decr smult-closed*
by (*metis deg-gtE le-neq-implies-less*)

lemma(**in** *UP-cring*) *deg-smult'*:
assumes $a \in$ *Units R*
assumes $f \in$ *carrier* (*UP R*)
shows *deg R* ($a \odot_{UP\ R} f$) = *deg R f*
apply(*cases deg R f = 0*)
apply (*metis P-def R.Units-closed assms*(*1*) *assms*(*2*) *deg-smult-decr le-zero-eq*)
apply(*rule deg-smult*)
using *assms* **apply** *blast*
using *assms* **apply** *blast*
proof
assume *A*: *deg R f* $\neq$ *0* $a \otimes f$ (*deg R f*) = **0**
have *0*: *f* (*deg R f*) = **0**
using *A assms R.Units-not-right-zero-divisor*[*of a f* (*deg R f*)] *UP-car-memE*(*1*)
by *blast*
then show *False* **using** *assms A*
by (*metis P-def deg-zero deg-ltrm monom-zero*)
qed

lemma(**in** *UP-domain*) *pow-sum0*:
$\bigwedge$ *p q*. *p* $\in$ *carrier P* $\Longrightarrow$ *q* $\in$ *carrier P* $\Longrightarrow$ *degree q* < *degree p* $\Longrightarrow$ *degree* ((*p* $\oplus_P$ *q*)[$\hat{}$]$_P$*n*) = (*degree p*)$*$*n*
proof(*induction n*)
case *0*
then show *?case*
by (*metis Group.nat-pow-0 deg-one mult-is-0*)
next
case (*Suc n*)
fix *n*
assume *IH*: $\bigwedge$ *p q*. *p* $\in$ *carrier P* $\Longrightarrow$ *q* $\in$ *carrier P* $\Longrightarrow$
degree q < *degree p* $\Longrightarrow$ *degree* ((*p* $\oplus_P$ *q*)[$\hat{}$]$_P$*n*) = (*degree p*)$*$*n*
then show $\bigwedge$ *p q*. *p* $\in$ *carrier P* $\Longrightarrow$ *q* $\in$ *carrier P* $\Longrightarrow$
degree q < *degree p* $\Longrightarrow$ *degree* ((*p* $\oplus_P$ *q*)[$\hat{}$]$_P$(*Suc n*)) = (*degree p*)$*$(*Suc n*)
proof−
fix *p q*
assume *A0*: *p* $\in$ *carrier P* **and**
A1: *q* $\in$ *carrier P* **and**
A2: *degree q* < *degree p*
show *degree* ((*p* $\oplus_P$ *q*)[$\hat{}$]$_P$(*Suc n*)) = (*degree p*)$*$(*Suc n*)
proof(*cases q* = **0**$_P$)
case *True*
then show *?thesis*
by (*metis A0 A1 A2 IH P.nat-pow-Suc2 P.nat-pow-closed P.r-zero deg-mult*
domain.nonzero-pow-nonzero local.domain-axioms mult-Suc-right nat-neq-iff)
next

```
      case False
      then show ?thesis
      proof−
        have P0: degree ((p ⊕P q )[^]P n) = (degree p)∗n
        using A0 A1 A2 IH by auto
        have P1: (p ⊕P q )[^]P(Suc n) = ((p ⊕P q )[^]P n) ⊗P (p ⊕P q )
          by simp
       then have P2: (p ⊕P q )[^]P(Suc n) = (((p ⊕P q )[^]P n) ⊗P p) ⊕P (((p
⊕P q )[^]P n) ⊗P q)
          by (simp add: A0 A1 UP-r-distr)
        have P3: degree (((p ⊕P q )[^]P n) ⊗P p) = (degree p)∗n + (degree p)
            using P0 A0 A1 A2 deg-nzero-nzero  degree-of-sum-diff-degree lo-
cal.nonzero-pow-nonzero by auto
        have P4: degree (((p ⊕P q )[^]P n) ⊗P q) = (degree p)∗n + (degree q)
            using P0 A0 A1 A2 deg-nzero-nzero  degree-of-sum-diff-degree lo-
cal.nonzero-pow-nonzero False deg-mult
          by simp
       have P5: degree (((p ⊕P q )[^]P n) ⊗P p) > degree (((p ⊕P q )[^]P n) ⊗P
q)
          using P3 P4 A2 by auto
        then show ?thesis using P5 P3 P2
          by (simp add: A0 A1 degree-of-sum-diff-degree)
      qed
    qed
  qed
qed

lemma(in UP-domain)  pow-sum:
  assumes p ∈ carrier P
  assumes q ∈ carrier P
  assumes degree q < degree p
  shows degree ((p ⊕P q )[^]P n) = (degree p)∗n
  using assms(1) assms(2) assms(3) pow-sum0 by blast

lemma(in UP-domain)  deg-pow0:
 ⋀ p. p ∈ carrier P ⟹ n ≥ degree p ⟹ degree (p [^]P m) = m∗(degree p)
proof(induction n)
  case 0
  show p ∈ carrier P ⟹ 0 ≥ degree p ⟹ degree (p [^]P m) = m∗(degree p)
  proof−
    assume B0:p ∈ carrier P
    assume B1: 0 ≥ degree p
    then obtain a where a-def: a ∈ carrier R ∧ p = monom P a 0
      using B0 deg-zero-impl-monom  by fastforce
    show degree (p [^]P m) = m∗(degree p)  using UP-cring.monom-pow
      by (metis P-def R.nat-pow-closed UP-cring-axioms a-def deg-const
        mult-0-right mult-zero-left)
  qed
next
```

```
  case (Suc n)
  fix n
  assume IH: ⋀p. (p ∈ carrier P ⟹ n ≥degree p ⟹ degree (p [^]_P m) = m ∗
(degree p))
  show p ∈ carrier P ⟹ Suc n ≥ degree p ⟹ degree (p [^]_P m) = m ∗ (degree
p)
  proof−
    assume A0: p ∈ carrier P
    assume A1: Suc n ≥ degree p
    show degree (p [^]_P m) = m ∗ (degree p)
    proof(cases Suc n > degree p)
      case True
      then show ?thesis using IH A0 by simp
    next
      case False
      then show ?thesis
      proof−
        obtain q where q-def: q = trunc p
          by simp
        obtain k where k-def: k = degree q
          by simp
        have q-is-poly: q ∈ carrier P
          by (simp add: A0 q-def trunc-closed)
        have k-bound0: k <degree p
          using k-def q-def trunc-degree[of p] A0 False by auto
        have k-bound1: k ≤ n
          using k-bound0 A0 A1 by auto
        have P-q:degree (q [^]_P m) = m ∗ k
          using IH[of q] k-bound1 k-def q-is-poly by auto
        have P-ltrm: degree ((ltrm p) [^]_P m) = m∗(degree p)
        proof−
          have degree p = degree (ltrm p)
            by (simp add: A0 deg-ltrm)
          then show ?thesis using monom-degree
           by (metis A0 P.r-zero P-def cfs-closed coeff-simp equal-deg-sum k-bound0
k-def lcoeff-nonzero2 nat-neq-iff q-is-poly)
        qed
        have p = q ⊕_P (ltrm p)
          by (simp add: A0 q-def trunc-simps(1))
        then show ?thesis
          using P-q pow-sum[of (ltrm p) q m] A0 UP-a-comm
            deg-ltrm k-bound0 k-def ltrm-closed q-is-poly by auto
      qed
    qed
  qed
qed

lemma(in UP-domain) deg-pow:
  assumes p ∈ carrier P
```

shows *degree* $(p\ [\hat{}]_P\ m) = m*(\mathit{degree}\ p)$
using *deg-pow0 assms* **by** *blast*

lemma(**in** *UP-domain*) *ltrm-pow0*:
$\bigwedge f.\ f \in \mathit{carrier}\ P \Longrightarrow \mathit{ltrm}\ (f\ [\hat{}]_P\ (n::\mathit{nat})) = (\mathit{ltrm}\ f)\ [\hat{}]_P\ n$
proof(*induction n*)
case *0*
then show *?case*
using *ltrm-deg-0 P.nat-pow-0 P.ring-simprules*(*6*) *deg-one* **by** *presburger*
next
case (*Suc n*)
fix *n*::*nat*
assume *IH*: $\bigwedge f.\ f \in \mathit{carrier}\ P \Longrightarrow \mathit{ltrm}\ (f\ [\hat{}]_P\ n) = (\mathit{ltrm}\ f)\ [\hat{}]_P\ n$
then show $\bigwedge f.\ f \in \mathit{carrier}\ P \Longrightarrow \mathit{ltrm}\ (f\ [\hat{}]_P\ (\mathit{Suc}\ n)) = (\mathit{ltrm}\ f)\ [\hat{}]_P\ (\mathit{Suc}\ n)$
proof−
fix *f*
assume *A*: $f \in \mathit{carrier}\ P$
show $\mathit{ltrm}\ (f\ [\hat{}]_P\ (\mathit{Suc}\ n)) = (\mathit{ltrm}\ f)\ [\hat{}]_P\ (\mathit{Suc}\ n)$
proof−
have *0*: $\mathit{ltrm}\ (f\ [\hat{}]_P\ n) = (\mathit{ltrm}\ f)\ [\hat{}]_P\ n$
using *A IH* **by** *blast*
have *1*: $\mathit{ltrm}\ (f\ [\hat{}]_P\ (\mathit{Suc}\ n)) = \mathit{ltrm}\ ((f\ [\hat{}]_P\ n) \otimes_P f)$
by *auto* **then**
show *?thesis* **using** *ltrm-mult 0 1*
by (*simp add*: *A*)
qed
qed
qed

lemma(**in** *UP-domain*) *ltrm-pow*:
assumes $f \in \mathit{carrier}\ P$
shows $\mathit{ltrm}\ (f\ [\hat{}]_P\ (n::\mathit{nat})) = (\mathit{ltrm}\ f)\ [\hat{}]_P\ n$
using *assms ltrm-pow0* **by** *blast*

lemma on the leading coefficient

lemma *lcf-eq*:
assumes $f \in \mathit{carrier}\ P$
shows $\mathit{lcf}\ f = \mathit{lcf}\ (\mathit{ltrm}\ f)$
using *ltrm-deg-0*
by (*simp add*: *ltrm-cfs assms deg-ltrm*)

lemma *lcf-eq-deg-eq-imp-ltrm-eq*:
assumes $p \in \mathit{carrier}\ P$
assumes $q \in \mathit{carrier}\ P$
assumes $\mathit{degree}\ p > 0$
assumes $\mathit{degree}\ p = \mathit{degree}\ q$
assumes $\mathit{lcf}\ p = \mathit{lcf}\ q$
shows $\mathit{ltrm}\ p = \mathit{ltrm}\ q$
using *assms*(*4*) *assms*(*5*)

by (*simp add*: *leading-term-def*)

lemma *ltrm-eq-imp-lcf-eq*:
assumes $p \in$ *carrier P*
assumes $q \in$ *carrier P*
assumes *ltrm p* = *ltrm q*
shows *lcf p* = *lcf q*
using *assms*
by (*metis lcf-eq*)

lemma *ltrm-eq-imp-deg-drop*:
assumes $p \in$ *carrier P*
assumes $q \in$ *carrier P*
assumes *ltrm p* = *ltrm q*
assumes *degree p* >*0*
shows *degree* ($p \ominus_P q$) < *degree p*
proof−
have *P0*: *degree p* = *degree q*
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *deg-ltrm*)
then have *P1*: *degree* ($p \ominus_P q$) ≤ *degree p*
by (*metis P.add.inv-solve-right P.minus-closed P.minus-eq assms*(*1*)
assms(*2*) *degree-of-sum-diff-degree neqE order.strict-implies-order order-refl*)
have *degree* ($p \ominus_P q$) ≠ *degree p*
proof
assume *A*: *degree* ($p \ominus_P q$) = *degree p*
have *Q0*: $p \ominus_P q$ = ((*trunc p*) $\oplus_P$ (*ltrm p*)) $\ominus_P$ ((*trunc q*) $\oplus_P$ (*ltrm p*))
using *assms*(*1*) *assms*(*2*) *assms*(*3*) *trunc-simps*(*1*) **by** *force*
have *Q1*: $p \ominus_P q$ = (*trunc p*) $\ominus_P$ (*trunc q*)
proof−
have $p \ominus_P q$ = ((*trunc p*) $\oplus_P$ (*ltrm p*)) $\ominus_P$ (*trunc q*) $\ominus_P$ (*ltrm p*)
using *Q0*
by (*simp add*: *P.minus-add P.minus-eq UP-a-assoc assms*(*1*) *assms*(*2*)
ltrm-closed trunc-closed)
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *ltrm-closed P.add.inv-mult-group P.minus-eq*
P.r-neg2 UP-a-assoc assms(*1*) *assms*(*2*) *assms*(*3*) *carrier-is-submodule*
submoduleE(*6*) *trunc-closed trunc-simps*(*1*))
qed
have *Q2*: *degree* (*trunc p*) < *degree p*
by (*simp add*: *assms*(*1*) *assms*(*4*) *trunc-degree*)
have *Q3*: *degree* (*trunc q*) < *degree q*
using *P0 assms*(*2*) *assms*(*4*) *trunc-degree* **by** *auto*
then show *False* **using** *A Q1 Q2 Q3* **by** (*simp add*: *P.add.inv-solve-right*
P.minus-eq P0 assms(*1*) *assms*(*2*) *degree-of-sum-diff-degree trunc-closed*)
qed
then show *?thesis*
using *P1* **by** *auto*
qed

lemma(**in** *UP-cring*) *cring-lcf-scalar-mult*:
assumes $p \in$ *carrier P*
assumes $a \in$ *carrier R*
assumes $a \otimes (lcf\ p) \neq \mathbf{0}$
shows $lcf\ (a \odot_P p) = a \otimes (lcf\ p)$
proof−
have *0*: $lcf\ (a \odot_P p) = lcf\ (ltrm\ (a \odot_P p))$
using *assms lcf-eq smult-closed* **by** *blast*
have *1*: $degree\ (a \odot_P p) = degree\ p$
by (*smt* (*verit*) *lcf-monom*(*1*) *P-def R.one-closed R.r-null UP-ring.coeff-smult UP-ring-axioms*
assms(*1*) *assms*(*2*) *assms*(*3*) *coeff-simp cring-deg-mult deg-const monom-closed monom-mult-is-smult smult-one*)
then have $lcf\ (a \odot_P p) = lcf\ (a \odot_P (ltrm\ p))$
using *lcf-eq*[*of* $a \odot_P p$] *smult-closed assms 0*
by (*metis cfs-closed cfs-smult monom-mult-smult*)
then show *?thesis*
unfolding *leading-term-def*
by (*metis P-def R.m-closed UP-cring.lcf-monom UP-cring-axioms assms*(*1*) *assms*(*2*) *cfs-closed monom-mult-smult*)
qed

lemma(**in** *UP-domain*) *lcf-scalar-mult*:
assumes $p \in$ *carrier P*
assumes $a \in$ *carrier R*
shows $lcf\ (a \odot_P p) = a \otimes (lcf\ p)$
proof−
have $lcf\ (a \odot_P p) = lcf\ (ltrm\ (a \odot_P p))$
using *lcf-eq UP-smult-closed assms*(*1*) *assms*(*2*) **by** *blast*
then have $lcf\ (a \odot_P p) = lcf\ (a \odot_P (ltrm\ p))$
using *ltrm-smult assms*(*1*) *assms*(*2*) **by** *metis*
then show *?thesis*
by (*metis* (*full-types*) *UP-smult-zero assms*(*1*) *assms*(*2*) *cfs-smult cfs-zero deg-smult*)
qed

lemma(**in** *UP-cring*) *cring-lcf-mult*:
assumes $p \in$ *carrier P*
assumes $q \in$ *carrier P*
assumes $(lcf\ p) \otimes (lcf\ q) \neq \mathbf{0}$
shows $lcf\ (p \otimes_P q) = (lcf\ p) \otimes (lcf\ q)$
using *assms cring-ltrm-mult*
by (*smt* (*verit*) *lcf-monom*(*1*) *P.m-closed R.m-closed cfs-closed monom-mult*)

lemma(**in** *UP-domain*) *lcf-mult*:
assumes $p \in$ *carrier P*
assumes $q \in$ *carrier P*
shows $lcf\ (p \otimes_P q) = (lcf\ p) \otimes (lcf\ q)$
by (*metis P-def R.integral-iff assms*(*1*) *assms*(*2*) *cfs-closed coeff-simp cring-lcf-mult*

lcoeff-nonzero local.integral-iff)

lemma(**in** *UP-cring*) *cring-lcf-pow*:
assumes $p \in$ *carrier P*
assumes $(lcf\ p)[\hat{}]n \neq \mathbf{0}$
shows $lcf\ (p[\hat{}]_P(n::nat)) = (lcf\ p)[\hat{}]n$
by (*smt* (*verit*) *P.nat-pow-closed R.nat-pow-closed assms*(*1*) *assms*(*2*) *cring-pow-ltrm lcf-closed lcf-ltrm lcf-monom monom-pow*)

lemma(**in** *UP-domain*) *lcf-pow*:
assumes $p \in$ *carrier P*
shows $lcf\ (p[\hat{}]_P(n::nat)) = (lcf\ p)[\hat{}]n$
proof−
show *?thesis*
proof(*induction n*)
case *0*
then show *?case*
by (*metis Group.nat-pow-0 P-def R.one-closed UP-cring.lcf-monom UP-cring-axioms monom-one*)
next
case (*Suc n*)
fix *n*
assume *IH*: $lcf\ (p[\hat{}]_P(n::nat)) = (lcf\ p)[\hat{}]n$
show $lcf\ (p[\hat{}]_P(Suc\ n)) = (lcf\ p)[\hat{}](Suc\ n)$
proof−
have $lcf\ (p[\hat{}]_P(Suc\ n)) = lcf\ ((p[\hat{}]_P n) \otimes_P p)$
by *simp*
then have $lcf\ (p[\hat{}]_P(Suc\ n)) = (lcf\ p)[\hat{}]n \otimes (lcf\ p)$
by (*simp add*: *IH assms lcf-mult*)
then show *?thesis* **by** *auto*
qed
qed
qed
end

3.5 Constant Terms and Constant Coefficients

Constant term and coefficient function

definition *zcf* **where**
$zcf\ f = (f\ 0)$

abbreviation(**in** *UP-cring*)(*input*) *ctrm* **where**
$ctrm\ f \equiv monom\ P\ (f\ 0)\ 0$

context *UP-cring*
begin

lemma *ctrm-is-poly*:
assumes $p \in$ *carrier P*

shows *ctrm p* ∈ *carrier P*
by (*simp add*: *assms cfs-closed*)

lemma *ctrm-degree*:
assumes *p* ∈ *carrier P*
shows *degree* (*ctrm p*) = *0*
by (*simp add*: *assms cfs-closed*)

lemma *ctrm-zcf*:
assumes *f* ∈ *carrier P*
assumes *zcf f* = **0**
shows *ctrm f* = $\mathbf{0}_P$
by (*metis P-def UP-ring.monom-zero UP-ring-axioms zcf-def assms*(*2*))

lemma *zcf-degree-zero*:
assumes *f* ∈ *carrier P*
assumes *degree f* = *0*
shows *lcf f* = *zcf f*
by (*simp add*: *zcf-def assms*(*2*))

lemma *zcf-zero-degree-zero*:
assumes *f* ∈ *carrier P*
assumes *degree f* = *0*
assumes *zcf f* = **0**
shows *f* = $\mathbf{0}_P$
using *zcf-degree-zero*[*of f*] *assms ltrm-deg-0*[*of f*]
by *simp*

lemma *zcf-ctrm*:
assumes *p* ∈ *carrier P*
shows *zcf* (*ctrm p*) = *zcf p*
unfolding *zcf-def*
using *P-def UP-ring.cfs-monom UP-ring-axioms assms cfs-closed* **by** *fastforce*

lemma *ctrm-trunc*:
assumes *p* ∈ *carrier P*
assumes *degree p* >*0*
shows *zcf*(*trunc p*) = *zcf p*
by (*simp add*: *zcf-def assms*(*1*) *assms*(*2*) *trunc-cfs*)

Constant coefficient function is a ring homomorphism

lemma *zcf-add*:
assumes *p* ∈ *carrier P*
assumes *q* ∈ *carrier P*
shows *zcf*(*p* ⊕$_P$ *q*) = (*zcf p*) ⊕ (*zcf q*)
by (*simp add*: *zcf-def assms*(*1*) *assms*(*2*))

lemma *coeff-ltrm*[*simp*]:
assumes *p* ∈ *carrier P*

assumes *degree p > 0*
shows *zcf*(*ltrm p*) = **0**
by (*metis ltrm-cfs-above-deg ltrm-cfs zcf-def assms*(*1*) *assms*(*2*))

lemma *zcf-zero*[*simp*]:
zcf $\mathbf{0}_P = \mathbf{0}$
using *zcf-degree-zero* **by** *auto*

lemma *zcf-one*[*simp*]:
zcf $\mathbf{1}_P = \mathbf{1}$
by (*simp add*: *zcf-def*)

lemma *ctrm-smult*:
assumes *f* ∈ *carrier P*
assumes *a* ∈ *carrier R*
shows *ctrm* (*a* $\odot_P$ *f*) = *a* $\odot_P$(*ctrm f*)
using *P-def UP-ring.monom-mult-smult UP-ring-axioms assms*(*1*) *assms*(*2*) *cfs-smult coeff-simp*
by (*simp add*: *UP-ring.monom-mult-smult cfs-closed*)

lemma *ctrm-monom*[*simp*]:
assumes *a* ∈ *carrier R*
shows *ctrm* (*monom P a* (*Suc k*)) = $\mathbf{0}_P$
by (*simp add*: *assms cfs-monom*)
end

3.6 Polynomial Induction Rules

context *UP-ring*
begin

Rule for strong induction on polynomial degree

lemma *poly-induct*:
assumes *p* ∈ *carrier P*
assumes *Deg-0*: ⋀*p*. *p* ∈ *carrier P* ⟹ *degree p* = *0* ⟹ *Q p*
assumes *IH*: ⋀*p*. (⋀*q*. *q* ∈ *carrier P* ⟹ *degree q* < *degree p* ⟹ *Q q*) ⟹ *p* ∈ *carrier P* ⟹ *degree p* > *0* ⟹ *Q p*
shows *Q p*
proof−
have ⋀*n*. ⋀*p*. *p* ∈ *carrier P* ⟹ *degree p* ≤ *n* ⟹ *Q p*
proof−
fix *n*
show ⋀*p*. *p* ∈ *carrier P* ⟹ *degree p* ≤ *n* ⟹ *Q p*
proof(*induction n*)
case *0*
then show *?case*
using *Deg-0* **by** *simp*
next
case (*Suc n*)

```
    fix n
    assume I: ⋀p. p ∈ carrier P ⟹ degree p ≤ n ⟹ Q p
    show ⋀p. p ∈ carrier P ⟹ degree p ≤ (Suc n) ⟹ Q p
    proof−
      fix p
      assume A0: p ∈ carrier P
      assume A1: degree p ≤Suc n
      show Q p
      proof(cases degree p < Suc n)
        case True
        then show ?thesis
          using I A0 by auto
      next
        case False
        then have D: degree p = Suc n
          by (simp add: A1 nat-less-le)
        then have (⋀q. q ∈ carrier P ⟹ degree q < degree p ⟹ Q q)
            using I by simp
          then show Q p
              using IH D A0 A1 Deg-0 by blast
      qed
    qed
  qed
  then show ?thesis using assms by blast
qed
```

Variant on induction on degree

```
lemma poly-induct2:
  assumes p ∈ carrier P
  assumes Deg-0: ⋀p. p ∈ carrier P ⟹ degree p = 0 ⟹ Q p
  assumes IH: ⋀p. degree p > 0  ⟹ p ∈ carrier P ⟹ Q (trunc p)  ⟹ Q p
  shows Q p
proof(rule poly-induct)
  show p ∈ carrier P
    by (simp add: assms(1))
  show ⋀p. p ∈ carrier P ⟹ degree p = 0 ⟹ Q p
      by (simp add: Deg-0)
   show ⋀p. (⋀q. q ∈ carrier P ⟹ degree q < degree p ⟹ Q q) ⟹ p ∈ carrier
P ⟹ 0 < degree p ⟹ Q p
    proof−
      fix p
      assume A0: (⋀q. q ∈ carrier P ⟹ degree q < degree p ⟹ Q q)
      assume A1: p ∈ carrier P
      assume A2: 0 < degree p
      show Q p
      proof−
        have degree (trunc p) < degree p
          by (simp add: A1 A2 trunc-degree)
```

```
      have Q (trunc p)
        by (simp add: A0 A1 ‹degree (trunc p) < degree p› trunc-closed)
      then show ?thesis
        by (simp add: A1 A2 IH)
    qed
  qed
qed
```

Additive properties which are true for all monomials are true for all polynomials

```
lemma poly-induct3:
  assumes p ∈ carrier P
  assumes add: ⋀p q. q ∈ carrier P ⟹ p ∈ carrier P ⟹ Q p ⟹ Q q ⟹ Q
(p ⊕_P q)
  assumes monom: ⋀a n. a ∈ carrier R ⟹ Q (monom P a n)
  shows Q p
  apply(rule poly-induct2)
  apply (simp add: assms(1))
  apply (metis lcf-closed P-def coeff-simp deg-zero-impl-monom monom)
  by (metis lcf-closed ltrm-closed add monom trunc-closed trunc-simps(1))

lemma poly-induct4:
  assumes p ∈ carrier P
  assumes add: ⋀p q. q ∈ carrier P ⟹ p ∈ carrier P ⟹ Q p ⟹ Q q ⟹ Q
(p ⊕_P q)
  assumes monom-zero: ⋀a. a ∈ carrier R ⟹ Q (monom P a 0)
  assumes monom-Suc: ⋀a n. a ∈ carrier R ⟹ Q (monom P a (Suc n))
  shows Q p
  apply(rule poly-induct3)
  using assms(1) apply auto[1]
  using add apply blast
  using monom-zero monom-Suc
  by (metis P-def UP-ring.monom-zero UP-ring-axioms deg-monom deg-monom-le
le-0-eq le-SucE zero-induct)

lemma monic-monom-smult:
  assumes a ∈ carrier R
  shows a ⊙_P monom P 1 n = monom P a n
  using assms
  by (metis R.one-closed R.r-one monom-mult-smult)

lemma poly-induct5:
  assumes p ∈ carrier P
  assumes add: ⋀p q. q ∈ carrier P ⟹ p ∈ carrier P ⟹ Q p ⟹ Q q ⟹ Q
(p ⊕_P q)
  assumes monic-monom: ⋀n. Q (monom P 1 n)
  assumes smult: ⋀p a . a ∈ carrier R ⟹ p ∈ carrier P ⟹ Q p ⟹ Q (a ⊙_P
p)
  shows Q p
```

apply(*rule poly-induct3*)
apply (*simp add*: *assms*(*1*))
using *add* **apply** *blast*
proof−
fix *a n* **assume** *A*: *a* ∈ *carrier R* **show** *Q* (*monom P a n*)
using *monic-monom*[*of n*] *smult*[*of a monom P* **1** *n*] *monom-mult-smult*[*of a* **1** *n*]
by (*simp add*: *A*)
qed

lemma *poly-induct6*:
assumes *p* ∈ *carrier P*
assumes *monom*: ⋀*a n*. *a* ∈ *carrier R* ⟹ *Q* (*monom P a 0*)
assumes *plus-monom*: ⋀*a n p*. *a* ∈ *carrier R* ⟹ *a* ≠ **0** ⟹ *p* ∈ *carrier P* ⟹ *degree p* < *n* ⟹ *Q p* ⟹
Q(*p* ⊕$_P$ *monom P a n*)
shows *Q p*
apply(*rule poly-induct2*)
using *assms*(*1*) **apply** *auto*[*1*]
apply (*metis lcf-closed P-def coeff-simp deg-zero-impl-monom monom*)
using *plus-monom*
by (*metis lcf-closed P-def coeff-simp lcoeff-nonzero-deg nat-less-le trunc-closed trunc-degree trunc-simps*(*1*))

end

4 Mapping a Polynomial to its Associated Ring Function

Turning a polynomial into a function on R:

definition *to-function* **where**
to-function S f = (λ*s* ∈ *carrier S*. *eval S S* (λ*x*. *x*) *s f*)

context *UP-cring*
begin

definition *to-fun* **where**
to-fun f ≡ *to-function R f*

Explicit formula for evaluating a polynomial function:

lemma *to-fun-eval*:
assumes *f* ∈ *carrier P*
assumes *x* ∈ *carrier R*
shows *to-fun f x* = *eval R R* (λ*x*. *x*) *x f*
using *assms* **unfolding** *to-function-def to-fun-def*
by *auto*

lemma *to-fun-formula*:
assumes $f \in$ *carrier* P
assumes $x \in$ *carrier* R
shows *to-fun* $f\ x = (\bigoplus i \in \{..degree\ f\}.\ (f\ i) \otimes x\ [\hat{}]\ i)$
proof−
have $f \in$ *carrier* $(UP\ R)$
using *assms P-def* **by** *auto*
then have *eval* $R\ R\ (\lambda x.\ x)\ x\ f = (\bigoplus_R i \in \{..deg\ R\ f\}.\ (\lambda x.\ x)\ (coeff\ (UP\ R)\ f\ i) \otimes_R x\ [\hat{}]_R\ i)$
apply(*simp add*:*UnivPoly.eval-def*) **done**
then have *to-fun* $f\ x = (\bigoplus_R i \in \{..deg\ R\ f\}.\ (\lambda x.\ x)\ (coeff\ (UP\ R)\ f\ i) \otimes_R x\ [\hat{}]_R\ i)$
using *to-function-def assms* **unfolding** *to-fun-def*
by (*simp add*: *to-function-def*)
then show *?thesis*
by(*simp add*: *assms coeff-simp*)
qed

lemma *eval-ring-hom*:
assumes $a \in$ *carrier* R
shows *eval* $R\ R\ (\lambda x.\ x)\ a \in$ *ring-hom* $P\ R$
proof−
have $(\lambda x.\ x) \in$ *ring-hom* $R\ R$
apply(*rule ring-hom-memI*)
apply *auto* **done**
then have *UP-pre-univ-prop* $R\ R\ (\lambda x.\ x)$
using *R-cring UP-pre-univ-propI* **by** *blast*
then show *?thesis*
by (*simp add*: *P-def UP-pre-univ-prop.eval-ring-hom assms*)
qed

lemma *to-fun-closed*:
assumes $f \in$ *carrier* P
assumes $x \in$ *carrier* R
shows *to-fun* $f\ x\ \in$ *carrier* R
using *assms to-fun-eval*[*of f x*] *eval-ring-hom*[*of x*]
ring-hom-closed
by *fastforce*

lemma *to-fun-plus*:
assumes $g \in$ *carrier* P
assumes $f \in$ *carrier* P
assumes $x \in$ *carrier* R
shows *to-fun* $(f \oplus_P g)\ x =$ (*to-fun* $f\ x$) $\oplus$ (*to-fun* $g\ x$)
using *assms to-fun-eval*[*of*] *eval-ring-hom*[*of x*]
by (*simp add*: *ring-hom-add*)

lemma *to-fun-mult*:

assumes $g \in carrier\ P$
assumes $f \in carrier\ P$
assumes $x \in carrier\ R$
shows $to\text{-}fun\ (f \otimes_P g)\ x = (to\text{-}fun\ f\ x) \otimes (to\text{-}fun\ g\ x)$
using *assms to-fun-eval*[*of*] *eval-ring-hom*[*of x*]
by (*simp add*: *ring-hom-mult*)

lemma *to-fun-ring-hom*:
assumes $a \in carrier\ R$
shows $(\lambda p.\ to\text{-}fun\ p\ a) \in ring\text{-}hom\ P\ R$
apply(*rule ring-hom-memI*)
apply (*simp add*: *assms to-fun-closed*)
apply (*simp add*: *assms to-fun-mult*)
apply (*simp add*: *assms to-fun-plus*)
using *to-fun-eval*[*of* $\mathbf{1}_P$ *a*] *eval-ring-hom*[*of a*]
ring-hom-closed
by (*simp add*: *assms ring-hom-one*)

lemma *ring-hom-uminus*:
assumes *ring S*
assumes $f \in (ring\text{-}hom\ S\ R)$
assumes $a \in carrier\ S$
shows $f\ (\ominus_S\ a) = \ominus\ (f\ a)$
proof−
have $f\ (a \ominus_S a) = (f\ a) \oplus f\ (\ominus_S\ a)$
unfolding *a-minus-def*
by (*simp add*: *assms*(*1*) *assms*(*2*) *assms*(*3*) *ring.ring-simprules*(*3*) *ring-hom-add*)
then have $(f\ a) \oplus f\ (\ominus_S\ a) = \mathbf{0}$
by (*metis R.ring-axioms a-minus-def assms*(*1*) *assms*(*2*) *assms*(*3*)
ring.ring-simprules(*16*) *ring-hom-zero*)
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *R.add.m-comm R.minus-equality assms*(*1*)
assms(*2*) *assms*(*3*) *ring.ring-simprules*(*3*) *ring-hom-closed*)
qed

lemma *to-fun-minus*:
assumes $f \in carrier\ P$
assumes $x \in carrier\ R$
shows $to\text{-}fun\ (\ominus_P f)\ x = \ominus\ (to\text{-}fun\ f\ x)$
unfolding *to-function-def to-fun-def*
using *eval-ring-hom*[*of x*] *assms*
by (*simp add*: *UP-ring ring-hom-uminus*)

lemma *id-is-hom*:
ring-hom-cring R R $(\lambda x.\ x)$
proof(*rule ring-hom-cringI*)
show *cring R*
by (*simp add*: *R-cring*)
show *cring R*

by (*simp add*: *R-cring*)
show ($\lambda x.\ x$) $\in$ *ring-hom R R*
unfolding *ring-hom-def*
apply(*auto*)
done
qed

lemma *UP-pre-univ-prop-fact*:
UP-pre-univ-prop R R ($\lambda x.\ x$)
unfolding *UP-pre-univ-prop-def*
by (*simp add*: *UP-cring-def R-cring id-is-hom*)

end

4.1 to-fun is a Ring Homomorphism from Polynomials to Functions

context *UP-cring*
begin

lemma *to-fun-is-Fun*:
assumes $x \in$ *carrier P*
shows *to-fun x* $\in$ *carrier* (*Fun R*)
apply(*rule ring-functions.function-ring-car-memI*)
unfolding *ring-functions-def* **apply**(*simp add*: *R.ring-axioms*)
using *to-fun-closed assms* **apply** *auto*[*1*]
unfolding *to-function-def to-fun-def* **by** *auto*

lemma *to-fun-Fun-mult*:
assumes $x \in$ *carrier P*
assumes $y \in$ *carrier P*
shows *to-fun* ($x \otimes_P y$) = *to-fun x* $\otimes_{\textit{function-ring}}$ (*carrier R*) *R to-fun y*
apply(*rule ring-functions.function-ring-car-eqI*[*of R - carrier R*])
apply (*simp add*: *R.ring-axioms ring-functions-def*)
apply (*simp add*: *assms*(*1*) *assms*(*2*) *to-fun-is-Fun*)
apply (*simp add*: *R.ring-axioms assms*(*1*) *assms*(*2*) *ring-functions.fun-mult-closed ring-functions.intro to-fun-is-Fun*)
by (*simp add*: *R.ring-axioms assms*(*1*) *assms*(*2*) *ring-functions.function-mult-eval-car ring-functions.intro to-fun-is-Fun to-fun-mult*)

lemma *to-fun-Fun-add*:
assumes $x \in$ *carrier P*
assumes $y \in$ *carrier P*
shows *to-fun* ($x \oplus_P y$) = *to-fun x* $\oplus_{\textit{function-ring}}$ (*carrier R*) *R to-fun y*
apply(*rule ring-functions.function-ring-car-eqI*[*of R - carrier R*])
apply (*simp add*: *R.ring-axioms ring-functions-def*)
apply (*simp add*: *assms*(*1*) *assms*(*2*) *to-fun-is-Fun*)
apply (*simp add*: *R.ring-axioms assms*(*1*) *assms*(*2*) *ring-functions.fun-add-closed ring-functions.intro to-fun-is-Fun*)

by (*simp add*: *R.ring-axioms assms*(*1*) *assms*(*2*) *ring-functions.fun-add-eval-car ring-functions.intro to-fun-is-Fun to-fun-plus*)

lemma *to-fun-Fun-one*:
to-fun $\mathbf{1}_P = \mathbf{1}_{Fun\ R}$
apply(*rule ring-functions.function-ring-car-eqI*[*of R - carrier R*])
apply (*simp add*: *R.ring-axioms ring-functions-def*)
apply (*simp add*: *to-fun-is-Fun*)
apply (*simp add*: *R.ring-axioms ring-functions.function-one-closed ring-functions-def*)
using *P-def R.ring-axioms UP-cring.eval-ring-hom UP-cring.to-fun-eval UP-cring-axioms UP-one-closed ring-functions.function-one-eval ring-functions.intro ring-hom-one*
by *fastforce*

lemma *to-fun-Fun-zero*:
to-fun $\mathbf{0}_P = \mathbf{0}_{Fun\ R}$
apply(*rule ring-functions.function-ring-car-eqI*[*of R - carrier R*])
apply (*simp add*: *R.ring-axioms ring-functions-def*)
apply (*simp add*: *to-fun-is-Fun*)
apply (*simp add*: *R.ring-axioms ring-functions.function-zero-closed ring-functions-def*)
using *P-def R.ring-axioms UP-cring.eval-ring-hom UP-cring.to-fun-eval UP-cring-axioms UP-zero-closed ring-functions.function-zero-eval ring-functions.intro ring-hom-zero*
by (*metis UP-ring eval-ring-hom*)

lemma *to-fun-function-ring-hom*:
to-fun $\in$ *ring-hom P* (*Fun R*)
apply(*rule ring-hom-memI*)
using *to-fun-is-Fun* **apply** *auto*[*1*]
apply (*simp add*: *to-fun-Fun-mult*)
apply (*simp add*: *to-fun-Fun-add*)
by (*simp add*: *to-fun-Fun-one*)

lemma(**in** *UP-cring*) *to-fun-one*:
assumes $a \in$ *carrier R*
shows *to-fun* $\mathbf{1}_P$ $a = \mathbf{1}$
using *assms to-fun-Fun-one*
by (*metis P-def UP-cring.to-fun-eval UP-cring-axioms UP-one-closed eval-ring-hom ring-hom-one*)

lemma(**in** *UP-cring*) *to-fun-zero*:
assumes $a \in$ *carrier R*
shows *to-fun* $\mathbf{0}_P$ $a = \mathbf{0}$
by (*simp add*: *assms R.ring-axioms ring-functions.function-zero-eval ring-functions.intro to-fun-Fun-zero*)

lemma(**in** *UP-cring*) *to-fun-nat-pow*:
assumes $h \in$ *carrier* (*UP R*)
assumes $a \in$ *carrier R*

shows *to-fun* $(h[\hat{}]_{UP\ R}(n::nat))\ a = (to\text{-}fun\ h\ a)[\hat{}]n$
apply(*induction n*)
using *assms to-fun-one*
apply (*metis P.nat-pow-0 P-def R.nat-pow-0*)
using *assms to-fun-mult P.nat-pow-closed P-def* **by** *auto*

lemma(**in** *UP-cring*) *to-fun-finsum*:
assumes *finite* $(Y::'d\ set)$
assumes $f \in UNIV \to carrier\ (UP\ R)$
assumes $t \in carrier\ R$
shows *to-fun* (*finsum* $(UP\ R)\ f\ Y)\ t = finsum\ R\ (\lambda i.\ (to\text{-}fun\ (f\ i)\ t))\ Y$
proof(*rule finite.induct*[*of Y*])
show *finite Y*
using *assms* **by** *blast*
show *to-fun* (*finsum* $(UP\ R)\ f\ \{\})\ t = (\bigoplus i{\in}\{\}.\ to\text{-}fun\ (f\ i)\ t)$
using *P.finsum-empty*[*of f*] *assms* **unfolding** *P-def R.finsum-empty*
using *P-def to-fun-zero* **by** *presburger*
show $\bigwedge A\ a.\ finite\ A \Longrightarrow$
to-fun (*finsum* $(UP\ R)\ f\ A)\ t = (\bigoplus i{\in}A.\ to\text{-}fun\ (f\ i)\ t) \Longrightarrow$ *to-fun* (*finsum* $(UP\ R)\ f$ (*insert a A*)) $t = (\bigoplus i{\in}insert\ a\ A.\ to\text{-}fun\ (f\ i)\ t)$
proof−
fix $A :: 'd\ set$ **fix** *a*
assume *A*: *finite A to-fun* (*finsum* $(UP\ R)\ f\ A)\ t = (\bigoplus i{\in}A.\ to\text{-}fun\ (f\ i)\ t)$
show *to-fun* (*finsum* $(UP\ R)\ f$ (*insert a A*)) $t = (\bigoplus i{\in}insert\ a\ A.\ to\text{-}fun\ (f\ i)\ t)$
proof(*cases* $a \in A$)
case *True*
then show *?thesis* **using** *A*
by (*metis insert-absorb*)
next
case *False*
have *0*: *finsum* $(UP\ R)\ f$ (*insert a A*) $= f\ a \oplus_{UP\ R} finsum\ (UP\ R)\ f\ A$
using *A False finsum-insert*[*of A a f*] *assms* **unfolding** *P-def* **by** *blast*
have *1*: *to-fun* $(f\ a \oplus_P finsum\ (UP\ R)\ f\ A\ )\ t =$ *to-fun* $(f\ a)\ t \oplus$ *to-fun* (*finsum* $(UP\ R)\ f\ A)\ t$
apply(*rule to-fun-plus*[*of finsum* $(UP\ R)\ f\ A\ f\ a\ t$])
using *assms*(*2*) *finsum-closed*[*of f A*] *A* **unfolding** *P-def* **apply** *blast*
using *P-def assms* **apply** *blast*
using *assms* **by** *blast*
have *2*: *to-fun* $(f\ a \oplus_P finsum\ (UP\ R)\ f\ A\ )\ t =$ *to-fun* $(f\ a)\ t \oplus (\bigoplus i{\in}A.$ *to-fun* $(f\ i)\ t)$
unfolding *1 A* **by** *blast*
have *3*: $(\bigoplus i{\in}insert\ a\ A.\ to\text{-}fun\ (f\ i)\ t) = to\text{-}fun\ (f\ a)\ t \oplus (\bigoplus i{\in}A.\ to\text{-}fun\ (f\ i)\ t)$
apply(*rule R.finsum-insert*, *rule A*, *rule False*)
using *to-fun-closed assms* **unfolding** *P-def* **apply** *blast*
apply(*rule to-fun-closed*) **using** *assms* **unfolding** *P-def* **apply** *blast* **using** *assms* **by** *blast*
show *?thesis*

unfolding *0* **unfolding** *3* **using** *2* **unfolding** *P-def* **by** *blast*
qed
qed
qed

end

4.2 Inclusion of a Ring into its Polynomials Ring via Constants

definition *to-polynomial* **where**
to-polynomial R = (λ*a*. *monom* (*UP R*) *a* 0)

context *UP-cring*
begin

abbreviation(*input*) *to-poly* **where**
to-poly ≡ *to-polynomial R*

lemma *to-poly-mult-simp*:
assumes *b* ∈ *carrier R*
assumes *f* ∈ *carrier* (*UP R*)
shows (*to-polynomial R b*) ⊗$_{UP\ R}$ *f* = *b* ⊙$_{UP\ R}$ *f*
f ⊗$_{UP\ R}$ (*to-polynomial R b*) = *b* ⊙$_{UP\ R}$ *f*
unfolding *to-polynomial-def*
using *assms P-def monom-mult-is-smult* **apply** *auto*[*1*]
using *UP-cring.UP-m-comm UP-cring-axioms UP-ring.monom-closed*
UP-ring.monom-mult-is-smult UP-ring-axioms assms(*1*) *assms*(*2*)
by *fastforce*

lemma *to-fun-to-poly*:
assumes *a* ∈ *carrier R*
assumes *b* ∈ *carrier R*
shows *to-fun* (*to-poly a*) *b* = *a*
unfolding *to-function-def to-fun-def to-polynomial-def*
by (*simp add*: *UP-pre-univ-prop.eval-const UP-pre-univ-prop-fact assms*(*1*) *assms*(*2*))

lemma *to-poly-inverse*:
assumes *f* ∈ *carrier P*
assumes *degree f* = *0*
shows *f* = *to-poly* (*f* 0)
using *P-def assms*(*1*) *assms*(*2*)
by (*metis ltrm-deg-0 to-polynomial-def*)

lemma *to-poly-closed*:
assumes *a* ∈ *carrier R*
shows *to-poly a* ∈ *carrier P*
by (*metis P-def assms monom-closed to-polynomial-def*)

lemma *degree-to-poly*[*simp*]:
assumes $a \in$ *carrier R*
shows *degree* (*to-poly a*) = *0*
by (*metis P-def assms deg-const to-polynomial-def*)

lemma *to-poly-is-ring-hom*:
to-poly $\in$ *ring-hom R P*
unfolding *to-polynomial-def*
unfolding *P-def*
using *UP-ring.const-ring-hom*[*of R*]
UP-ring-axioms **by** *simp*

lemma *to-poly-add*:
assumes $a \in$ *carrier R*
assumes $b \in$ *carrier R*
shows *to-poly* $(a \oplus b)$ = *to-poly a* $\oplus_P$ *to-poly b*
by (*simp add*: *assms*(*1*) *assms*(*2*) *ring-hom-add to-poly-is-ring-hom*)

lemma *to-poly-mult*:
assumes $a \in$ *carrier R*
assumes $b \in$ *carrier R*
shows *to-poly* $(a \otimes b)$ = *to-poly a* $\otimes_P$ *to-poly b*
by (*simp add*: *assms*(*1*) *assms*(*2*) *ring-hom-mult to-poly-is-ring-hom*)

lemma *to-poly-minus*:
assumes $a \in$ *carrier R*
assumes $b \in$ *carrier R*
shows *to-poly* $(a \ominus b)$ = *to-poly a* $\ominus_P$ *to-poly b*
by (*metis P.minus-eq P-def R.add.inv-closed R.ring-axioms UP-ring.monom-add*

UP-ring-axioms assms(*1*) *assms*(*2*) *monom-a-inv ring.ring-simprules*(*14*)
to-polynomial-def)

lemma *to-poly-a-inv*:
assumes $a \in$ *carrier R*
shows *to-poly* $(\ominus a) = \ominus_P$ *to-poly a*
by (*metis P-def assms monom-a-inv to-polynomial-def*)

lemma *to-poly-nat-pow*:
assumes $a \in$ *carrier R*
shows (*to-poly a*) $[\hat{}]_P$ (*n*::*nat*)= *to-poly* $(a[\hat{}]n)$
using *assms UP-cring UP-cring-axioms UP-cring-def UnivPoly.ring-hom-cringI*
ring-hom-cring.hom-pow to-poly-is-ring-hom
by *fastforce*

end

5 Polynomial Substitution

definition *compose* **where**
compose R f g = eval R (UP R) (to-polynomial R) g f

abbreviation(**in** *UP-cring*)(*input*) *sub* (**infixl** ‹*of*› *70*) **where**
sub f g ≡ compose R f g

definition *rev-compose* **where**
rev-compose R = eval R (UP R) (to-polynomial R)

abbreviation(**in** *UP-cring*)(*input*) *rev-sub* **where**
rev-sub ≡ rev-compose R

context *UP-cring*
begin

lemma *sub-rev-sub*:
sub f g = rev-sub g f
unfolding *compose-def rev-compose-def*
by *simp*

lemma(**in** *UP-cring*) *to-poly-UP-pre-univ-prop*:
UP-pre-univ-prop R P to-poly
proof
show *to-poly ∈ ring-hom R P*
by (*simp add*: *to-poly-is-ring-hom*)
qed

lemma *rev-sub-is-hom*:
assumes *g ∈ carrier P*
shows *rev-sub g ∈ ring-hom P P*
unfolding *rev-compose-def*
using *to-poly-UP-pre-univ-prop assms*(*1*) *UP-pre-univ-prop.eval-ring-hom*[*of R P to-poly g*]
unfolding *P-def* **apply** *auto*
done

lemma *rev-sub-closed*:
assumes *p ∈ carrier P*
assumes *q ∈ carrier P*
shows *rev-sub q p ∈ carrier P*
using *rev-sub-is-hom*[*of q*] *assms ring-hom-closed*[*of rev-sub q P P p*] **by** *auto*

lemma *sub-closed*:
assumes *p ∈ carrier P*
assumes *q ∈ carrier P*
shows *sub q p ∈ carrier P*
by (*simp add*: *assms*(*1*) *assms*(*2*) *rev-sub-closed sub-rev-sub*)

lemma *rev-sub-add*:
assumes $g \in carrier\ P$
assumes $f \in carrier\ P$
assumes $h \in carrier\ P$
shows $rev\text{-}sub\ g\ (f \oplus_P h) = (rev\text{-}sub\ g\ f) \oplus_P (rev\text{-}sub\ g\ h)$
using *rev-sub-is-hom assms ring-hom-add* **by** *fastforce*

lemma *sub-add*:
assumes $g \in carrier\ P$
assumes $f \in carrier\ P$
assumes $h \in carrier\ P$
shows $((f \oplus_P h)\ of\ g) = ((f\ of\ g) \oplus_P (h\ of\ g))$
by (*simp add*: *assms*(*1*) *assms*(*2*) *assms*(*3*) *rev-sub-add sub-rev-sub*)

lemma *rev-sub-mult*:
assumes $g \in carrier\ P$
assumes $f \in carrier\ P$
assumes $h \in carrier\ P$
shows $rev\text{-}sub\ g\ (f \otimes_P h) = (rev\text{-}sub\ g\ f) \otimes_P (rev\text{-}sub\ g\ h)$
using *rev-sub-is-hom assms ring-hom-mult* **by** *fastforce*

lemma *sub-mult*:
assumes $g \in carrier\ P$
assumes $f \in carrier\ P$
assumes $h \in carrier\ P$
shows $((f \otimes_P h)\ of\ g) = ((f\ of\ g) \otimes_P (h\ of\ g))$
by (*simp add*: *assms*(*1*) *assms*(*2*) *assms*(*3*) *rev-sub-mult sub-rev-sub*)

lemma *sub-monom*:
assumes $g \in carrier\ (UP\ R)$
assumes $a \in carrier\ R$
shows $sub\ (monom\ (UP\ R)\ a\ n)\ g = to\text{-}poly\ a\ \otimes_{UP\ R} (g[\hat{}]_{UP\ R}\ (n::nat))$
$sub\ (monom\ (UP\ R)\ a\ n)\ g = a \odot_{UP\ R} (g[\hat{}]_{UP\ R}\ (n::nat))$
apply (*simp add*: *UP-cring.to-poly-UP-pre-univ-prop UP-cring-axioms*
UP-pre-univ-prop.eval-monom assms(*1*) *assms*(*2*) *Cring-Poly.compose-def*)
by (*metis P-def UP-cring.to-poly-mult-simp*(*1*) *UP-cring-axioms UP-pre-univ-prop.eval-monom*

UP-ring assms(*1*) *assms*(*2*) *Cring-Poly.compose-def monoid.nat-pow-closed*
ring-def to-poly-UP-pre-univ-prop)

Subbing into a constant does nothing

lemma *rev-sub-to-poly*:
assumes $g \in carrier\ P$
assumes $a \in carrier\ R$
shows $rev\text{-}sub\ g\ (to\text{-}poly\ a) = to\text{-}poly\ a$
unfolding *to-polynomial-def rev-compose-def*
using *to-poly-UP-pre-univ-prop*
unfolding *to-polynomial-def*

using *P-def UP-pre-univ-prop.eval-const assms(1) assms(2)* **by** *fastforce*

lemma *sub-to-poly*:
assumes $g \in$ *carrier P*
assumes $a \in$ *carrier R*
shows (*to-poly a*) *of g* = *to-poly a*
by (*simp add*: *assms(1) assms(2) rev-sub-to-poly sub-rev-sub*)

lemma *sub-const*:
assumes $g \in$ *carrier P*
assumes $f \in$ *carrier P*
assumes *degree f* = *0*
shows *f of g* = *f*
by (*metis lcf-closed assms(1) assms(2) assms(3) sub-to-poly to-poly-inverse*)

Substitution into a monomial

lemma *monom-sub*:
assumes $a \in$ *carrier R*
assumes $g \in$ *carrier P*
shows (*monom P a n*) *of g* = $a \odot_P g[\hat{}]_P n$
unfolding *compose-def*
using *assms UP-pre-univ-prop.eval-monom*[*of R P to-poly a g n*] *to-poly-UP-pre-univ-prop*

unfolding *P-def*
using *P.nat-pow-closed P-def to-poly-mult-simp*(*1*)
by (*simp add*: *to-poly-mult-simp*(*1*) *UP-cring-axioms*)

lemma(**in** *UP-cring*) *cring-sub-monom-bound*:
assumes $a \in$ *carrier R*
assumes $a \neq \mathbf{0}$
assumes *f* = *monom P a n*
assumes $g \in$ *carrier P*
shows *degree* (*f of g*) $\leq n*$(*degree g*)
proof−
have *f of g* = (*to-poly a*) $\otimes_P$ ($g[\hat{}]_P n$)
unfolding *compose-def*
using *assms UP-pre-univ-prop.eval-monom*[*of R P to-poly a g*] *to-poly-UP-pre-univ-prop*

unfolding *P-def*
by *blast*
then show *?thesis*
by (*smt* (*verit*) *P.nat-pow-closed assms*(*1*) *assms*(*4*) *cring-pow-deg-bound deg-mult-ring degree-to-poly le-trans plus-nat.add-0 to-poly-closed*)
qed

lemma(**in** *UP-cring*) *cring-sub-monom*:
assumes $a \in$ *carrier R*
assumes $a \neq \mathbf{0}$
assumes *f* = *monom P a n*

assumes $g \in$ *carrier* P
assumes $a \otimes (lcf\ g\ [\widehat{}]\ n) \neq \mathbf{0}$
shows *degree* $(f\ of\ g) = n*(degree\ g)$
proof−
have *0*: $f\ of\ g = (to\text{-}poly\ a) \otimes_P (g[\widehat{}]_P n)$
unfolding *compose-def*
using *assms UP-pre-univ-prop.eval-monom*[*of R P to-poly a g*] *to-poly-UP-pre-univ-prop*

unfolding *P-def*
by *blast*
have *1*: $lcf\ (to\text{-}poly\ a) \otimes lcf\ (g\ [\widehat{}]_P\ n) \neq \mathbf{0}$
using *assms*
by (*smt* (*verit*) *P.nat-pow-closed P-def R.nat-pow-closed R.r-null cring-pow-ltrm lcf-closed lcf-ltrm lcf-monom monom-pow to-polynomial-def*)
then show *?thesis*
using *0 1 assms cring-pow-deg*[*of g n*] *cring-deg-mult*[*of to-poly a g*[$\widehat{}$]$_P n$]
by (*metis P.nat-pow-closed R.r-null add.right-neutral degree-to-poly to-poly-closed*)
qed

lemma(**in** *UP-domain*) *sub-monom*:
assumes $a \in$ *carrier* R
assumes $a \neq \mathbf{0}$
assumes $f =$ *monom* $P\ a\ n$
assumes $g \in$ *carrier* P
shows *degree* $(f\ of\ g) = n*(degree\ g)$
proof−
have $f\ of\ g = (to\text{-}poly\ a) \otimes_P (g[\widehat{}]_P n)$
unfolding *compose-def*
using *assms UP-pre-univ-prop.eval-monom*[*of R P to-poly a g*] *to-poly-UP-pre-univ-prop*

unfolding *P-def*
by *blast*
then show *?thesis* **using** *deg-pow deg-mult*
by (*metis P.nat-pow-closed P-def assms*(*1*) *assms*(*2*)
assms(*4*) *deg-smult monom-mult-is-smult to-polynomial-def*)
qed

Subbing a constant into a polynomial yields a constant

lemma *sub-in-const*:
assumes $g \in$ *carrier* P
assumes $f \in$ *carrier* P
assumes *degree* $g = 0$
shows *degree* $(f\ of\ g) = 0$
proof−
have $\bigwedge n.\ (\bigwedge p.\ p \in carrier\ P \Longrightarrow degree\ p \leq n \Longrightarrow degree\ (p\ of\ g) = 0)$
proof−
fix n
show $\bigwedge p.\ p \in carrier\ P \Longrightarrow degree\ p \leq n \Longrightarrow degree\ (p\ of\ g) = 0$
proof(*induction n*)

```
    case 0
    then show ?case
      by (simp add: assms(1) sub-const)
  next
    case (Suc n)
    fix n
    assume IH: ⋀p. p ∈ carrier P ⟹ degree p ≤ n ⟹ degree (p of g) = 0
    show  ⋀p. p ∈ carrier P ⟹ degree p ≤ (Suc n) ⟹ degree (p of g) = 0
    proof−
      fix p
      assume A0: p ∈ carrier P
      assume A1: degree p ≤ (Suc n)
      show degree (p of g) = 0
      proof(cases degree p < Suc n)
        case True
        then show ?thesis using IH
          using A0 by auto
      next
        case False
        then have D: degree p = Suc n
          by (simp add: A1 nat-less-le)
        show ?thesis
        proof−
          have P0: degree ((trunc p) of g) = 0 using IH
            by (metis A0 D less-Suc-eq-le trunc-degree trunc-closed zero-less-Suc)
          have P1: degree ((ltrm p) of g) = 0
          proof−
            obtain a n where an-def: ltrm p = monom P a n ∧ a ∈ carrier R
              unfolding leading-term-def
              using A0 P-def cfs-closed by blast
            obtain b where b-def: g = monom P b 0 ∧ b ∈ carrier R
              using assms deg-zero-impl-monom  coeff-closed
              by blast
            have 0:  monom P b 0 [^]_P n = monom P (b[^]n) 0
              apply(induction n)
               apply fastforce[1]
             proof− fix n::nat assume IH: monom P b 0 [^]_P n = monom P (b
[^] n) 0
              have monom P b 0 [^]_P Suc n = (monom P (b[^]n) 0) ⊗_P monom
P b 0
                using IH by simp
              then have monom P b 0 [^]_P Suc n = (monom P ((b[^]n)⊗b) 0)
                using b-def
                by (simp add: monom-mult-is-smult monom-mult-smult)
              then show monom P b 0 [^]_P Suc n = monom P (b [^] Suc n) 0
                by simp
            qed

            then have 0: a ⊙_P monom P b 0 [^]_P n = monom P (a ⊗ b[^]n) 0
```

by (*simp add*: *an-def b-def monom-mult-smult*)

then show *?thesis* **using** *monom-sub*[*of a monom P b 0 n*] *assms an-def*
by (*simp add*: ‹⟦*a* ∈ *carrier R*; *monom P b 0* ∈ *carrier P*⟧ ⟹ *monom P a n of monom P b 0 = a* ⊙$_P$ *monom P b 0* [^]$_P$ *n*› *b-def*)
qed
have *P2*: *p of g* = (*trunc p of g*) ⊕$_P$ ((*ltrm p*) *of g*)
by (*metis A0 assms*(*1*) *ltrm-closed sub-add trunc-simps*(*1*) *trunc-closed*)
then show *?thesis*
using *P0 P1 P2 deg-add*[*of trunc p of g ltrm p of g*]
by (*metis A0 assms*(*1*) *le-0-eq ltrm-closed max-0R sub-closed trunc-closed*)

qed
qed
qed
qed
qed
then show *?thesis*
using *assms*(*2*) **by** *blast*
qed

lemma (**in** *UP-cring*) *cring-sub-deg-bound*:
assumes *g* ∈ *carrier P*
assumes *f* ∈ *carrier P*
shows *degree* (*f of g*) ≤ *degree f* ∗ *degree g*
proof−
have ⋀*n*. ⋀ *p*. *p* ∈ *carrier P* ⟹ (*degree p*) ≤ *n* ⟹ *degree* (*p of g*) ≤ *degree p* ∗ *degree g*
proof−
fix *n*::*nat*
show ⋀ *p*. *p* ∈ *carrier P* ⟹ (*degree p*) ≤ *n* ⟹ *degree* (*p of g*) ≤ *degree p* ∗ *degree g*
proof(*induction n*)
case *0*
then have *B0*: *degree p* = *0* **by** *auto*
then show *?case* **using** *sub-const*[*of g p*]
by (*simp add*: *0.prems*(*1*) *assms*(*1*))
next
case (*Suc n*)
fix *n*
assume *IH*: (⋀*p*. *p* ∈ *carrier P* ⟹ *degree p* ≤ *n* ⟹ *degree* (*p of g*) ≤ *degree p* ∗ *degree g*)
show *p* ∈ *carrier P* ⟹ *degree p* ≤ *Suc n* ⟹ *degree* (*p of g*) ≤ *degree p* ∗ *degree g*
proof−
assume *A0*: *p* ∈ *carrier P*
assume *A1*: *degree p* ≤ *Suc n*

show *?thesis*
proof(*cases degree p < Suc n*)
case *True*
then show *?thesis* **using** *IH*
by (*simp add*: *A0*)
next
case *False*
then have *D*: *degree p = Suc n*
using *A1* **by** *auto*
have *P0*: (*p of g*) = ((*trunc p*) *of g*) ⊕*P* ((*ltrm p*) *of g*)
by (*metis A0 assms*(*1*) *ltrm-closed sub-add trunc-simps*(*1*) *trunc-closed*)
have *P1*: *degree* ((*trunc p*) *of g*) ≤ (*degree* (*trunc p*))∗(*degree g*)
using *IH* **by** (*metis A0 D less-Suc-eq-le trunc-degree trunc-closed zero-less-Suc*)
have *P2*: *degree* ((*ltrm p*) *of g*) ≤ (*degree p*) ∗ *degree g*
using *A0 D P-def UP-cring-axioms assms*(*1*)
by (*metis False cfs-closed coeff-simp cring-sub-monom-bound deg-zero lcoeff-nonzero2 less-Suc-eq-0-disj*)
then show *?thesis*
proof(*cases degree g = 0*)
case *True*
then show *?thesis*
by (*simp add*: *Suc*(*2*) *assms*(*1*) *sub-in-const*)
next
case *F*: *False*
then show *?thesis*
proof−
have *P3*: *degree* ((*trunc p*) *of g*) ≤ *n*∗*degree g*
using *A0 False D P1 P2 IH*[*of trunc p*] *trunc-degree*[*of p*]
proof −
{ assume *degree* (*trunc p*) < *degree p*
then have *degree* (*trunc p*) ≤ *n*
using *D* **by** *auto*
then have *?thesis*
by (*meson P1 le-trans mult-le-cancel2*) **}**
then show *?thesis*
by (*metis* (*full-types*) *A0 D Suc-mult-le-cancel1 nat-mult-le-cancel-disj trunc-degree*)
qed
then have *P3′*: *degree* ((*trunc p*) *of g*) < (*degree p*)∗*degree g*
using *F D* **by** *auto*
have *P4*: *degree* (*ltrm p of g*) ≤ (*degree p*)∗*degree g*
using *cring-sub-monom-bound D P2*
by *auto*
then show *?thesis*
using *D P0 P1 P3 P4 A0 P3′ assms*(*1*) *bound-deg-sum less-imp-le-nat ltrm-closed sub-closed trunc-closed*
by *metis*
qed

```
        qed
      qed
    qed
  qed
 qed
 then show ?thesis
  using assms(2) by blast
qed

lemma (in UP-cring) cring-sub-deg:
 assumes g ∈ carrier P
 assumes f ∈ carrier P
 assumes lcf f ⊗ (lcf g [^] (degree f)) ≠ 𝟬
 shows degree (f of g) = degree f * degree g
proof−
 have 0: f of g = (trunc f of g) ⊕_P ((ltrm f) of g)
  by (metis assms(1) assms(2) ltrm-closed rev-sub-add sub-rev-sub trunc-simps(1)
trunc-closed)
 have 1: lcf f ≠ 𝟬
  using assms cring.cring-simprules(26) lcf-closed
  by auto
 have 2: degree ((ltrm f) of g) = degree f * degree g
  using 0 1 assms cring-sub-monom[of lcf f ltrm f degree f g] lcf-closed lcf-ltrm
  by blast
 show ?thesis
  apply(cases degree f = 0)
   apply (simp add: assms(1) assms(2))
  apply(cases degree g = 0)
   apply (simp add: assms(1) assms(2) sub-in-const)
  using 0 1 assms cring-sub-deg-bound[of g trunc f] trunc-degree[of f]
  using sub-const apply auto[1]
  apply(cases degree g = 0)
  using 0 1 assms cring-sub-deg-bound[of g trunc f] trunc-degree[of f]
  using sub-in-const apply fastforce
  unfolding 0 using 1 2
   by (smt (verit) 0 ltrm-closed ‹⟦f ∈ carrier P; 0 < deg R f⟧ ⟹ deg R
(Cring-Poly.truncate R f) < deg R f›
    assms(1) assms(2) cring-sub-deg-bound degree-of-sum-diff-degree equal-deg-sum
    le-eq-less-or-eq mult-less-cancel2 nat-neq-iff neq0-conv sub-closed trunc-closed)

qed

lemma (in UP-domain) sub-deg0:
 assumes g ∈ carrier P
 assumes f ∈ carrier P
 assumes g ≠ 𝟬_P
 assumes f ≠ 𝟬_P
 shows degree (f of g) = degree f * degree g
proof−
```

have $\bigwedge n.\ \bigwedge p.\ p \in carrier\ P \Longrightarrow (degree\ p) \leq n \Longrightarrow degree\ (p\ of\ g) = degree\ p * degree\ g$
proof−
fix n::nat
show $\bigwedge p.\ p \in carrier\ P \Longrightarrow (degree\ p) \leq n \Longrightarrow degree\ (p\ of\ g) = degree\ p * degree\ g$
proof(*induction n*)
case *0*
then have *B0*: $degree\ p = 0$ **by** *auto*
then show *?case* **using** *sub-const*[*of g p*]
by (*simp add*: *0.prems*(*1*) *assms*(*1*))
next
case (*Suc n*)
fix n
assume *IH*: $(\bigwedge p.\ p \in carrier\ P \Longrightarrow degree\ p \leq n \Longrightarrow degree\ (p\ of\ g) = degree\ p * degree\ g)$
show $p \in carrier\ P \Longrightarrow degree\ p \leq Suc\ n \Longrightarrow degree\ (p\ of\ g) = degree\ p * degree\ g$
proof−
assume *A0*: $p \in carrier\ P$
assume *A1*: $degree\ p \leq Suc\ n$
show *?thesis*
proof(*cases* $degree\ p < Suc\ n$)
case *True*
then show *?thesis* **using** *IH*
by (*simp add*: *A0*)
next
case *False*
then have *D*: $degree\ p = Suc\ n$
using *A1* **by** *auto*
have *P0*: $(p\ of\ g) = ((trunc\ p)\ of\ g) \oplus_P ((ltrm\ p)\ of\ g)$
by (*metis A0 assms*(*1*) *ltrm-closed sub-add trunc-simps*(*1*) *trunc-closed*)
have *P1*: $degree\ ((trunc\ p)\ of\ g) = (degree\ (trunc\ p)) * (degree\ g)$
using *IH* **by** (*metis A0 D less-Suc-eq-le trunc-degree trunc-closed zero-less-Suc*)
have *P2*: $degree\ ((ltrm\ p)\ of\ g) = (degree\ p) * degree\ g$
using *A0 D P-def UP-domain.sub-monom UP-cring-axioms assms*(*1*)
by (*metis False UP-domain-axioms UP-ring.coeff-simp UP-ring.lcoeff-nonzero2 UP-ring-axioms cfs-closed deg-nzero-nzero less-Suc-eq-0-disj*)

then show *?thesis*
proof(*cases* $degree\ g = 0$)
case *True*
then show *?thesis*
by (*simp add*: *Suc*(*2*) *assms*(*1*) *sub-in-const*)
next
case *False*
then show *?thesis*
proof−

```
            have P3: degree ((trunc p) of g) < degree ((ltrm p) of g)
              using False D  P1 P2
              by (metis (no-types, lifting) A0 mult.commute mult-right-cancel
                  nat-less-le nat-mult-le-cancel-disj trunc-degree zero-less-Suc)
            then show ?thesis
          by (simp add: A0 ltrm-closed P0 P2 assms(1) equal-deg-sum sub-closed
trunc-closed)
        qed
      qed
    qed
  qed
 qed
 then show ?thesis
    using assms(2) by blast
qed

lemma(in UP-domain)  sub-deg:
 assumes g ∈ carrier P
 assumes f ∈ carrier P
 assumes g ≠ 0_P
 shows degree (f of g) = degree f * degree g
proof(cases f = 0_P)
 case True
 then show ?thesis
   using assms(1)  sub-const by auto
next
 case False
 then show ?thesis
   by (simp add: assms(1) assms(2) assms(3) sub-deg0)
qed

lemma(in UP-cring)  cring-ltrm-sub:
 assumes g ∈ carrier P
 assumes f ∈ carrier P
 assumes degree g > 0
 assumes lcf f ⊗ (lcf g [^] (degree f)) ≠ 0
 shows ltrm (f of g) = ltrm ((ltrm f) of g)
proof−
 have P0: degree (f of g) = degree ((ltrm f) of g)
   using assms(1) assms(2) assms(4) cring-sub-deg lcf-eq ltrm-closed deg-ltrm
   by auto
 have P1: f of g = ((trunc f) of g) ⊕_P((ltrm f) of g)
  by (metis assms(1) assms(2) ltrm-closed rev-sub-add sub-rev-sub trunc-simps(1)
trunc-closed)
 then show ?thesis
 proof(cases degree f = 0)
   case True
   then show ?thesis
```

```
    using ltrm-deg-0 assms(2) by auto
  next
    case False
    have P2: degree (f of g) = degree f * degree g
      by (simp add: assms(1) assms(2) assms(4) cring-sub-deg)
    then have P3: degree ((trunc f) of g) < degree ((ltrm f) of g)
      using False P0 P1 P-def UP-cring.sub-closed trunc-closed UP-cring-axioms
            UP-ring.degree-of-sum-diff-degree UP-ring.ltrm-closed UP-ring-axioms
assms(1)
       assms(2) assms(4) cring-sub-deg-bound le-antisym less-imp-le-nat less-nat-zero-code
         mult-right-le-imp-le nat-neq-iff trunc-degree
      by (smt (verit) assms(3))
    then show ?thesis using P0 P1 P2
     by (metis (no-types, lifting) ltrm-closed ltrm-of-sum-diff-degree P.add.m-comm
assms(1) assms(2) sub-closed trunc-closed)
  qed
qed

lemma(in UP-domain) ltrm-sub:
  assumes g ∈ carrier P
  assumes f ∈ carrier P
  assumes degree g > 0
  shows ltrm (f of g) = ltrm ((ltrm f) of g)
proof−
  have P0: degree (f of g) = degree ((ltrm f) of g)
    using sub-deg
   by (metis ltrm-closed assms(1) assms(2) assms(3) deg-zero deg-ltrm nat-neq-iff)

  have P1: f of g = ((trunc f) of g) ⊕_P((ltrm f) of g)
   by (metis assms(1) assms(2) ltrm-closed rev-sub-add sub-rev-sub trunc-simps(1)
trunc-closed)
  then show ?thesis
  proof(cases degree f = 0)
    case True
    then show ?thesis
      using ltrm-deg-0 assms(2) by auto
  next
    case False
    then have P2: degree ((trunc f) of g) < degree ((ltrm f) of g)
      using sub-deg
     by (metis (no-types, lifting) ltrm-closed assms(1) assms(2) assms(3) deg-zero
         deg-ltrm mult-less-cancel2 neq0-conv trunc-closed trunc-degree)
    then show ?thesis
      using P0 P1 P2
     by (metis (no-types, lifting) ltrm-closed ltrm-of-sum-diff-degree P.add.m-comm
assms(1) assms(2) sub-closed trunc-closed)
  qed
qed
```

lemma(**in** *UP-cring*) *cring-lcf-of-sub-in-ltrm*:
assumes $g \in$ *carrier* P
assumes $f \in$ *carrier* P
assumes *degree* $f = n$
assumes *degree* $g > 0$
assumes $(lcf\ f) \otimes ((lcf\ g)[\hat{}]n) \neq \mathbf{0}$
shows $lcf\ ((ltrm\ f)\ of\ g) = (lcf\ f) \otimes ((lcf\ g)[\hat{}]n)$
by (*metis* (*no-types*, *lifting*) *P.nat-pow-closed P-def R.r-null UP-cring.monom-sub UP-cring-axioms*
assms(*1*) *assms*(*2*) *assms*(*3*) *assms*(*5*) *cfs-closed cring-lcf-pow cring-lcf-scalar-mult*)

lemma(**in** *UP-domain*) *lcf-of-sub-in-ltrm*:
assumes $g \in$ *carrier* P
assumes $f \in$ *carrier* P
assumes *degree* $f = n$
assumes *degree* $g > 0$
shows $lcf\ ((ltrm\ f)\ of\ g) = (lcf\ f) \otimes ((lcf\ g)[\hat{}]n)$
proof(*cases degree* $f = 0$)
case *True*
then show *?thesis*
using *ltrm-deg-0 assms*(*1*) *assms*(*2*) *assms*(*3*) *cfs-closed*
by (*simp add*: *sub-const*)
next
case *False*
then show *?thesis*
proof−
have *P0*: $(ltrm\ f)\ of\ g = (to\text{-}poly\ (lcf\ f)) \otimes_P (g[\hat{}]_P n)$
unfolding *compose-def*
using *assms UP-pre-univ-prop.eval-monom*[*of R P to-poly* (*lcf f*) *g n*] *to-poly-UP-pre-univ-prop*

unfolding *P-def*
using *P-def cfs-closed* **by** *blast*
have *P1*: $(ltrm\ f)\ of\ g = (lcf\ f) \odot_P (g[\hat{}]_P n)$
using *P0 P.nat-pow-closed*
by (*simp add*: *assms*(*1*) *assms*(*2*) *assms*(*3*) *cfs-closed monom-sub*)
have *P2*: $ltrm\ ((ltrm\ f)\ of\ g) = (ltrm\ (to\text{-}poly\ (lcf\ f))) \otimes_P (ltrm\ (g[\hat{}]_P n))$
using *P0 ltrm-mult P.nat-pow-closed P-def assms*(*1*) *assms*(*2*)
to-poly-closed
by (*simp add*: *cfs-closed*)
have *P3*: $ltrm\ ((ltrm\ f)\ of\ g) = (to\text{-}poly\ (lcf\ f)) \otimes_P (ltrm\ (g[\hat{}]_P n))$
using *P2 ltrm-deg-0 assms*(*2*) *to-poly-closed*
by (*simp add*: *cfs-closed*)
have *P4*: $ltrm\ ((ltrm\ f)\ of\ g) = (lcf\ f) \odot_P ((ltrm\ g)[\hat{}]_P n)$
using *P.nat-pow-closed P1 P-def assms*(*1*) *assms*(*2*) *ltrm-pow0 ltrm-smult*
by (*simp add*: *cfs-closed*)
have *P5*: $lcf\ ((ltrm\ f)\ of\ g) = (lcf\ f) \otimes (lcf\ ((ltrm\ g)[\hat{}]_P n))$
using *lcf-scalar-mult P4* **by** (*metis P.nat-pow-closed P1 cfs-closed*
UP-smult-closed assms(*1*) *assms*(*2*) *assms*(*3*) *lcf-eq ltrm-closed sub-rev-sub*)
show *?thesis*

using *P5 ltrm-pow lcf-pow assms(1) lcf-eq ltrm-closed* **by** *presburger*
qed
qed

lemma(**in** *UP-cring*) *cring-ltrm-of-sub-in-ltrm*:
assumes $g \in$ *carrier* P
assumes $f \in$ *carrier* P
assumes *degree* $f = n$
assumes *degree* $g > 0$
assumes $(lcf\ f) \otimes ((lcf\ g)[\hat{}]n) \neq \mathbf{0}$
shows $ltrm\ ((ltrm\ f)\ of\ g) = (lcf\ f) \odot_P ((ltrm\ g)[\hat{}]_P n)$
by (*smt* (*verit*) *lcf-eq ltrm-closed R.nat-pow-closed R.r-null assms(1) assms(2) assms(3) assms(4) assms(5) cfs-closed cring-lcf-of-sub-in-ltrm cring-lcf-pow cring-pow-ltrm cring-pow-deg cring-sub-deg deg-zero deg-ltrm monom-mult-smult neq0-conv*)

lemma(**in** *UP-domain*) *ltrm-of-sub-in-ltrm*:
assumes $g \in$ *carrier* P
assumes $f \in$ *carrier* P
assumes *degree* $f = n$
assumes *degree* $g > 0$
shows $ltrm\ ((ltrm\ f)\ of\ g) = (lcf\ f) \odot_P ((ltrm\ g)[\hat{}]_P n)$
using *assms(1) assms(2) assms(3) lcf-closed ltrm-pow0 ltrm-smult monom-sub* **by** *force*

formula for the leading term of a composition

lemma(**in** *UP-domain*) *cring-ltrm-of-sub*:
assumes $g \in$ *carrier* P
assumes $f \in$ *carrier* P
assumes *degree* $f = n$
assumes *degree* $g > 0$
assumes $(lcf\ f) \otimes ((lcf\ g)[\hat{}]n) \neq \mathbf{0}$
shows $ltrm\ (f\ of\ g) = (lcf\ f) \odot_P ((ltrm\ g)[\hat{}]_P n)$
using *ltrm-of-sub-in-ltrm ltrm-sub assms(1) assms(2) assms(3) assms(4)* **by** *presburger*

lemma(**in** *UP-domain*) *ltrm-of-sub*:
assumes $g \in$ *carrier* P
assumes $f \in$ *carrier* P
assumes *degree* $f = n$
assumes *degree* $g > 0$
shows $ltrm\ (f\ of\ g) = (lcf\ f) \odot_P ((ltrm\ g)[\hat{}]_P n)$
using *ltrm-of-sub-in-ltrm ltrm-sub assms(1) assms(2) assms(3) assms(4)* **by** *presburger*

subtitution is associative

lemma *sub-assoc-monom*:
assumes $f \in$ *carrier* P
assumes $q \in$ *carrier* P

assumes $r \in$ *carrier* P
shows (*ltrm* f) *of* (q *of* r) = ((*ltrm* f) *of* q) *of* r
proof−
obtain n **where** *n-def*: n = *degree* f
by *simp*
obtain a **where** *a-def*: $a \in$ *carrier* $R \wedge$ (*ltrm* f) = *monom* P a n
using *assms*(1) *cfs-closed n-def* **by** *blast*
have *LHS*: (*ltrm* f) *of* (q *of* r) = $a \odot_P$ (q *of* r)[^]$_P$ n
by (*metis P.nat-pow-closed P-def UP-pre-univ-prop.eval-monom a-def assms*(2)
assms(3) *compose-def monom-mult-is-smult sub-closed to-poly-UP-pre-univ-prop*
to-polynomial-def)
have *RHS0*: ((*ltrm* f) *of* q) *of* r = ($a \odot_P$ q[^]$_P$ n)*of* r
by (*metis P.nat-pow-closed P-def UP-pre-univ-prop.eval-monom a-def*
assms(2) *compose-def monom-mult-is-smult to-poly-UP-pre-univ-prop to-polynomial-def*)
have *RHS1*: ((*ltrm* f) *of* q) *of* r = ((*to-poly* a) $\otimes_P$ q[^]$_P$ n)*of* r
using *RHS0* **by** (*metis P.nat-pow-closed P-def a-def*
assms(2) *monom-mult-is-smult to-polynomial-def*)
have *RHS2*: ((*ltrm* f) *of* q) *of* r = ((*to-poly* a) *of* r) $\otimes_P$ (q[^]$_P$ n *of* r)
using *RHS1 a-def assms*(2) *assms*(3) *sub-mult to-poly-closed* **by** *auto*
have *RHS3*: ((*ltrm* f) *of* q) *of* r = (*to-poly* a) $\otimes_P$ (q[^]$_P$ n *of* r)
using *RHS2 a-def assms*(3) *sub-to-poly* **by** *auto*
have *RHS4*: ((*ltrm* f) *of* q) *of* r = $a \odot_P$ ((q[^]$_P$ n)*of* r)
using *RHS3*
by (*metis P.nat-pow-closed P-def a-def assms*(2) *assms*(3)
monom-mult-is-smult sub-closed to-polynomial-def)
have (q *of* r)[^]$_P$ n = ((q[^]$_P$ n)*of* r)
apply(*induction* n)
apply (*metis Group.nat-pow-0 P.ring-simprules*(6) *assms*(3) *deg-one sub-const*)
by (*simp add*: *assms*(2) *assms*(3) *sub-mult*)
then show *?thesis* **using** *RHS4 LHS* **by** *simp*
qed

lemma *sub-assoc*:
assumes $f \in$ *carrier* P
assumes $q \in$ *carrier* P
assumes $r \in$ *carrier* P
shows f *of* (q *of* r) = (f *of* q) *of* r
proof−
have $\bigwedge n.\ \bigwedge p.\ p \in$ *carrier* $P \Longrightarrow$ *degree* $p \leq n \Longrightarrow p$ *of* (q *of* r) = (p *of* q) *of* r
proof−
fix n
show $\bigwedge p.\ p \in$ *carrier* $P \Longrightarrow$ *degree* $p \leq n \Longrightarrow p$ *of* (q *of* r) = (p *of* q) *of* r
proof(*induction* n)
case *0*
then have *deg-p*: *degree* p = *0*
by *blast*
then have *B0*: p *of* (q *of* r) = p
using *sub-const*[*of q of r p*] *assms 0.prems*(1) *sub-closed* **by** *blast*
have *B1*: (p *of* q) *of* r = p

proof−
have *p0*: *p of q = p*
using *deg-p 0 assms(2)*
by (*simp add*: *P-def UP-cring.sub-const UP-cring-axioms*)
show *?thesis*
unfolding *p0* **using** *deg-p 0 assms(3)*
by (*simp add*: *P-def UP-cring.sub-const UP-cring-axioms*)
qed
then show *p of (q of r) = (p of q) of r* **using** *B0 B1* **by** *auto*
next
case (*Suc n*)
fix *n*
assume *IH*: $\bigwedge$ *p. p ∈ carrier P ⟹ degree p ≤ n ⟹ p of (q of r) = (p of q) of r*
then show $\bigwedge$ *p. p ∈ carrier P ⟹ degree p ≤ Suc n ⟹ p of (q of r) = (p of q) of r*
proof−
fix *p*
assume *A0*: *p ∈ carrier P*
assume *A1*: *degree p ≤ Suc n*
show *p of (q of r) = (p of q) of r*
proof(*cases degree p < Suc n*)
case *True*
then show *?thesis* **using** *A0 A1 IH* **by** *auto*
next
case *False*
then have *degree p = Suc n*
using *A1* **by** *auto*
have *I0*: *p of (q of r) = ((trunc p) ⊕$_P$ (ltrm p)) of (q of r)*
using *A0 trunc-simps(1)* **by** *auto*
have *I1*: *p of (q of r) = ((trunc p) of (q of r)) ⊕$_P$ ((ltrm p) of (q of r))*
using *I0 sub-add*
by (*simp add*: *A0 assms(2) assms(3) ltrm-closed rev-sub-closed sub-rev-sub trunc-closed*)
have *I2*: *p of (q of r) = (((trunc p) of q) of r) ⊕$_P$ (((ltrm p) of q) of r)*
using *IH*[*of trunc p*] *sub-assoc-monom*[*of p q r*]
by (*metis A0 I1 ‹degree p = Suc n› assms(2) assms(3) less-Suc-eq-le trunc-degree trunc-closed zero-less-Suc*)
have *I3*: *p of (q of r) = (((trunc p) of q) ⊕$_P$ ((ltrm p) of q)) of r*
using *sub-add trunc-simps(1) assms*
by (*simp add*: *A0 I2 ltrm-closed sub-closed trunc-closed*)
have *I4*: *p of (q of r) = (((trunc p)⊕$_P$(ltrm p)) of q) of r*
using *sub-add trunc-simps(1) assms*
by (*simp add*: *trunc-simps(1) A0 I3 ltrm-closed trunc-closed*)
then show *?thesis*
using *A0 trunc-simps(1)* **by** *auto*
qed
qed
qed

qed
then show *?thesis*
using *assms(1)* **by** *blast*
qed

lemma *sub-smult*:
assumes $f \in$ *carrier P*
assumes $q \in$ *carrier P*
assumes $a \in$ *carrier R*
shows $(a \odot_P f\ )$ *of* $q = a \odot_P (f$ *of* $q)$
proof−
have $(a \odot_P f\ )$ *of* $q = ((to\text{-}poly\ a) \otimes_P f)$ *of* q
using *assms* **by** (*metis P-def monom-mult-is-smult to-polynomial-def*)
then have $(a \odot_P f\ )$ *of* $q = ((to\text{-}poly\ a)$ *of* $q) \otimes_P (f$ *of* $q)$
by (*simp add*: *assms(1) assms(2) assms(3) sub-mult to-poly-closed*)
then have $(a \odot_P f\ )$ *of* $q = (to\text{-}poly\ a) \otimes_P (f$ *of* $q)$
by (*simp add*: *assms(2) assms(3) sub-to-poly*)
then show *?thesis*
by (*metis P-def assms(1) assms(2) assms(3)*
monom-mult-is-smult sub-closed to-polynomial-def)
qed

lemma *to-fun-sub-monom*:
assumes *is-UP-monom f*
assumes $g \in$ *carrier P*
assumes $a \in$ *carrier R*
shows *to-fun* $(f$ *of* $g)\ a =$ *to-fun* f $(to\text{-}fun\ g\ a)$
proof−
obtain $b\ n$ **where** *b-def*: $b \in$ *carrier* $R \wedge f = monom\ P\ b\ n$
using *assms* **unfolding** *is-UP-monom-def*
using *P-def cfs-closed* **by** *blast*
then have *P0*: f *of* $g = b \odot_P (g[\widehat{\ }]_P n)$
using *b-def assms(2) monom-sub* **by** *blast*
have *P1*: *UP-pre-univ-prop R R* $(\lambda x.\ x)$
by (*simp add*: *UP-pre-univ-prop-fact*)
then have *P2*: *to-fun f* $(to\text{-}fun\ g\ a) = b \otimes ((to\text{-}fun\ g\ a)[\widehat{\ }]n)$
using *P1 to-fun-eval*[*of f to-fun g a*] *P-def UP-pre-univ-prop.eval-monom assms(1)*
assms(2) assms(3) b-def is-UP-monomE(1) to-fun-closed
by *force*
have *P3*: *to-fun* $(monom\ P\ b\ n$ *of* $g)\ a = \ b \otimes ((to\text{-}fun\ g\ a)[\widehat{\ }]n)$
proof−
have *0*: *to-fun* $(monom\ P\ b\ n$ *of* $g)\ a = eval\ R\ R\ (\lambda x.\ x)\ a\ \ (b \odot_P (g[\widehat{\ }]_P n)\ )$

using *UP-pre-univ-prop.eval-monom*[*of R* $(UP\ R)$ *to-poly b g n*]
P-def assms(2) b-def to-poly-UP-pre-univ-prop to-fun-eval P0
by (*metis assms(3) monom-closed sub-closed*)
have *1*: *to-fun* $(monom\ P\ b\ n$ *of* $g)\ a = (eval\ R\ R\ (\lambda x.\ x)\ a\ \ (to\text{-}poly\ b)) \otimes ($
eval R R $(\lambda x.\ x)\ a\ (\ g\ [\widehat{\ }]_{UP\ R}\ n\ ))$
using *0 eval-ring-hom*

by (*metis P.nat-pow-closed P0 P-def assms*(*2*) *assms*(*3*) *b-def monom-mult-is-smult to-fun-eval to-fun-mult to-poly-closed to-polynomial-def*)
have *2*: *to-fun* (*monom P b n of g*) *a* = *b* ⊗ (*eval R R* (λ*x*. *x*) *a* (*g* [^]$_{UP\ R}$ *n*))
using *1 assms*(*3*) *b-def to-fun-eval to-fun-to-poly to-poly-closed* **by** *auto*
then show *?thesis*
unfolding *to-function-def to-fun-def*
using *eval-ring-hom P-def UP-pre-univ-prop.ring-homD UP-pre-univ-prop-fact*

assms(*2*) *assms*(*3*) *ring-hom-cring.hom-pow* **by** *fastforce*
qed
then show *?thesis*
using *b-def P2* **by** *auto*
qed

lemma *to-fun-sub*:
assumes *g* ∈ *carrier P*
assumes *f* ∈ *carrier P*
assumes *a* ∈ *carrier R*
shows *to-fun* (*f of g*) *a* = (*to-fun f*) (*to-fun g a*)
proof(*rule poly-induct2*[*of f*])
show *f* ∈ *carrier P*
using *assms* **by** *auto*
show ⋀*p*. *p* ∈ *carrier P* ⟹ *degree p* = *0* ⟹ *to-fun* (*p of g*) *a* = *to-fun p* (*to-fun g a*)
proof−
fix *p*
assume *A0*: *p* ∈ *carrier P*
assume *A1*: *degree p* = *0*
then have *P0*: *degree* (*p of g*) = *0*
by (*simp add*: *A0 assms*(*1*) *sub-const*)
then obtain *b* **where** *b-def*: *p of g* = *to-poly b* ∧ *b* ∈ *carrier R*
using *A0 A1 cfs-closed assms*(*1*) *to-poly-inverse*
by (*meson sub-closed*)
then have *to-fun* (*p of g*) *a* = *b*
by (*simp add*: *assms*(*3*) *to-fun-to-poly*)
have *p of g* = *p*
using *A0 A1 P-def sub-const UP-cring-axioms assms*(*1*) **by** *blast*
then have *P1*: *p* = *to-poly b*
using *b-def* **by** *auto*
have *to-fun g a* ∈ *carrier R*
using *assms*
by (*simp add*: *to-fun-closed*)
then show *to-fun* (*p of g*) *a* = *to-fun p* (*to-fun g a*)
using *P1* ‹*to-fun* (*p of g*) *a* = *b*› *b-def*
by (*simp add*: *to-fun-to-poly*)
qed
show ⋀*p*. *0* < *degree p* ⟹ *p* ∈ *carrier P* ⟹
to-fun (*trunc p of g*) *a* = *to-fun* (*trunc p*) (*to-fun g a*) ⟹

to-fun (*p of g*) *a* = *to-fun p* (*to-fun g a*)
proof−
fix *p*
assume *A0*: *0* < *degree p*
assume *A1*: *p* ∈ *carrier P*
assume *A2*: *to-fun* (*trunc p of g*) *a* = *to-fun* (*trunc p*) (*to-fun g a*)
show *to-fun* (*p of g*) *a* = *to-fun p* (*to-fun g a*)
proof−
have *p of g* = (*trunc p*) *of g* ⊕$_P$ (*ltrm p*) *of g*
by (*metis A1 assms*(*1*) *ltrm-closed sub-add trunc-simps*(*1*) *trunc-closed*)
then have *to-fun* (*p of g*) *a* = *to-fun* ((*trunc p*) *of g*) *a* ⊕ (*to-fun* ((*ltrm p*) *of g*) *a*)
by (*simp add*: *A1 assms*(*1*) *assms*(*3*) *to-fun-plus ltrm-closed sub-closed trunc-closed*)
then have *0*: *to-fun* (*p of g*) *a* = *to-fun* (*trunc p*) (*to-fun g a*) ⊕ (*to-fun* ((*ltrm p*) *of g*) *a*)
by (*simp add*: *A2*)
have (*to-fun* ((*ltrm p*) *of g*) *a*) = *to-fun* (*ltrm p*) (*to-fun g a*)
using *to-fun-sub-monom*
by (*simp add*: *A1 assms*(*1*) *assms*(*3*) *ltrm-is-UP-monom*)
then have *to-fun* (*p of g*) *a* = *to-fun* (*trunc p*) (*to-fun g a*) ⊕ *to-fun* (*ltrm p*) (*to-fun g a*)
using *0* **by** *auto*
then show *?thesis*
by (*metis A1 assms*(*1*) *assms*(*3*) *to-fun-closed to-fun-plus ltrm-closed trunc-simps*(*1*) *trunc-closed*)
qed
qed
qed
end

More material on constant terms and constant coefficients

context *UP-cring*
begin

lemma *to-fun-ctrm*:
assumes *f* ∈ *carrier P*
assumes *b* ∈ *carrier R*
shows *to-fun* (*ctrm f*) *b* = (*f 0*)
using *assms*
by (*metis ctrm-degree ctrm-is-poly lcf-monom*(*2*) *P-def cfs-closed to-fun-to-poly to-poly-inverse*)

lemma *to-fun-smult*:
assumes *f* ∈ *carrier P*
assumes *b* ∈ *carrier R*
assumes *c* ∈ *carrier R*
shows *to-fun* (*c* ⊙$_P$ *f*) *b* = *c* ⊗(*to-fun f b*)
proof−

have $(c \odot_P f) = (\textit{to-poly}\ c) \otimes_P f$
by (*metis P-def assms*(*1*) *assms*(*3*) *monom-mult-is-smult to-polynomial-def*)
then have *to-fun* $(c \odot_P f)\ b =$ *to-fun* $(\textit{to-poly}\ c)\ b \otimes$ *to-fun f b*
by (*simp add*: *assms*(*1*) *assms*(*2*) *assms*(*3*) *to-fun-mult to-poly-closed*)
then show *?thesis*
by (*simp add*: *assms*(*2*) *assms*(*3*) *to-fun-to-poly*)
qed

lemma *to-fun-monom*:
assumes $c \in$ *carrier R*
assumes $x \in$ *carrier R*
shows *to-fun* $(\textit{monom}\ P\ c\ n)\ x = c \otimes x\ [\hat{}]\ n$
by (*smt* (*verit*) *P-def R.m-comm R.nat-pow-closed UP-cring.to-poly-nat-pow UP-cring-axioms assms*(*1*)
assms(*2*) *monom-is-UP-monom*(*1*) *sub-monom*(*1*) *to-fun-smult to-fun-sub-monom to-fun-to-poly*
to-poly-closed to-poly-mult-simp(*2*))

lemma *zcf-monom*:
assumes $a \in$ *carrier R*
shows *zcf* $(\textit{monom}\ P\ a\ n) =$ *to-fun* $(\textit{monom}\ P\ a\ n)\ \mathbf{0}$
using *to-fun-monom* **unfolding** *zcf-def*
by (*simp add*: *R.nat-pow-zero assms cfs-monom*)

lemma *zcf-to-fun*:
assumes $p \in$ *carrier P*
shows *zcf p* = *to-fun p* $\mathbf{0}$
apply(*rule poly-induct3*[*of p*])
apply (*simp add*: *assms*)
using *R.zero-closed zcf-add to-fun-plus* **apply** *presburger*
using *zcf-monom* **by** *blast*

lemma *zcf-to-poly*[*simp*]:
assumes $a \in$ *carrier R*
shows *zcf* $(\textit{to-poly}\ a) = a$
by (*metis assms cfs-closed degree-to-poly to-fun-to-poly to-poly-inverse to-poly-closed zcf-def*)

lemma *zcf-ltrm-mult*:
assumes $p \in$ *carrier P*
assumes $q \in$ *carrier P*
assumes *degree p* > *0*
shows $\textit{zcf}((\textit{ltrm}\ p) \otimes_P q) = \mathbf{0}$
using *zcf-to-fun*[*of ltrm p* $\otimes_P$ *q*]
by (*metis ltrm-closed P.l-null P.m-closed R.zero-closed UP-zero-closed zcf-to-fun*
zcf-zero assms(*1*) *assms*(*2*) *assms*(*3*) *coeff-ltrm to-fun-mult*)

lemma *zcf-mult*:
assumes $p \in$ *carrier P*

assumes $q \in$ *carrier* P
shows *zcf*$(p \otimes_P q) = ($*zcf* $p) \otimes ($*zcf* $q)$
using *zcf-to-fun*[*of* $p \otimes_P q$] *zcf-to-fun*[*of* p] *zcf-to-fun*[*of* q] *to-fun-mult*[*of* q p **0**]
by (*simp add*: *assms*(*1*) *assms*(*2*))

lemma *zcf-is-ring-hom*:
zcf$\in$ *ring-hom* P R
apply(*rule ring-hom-memI*)
using *zcf-mult zcf-add*
apply (*simp add*: *P-def UP-ring.cfs-closed UP-ring-axioms zcf-def*)
apply (*simp add*: *zcf-mult*)
using *zcf-add* **apply** *auto*[*1*]
by *simp*

lemma *ctrm-is-ring-hom*:
ctrm $\in$ *ring-hom* P P
apply(*rule ring-hom-memI*)
apply (*simp add*: *ctrm-is-poly*)
apply (*metis zcf-def zcf-mult cfs-closed monom-mult zero-eq-add-iff-both-eq-0*)

using *cfs-add*[*of - - 0*]
apply (*simp add*: *cfs-closed*)
by *auto*

6 Describing the Image of (UP R) in the Ring of Functions from R to R

lemma *to-fun-diff*:
assumes $p \in$ *carrier* P
assumes $q \in$ *carrier* P
assumes $a \in$ *carrier* R
shows *to-fun* $(p \ominus_P q)$ $a =$ *to-fun* p $a \ominus$ *to-fun* q a
using *to-fun-plus*[*of* $\ominus_P$ q p a]
by (*simp add*: *P.minus-eq R.minus-eq assms*(*1*) *assms*(*2*) *assms*(*3*) *to-fun-minus*)

lemma *to-fun-const*:
assumes $a \in$ *carrier* R
assumes $b \in$ *carrier* R
shows *to-fun* (*monom* P a *0*) $b = a$
by (*metis lcf-monom*(*2*) *P-def UP-cring.to-fun-ctrm UP-cring-axioms assms*(*1*) *assms*(*2*) *deg-const monom-closed*)

lemma *to-fun-monic-monom*:
assumes $b \in$ *carrier* R
shows *to-fun* (*monom* P **1** n) $b = b[\hat{}]n$
by (*simp add*: *assms to-fun-monom*)

Constant polynomials map to constant polynomials

lemma *const-to-constant*:
assumes $a \in$ *carrier R*
shows *to-fun* (*monom P a 0*) = *constant-function* (*carrier R*) *a*
apply(*rule ring-functions.function-ring-car-eqI*[*of R - carrier R*])
unfolding *ring-functions-def* **apply**(*simp add*: *R.ring-axioms*)
apply (*simp add*: *assms to-fun-is-Fun*)
using *assms ring-functions.constant-function-closed*[*of R a carrier R*]
unfolding *ring-functions-def* **apply** (*simp add*: *R.ring-axioms*)
using *assms to-fun-const*[*of a*] **unfolding** *constant-function-def*
by *auto*

Monomial polynomials map to monomial functions

lemma *monom-to-monomial*:
assumes $a \in$ *carrier R*
shows *to-fun* (*monom P a n*) = *monomial-function R a n*
apply(*rule ring-functions.function-ring-car-eqI*[*of R - carrier R*])
unfolding *ring-functions-def* **apply**(*simp add*: *R.ring-axioms*)
apply (*simp add*: *assms to-fun-is-Fun*)
using *assms U-function-ring.monomial-functions*[*of R a n*] *R.ring-axioms*
unfolding *U-function-ring-def*
apply *auto*[*1*]
unfolding *monomial-function-def*
using *assms to-fun-monom*[*of a - n*]
by *auto*
end

7 Taylor Expansions

7.1 Monic Linear Polynomials

The polynomial representing the variable X

definition *X-poly* **where**
X-poly R = *monom* (*UP R*) $\mathbf{1}_R$ *1*

context *UP-cring*
begin

abbreviation(*input*) *X* **where**
$X \equiv$ *X-poly R*

lemma *X-closed*:
$X \in$ *carrier P*
unfolding *X-poly-def*
using *P-def monom-closed* **by** *blast*

lemma *degree-X*[*simp*]:
assumes 1 $\neq$**0**
shows*degree X* = *1*

unfolding *X-poly-def*
using *assms P-def deg-monom*[*of* **1** *1*]
by *blast*

lemma *X-not-zero*:
assumes 1 $\neq$**0**
shows$X \neq \mathbf{0}_P$
using *degree-X assms* **by** *force*

lemma *sub-X*[*simp*]:
assumes $p \in$ *carrier P*
shows *X of p* = *p*
unfolding *X-poly-def*
using *P-def UP-pre-univ-prop.eval-monom1 assms compose-def to-poly-UP-pre-univ-prop*

by *metis*

lemma *sub-monom-deg-one*:
assumes $p \in$ *carrier P*
assumes $a \in$ *carrier R*
shows *monom P a 1 of p* = $a \odot_P p$
using *assms sub-smult*[*of X p a*] **unfolding** *X-poly-def*
by (*metis P-def R.one-closed R.r-one X-closed X-poly-def monom-mult-smult sub-X*)

lemma *monom-rep-X-pow*:
assumes $a \in$ *carrier R*
shows *monom P a n* = $a \odot_P (X[\hat{}]_P n)$
proof−
have *monom P a n* = $a \odot_P$*monom P* **1** *n*
by (*metis R.one-closed R.r-one assms monom-mult-smult*)
then show *?thesis*
unfolding *X-poly-def*
using *monom-pow*
by (*simp add*: *P-def*)
qed

lemma *X-sub*[*simp*]:
assumes $p \in$ *carrier P*
shows *p of X* = *p*
apply(*rule poly-induct3*)
apply (*simp add*: *assms*)
using *X-closed sub-add* **apply** *presburger*
using *sub-monom*[*of X*] *P-def monom-rep-X-pow X-closed* **by** *auto*

representation of monomials as scalar multiples of powers of X

lemma *ltrm-rep-X-pow*:
assumes $p \in$ *carrier P*
shows *ltrm p* = $(lcf\ p) \odot_P (X[\hat{}]_P (degree\ p))$

proof−
have *ltrm p* = *monom P* (*lcf p*) (*degree p*)
using *assms* **unfolding** *leading-term-def* **by** (*simp add*: *P-def*)
then show *?thesis*
using *monom-rep-X-pow P-def assms*
by (*simp add*: *cfs-closed*)
qed

lemma *to-fun-monom′*:
assumes *c* ∈ *carrier R*
assumes *c* ≠**0**
assumes *x* ∈ *carrier R*
shows *to-fun* (*c* ⊙$_P$ *X*[^]$_P$(*n*::*nat*)) *x* = *c* ⊗ *x* [^] *n*
using *P-def to-fun-monom monom-rep-X-pow UP-cring-axioms assms*(*1*) *assms*(*2*) *assms*(*3*) **by** *fastforce*

lemma *to-fun-X-pow*:
assumes *x* ∈ *carrier R*
shows *to-fun* (*X*[^]$_P$(*n*::*nat*)) *x* = *x* [^] *n*
using *to-fun-monom′*[*of* **1** *x n*] *assms*
by (*metis P.nat-pow-closed R.l-one R.nat-pow-closed R.one-closed R.r-null R.r-one*

UP-one-closed X-closed to-fun-to-poly ring-hom-one smult-l-null smult-one to-poly-is-ring-hom)
end

Monic linear polynomials

definition *X-poly-plus* **where**
X-poly-plus R a = (*X-poly R*) ⊕$_{(UP\ R)}$ *to-polynomial R a*

definition *X-poly-minus* **where**
X-poly-minus R a = (*X-poly R*) ⊖$_{(UP\ R)}$ *to-polynomial R a*

context *UP-cring*
begin

abbreviation(*input*) *X-plus* **where**
X-plus ≡ *X-poly-plus R*

abbreviation(*input*) *X-minus* **where**
X-minus ≡ *X-poly-minus R*

lemma *X-plus-closed*:
assumes *a* ∈ *carrier R*
shows (*X-plus a*) ∈ *carrier P*
unfolding *X-poly-plus-def* **using** *X-closed to-poly-closed*
using *P-def UP-a-closed assms* **by** *auto*

lemma *X-minus-closed*:

assumes $a \in$ *carrier* R
shows (X-*minus* a) $\in$ *carrier* P
unfolding X-*poly-minus-def* **using** X-*closed to-poly-closed*
by (*simp add*: P-*def UP-cring.UP-cring UP-cring-axioms assms cring.cring-simprules*(4))

lemma X-*minus-plus*:
assumes $a \in$ *carrier* R
shows (X-*minus* a) = X-*plus* ($\ominus a$)
using P-*def UP-ring.UP-ring UP-ring-axioms*
by (*simp add*: X-*poly-minus-def X-poly-plus-def a-minus-def assms to-poly-a-inv*)

lemma *degree-of-X-plus*:
assumes $a \in$ *carrier* R
assumes 1 $\neq$**0**
shows *degree* (X-*plus* a) = *1*
proof−
have *0*:*degree* (X-*plus* a) $\leq$ *1*
using *deg-add degree-X P-def* **unfolding** X-*poly-plus-def*
using *UP-cring.to-poly-closed UP-cring-axioms X-closed assms*(*1*) *assms*(*2*)
by *fastforce*
have *1*:*degree* (X-*plus* a) > *0*
by (*metis One-nat-def P-def R.one-closed R.r-zero X-poly-def*
X-closed X-poly-plus-def X-plus-closed assms coeff-add coeff-monom deg-aboveD
gr0I lessI n-not-Suc-n to-polynomial-def to-poly-closed)
then show *?thesis*
using *0* **by** *linarith*
qed

lemma *degree-of-X-minus*:
assumes $a \in$ *carrier* R
assumes 1 $\neq$**0**
shows *degree* (X-*minus* a) = *1*
using *degree-of-X-plus*[*of* $\ominus a$] X-*minus-plus*[*simp*] *assms* **by** *auto*

lemma *ltrm-of-X*:
shows*ltrm* $X = X$
unfolding *leading-term-def*
by (*metis P-def R.one-closed X-poly-def is-UP-monom-def is-UP-monomI leading-term-def*)

lemma *ltrm-of-X-plus*:
assumes $a \in$ *carrier* R
assumes 1 $\neq$**0**
shows *ltrm* (X-*plus* a) = X
unfolding X-*poly-plus-def*
using X-*closed assms ltrm-of-sum-diff-degree*[*of X to-poly a*]
degree-to-poly[*of a*] *to-poly-closed*[*of a*] *degree-X ltrm-of-X*

by (*simp add*: *P-def*)

lemma *ltrm-of-X-minus*:
assumes $a \in$ *carrier R*
assumes $\mathbf{1} \neq \mathbf{0}$
shows *ltrm* (*X-minus a*) = *X*
using *X-minus-plus*[*of a*] *assms*
by (*simp add*: *ltrm-of-X-plus*)

lemma *lcf-of-X-minus*:
assumes $a \in$ *carrier R*
assumes $\mathbf{1} \neq \mathbf{0}$
shows *lcf* (*X-minus a*) = **1**
using *ltrm-of-X-minus* **unfolding** *X-poly-def*
using *P-def UP-cring.X-minus-closed UP-cring.lcf-eq UP-cring-axioms assms*(*1*) *assms*(*2*) *lcf-monom*
by (*metis R.one-closed*)

lemma *lcf-of-X-plus*:
assumes $a \in$ *carrier R*
assumes $\mathbf{1} \neq \mathbf{0}$
shows *lcf* (*X-plus a*) = **1**
using *ltrm-of-X-plus* **unfolding** *X-poly-def*
by (*metis lcf-of-X-minus P-def UP-cring.lcf-eq UP-cring.X-plus-closed UP-cring-axioms X-minus-closed assms*(*1*) *assms*(*2*) *degree-of-X-minus*)

lemma *to-fun-X*[*simp*]:
assumes $a \in$ *carrier R*
shows *to-fun X a* = *a*
using *X-closed assms to-fun-sub-monom ltrm-is-UP-monom ltrm-of-X to-poly-closed*

by (*metis sub-X to-fun-to-poly*)

lemma *to-fun-X-plus*[*simp*]:
assumes $a \in$ *carrier R*
assumes $b \in$ *carrier R*
shows *to-fun* (*X-plus a*) *b* = $b \oplus a$
unfolding *X-poly-plus-def*
using *assms to-fun-X*[*of b*] *to-fun-plus*[*of to-poly a X b*] *to-fun-to-poly*[*of a b*]
using *P-def X-closed to-poly-closed* **by** *auto*

lemma *to-fun-X-minus*[*simp*]:
assumes $a \in$ *carrier R*
assumes $b \in$ *carrier R*
shows *to-fun* (*X-minus a*) *b* = $b \ominus a$
using *to-fun-X-plus*[*of* $\ominus$ *a b*] *X-minus-plus*[*of a*] *assms*
by (*simp add*: *R.minus-eq*)

lemma *cfs-X-plus*:

assumes $a \in$ *carrier* R
shows *X-plus a n* = (*if n* = *0 then a else* (*if n* = *1 then* **1** *else* **0**))
using *assms cfs-add monom-closed UP-ring-axioms cfs-monom*
unfolding *X-poly-plus-def to-polynomial-def X-poly-def P-def*
by *auto*

lemma *cfs-X-minus*:
assumes $a \in$ *carrier* R
shows *X-minus a n* = (*if n* = *0 then* $\ominus$ *a else* (*if n* = *1 then* **1** *else* **0**))
using *cfs-X-plus*[*of* $\ominus$ *a*] *assms*
unfolding *X-poly-plus-def X-poly-minus-def*
by (*simp add*: *P-def a-minus-def to-poly-a-inv*)

Linear substituions

lemma *X-plus-sub-deg*:
assumes $a \in$ *carrier* R
assumes $f \in$ *carrier* P
shows *degree* (*f of* (*X-plus a*)) = *degree f*
apply(*cases* **1** = **0**)
apply (*metis P-def UP-one-closed X-plus-closed X-poly-def sub-X assms*(*1*) *assms*(*2*) *deg-one monom-one monom-zero sub-const*)
using *cring-sub-deg*[*of X-plus a f*] *assms X-plus-closed*[*of a*] *lcf-of-X-plus*[*of a*] *ltrm-of-X-plus degree-of-X-plus*[*of a*] *P-def*
by (*metis lcf-eq R.nat-pow-one R.r-one UP-cring.cring-sub-deg UP-cring-axioms X-closed X-sub cfs-closed coeff-simp deg-nzero-nzero degree-X lcoeff-nonzero2 sub-const*)

lemma *X-minus-sub-deg*:
assumes $a \in$ *carrier* R
assumes $f \in$ *carrier* P
shows *degree* (*f of* (*X-minus a*)) = *degree f*
using *X-plus-sub-deg*[*of* $\ominus a$] *assms X-minus-plus*[*of a*]
by *simp*

lemma *plus-minus-sub*:
assumes $a \in$ *carrier* R
shows *X-plus a of X-minus a* = *X*
unfolding *X-poly-plus-def*
proof−
have (*X* $\oplus_P$ *to-poly a*) *of X-minus a* = (*X of X-minus a*) $\oplus_P$ (*to-poly a*) *of X-minus a*
using *sub-add*
by (*simp add*: *X-closed X-minus-closed assms to-poly-closed*)
then have (*X* $\oplus_P$ *to-poly a*) *of X-minus a* = (*X-minus a*) $\oplus_P$ (*to-poly a*)
by (*simp add*: *X-minus-closed assms sub-to-poly*)
then show (*X* $\oplus_{UP\ R}$ *to-poly a*) *of X-minus a* = *X*
unfolding *to-polynomial-def X-poly-minus-def*
by (*metis P.add.inv-solve-right P.minus-eq P-def X-closed X-poly-minus-def X-minus-closed assms monom-closed to-polynomial-def*)

qed

lemma *minus-plus-sub*:
assumes $a \in$ *carrier R*
shows *X-minus a of X-plus a* $= X$
using *plus-minus-sub*[*of* $\ominus a$]
unfolding *X-poly-minus-def*
unfolding *X-poly-plus-def*
using *assms* **apply** *simp*
by (*metis P-def R.add.inv-closed R.minus-minus a-minus-def to-poly-a-inv*)

lemma *ltrm-times-X*:
assumes $p \in$ *carrier P*
shows *ltrm* $(X \otimes_P p) = X \otimes_P$ (*ltrm p*)
using *assms ltrm-of-X cring-ltrm-mult*[*of X p*]
by (*metis ltrm-deg-0 P.r-null R.l-one R.one-closed UP-cring.lcf-monom*(*1*)
UP-cring-axioms X-closed X-poly-def cfs-closed deg-zero deg-ltrm monom-zero)

lemma *times-X-not-zero*:
assumes $p \in$ *carrier P*
assumes $p \neq \mathbf{0}_P$
shows $(X \otimes_P p) \neq \mathbf{0}_P$
by (*metis* (*no-types*, *opaque-lifting*) *lcf-monom*(*1*) *lcf-of-X-minus ltrm-of-X-minus P.inv-unique*
P.r-null R.l-one R.one-closed UP-zero-closed X-closed zcf-def
zcf-zero-degree-zero assms(*1*) *assms*(*2*) *cfs-closed cfs-zero cring-lcf-mult*
deg-monom deg-nzero-nzero deg-ltrm degree-X degree-of-X-minus
monom-one monom-zero)

lemma *degree-times-X*:
assumes $p \in$ *carrier P*
assumes $p \neq \mathbf{0}_P$
shows *degree* $(X \otimes_P p) =$ *degree p* $+ 1$
using *cring-deg-mult*[*of X p*] *assms times-X-not-zero*[*of p*]
by (*metis* (*no-types*, *lifting*) *P.r-null P.r-one P-def R.l-one R.one-closed*
UP-cring.lcf-monom(*1*) *UP-cring-axioms UP-zero-closed X-closed X-poly-def cfs-closed*
deg-zero deg-ltrm degree-X monom-one monom-zero to-poly-inverse)

end

7.2 Basic Facts About Taylor Expansions

definition *taylor-expansion* **where**
taylor-expansion R a p $=$ *compose R p* (*X-poly-plus R a*)

definition(**in** *UP-cring*) *taylor* **where**
taylor $\equiv$ *taylor-expansion R*

context *UP-cring*
begin

lemma *taylor-expansion-ring-hom*:
assumes $c \in$ *carrier R*
shows *taylor-expansion R c* $\in$ *ring-hom P P*
unfolding *taylor-expansion-def*
using *rev-sub-is-hom*[*of X-plus c*]
unfolding *rev-compose-def compose-def*
using *X-plus-closed assms* **by** *auto*

notation *taylor* ($\langle T\_ \rangle$)

lemma(**in** *UP-cring*) *taylor-closed*:
assumes $f \in$ *carrier P*
assumes $a \in$ *carrier R*
shows $T_a\ f \in$ *carrier P*
unfolding *taylor-def*
by (*simp add*: *X-plus-closed assms*(*1*) *assms*(*2*) *sub-closed taylor-expansion-def*)

lemma *taylor-deg*:
assumes $a \in$ *carrier R*
assumes $p \in$ *carrier P*
shows *degree* (T_a *p*) = *degree p*
unfolding *taylor-def taylor-expansion-def*
using *X-plus-sub-deg*[*of a p*] *assms*
by (*simp add*: *taylor-expansion-def*)

lemma *taylor-id*:
assumes $a \in$ *carrier R*
assumes $p \in$ *carrier P*
shows *p* = (T_a *p*) *of* (*X-minus a*)
unfolding *taylor-expansion-def taylor-def*
using *assms sub-assoc*[*of p X-plus a X-minus a*] *X-plus-closed*[*of a*] *X-minus-closed*[*of a*]
by (*metis X-sub plus-minus-sub taylor-expansion-def*)

lemma *taylor-eval*:
assumes $a \in$ *carrier R*
assumes $f \in$ *carrier P*
assumes $b \in$ *carrier R*
shows *to-fun* (T_a *f*) *b* = *to-fun f* (*b* $\oplus$ *a*)
unfolding *taylor-expansion-def taylor-def*
using *to-fun-sub*[*of* (*X-plus a*) *f b*] *to-fun-X-plus*[*of a b*]
assms X-plus-closed[*of a*] **by** *auto*

lemma *taylor-eval'*:
assumes $a \in$ *carrier R*
assumes $f \in$ *carrier P*

assumes $b \in$ *carrier* R
shows *to-fun* f (b) = *to-fun* $(T_a\ f)$ $(b \ominus a)$
unfolding *taylor-expansion-def taylor-def*
using *to-fun-sub*[*of* (*X-minus* a) $T_a\ f\ b$] *to-fun-X-minus*[*of* b a]
assms X-minus-closed[*of* a]
by (*metis taylor-closed taylor-def taylor-id taylor-expansion-def to-fun-X-minus*)

lemma(**in** *UP-cring*) *degree-monom*:
assumes $a \in$ *carrier* R
shows *degree* $(a \odot_{UP\ R}$ (*X-poly* R)[^]$_{UP\ R} n) =$ (*if* $a = \mathbf{0}$ *then* 0 *else* n)
apply(*cases* $a = \mathbf{0}$)
apply (*metis* (*full-types*) *P.nat-pow-closed P-def R.one-closed UP-smult-zero X-poly-def deg-zero monom-closed*)
using *P-def UP-cring.monom-rep-X-pow UP-cring-axioms assms deg-monom* **by** *fastforce*

lemma(**in** *UP-cring*) *poly-comp-finsum*:
assumes $\bigwedge i$::*nat*. $i \leq n \Longrightarrow g\ i \in$ *carrier* P
assumes $q \in$ *carrier* P
assumes $p = (\bigoplus_P i \in \{..n\}.\ g\ i)$
shows p *of* $q = (\bigoplus_P i \in \{..n\}.\ (g\ i)$ *of* $q)$
proof−
have *0*: p *of* q = *rev-sub* $q\ p$
unfolding *compose-def rev-compose-def* **by** *blast*
have *1*: p *of* q = *finsum* P (*rev-compose* $R\ q \circ g$) $\{..n\}$
unfolding *0* **unfolding** *assms*
apply(*rule ring-hom-finsum*[*of rev-compose* $R\ q\ P\ \{..n\}\ g$])
using *assms*(*2*) *rev-sub-is-hom* **apply** *blast*
apply (*simp add*: *UP-ring*)
apply *simp*
by (*simp add*: *assms*(*1*))
show *?thesis* **unfolding** *1*
unfolding *comp-apply rev-compose-def compose-def*
by *auto*
qed

lemma(**in** *UP-cring*) *poly-comp-expansion*:
assumes $p \in$ *carrier* P
assumes $q \in$ *carrier* P
assumes *degree* $p \leq n$
shows p *of* $q = (\bigoplus_P i \in \{..n\}.\ (p\ i) \odot_P q[\hat{}]_P i)$
proof−
obtain g **where** *g-def*: $g = (\lambda i.$ *monom* $P\ (p\ i)\ i)$
by *blast*
have *0*: $\bigwedge i.\ (g\ i)$ *of* $q = (p\ i) \odot_P q[\hat{}]_P i$
proof− **fix** i **show** $g\ i$ *of* $q = p\ i \odot_P q\ [\hat{}]_P\ i$
using *assms g-def P-def coeff-simp monom-sub*
by (*simp add*: *cfs-closed*)
qed

have *1*: $(\bigwedge i.\ i \leq n \Longrightarrow g\ i \in carrier\ P)$
using *g-def assms*
by (*simp add*: *cfs-closed*)
have $(\bigoplus_P i \in \{..n\}.\ monom\ P\ (p\ i)\ i) = p$
using *assms up-repr-le*[*of p n*] *coeff-simp*[*of p*] **unfolding** *P-def*
by *auto*
then have $p = (\bigoplus_P i \in \{..n\}.\ g\ i)$
using *g-def* **by** *auto*
then have $p\ of\ q = (\bigoplus_P i \in \{..n\}.\ g\ i\ of\ q)$
using *0 1 poly-comp-finsum*[*of n g q p*]
using *assms*(*2*)
by *blast*
then show *?thesis*
by(*simp add*: *0*)
qed

lemma(**in** *UP-cring*) *taylor-sum*:
assumes $p \in carrier\ P$
assumes $degree\ p \leq n$
assumes $a \in carrier\ R$
shows $p = (\bigoplus_P i \in \{..n\}.\ T_a\ p\ i \odot_P (X\text{-}minus\ a)[\hat{}]_P i)$
proof−
have *0*: $(T_a\ p)\ of\ X\text{-}minus\ a = p$
using *P-def taylor-id assms*(*1*) *assms*(*3*)
by *fastforce*
have *1*: $degree\ (T_a\ p) \leq n$
using *assms*
by (*simp add*: *taylor-deg*)
have *2*: $T_a\ p\ of\ X\text{-}minus\ a = (\bigoplus_P i \in \{..n\}.\ T_a\ p\ i \odot_P X\text{-}minus\ a\ [\hat{}]_P\ i)$
using *1 X-minus-closed*[*of a*] *poly-comp-expansion*[*of* T_a *p X-minus a n*]
assms taylor-closed
by *blast*
then show *?thesis*
using *0*
by *simp*
qed

The i^{th} term in the taylor expansion

definition *taylor-term* **where**
$taylor\text{-}term\ c\ p\ i = (taylor\text{-}expansion\ R\ c\ p\ i) \odot_{UP\ R} (UP\text{-}cring.X\text{-}minus\ R\ c)[\hat{}]_{UP\ R} i$

lemma (**in** *UP-cring*) *taylor-term-closed*:
assumes $p \in carrier\ P$
assumes $a \in carrier\ R$
shows $taylor\text{-}term\ a\ p\ i \in carrier\ (UP\ R)$
unfolding *taylor-term-def*
using *P.nat-pow-closed P-def taylor-closed taylor-def X-minus-closed assms*(*1*) *assms*(*2*) *smult-closed*

by (*simp add*: *cfs-closed*)

lemma(**in** *UP-cring*) *taylor-term-sum*:
assumes $p \in carrier\ P$
assumes $degree\ p \leq n$
assumes $a \in carrier\ R$
shows $p = (\bigoplus_P\ i \in \{..n\}.\ taylor\text{-}term\ a\ p\ i)$
unfolding *taylor-term-def taylor-def*
using *assms taylor-sum*[*of p n a*] *P-def*
using *taylor-def* **by** *auto*

lemma (**in** *UP-cring*) *taylor-expansion-add*:
assumes $p \in carrier\ P$
assumes $q \in carrier\ P$
assumes $c \in carrier\ R$
shows $taylor\text{-}expansion\ R\ c\ (p \oplus_{UP\ R}\ q) = (taylor\text{-}expansion\ R\ c\ p) \oplus_{UP\ R} (taylor\text{-}expansion\ R\ c\ q)$
unfolding *taylor-expansion-def*
using *assms X-plus-closed*[*of c*] *P-def sub-add*
by *blast*

lemma (**in** *UP-cring*) *taylor-term-add*:
assumes $p \in carrier\ P$
assumes $q \in carrier\ P$
assumes $a \in carrier\ R$
shows $taylor\text{-}term\ a\ (p \oplus_{UP\ R} q)\ i = taylor\text{-}term\ a\ p\ i \oplus_{UP\ R} taylor\text{-}term\ a\ q\ i$
using *assms taylor-expansion-add*[*of p q a*]
unfolding *taylor-term-def*
using *P.nat-pow-closed P-def taylor-closed X-minus-closed cfs-add smult-l-distr*
by (*simp add*: *taylor-def cfs-closed*)

lemma (**in** *UP-cring*) *to-fun-taylor-term*:
assumes $p \in carrier\ P$
assumes $a \in carrier\ R$
assumes $c \in carrier\ R$
shows $to\text{-}fun\ (taylor\text{-}term\ c\ p\ i)\ a = (T_c\ p\ i) \otimes (a \ominus c)[\hat{}]i$
using *assms to-fun-smult*[*of X-minus c* $[\hat{}]_{UP\ R}$ *i a taylor-expansion R c p i*]
to-fun-X-minus[*of c a*] *to-fun-nat-pow*[*of X-minus c a i*]
unfolding *taylor-term-def*
using *P.nat-pow-closed P-def taylor-closed taylor-def X-minus-closed*
by (*simp add*: *cfs-closed*)

end

7.3 Defining the (Scalar-Valued) Derivative of a Polynomial Using the Taylor Expansion

definition *derivative* **where**

derivative R f a = (taylor-expansion R a f) 1

context *UP-cring*
begin

abbreviation(**in** *UP-cring*) *deriv* **where**
deriv ≡ derivative R

lemma(**in** *UP-cring*) *deriv-closed*:
assumes *f ∈ carrier P*
assumes *a ∈ carrier R*
shows *(derivative R f a) ∈ carrier R*
unfolding *derivative-def*
using *taylor-closed taylor-def assms(1) assms(2) cfs-closed* **by** *auto*

lemma(**in** *UP-cring*) *deriv-add*:
assumes *f ∈ carrier P*
assumes *g ∈ carrier P*
assumes *a ∈ carrier R*
shows *deriv (f ⊕$_P$ g) a = deriv f a ⊕ deriv g a*
unfolding *derivative-def taylor-expansion-def* **using** *assms*
by (*simp add*: *X-plus-closed sub-add sub-closed*)

end

8 The Polynomial-Valued Derivative Operator

context *UP-cring*
begin

8.1 Operator Which Shifts Coefficients

lemma *cfs-times-X*:
assumes *g ∈ carrier P*
shows *(X ⊗$_P$ g) (Suc n) = g n*
apply(*rule poly-induct3*[*of g*])
apply (*simp add*: *assms*)
apply (*metis* (*no-types*, *lifting*) *P.m-closed P.r-distr X-closed cfs-add*)
by (*metis* (*no-types*, *lifting*) *P-def R.l-one R.one-closed R.r-null Suc-eq-plus1 X-poly-def*
cfs-monom coeff-monom-mult coeff-simp monom-closed monom-mult)

lemma *times-X-pow-coeff*:
assumes *g ∈ carrier P*
shows *(monom P* **1** *k ⊗$_P$ g) (n + k) = g n*
using *coeff-monom-mult P.m-closed P-def assms coeff-simp monom-closed*
by (*simp add*: *cfs-closed*)

lemma *zcf-eq-zero-unique*:

assumes $f \in$ *carrier* P
assumes $g \in$ *carrier* $P \wedge (f = X \otimes_P g)$
shows $\bigwedge h.\ h \in$ *carrier* $P \wedge (f = X \otimes_P h) \Longrightarrow h = g$
proof−
fix h
assume *A*: $h \in$ *carrier* $P \wedge (f = X \otimes_P h)$
then have *0*: $X \otimes_P g = X \otimes_P h$
using *assms*(*2*) **by** *auto*
show $h = g$
using *0 A assms*
by (*metis P-def coeff-simp cfs-times-X up-eqI*)
qed

lemma *f-minus-ctrm*:
assumes $f \in$ *carrier* P
shows *zcf*$(f \ominus_P$ *ctrm* $f) = \mathbf{0}$
using *assms*
by (*smt* (*verit*) *ctrm-is-poly P.add.inv-closed P.minus-closed P-def R.r-neg R.zero-closed zcf-to-fun*
to-fun-minus to-fun-plus UP-cring-axioms zcf-ctrm zcf-def a-minus-def cfs-closed)

definition *poly-shift* **where**
poly-shift f n = *f* (*Suc n*)

lemma *poly-shift-closed*:
assumes $f \in$ *carrier* P
shows *poly-shift f* ∈ *carrier P*
apply(*rule UP-car-memI*[*of deg R f*])
unfolding *poly-shift-def*
proof −
fix *n* :: *nat*
assume *deg R f* < *n*
then have *deg R f* < *Suc n*
using *Suc-lessD* **by** *blast*
then have *f* (*Suc n*) = $\mathbf{0}_P$ (*Suc n*)
by (*metis P.l-zero UP-zero-closed assms coeff-of-sum-diff-degree0*)
then show *f* (*Suc n*) = **0**
by *simp*
next
show ⋀*n*. *f* (*Suc n*) ∈ *carrier R*
by(*rule cfs-closed*, *rule assms*)
qed

lemma *poly-shift-eq-0*:
assumes $f \in$ *carrier* P
shows *f n* = (*ctrm f* ⊕$_P$ *X* ⊗$_P$ *poly-shift f*) *n*
apply(*cases n* = *0*)
apply (*smt* (*verit*) *ctrm-degree ctrm-is-poly ltrm-of-X One-nat-def P.r-null P.r-zero*

P-def UP-cring.lcf-monom(1) UP-cring-axioms UP-mult-closed UP-r-one UP-zero-closed X-closed zcf-ltrm-mult zcf-def zcf-zero assms cfs-add cfs-closed deg-zero degree-X lessI monom-one poly-shift-closed to-poly-inverse)
proof− **assume** *A*: $n \neq 0$
then obtain *k* **where** *k-def*: $n = Suc\ k$
by (*meson lessI less-Suc-eq-0-disj*)
show *?thesis*
using *cfs-times-X*[*of poly-shift f k*] *poly-shift-def*[*of f k*] *poly-shift-closed assms cfs-add*[*of ctrm f X* $\otimes_P$ *poly-shift f n*] **unfolding** *k-def*
by (*simp add*: *X-closed cfs-closed cfs-monom*)
qed

lemma *poly-shift-eq*:
assumes $f \in carrier\ P$
shows $f = (ctrm\ f \oplus_P X \otimes_P poly\text{-}shift\ f)$
by(*rule ext*, *rule poly-shift-eq-0*, *rule assms*)

lemma *poly-shift-id*:
assumes $f \in carrier\ P$
shows $f \ominus_P ctrm\ f = X \otimes_P poly\text{-}shift\ f$
using *assms poly-shift-eq*[*of f*] *poly-shift-closed* **unfolding** *a-minus-def*
by (*metis ctrm-is-poly P.add.inv-solve-left P.m-closed UP-a-comm UP-a-inv-closed X-closed*)

lemma *poly-shift-degree-zero*:
assumes $p \in carrier\ P$
assumes $degree\ p = 0$
shows $poly\text{-}shift\ p = \mathbf{0}_P$
by (*metis ltrm-deg-0 P.r-neg P.r-null UP-ring UP-zero-closed X-closed zcf-eq-zero-unique*

abelian-group.minus-eq assms(*1*) *assms*(*2*) *poly-shift-closed poly-shift-id ring-def*)

lemma *poly-shift-degree*:
assumes $p \in carrier\ P$
assumes $degree\ p > 0$
shows $degree\ (poly\text{-}shift\ p) = degree\ p - 1$
using *poly-shift-id*[*of p*]
by (*metis ctrm-degree ctrm-is-poly P.r-null X-closed add-diff-cancel-right' assms*(*1*) *assms*(*2*)
deg-zero degree-of-difference-diff-degree degree-times-X nat-less-le poly-shift-closed)

lemma *poly-shift-monom*:
assumes $a \in carrier\ R$
shows $poly\text{-}shift\ (monom\ P\ a\ (Suc\ k)) = (monom\ P\ a\ k)$
proof−
have $(monom\ P\ a\ (Suc\ k)) = ctrm\ (monom\ P\ a\ (Suc\ k)) \oplus_P X \otimes_P poly\text{-}shift\ (monom\ P\ a\ (Suc\ k))$
using *poly-shift-eq*[*of monom P a* (*Suc k*)] *assms monom-closed*
by *blast*

then have (*monom P a* (*Suc k*)) = $\mathbf{0}_P \oplus_P X \otimes_P$*poly-shift* (*monom P a* (*Suc k*))
using *assms* **by** *simp*
then have (*monom P a* (*Suc k*)) = $X \otimes_P$*poly-shift* (*monom P a* (*Suc k*))
using *X-closed assms poly-shift-closed* **by** *auto*
then have $X \otimes_P$(*monom P a k*) = $X \otimes_P$*poly-shift* (*monom P a* (*Suc k*))
by (*metis P-def R.l-one R.one-closed X-poly-def assms monom-mult plus-1-eq-Suc*)
then show *?thesis*
using *X-closed X-not-zero assms*
by (*meson UP-mult-closed zcf-eq-zero-unique monom-closed poly-shift-closed*)

qed

lemma(**in** *UP-cring*) *poly-shift-add*:
assumes $f \in$ *carrier P*
assumes $g \in$ *carrier P*
shows *poly-shift* ($f \oplus_P g$) = (*poly-shift f*) $\oplus_P$ (*poly-shift g*)
apply(*rule ext*)
using *cfs-add*[*of poly-shift f poly-shift g*] *poly-shift-closed poly-shift-def*
by (*simp add*: *poly-shift-def assms*(*1*) *assms*(*2*))

lemma(**in** *UP-cring*) *poly-shift-s-mult*:
assumes $f \in$ *carrier P*
assumes $s \in$ *carrier R*
shows *poly-shift* ($s \odot_P f$) = $s \odot_P$ (*poly-shift f*)
proof−
have ($s \odot_P f$) = (*ctrm* ($s \odot_P f$)) $\oplus_P$($X \otimes_P$ *poly-shift* ($s \odot_P f$))
using *poly-shift-eq*[*of* ($s \odot_P f$)] *assms*(*1*) *assms*(*2*)
by *blast*
then have *0*: ($s \odot_P f$) = ($s \odot_P$(*ctrm f*)) $\oplus_P$($X \otimes_P$ *poly-shift* ($s \odot_P f$))
using *ctrm-smult assms*(*1*) *assms*(*2*) **by** *auto*
have *1*: ($s \odot_P f$) = $s \odot_P$ ((*ctrm f*) $\oplus_P$ ($X \otimes_P$ (*poly-shift f*)))
using *assms*(*1*) *poly-shift-eq* **by** *auto*
have *2*: ($s \odot_P f$) = ($s \odot_P$(*ctrm f*)) $\oplus_P$ ($s \odot_P$($X \otimes_P$ (*poly-shift f*)))
by (*simp add*: *1 X-closed assms*(*1*) *assms*(*2*) *ctrm-is-poly poly-shift-closed smult-r-distr*)
have *3*: ($s \odot_P f$) = ($s \odot_P$(*ctrm f*)) $\oplus_P$ ($X \otimes_P$ ($s \odot_P$(*poly-shift f*)))
using *2 UP-m-comm X-closed assms*(*1*) *assms*(*2*) *smult-assoc2*
by (*simp add*: *poly-shift-closed*)
have *4*: ($X \otimes_P$ *poly-shift* ($s \odot_P f$)) = ($X \otimes_P$ ($s \odot_P$(*poly-shift f*)))
using *3 0 X-closed assms*(*1*) *assms*(*2*) *ctrm-is-poly poly-shift-closed*
by *auto*
then show *?thesis*
using *X-closed X-not-zero assms*(*1*) *assms*(*2*)
by (*metis UP-mult-closed UP-smult-closed zcf-eq-zero-unique poly-shift-closed*)

qed

lemma *zcf-poly-shift*:

assumes $f \in$ *carrier P*
shows *zcf* (*poly-shift f*) = *f 1*
apply(*rule poly-induct3*)
apply (*simp add*: *assms*)
using *poly-shift-add zcf-add cfs-add poly-shift-closed* **apply** *metis*
unfolding *zcf-def* **using** *poly-shift-monom poly-shift-degree-zero*
by (*simp add*: *poly-shift-def*)

fun *poly-shift-iter* (‹*shift*›) **where**
Base:*poly-shift-iter 0 f* = *f*|
Step:*poly-shift-iter* (*Suc n*) *f* = *poly-shift* (*poly-shift-iter n f*)

lemma *shift-closed*:
assumes *f* ∈ *carrier P*
shows *shift n f* ∈ *carrier P*
apply(*induction n*)
using *assms poly-shift-closed* **by** *auto*

8.2 Operator Which Multiplies Coefficients by Their Degree

definition *n-mult* **where**
n-mult f = (λ*n*. [*n*]·$_R$(*f n*))

lemma(**in** *UP-cring*) *n-mult-closed*:
assumes *f* ∈ *carrier P*
shows *n-mult f* ∈ *carrier P*
apply(*rule UP-car-memI*[*of deg R f*])
unfolding *n-mult-def*
apply (*metis P.l-zero R.add.nat-pow-one UP-zero-closed assms cfs-zero coeff-of-sum-diff-degree0*)
using *assms cfs-closed* **by** *auto*

Facts about the shift function

lemma *shift-one*:
shift (*Suc 0*) = *poly-shift*
by *auto*

lemma *shift-factor0*:
assumes *f* ∈ *carrier P*
shows *degree f* ≥ (*Suc k*) ⟹ *degree* (*f* ⊖$_P$ ((*shift* (*Suc k*) *f*) ⊗$_P$(*X*[^]$_P$(*Suc k*)))) < (*Suc k*)
proof(*induction k*)
case *0*
have *0*: *f* ⊖$_P$ (*ctrm f*) = (*shift* (*Suc 0*) *f*)⊗$_P$*X*
by (*metis UP-m-comm X-closed assms poly-shift-id shift-closed shift-one*)
then have *f* ⊖$_P$(*shift* (*Suc 0*) *f*)⊗$_P$*X* = (*ctrm f*)
proof−
have *f* ⊖$_P$ (*ctrm f*) ⊖$_P$ (*shift* (*Suc 0*) *f*)⊗$_P$*X*= (*shift* (*Suc 0*) *f*)⊗$_P$*X* ⊖$_P$ (*shift* (*Suc 0*) *f*)⊗$_P$*X*
using *0* **by** *simp*

```
    then have  f ⊖P (ctrm f) ⊖P (shift (Suc 0) f)⊗PX = 0P
      using UP-cring.UP-cring[of R] assms
      by (metis 0 P.ring-simprules(4) P-def UP-ring.UP-ring UP-ring-axioms
          a-minus-def abelian-group.r-neg ctrm-is-poly ring-def)
    then have  f ⊖P ((ctrm f) ⊕P (shift (Suc 0) f)⊗PX) = 0P
      using assms P.ring-simprules
      by (metis 0 poly-shift-id poly-shift-eq)
    then have  f ⊖P ((shift (Suc 0) f)⊗PX ⊕P (ctrm f) ) = 0P
      using P.m-closed UP-a-comm X-closed assms ctrm-is-poly shift-closed
      by presburger
    then have f ⊖P ((shift (Suc 0) f)⊗PX) ⊖P (ctrm f)= 0P
      using P.add.m-assoc P.ring-simprules(14) P.ring-simprules(19) assms 0
          P.add.inv-closed P.r-neg P.r-zero ctrm-is-poly
      by (smt (verit, ccfv-threshold))
    then show ?thesis
    by (metis 0 P.add.m-comm P.m-closed P.ring-simprules(14) P.ring-simprules(18)

        P.ring-simprules(3) X-closed assms ctrm-is-poly poly-shift-id poly-shift-eq
        shift-closed)
  qed
  then have   f ⊖P(shift (Suc 0) f)⊗P(X[^]P(Suc 0)) =    (ctrm f)
  proof−
    have X = X[^]P(Suc 0)
      by (simp add: X-closed)
    then show ?thesis
      using 0 ‹f ⊖P shift (Suc 0) f ⊗P X = ctrm f›
      by auto
  qed
  then have  degree (f ⊖P(shift (Suc 0) f)⊗P(X[^]P(Suc 0))) < 1
    using ctrm-degree[of f] assms   by simp
  then show ?case
    by blast
next
  case (Suc n)
  fix k
  assume IH: degree f ≥ (Suc k) ⟹ degree (f ⊖P ((shift (Suc k) f) ⊗P(X[^]P(Suc
k)))) < (Suc k)
   show   degree f ≥ (Suc (Suc k)) ⟹ degree (f ⊖P ((shift (Suc (Suc k)) f)
⊗P(X[^]P(Suc (Suc k))))) < (Suc (Suc k))
  proof−
    obtain n where n-def: n = Suc k
      by simp
    have IH': degree f ≥ n ⟹ degree (f ⊖P ((shift n f) ⊗P(X[^]Pn))) < n
      using n-def IH by auto
   have P: degree f ≥ (Suc n) ⟹ degree (f ⊖P ((shift (Suc n) f) ⊗P(X[^]P(Suc
n)))) < (Suc n)
    proof−
      obtain g where g-def: g = (f ⊖P ((shift n f) ⊗P(X[^]Pn)))
        by simp
```

obtain s **where** *s-def*: $s = shift\ n\ f$
by *simp*
obtain s' **where** *s'-def*: $s' = shift\ (Suc\ n)\ f$
by *simp*
have *P*: $g \in carrier\ P\ s \in carrier\ P\ s' \in carrier\ P\ (X[\hat{}]_P n) \in carrier\ P$
using *s-def s'-def g-def assms shift-closed*[*of f n*]
apply (*simp add*: *X-closed*)
apply (*simp add*: ‹$f \in carrier\ P \Longrightarrow shift\ n\ f \in carrier\ P$› *assms s-def*)
using *P-def UP-cring.shift-closed UP-cring-axioms assms s'-def* **apply** *blast*
using *X-closed* **by** *blast*
have *g-def'*: $g = (f \ominus_P (s \otimes_P (X[\hat{}]_P n)))$
using *g-def s-def* **by** *auto*
assume *degree* $f \geq (Suc\ n)$
then have *degree* $(f \ominus_P (s \otimes_P (X[\hat{}]_P n))) < n$
using *IH' Suc-leD s-def* **by** *blast*
then have *d-g*: *degree* $g < n$ **using** *g-def'* **by** *auto*
have *P0*: $f \ominus_P (s' \otimes_P (X[\hat{}]_P (Suc\ n))) = ((ctrm\ s) \otimes_P (X[\hat{}]_P n)) \oplus_P g$
proof−
have $s = (ctrm\ s) \oplus_P (X \otimes_P s')$
using *s-def s'-def P-def poly-shift-eq UP-cring-axioms assms shift-closed*
by (*simp add*: *UP-cring.poly-shift-eq*)
then have *0*: $g = f \ominus_P ((ctrm\ s) \oplus_P (X \otimes_P s')) \otimes_P (X[\hat{}]_P n)$
using *g-def'* **by** *auto*
then have $g = f \ominus_P ((ctrm\ s) \otimes_P (X[\hat{}]_P n)) \ominus_P ((X \otimes_P s') \otimes_P (X[\hat{}]_P n))$
using *P cring-axioms X-closed P.l-distr P.ring-simprules*(*19*) *UP-a-assoc a-minus-def assms*
by (*simp add*: *a-minus-def ctrm-is-poly*)
then have $g \oplus_P ((X \otimes_P s') \otimes_P (X[\hat{}]_P n)) = f \ominus_P ((ctrm\ s) \otimes_P (X[\hat{}]_P n))$
using *P cring-axioms X-closed P.l-distr P.ring-simprules UP-a-assoc a-minus-def assms*
by (*simp add*: *P.r-neg2 ctrm-is-poly*)
then have $((ctrm\ s) \otimes_P (X[\hat{}]_P n)) = f \ominus_P (g \oplus_P ((X \otimes_P s') \otimes_P (X[\hat{}]_P n)))$
using *P cring-axioms X-closed P.ring-simprules UP-a-assoc a-minus-def assms*
by (*simp add*: *P.ring-simprules*(*17*) *ctrm-is-poly*)
then have $((ctrm\ s) \otimes_P (X[\hat{}]_P n)) = f \ominus_P (((X \otimes_P s') \otimes_P (X[\hat{}]_P n)) \oplus_P g)$
by (*simp add*: *P*(*1*) *P*(*3*) *UP-a-comm X-closed*)
then have $((ctrm\ s) \otimes_P (X[\hat{}]_P n)) = f \ominus_P ((X \otimes_P s') \otimes_P (X[\hat{}]_P n)) \ominus_P g$
using *P*(*1*) *P*(*3*) *P.ring-simprules*(*19*) *UP-a-assoc a-minus-def assms*
by (*simp add*: *a-minus-def X-closed*)
then have $((ctrm\ s) \otimes_P (X[\hat{}]_P n)) \oplus_P g = f \ominus_P ((X \otimes_P s') \otimes_P (X[\hat{}]_P n))$
by (*metis P*(*1*) *P*(*3*) *P*(*4*) *P.add.inv-solve-right P.m-closed P.ring-simprules*(*14*) *P.ring-simprules*(*4*) *P-def UP-cring.X-closed UP-cring-axioms assms*)
then have $((ctrm\ s) \otimes_P (X[\hat{}]_P n)) \oplus_P g = f \ominus_P ((s' \otimes_P X) \otimes_P (X[\hat{}]_P n))$
by (*simp add*: *P*(*3*) *UP-m-comm X-closed*)
then have $((ctrm\ s) \otimes_P (X[\hat{}]_P n)) \oplus_P g = f \ominus_P (s' \otimes_P (X[\hat{}]_P (Suc\ n)))$
using *P*(*3*) *P.nat-pow-Suc2 UP-m-assoc X-closed* **by** *auto*
then show *?thesis*

```
      by auto
    qed
    have P1: degree (((ctrm s)⊗P(X[^]Pn)) ⊕P g) ≤ n
    proof−
      have Q0: degree ((ctrm s)⊗P(X[^]Pn)) ≤ n
      proof(cases ctrm s = 0P)
        case True
        then show ?thesis
          by (simp add: P(4))
      next
        case False
        then have F0: degree ((ctrm s)⊗P(X[^]Pn)) ≤ degree (ctrm s) + degree
(X[^]Pn)
          by (meson ctrm-is-poly P(2) P(4) deg-mult-ring)
        have F1: 1≠0⟹ degree (X[^]Pn) = n
          unfolding X-poly-def
          using P-def cring-monom-degree by auto
        show ?thesis
         by (metis (no-types, opaque-lifting) F0 F1 ltrm-deg-0 P(2) P.r-null P-def
R.l-null R.l-one
              R.nat-pow-closed R.zero-closed X-poly-def assms cfs-closed
              add-0 deg-const deg-zero deg-ltrm
              monom-pow monom-zero zero-le)
      qed
      then show ?thesis
        using d-g
        by (simp add: P(1) P(2) P(4) bound-deg-sum ctrm-is-poly)
    qed
    then show ?thesis
      using s'-def P0 by auto
  qed
  assume degree f ≥ (Suc (Suc k))
  then show degree (f ⊖P ((shift (Suc (Suc k)) f) ⊗P(X[^]P(Suc (Suc k))))) <
(Suc (Suc k))
    using P by(simp add: n-def)
 qed
qed

lemma(in UP-cring) shift-degree0:
 assumes f ∈ carrier P
 shows degree f >n ⟹ Suc (degree (shift (Suc n) f)) = degree (shift n f)
proof(induction n)
 case 0
 assume B: 0< degree f
 have 0: degree (shift 0 f) = degree f
   by simp
 have 1: degree f = degree (f ⊖P (ctrm f))
   using assms(1) B ctrm-degree degree-of-difference-diff-degree
   by (simp add: ctrm-is-poly)
```

```
  have (f ⊖_P (ctrm f)) = X ⊗_P(shift 1 f)
    using P-def poly-shift-id UP-cring-axioms assms(1) by auto
  then have degree (f ⊖_P (ctrm f)) = 1 + (degree (shift 1 f))
   by (metis 1 B P.r-null X-closed add.commute assms deg-nzero-nzero degree-times-X
not-gr-zero shift-closed)
  then have  degree (shift 0 f) = 1 + (degree (shift 1 f))
    using 0 1 by auto
  then show ?case
    by simp
next
  case (Suc n)
  fix n
  assume IH: (n < degree f ⟹ Suc (degree (shift (Suc n) f)) = degree (shift n
f))
  show Suc n < degree f ⟹ Suc (degree (shift (Suc (Suc n)) f)) = degree (shift
(Suc n) f)
  proof−
    assume A:  Suc n < degree f
   then have 0: (shift (Suc n) f) = ctrm ((shift (Suc n) f)) ⊕_P  (shift (Suc (Suc
n)) f)⊗_P X
       by (metis UP-m-comm X-closed assms local.Step poly-shift-eq shift-closed)

    have N: (shift (Suc (Suc n)) f) ≠ 0_P
    proof
      assume C: shift (Suc (Suc n)) f = 0_P
     obtain g where g-def: g = f ⊖_P  (shift (Suc (Suc n)) f)⊗_P(X[ ^ ]_P(Suc (Suc
n)))
       by simp
      have C0: degree g < degree f
        using g-def assms A
        by (meson Suc-leI Suc-less-SucD Suc-mono less-trans-Suc shift-factor0)
      have C1: g = f
        using C
        by (simp add: P.minus-eq X-closed assms g-def)
      then show False
        using C0 by auto
    qed
     have 1: degree (shift (Suc n) f) = degree ((shift (Suc n) f) ⊖_P ctrm ((shift
(Suc n) f)))
    proof(cases degree (shift (Suc n) f) = 0)
      case True
      then show ?thesis
         using N assms poly-shift-degree-zero poly-shift-closed shift-closed by auto

    next
      case False
      then have degree (shift (Suc n) f) > degree  (ctrm ((shift (Suc n) f)))
      proof −
        have shift (Suc n) f ∈ carrier P
```

```
        using assms shift-closed by blast
      then show ?thesis
        using False ctrm-degree by auto
    qed
    then show ?thesis
    proof −
      show ?thesis
        using ‹degree (ctrm (shift (Suc n) f)) < degree (shift (Suc n) f)›
        assms ctrm-is-poly degree-of-difference-diff-degree shift-closed by presburger
    qed
  qed
  have 2: (shift (Suc n) f) ⊖_P ctrm ((shift (Suc n) f)) = (shift (Suc (Suc n))
f)⊗_P X
    using 0
     by (metis Cring-Poly.INTEG.Step P.m-comm X-closed assms poly-shift-id
shift-closed)
  have 3: degree ((shift (Suc n) f) ⊖_P ctrm ((shift (Suc n) f))) = degree (shift
(Suc (Suc n)) f) + 1
    using 2 N X-closed X-not-zero assms degree-X shift-closed
    by (metis UP-m-comm degree-times-X)
  then show ?thesis using 1
    by linarith
 qed
qed

lemma(in UP-cring) shift-degree:
 assumes f ∈ carrier P
 shows degree f ≥ n ⟹ degree (shift n f) + n = degree f
proof(induction n)
 case 0
 then show ?case
  by auto
next
 case (Suc n)
 fix n
 assume IH: (n ≤ degree f ⟹ degree (shift n f) + n = degree f)
 show Suc n ≤ degree f ⟹ degree (shift (Suc n) f) + Suc n = degree f
 proof−
  assume A: Suc n ≤ degree f
  have 0: degree (shift n f) + n = degree f
   using IH A by auto
  have 1: degree (shift n f) = Suc (degree (shift (Suc n) f))
   using A assms shift-degree0 by auto
  show degree (shift (Suc n) f) + Suc n = degree f
   using 0 1 by simp
 qed
qed

lemma(in UP-cring) shift-degree′:
```

assumes $f \in carrier\ P$
shows $degree\ (shift\ (degree\ f)\ f) = 0$
using *shift-degree assms*
by *fastforce*

lemma(**in** *UP-cring*) *shift-above-degree*:
assumes $f \in carrier\ P$
assumes $k > degree\ f$
shows $(shift\ k\ f) = \mathbf{0}_P$
proof−
have $\bigwedge n.\ shift\ ((degree\ f)+\ (Suc\ n))\ f = \mathbf{0}_P$
proof−
fix *n*
show $shift\ ((degree\ f)+\ (Suc\ n))\ f = \mathbf{0}_P$
proof(*induction n*)
case *0*
have *B0*:$shift\ (degree\ f)\ f\ = ctrm(shift\ (degree\ f)\ f) \oplus_P (shift\ (degree\ f +$
$Suc\ 0)\ f)\otimes_P X$
proof −
have *f1*: $\forall f\ n.\ f \notin carrier\ P \vee shift\ n\ f \in carrier\ P$
by (*meson shift-closed*)
then have $shift\ (degree\ f + Suc\ 0)\ f \in carrier\ P$
using *assms*(*1*) **by** *blast*
then show *?thesis*
using *f1* **by** (*simp add*: *P.m-comm X-closed assms*(*1*) *poly-shift-eq*)
qed
have *B1*:$shift\ (degree\ f)\ f = ctrm(shift\ (degree\ f)\ f)$
proof −
have $shift\ (degree\ f)\ f \in carrier\ P$
using *assms*(*1*) *shift-closed* **by** *blast*
then show *?thesis*
using *ltrm-deg-0 assms*(*1*) *shift-degree′* **by** *auto*
qed
have *B2*: $(shift\ (degree\ f + Suc\ 0)\ f)\otimes_P X = \mathbf{0}_P$
using *B0 B1 X-closed assms*(*1*)
proof −
have $\forall f\ n.\ f \notin carrier\ P \vee shift\ n\ f \in carrier\ P$
using *shift-closed* **by** *blast*
then show *?thesis*
by (*metis* (*no-types*) *B0 B1 P.add.l-cancel-one UP-mult-closed X-closed*
assms(*1*))
qed
then show *?case*
by (*metis P.r-null UP-m-comm UP-zero-closed X-closed assms*(*1*) *zcf-eq-zero-unique*
shift-closed)
next
case (*Suc n*)
fix *n*
assume $shift\ (degree\ f + Suc\ n)\ f = \mathbf{0}_P$

```
    then show shift (degree f + Suc (Suc n)) f = 0_P
      by (simp add: poly-shift-degree-zero)
   qed
  qed
  then show ?thesis
   using assms(2) less-iff-Suc-add by auto
qed

lemma(in UP-domain) shift-cfs0:
  assumes f ∈ carrier P
  shows zcf(shift 1 f) = f 1
  using assms
  by (simp add: zcf-poly-shift)

lemma(in UP-cring) X-mult-cf:
  assumes p ∈ carrier P
  shows (p ⊗_P X) (k+1) = p k
  unfolding X-poly-def
  using assms
 by (metis UP-m-comm X-closed X-poly-def add.commute plus-1-eq-Suc cfs-times-X)

lemma(in UP-cring) X-pow-cf:
  assumes p ∈ carrier P
  shows (p ⊗_P X[^]_P(n::nat)) (n + k) = p k
proof−
  have P: ⋀f. f ∈ carrier P ⟹ (f ⊗_P X[^]_P(n::nat)) (n + k) = f k
  proof(induction n)
   show ⋀f. f ∈ carrier P ⟹ (f ⊗_P X [^]_P (0::nat)) (0 + k) = f k
   proof−
    fix f
    assume B0: f ∈ carrier P
    show (f ⊗_P X [^]_P (0::nat)) (0 + k) = f k
     by (simp add: B0)
   qed
   fix n
   fix f
   assume IH: (⋀f. f ∈ carrier P ⟹ (f ⊗_P X [^]_P n) (n + k) = f k)
   assume A0:  f ∈ carrier P
   show (f ⊗_P X [^]_P Suc n) (Suc n + k) = f k
   proof−
    have 0: (f ⊗_P X [^]_P n)(n + k) = f k
     using A0 IH by simp
    have 1: ((f ⊗_P X [^]_P n)⊗_PX) (Suc n + k) = (f ⊗_P X [^]_P n)(n + k)
     using X-mult-cf A0 P.m-closed P.nat-pow-closed
          Suc-eq-plus1 X-closed add-Suc by presburger
    have 2: (f ⊗_P (X [^]_P n ⊗_PX)) (Suc n + k) = (f ⊗_P X [^]_P n)(n + k)
     using 1
     by (simp add: A0 UP-m-assoc X-closed)
    then show ?thesis
```

by (*simp add*: *0*)
qed
qed
show *?thesis* **using** *assms P*[*of p*] **by** *auto*
qed

lemma *poly-shift-cfs*:
assumes $f \in$ *carrier P*
shows *poly-shift f n* = *f* (*Suc n*)
proof−
have (*f* $\ominus_P$ *ctrm f*) (*Suc n*) = (*X* $\otimes_P$ (*poly-shift f*)) (*Suc n*)
using *assms poly-shift-id* **by** *auto*
then show *?thesis* **unfolding** *X-poly-def* **using** *poly-shift-closed assms*
by (*metis* (*no-types*, *lifting*) *ctrm-degree ctrm-is-poly*
P.add.m-comm P.minus-closed coeff-of-sum-diff-degree0 poly-shift-id poly-shift-eq
cfs-times-X zero-less-Suc)
qed

lemma(**in** *UP-cring*) *shift-cfs*:
assumes $p \in$ *carrier P*
shows (*shift k p*) *n* = *p* (*k* + *n*)
apply(*induction k arbitrary*: *n*)
by (*auto simp*: *assms poly-shift-cfs shift-closed*)

8.3 The Derivative Operator

definition *pderiv* **where**
pderiv p = *poly-shift* (*n-mult p*)

lemma *pderiv-closed*:
assumes $p \in$ *carrier P*
shows *pderiv p* $\in$ *carrier P*
unfolding *pderiv-def*
using *assms n-mult-closed*[*of p*] *poly-shift-closed*[*of n-mult p*]
by *blast*

Function which obtains the first n+1 terms of f, in ascending order of degree:

definition *trms-of-deg-leq* **where**
trms-of-deg-leq n f ≡ *f* $\ominus_{(UP\ R)}$ ((*shift* (*Suc n*) *f*) $\otimes_{UP\ R}$ *monom P* **1** (*Suc n*))

lemma *trms-of-deg-leq-closed*:
assumes $f \in$ *carrier P*
shows *trms-of-deg-leq n f* $\in$ *carrier P*
unfolding *trms-of-deg-leq-def* **using** *assms*
by (*metis P.m-closed P.minus-closed P-def R.one-closed monom-closed shift-closed*)

lemma *trms-of-deg-leq-id*:
assumes $f \in$ *carrier P*
shows *f* $\ominus_P$ (*trms-of-deg-leq k f*) = *shift* (*Suc k*) *f* $\otimes_P$ *monom P* **1** (*Suc k*)

unfolding *trms-of-deg-leq-def*
using *assms*
by (*smt* (*verit*) *P.add.inv-closed P.l-zero P.m-closed P.minus-add P.minus-minus P.r-neg*
P-def R.one-closed UP-a-assoc a-minus-def monom-closed shift-closed)

lemma *trms-of-deg-leq-id'*:
assumes $f \in$ *carrier P*
shows $f =$ (*trms-of-deg-leq k f*) $\oplus_P$ *shift* (*Suc k*) *f* $\otimes_P$ *monom P* **1** (*Suc k*)
using *trms-of-deg-leq-id assms trms-of-deg-leq-closed*[*of f*]
by (*smt* (*verit*, *ccfv-threshold*) *P.add.inv-closed P.l-zero P.m-closed P.minus-add P.minus-minus P.r-neg R.one-closed UP-a-assoc a-minus-def monom-closed shift-closed*)

lemma *deg-leqI*:
assumes $p \in$ *carrier P*
assumes $\bigwedge n.\ n > k \Longrightarrow p\ n = \mathbf{0}$
shows *degree p* $\leq k$
by (*metis assms*(*1*) *assms*(*2*) *deg-zero deg-ltrm le0 le-less-linear monom-zero*)

lemma *deg-leE*:
assumes $p \in$ *carrier P*
assumes *degree p* $< k$
shows *p k* $= \mathbf{0}$
using *assms coeff-of-sum-diff-degree0 P-def coeff-simp deg-aboveD*
by *auto*

lemma *trms-of-deg-leq-deg*:
assumes $f \in$ *carrier P*
shows *degree* (*trms-of-deg-leq k f*) $\leq k$
proof−
have $\bigwedge n.$ (*trms-of-deg-leq k f*) (*Suc k* + *n*) $= \mathbf{0}$
proof−
fix *n*
have *0*: (*shift* (*Suc k*) *f* $\otimes_{UP\ R}$ *monom P* **1** (*Suc k*)) (*Suc k* + *n*) = *shift* (*Suc k*) *f n*
using *assms shift-closed cfs-monom-mult-l*
by (*metis P.m-comm P-def R.one-closed add.commute monom-closed times-X-pow-coeff*)
then show *trms-of-deg-leq k f* (*Suc k* + *n*) $= \mathbf{0}$
unfolding *trms-of-deg-leq-def*
using *shift-cfs*[*of f Suc k n*]
cfs-minus[*of f shift* (*Suc k*) *f* $\otimes_{UP\ R}$ *monom P* **1** (*Suc k*) *Suc k* + *n*]
by (*metis P.m-closed P.r-neg P-def R.one-closed a-minus-def assms cfs-minus cfs-zero monom-closed shift-closed*)
qed
then show *?thesis* **using** *deg-leqI*
by (*metis* (*no-types*, *lifting*) *assms le-iff-add less-Suc-eq-0-disj less-Suc-eq-le trms-of-deg-leq-closed*)
qed

lemma *trms-of-deg-leq-zero-is-ctrm*:
assumes $f \in carrier\ P$
assumes $degree\ f > 0$
shows $trms\text{-}of\text{-}deg\text{-}leq\ 0\ f = ctrm\ f$
proof−
have $f = ctrm\ f \oplus_P (X \otimes_P (shift\ (Suc\ 0)\ f))$
using *assms poly-shift-eq*
by *simp*
then have $f = ctrm\ f \oplus_P (X\ [\widehat{}]_{UP\ R}\ Suc\ 0 \otimes_P (shift\ (Suc\ 0)\ f))$
using *P.nat-pow-eone P-def X-closed* **by** *auto*
then show *?thesis*
unfolding *trms-of-deg-leq-def*
by (*metis* (*no-types*, *lifting*) *ctrm-is-poly One-nat-def P.add.right-cancel P.m-closed*

P.minus-closed P.nat-pow-eone P-def UP-m-comm X-closed X-poly-def assms(*1*) *shift-closed*
trms-of-deg-leq-def trms-of-deg-leq-id′)
qed

lemma *cfs-monom-mult*:
assumes $p \in carrier\ P$
assumes $a \in carrier\ R$
assumes $k < n$
shows $(p \otimes_P (monom\ P\ a\ n))\ k = \mathbf{0}$
apply(*rule poly-induct3*[*of p*])
apply (*simp add*: *assms*(*1*))
apply (*metis* (*no-types*, *lifting*) *P.l-distr P.m-closed R.r-zero R.zero-closed assms*(*2*) *cfs-add monom-closed*)
using *assms monom-mult*[*of - a - n*]
by (*metis R.m-closed R.m-comm add.commute cfs-monom not-add-less1*)

lemma(**in** *UP-cring*) *cfs-monom-mult-2*:
assumes $f \in carrier\ P$
assumes $a \in carrier\ R$
assumes $m < n$
shows $((monom\ P\ a\ n) \otimes_P f)\ m = \mathbf{0}$
using *cfs-monom-mult*
by (*simp add*: *P.m-comm assms*(*1*) *assms*(*2*) *assms*(*3*))

lemma *trms-of-deg-leq-cfs*:
assumes $f \in carrier\ P$
shows $trms\text{-}of\text{-}deg\text{-}leq\ n\ f\ k = (if\ k \leq n\ then\ (f\ k)\ else\ \mathbf{0})$
unfolding *trms-of-deg-leq-def*
apply(*cases* $k \leq n$)
using *cfs-minus*[*of f shift* (*Suc n*) *f* $\otimes_{UP\ R}$ *monom P* **1** (*Suc n*)]
cfs-monom-mult[*of* - **1** *k Suc n*]
apply (*metis* (*no-types*, *lifting*) *P.m-closed P.minus-closed P-def R.one-closed R.r-zero assms*
cfs-add cfs-closed le-refl monom-closed nat-less-le nat-neq-iff not-less-eq-eq

shift-closed
trms-of-deg-leq-def trms-of-deg-leq-id')
using *trms-of-deg-leq-deg*[*of f n*] *deg-leE*
unfolding *trms-of-deg-leq-def*
using *assms trms-of-deg-leq-closed trms-of-deg-leq-def* **by** *auto*

lemma *trms-of-deg-leq-iter*:
assumes $f \in$ *carrier P*
shows *trms-of-deg-leq* (*Suc k*) *f* = (*trms-of-deg-leq k f*) $\oplus_P$ *monom P* (*f* (*Suc k*)) (*Suc k*)
proof fix *x*
show *trms-of-deg-leq* (*Suc k*) *f x* = (*trms-of-deg-leq k f* $\oplus_P$ *monom P* (*f* (*Suc k*)) (*Suc k*)) *x*
apply(*cases x* $\leq$ *k*)
using *trms-of-deg-leq-cfs trms-of-deg-leq-closed cfs-closed*[*of f Suc k*]
cfs-add[*of trms-of-deg-leq k f monom P* (*f* (*Suc k*)) (*Suc k*) *x*]
apply (*simp add*: *assms*)
using *deg-leE assms cfs-closed cfs-monom* **apply** *auto*[*1*]
by (*simp add*: *assms cfs-closed cfs-monom trms-of-deg-leq-cfs trms-of-deg-leq-closed*)

qed

lemma *trms-of-deg-leq-0*:
assumes $f \in$ *carrier P*
shows *trms-of-deg-leq 0 f* = *ctrm f*
by (*metis One-nat-def P.r-null P-def UP-m-comm UP-zero-closed X-closed X-poly-def assms not-gr-zero*
poly-shift-degree-zero shift-one trms-of-deg-leq-def trms-of-deg-leq-zero-is-ctrm trunc-simps(*2*) *trunc-zero*)

lemma *trms-of-deg-leq-degree-f*:
assumes $f \in$ *carrier P*
shows *trms-of-deg-leq* (*degree f*) *f* = *f*
proof fix *x*
show *trms-of-deg-leq* (*deg R f*) *f x* = *f x*
using *assms trms-of-deg-leq-cfs deg-leE*[*of f x*]
by *simp*
qed

definition(**in** *UP-cring*) *lin-part* **where**
lin-part f = *trms-of-deg-leq 1 f*

lemma(**in** *UP-cring*) *lin-part-id*:
assumes $f \in$ *carrier P*
shows *lin-part f* = (*ctrm f*) $\oplus_P$ *monom P* (*f 1*) *1*
unfolding *lin-part-def*
by (*simp add*: *assms trms-of-deg-leq-0 trms-of-deg-leq-iter*)

lemma(**in** *UP-cring*) *lin-part-eq*:

assumes $f \in$ *carrier* P
shows $f =$ *lin-part* $f \oplus_P$ (*shift* 2 f) $\otimes_P$ *monom* P **1** 2
unfolding *lin-part-def*
by (*metis Suc-1 assms trms-of-deg-leq-id'*)

Constant term of a substitution:

lemma *zcf-eval*:
assumes $f \in$ *carrier* P
shows *zcf* $f =$ *to-fun* f **0**
using *assms zcf-to-fun* **by** *blast*

lemma *ctrm-of-sub*:
assumes $f \in$ *carrier* P
assumes $g \in$ *carrier* P
shows *zcf*(*f of g*) = *to-fun f* (*zcf g*)
apply(*rule poly-induct3*[*of f*])
apply (*simp add*: *assms*(*1*))
using *P-def UP-cring.to-fun-closed UP-cring-axioms zcf-add zcf-to-fun assms*(*2*) *to-fun-plus sub-add sub-closed* **apply** *fastforce*
using *R.zero-closed zcf-to-fun assms*(*2*) *to-fun-sub monom-closed sub-closed* **by** *presburger*

Evaluation of linear part:

lemma *to-fun-lin-part*:
assumes $f \in$ *carrier* P
assumes $b \in$ *carrier* R
shows *to-fun* (*lin-part f*) $b = (f\ 0) \oplus (f\ 1) \otimes b$
using *assms lin-part-id*[*of f*] *to-fun-ctrm to-fun-monom monom-closed*
by (*simp add*: *cfs-closed to-fun-plus*)

Constant term of taylor expansion:

lemma *taylor-zcf*:
assumes $f \in$ *carrier* P
assumes $a \in$ *carrier* R
shows *zcf*(T_a f) = *to-fun f a*
unfolding *taylor-expansion-def*
using *ctrm-of-sub assms P-def zcf-eval X-plus-closed taylor-closed taylor-eval* **by** *auto*

lemma(**in** *UP-cring*) *taylor-eq-1*:
assumes $f \in$ *carrier* P
assumes $a \in$ *carrier* R
shows (T_a f) $\ominus_P$ (*trms-of-deg-leq 1* (T_a f)) = (*shift* (*2*::*nat*) (T_a f))$\otimes_P$ ($X[\hat{}]_P$(*2*::*nat*))
by (*metis P.nat-pow-eone P.nat-pow-mult P-def Suc-1 taylor-closed X-closed X-poly-def assms*(*1*)
assms(*2*) *monom-one-Suc2 one-add-one trms-of-deg-leq-id*)

lemma(**in** *UP-cring*) *taylor-deg-1*:

assumes $f \in$ *carrier* P
assumes $a \in$ *carrier* R
shows *f of* (*X-plus a*) = (*lin-part* (T_a *f*)) $\oplus_P$ (*shift* (*2*::*nat*) (T_a *f*))$\otimes_P$ ($X[\hat{}]_P$(*2*::*nat*))
using *taylor-eq-1*[*of f a*]
unfolding *taylor-expansion-def lin-part-def*
using *One-nat-def X-plus-closed assms*(*1*)
assms(*2*) *trms-of-deg-leq-id*′ *numeral-2-eq-2 sub-closed*
by (*metis P.nat-pow-Suc2 P.nat-pow-eone P-def taylor-def X-closed X-poly-def monom-one-Suc taylor-expansion-def*)

lemma(**in** *UP-cring*) *taylor-deg-1-eval*:
assumes $f \in$ *carrier* P
assumes $a \in$ *carrier* R
assumes $b \in$ *carrier* R
assumes c = *to-fun* (*shift* (*2*::*nat*) (T_a *f*)) *b*
assumes *fa* = *to-fun f a*
assumes *f*′*a* = *deriv f a*
shows *to-fun f* (*b* $\oplus$ *a*) = *fa* $\oplus$ (*f*′*a* $\otimes$ *b*) $\oplus$ (*c* $\otimes$ *b*[$\hat{}$](*2*::*nat*))
using *assms taylor-deg-1* **unfolding** *derivative-def*
proof−
have *0*: *to-fun f* (*b* $\oplus$ *a*) = *to-fun* (*f of* (*X-plus a*)) *b*
using *to-fun-sub assms X-plus-closed* **by** *auto*
have *1*: *to-fun* (*lin-part* (T_a *f*)) *b* = *fa* $\oplus$ (*f*′*a* $\otimes$ *b*)
using *assms to-fun-lin-part*[*of* (T_a *f*) *b*]
by (*metis P-def taylor-def UP-cring.taylor-zcf UP-cring.taylor-closed UP-cring-axioms zcf-def derivative-def*)
have *2*: (T_a *f*) = (*lin-part* (T_a *f*)) $\oplus_P$ ((*shift 2* (T_a *f*))$\otimes_P$$X[\hat{}]_P$(*2*::*nat*))
using *lin-part-eq*[*of* (T_a*f*)] *assms*(*1*) *assms*(*2*) *taylor-closed*
by (*metis taylor-def taylor-deg-1 taylor-expansion-def*)
then have *to-fun* (T_a*f*) *b* = *fa* $\oplus$ (*f*′*a* $\otimes$ *b*) $\oplus$ *to-fun* ((*shift 2* (T_a *f*))$\otimes_P$$X[\hat{}]_P$(*2*::*nat*)) *b*
using *1 2*
by (*metis P.nat-pow-closed taylor-closed UP-mult-closed X-closed assms*(*1*) *assms*(*2*) *assms*(*3*)
to-fun-plus lin-part-def shift-closed trms-of-deg-leq-closed)
then have *to-fun* (T_a*f*) *b* = *fa* $\oplus$ (*f*′*a* $\otimes$ *b*) $\oplus$ *c* $\otimes$ *to-fun* ($X[\hat{}]_P$(*2*::*nat*)) *b*
by (*simp add*: *taylor-closed X-closed assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *to-fun-mult shift-closed*)
then have *3*: *to-fun f* (*b* $\oplus$ *a*)= *fa* $\oplus$ (*f*′*a* $\otimes$ *b*) $\oplus$ *c* $\otimes$ *to-fun* ($X[\hat{}]_P$(*2*::*nat*)) *b*
using *taylor-eval assms*(*1*) *assms*(*2*) *assms*(*3*) **by** *auto*
have *to-fun* ($X[\hat{}]_P$(*2*::*nat*)) *b* = *b*[$\hat{}$](*2*::*nat*)
by (*metis P.nat-pow-Suc2 P.nat-pow-eone R.nat-pow-Suc2*
R.nat-pow-eone Suc-1 to-fun-X
X-closed assms(*3*) *to-fun-mult*)
then show *?thesis*
using *3* **by** *auto*
qed

lemma(**in** *UP-cring*) *taylor-deg-1-eval*′:

assumes $f \in carrier\ P$
assumes $a \in carrier\ R$
assumes $b \in carrier\ R$
assumes $c = to\text{-}fun\ (shift\ (2{::}nat)\ (T_a\ f))\ b$
assumes $fa = to\text{-}fun\ f\ a$
assumes $f'a = deriv\ f\ a$
shows $to\text{-}fun\ f\ (a \oplus b) = fa \oplus (f'a \otimes b) \oplus (c \otimes b[\hat{}](2{::}nat))$
using *R.add.m-comm taylor-deg-1-eval assms(1) assms(2) assms(3) assms(4) assms(5) assms(6)*
by *auto*

lemma(**in** *UP-cring*) *taylor-deg-1-eval''*:
assumes $f \in carrier\ P$
assumes $a \in carrier\ R$
assumes $b \in carrier\ R$
assumes $c = to\text{-}fun\ (shift\ (2{::}nat)\ (T_a\ f))\ (\ominus b)$
shows $to\text{-}fun\ f\ (a \ominus b) = (to\text{-}fun\ f\ a) \ominus (deriv\ f\ a \otimes b) \oplus (c \otimes b[\hat{}](2{::}nat))$
proof−
have $\ominus b \in carrier\ R$
using *assms*
by *blast*
then have *0*: $to\text{-}fun\ f\ (a \ominus b) = (to\text{-}fun\ f\ a) \oplus (deriv\ f\ a \otimes (\ominus b)) \oplus (c \otimes (\ominus b)[\hat{}](2{::}nat))$
unfolding *a-minus-def*
using *taylor-deg-1-eval'[of f a ⊖b c (to-fun f a) deriv f a] assms*
by *auto*
have *1*: $\ominus (deriv\ f\ a \otimes b) = (deriv\ f\ a \otimes (\ominus b))$
using *assms*
by (*simp add*: *R.r-minus deriv-closed*)
have *2*: $(c \otimes b[\hat{}](2{::}nat)) = (c \otimes (\ominus b)[\hat{}](2{::}nat))$
using *assms*
by (*metis R.add.inv-closed R.add.inv-solve-right R.l-zero R.nat-pow-Suc2 R.nat-pow-eone R.zero-closed Suc-1 UP-ring-axioms UP-ring-def ring.ring-simprules(26) ring.ring-simprules(27)*)
show *?thesis*
using *0 1 2*
unfolding *a-minus-def*
by *simp*
qed

lemma(**in** *UP-cring*) *taylor-deg-1-expansion*:
assumes $f \in carrier\ P$
assumes $a \in carrier\ R$
assumes $b \in carrier\ R$
assumes $c = to\text{-}fun\ (shift\ (2{::}nat)\ (T_a\ f))\ (b \ominus a)$
assumes $fa = to\text{-}fun\ f\ a$
assumes $f'a = deriv\ f\ a$
shows $to\text{-}fun\ f\ (b) = fa \oplus f'a \otimes (b \ominus a) \oplus (c \otimes (b \ominus a)[\hat{}](2{::}nat))$
proof−

obtain b' **where** b'*-def*: $b' = b \ominus a$
by *simp*
then have b'*-def'*: $b = b' \oplus a$
using *assms*
by (*metis R.add.inv-solve-right R.minus-closed R.minus-eq*)
have *to-fun f* $(b' \oplus a) = fa \oplus (f'a \otimes b') \oplus (c \otimes b'[\widehat{\ }](2::nat))$
using *assms taylor-deg-1-eval*[*of f a b' c fa f'a*] b'*-def*
by *blast*
then have *to-fun f* $(b) = fa \oplus (f'a \otimes b') \oplus (c \otimes b'[\widehat{\ }](2::nat))$
using b'*-def'*
by *auto*
then show *to-fun f* $(b) = fa \oplus f'a \otimes (b \ominus a) \oplus c \otimes (b \ominus a) [\widehat{\ }] (2::nat)$
using b'*-def*
by *auto*
qed

lemma(**in** *UP-cring*) *Taylor-deg-1-expansion'*:
assumes $f \in$ *carrier* $(UP\ R)$
assumes $a \in$ *carrier* R
assumes $b \in$ *carrier* R
shows $\exists c \in$ *carrier R. to-fun f* $(b) =$ (*to-fun f a*) $\oplus$ (*deriv f a*) $\otimes (b \ominus a) \oplus (c \otimes (b \ominus a)[\widehat{\ }](2::nat))$
using *taylor-deg-1-expansion*[*of f a b*] *assms* **unfolding** *P-def*
by (*metis P-def R.minus-closed taylor-closed shift-closed to-fun-closed*)

Basic Properties of deriv and pderiv:

lemma *n-mult-degree-bound*:
assumes $f \in$ *carrier P*
shows *degree* (*n-mult f*) $\leq$ *degree f*
apply(*rule deg-leqI*)
apply (*simp add*: *assms n-mult-closed*)
by (*simp add*: *assms deg-leE n-mult-def*)

lemma *pderiv-deg-0*[*simp*]:
assumes $f \in$ *carrier P*
assumes *degree f* $= 0$
shows *pderiv f* $= \mathbf{0}_P$
proof−
have *degree* (*n-mult f*) $= 0$
using *P-def n-mult-degree-bound assms*(*1*) *assms*(*2*) **by** *fastforce*
then show *?thesis*
unfolding *pderiv-def*
by (*simp add*: *assms*(*1*) *n-mult-closed poly-shift-degree-zero*)
qed

lemma *deriv-deg-0*:
assumes $f \in$ *carrier P*
assumes *degree f* $= 0$
assumes $a \in$ *carrier R*

shows *deriv f a* = **0**
unfolding *derivative-def taylor-expansion-def*
using *X-plus-closed assms*(*1*) *assms*(*2*) *assms*(*3*) *deg-leE sub-const* **by** *force*

lemma *poly-shift-monom′*:
assumes *a* ∈ *carrier R*
shows *poly-shift* (*a* ⊙$_P$ (*X*[^]$_P$(*Suc n*))) = *a*⊙$_P$(*X*[^]$_P$*n*)
using *assms monom-rep-X-pow poly-shift-monom* **by** *auto*

lemma *monom-coeff*:
assumes *a* ∈ *carrier R*
shows (*a* ⊙$_P$ *X* [^]$_P$ (*n*::*nat*)) *k* = (*if* (*k* = *n*) *then a else* **0**)
using *assms cfs-monom monom-rep-X-pow* **by** *auto*

lemma *cfs-n-mult*:
assumes *p* ∈ *carrier P*
shows *n-mult p n* = [*n*]·(*p n*)
by (*simp add*: *n-mult-def*)

lemma *cfs-add-nat-pow*:
assumes *p* ∈ *carrier P*
shows ([(*n*::*nat*)]·$_P$*p*) *k* = [*n*]·(*p k*)
apply(*induction n*) **by** (*auto simp*: *assms*)

lemma *cfs-add-int-pow*:
assumes *p* ∈ *carrier P*
shows ([(*n*::*int*)]·$_P$*p*) *k* = [*n*]·(*p k*)
apply(*induction n*)
by(*auto simp*: *add-pow-int-ge assms cfs-add-nat-pow add-pow-int-lt*)

lemma *add-nat-pow-monom*:
assumes *a* ∈ *carrier R*
shows [(*n*::*nat*)]·$_P$*monom P a k* = *monom P* ([*n*]·*a*) *k*
apply(*rule ext*)
by (*simp add*: *assms cfs-add-nat-pow cfs-monom*)

lemma *add-int-pow-monom*:
assumes *a* ∈ *carrier R*
shows [(*n*::*int*)]·$_P$*monom P a k* = *monom P* ([*n*]·*a*) *k*
apply(*rule ext*)
by (*simp add*: *assms cfs-add-int-pow cfs-monom*)

lemma *n-mult-monom*:
assumes *a* ∈ *carrier R*
shows *n-mult* (*monom P a* (*Suc n*)) = *monom P* ([*Suc n*]·*a*) (*Suc n*)
apply(*rule ext*)
unfolding *n-mult-def*
using *assms cfs-monom* **by** *auto*

lemma *pderiv-monom*:
assumes $a \in$ *carrier R*
shows *pderiv* (*monom P a n*) = *monom P* ([*n*]·*a*) (*n*−*1*)
apply(*cases n* = *0*)
apply (*simp add*: *assms*)
unfolding *pderiv-def*
using *assms Suc-diff-1*[*of n*] *n-mult-monom*[*of a n*−*1*] *poly-shift-monom*[*of* [*Suc* (*n*−*1*)]·*a Suc* (*n*−*1*)]
by (*metis R.add.nat-pow-closed neq0-conv poly-shift-monom*)

lemma *pderiv-monom′*:
assumes $a \in$ *carrier R*
shows *pderiv* ($a \odot_P X[\hat{}]_P$(*n*::*nat*)) = ([*n*]·*a*)$\odot_P X[\hat{}]_P$(*n*−*1*)
using *assms pderiv-monom*[*of a n*]
by (*simp add*: *P-def UP-cring.monom-rep-X-pow UP-cring-axioms*)

lemma *n-mult-add*:
assumes $p \in$ *carrier P*
assumes $q \in$ *carrier P*
shows *n-mult* ($p \oplus_P q$) = *n-mult p* $\oplus_P$ *n-mult q*
proof(*rule ext*) **fix** *x* **show** *n-mult* ($p \oplus_P q$) *x* = (*n-mult p* $\oplus_P$ *n-mult q*) *x*
using *assms R.add.nat-pow-distrib*[*of p x q x x*] *cfs-add*[*of p q x*]
cfs-add[*of n-mult p n-mult q x*] *n-mult-closed*
unfolding *n-mult-def*
by (*simp add*: *cfs-closed*)
qed

lemma *pderiv-add*:
assumes $p \in$ *carrier P*
assumes $q \in$ *carrier P*
shows *pderiv* ($p \oplus_P q$) = *pderiv p* $\oplus_P$ *pderiv q*
unfolding *pderiv-def*
using *assms poly-shift-add n-mult-add*
by (*simp add*: *n-mult-closed*)

lemma *zcf-monom-sub*:
assumes $p \in$ *carrier P*
shows *zcf* ((*monom P* **1** (*Suc n*)) *of* *p*) = *zcf p* [$\hat{}$] (*Suc n*)
apply(*induction n*)
using *One-nat-def P.nat-pow-eone R.nat-pow-eone R.one-closed R.zero-closed zcf-to-fun assms to-fun-closed monom-sub smult-one* **apply** *presburger*
using *P-def UP-cring.ctrm-of-sub UP-cring-axioms zcf-to-fun assms to-fun-closed to-fun-monom monom-closed*
by *fastforce*

lemma *zcf-monom-sub′*:
assumes $p \in$ *carrier P*
assumes $a \in$ *carrier R*
shows *zcf* ((*monom P a* (*Suc n*)) *of* *p*) = *a* $\otimes$ *zcf p* [$\hat{}$] (*Suc n*)

using *zcf-monom-sub assms P-def R.zero-closed UP-cring.ctrm-of-sub UP-cring.to-fun-monom UP-cring-axioms*
zcf-to-fun to-fun-closed monom-closed **by** *fastforce*

lemma *deriv-monom*:
assumes $a \in$ *carrier R*
assumes $b \in$ *carrier R*
shows *deriv* (*monom P a n*) *b* = $([n]\cdot a)\otimes(b[\hat{}](n-1))$
proof(*induction n*)
case *0*
have *0*: *b* [^] ((*0*::*nat*) − *1*) ∈ *carrier R*
using *assms*
by *simp*
then show *?case* **unfolding** *derivative-def* **using** *assms*
by (*metis R.add.nat-pow-0 R.l-null deg-const deriv-deg-0 derivative-def monom-closed*)
next
case (*Suc n*)
show *?case*
proof(*cases n* = *0*)
case *True*
have *T0*: [*Suc n*] · *a* ⊗ *b* [^] (*Suc n* − *1*) = *a*
by (*simp add*: *True assms*(*1*))
have *T1*: (*X-poly R* $\oplus_{UP\ R}$ *to-polynomial R b*) [^]$_{UP\ R}$ *Suc n* = *X-poly R* $\oplus_{UP\ R}$ *to-polynomial R b*
using *P.nat-pow-eone P-def True UP-a-closed X-closed assms*(*2*) *to-poly-closed* **by** *auto*
then show *?thesis*
unfolding *derivative-def taylor-expansion-def*
using *T0 T1 True sub-monom*(*2*)[*of X-plus b a Suc n*] *cfs-add assms*
unfolding *P-def X-poly-plus-def to-polynomial-def X-poly-def*
by (*metis One-nat-def P-def R.add.nat-pow-eone R.nat-pow-0 UP-cring.cfs-X-plus X-plus-closed X-poly-def X-poly-plus-def cfs-smult diff-Suc-1' is-UP-cring n-not-Suc-n to-polynomial-def*)
next
case *False*
have *deriv* (*monom P a* (*Suc n*)) *b* = ((*monom P a* (*Suc n*)) *of* (*X-plus b*)) *1*
unfolding *derivative-def taylor-expansion-def*
by *auto*
then have *deriv* (*monom P a* (*Suc n*)) *b* = (((*monom P a n*) *of* (*X-plus b*)) $\otimes_P$ (*X-plus b*)) *1*
using *monom-mult*[*of a* **1** *n 1*] *sub-mult*[*of X-plus b monom P a n monom P* **1** *1*] *X-plus-closed*[*of b*] *assms*
by (*metis lcf-monom*(*1*) *P.l-one P.nat-pow-eone P-def R.one-closed R.r-one Suc-eq-plus1*
deg-one monom-closed monom-one sub-monom(*1*) *to-poly-inverse*)
then have *deriv* (*monom P a* (*Suc n*)) *b* = (((*monom P a n*) *of* (*X-plus b*)) $\otimes_P$ (*monom P* **1** *1*) $\oplus_P$
(((*monom P a n*) *of* (*X-plus b*)) $\otimes_P$ *to-poly b*)) *1*
unfolding *X-poly-plus-def*

by (*metis P.r-distr P-def X-closed X-plus-closed X-poly-def X-poly-plus-def assms(1) assms(2) monom-closed sub-closed to-poly-closed*)
then have *deriv* (*monom P a* (*Suc n*)) *b* = ((*monom P a n*) *of* (*X-plus b*)) *0* ⊕ *b* ⊗ ((*monom P a n*) *of* (*X-plus b*)) *1*
unfolding *X-poly-plus-def*
by (*smt* (*verit*) *One-nat-def P.m-closed P-def UP-m-comm X-closed X-plus-closed X-poly-def X-poly-plus-def*
assms(1) assms(2) cfs-add cfs-monom-mult-l monom-closed plus-1-eq-Suc sub-closed cfs-times-X to-polynomial-def)
then have *deriv* (*monom P a* (*Suc n*)) *b* = ((*monom P a n*) *of* (*X-plus b*)) *0* ⊕ *b* ⊗ (*deriv* (*monom P a n*) *b*)
by (*simp add*: *derivative-def taylor-expansion-def*)
then have *deriv* (*monom P a* (*Suc n*)) *b* = ((*monom P a n*) *of* (*X-plus b*)) *0* ⊕ *b* ⊗ (([*n*]·*a*)⊗(*b*[^](*n*−*1*)))
by (*simp add*: *Suc*)
then have *0*: *deriv* (*monom P a* (*Suc n*)) *b* = ((*monom P a n*) *of* (*X-plus b*)) *0* ⊕ ([*n*]·*a*)⊗(*b*[^]*n*)
using *assms R.m-comm*[*of b*] *R.nat-pow-mult*[*of b n*−*1 1*] *False*
by (*metis* (*no-types*, *lifting*) *R.add.nat-pow-closed R.m-lcomm R.nat-pow-closed R.nat-pow-eone add.commute add-eq-if plus-1-eq-Suc*)
have *1*: ((*monom P a n*) *of* (*X-plus b*)) *0* = *a* ⊗ *b*[^]*n*
unfolding *X-poly-plus-def* **using** *zcf-monom-sub'*
by (*smt* (*verit*) *ctrm-of-sub One-nat-def P-def R.l-zero R.one-closed UP-cring.zcf-to-poly*

UP-cring.f-minus-ctrm UP-cring-axioms X-plus-closed X-poly-def X-poly-plus-def zcf-add
zcf-def assms(1) assms(2) to-fun-monom monom-closed monom-one-Suc2 poly-shift-id poly-shift-monom to-poly-closed)
show *?thesis*
using *0 1 R.add.nat-pow-Suc2 R.add.nat-pow-closed R.l-distr R.nat-pow-closed assms(1) assms(2) diff-Suc-1* **by** *presburger*
qed
qed

lemma *deriv-smult*:
assumes *a* ∈ *carrier R*
assumes *b* ∈ *carrier R*
assumes *g* ∈ *carrier P*
shows *deriv* (*a* ⊙$_P$ *g*) *b* = *a* ⊗ (*deriv g b*)
unfolding *derivative-def taylor-expansion-def*
using *assms sub-smult X-plus-closed cfs-smult*
by (*simp add*: *sub-closed*)

lemma *deriv-const*:
assumes *a* ∈ *carrier R*
assumes *b* ∈ *carrier R*
shows *deriv* (*monom P a 0*) *b* = **0**
unfolding *derivative-def*
using *assms taylor-closed taylor-def taylor-deg deg-leE* **by** *auto*

lemma *deriv-monom-deg-one*:
assumes $a \in carrier\ R$
assumes $b \in carrier\ R$
shows $deriv\ (monom\ P\ a\ 1)\ b = a$
unfolding *derivative-def taylor-expansion-def*
using *assms cfs-X-plus*[*of b 1*] *sub-monom-deg-one X-plus-closed*[*of b*]
by *simp*

lemma *monom-Suc*:
assumes $a \in carrier\ R$
shows $monom\ P\ a\ (Suc\ n) = monom\ P\ \mathbf{1}\ 1 \otimes_P monom\ P\ a\ n$
$monom\ P\ a\ (Suc\ n) = monom\ P\ a\ n \otimes_P monom\ P\ \mathbf{1}\ 1$
apply (*metis R.l-one R.one-closed Suc-eq-plus1-left assms monom-mult*)
by (*metis R.one-closed R.r-one Suc-eq-plus1 assms monom-mult*)

8.4 The Product Rule

lemma(**in** *UP-cring*) *times-x-product-rule*:
assumes $f \in carrier\ P$
shows $pderiv\ (f \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1) = f \oplus_P pderiv\ f \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1$
proof(*rule poly-induct3*[*of f*])
show $f \in carrier\ P$
using *assms* **by** *blast*
show $\bigwedge p\ q.\ q \in carrier\ P \Longrightarrow p \in carrier\ P \Longrightarrow pderiv\ (p \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1) = p \oplus_P pderiv\ p \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1 \Longrightarrow pderiv\ (q \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1) = q \oplus_P pderiv\ q \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1 \Longrightarrow pderiv\ ((p \oplus_P q) \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1) = p \oplus_P q \oplus_P pderiv\ (p \oplus_P q) \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1$
proof− **fix** $p\ q$ **assume** A: $q \in carrier\ P$ $p \in carrier\ P$ $pderiv\ (p \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1) = p \oplus_P pderiv\ p \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1$ $pderiv\ (q \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1) = q \oplus_P pderiv\ q \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1$
have *0*: $(p \oplus_P q) \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1 = (p \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1) \oplus_P (q \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1)$
using *A assms* **by** (*meson R.one-closed UP-l-distr is-UP-monomE*(*1*) *is-UP-monomI*)
have *1*: $pderiv\ ((p \oplus_P q) \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1) = pderiv\ (p \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1) \oplus_P pderiv\ (q \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1)$
unfolding *0* **apply**(*rule pderiv-add*)
using *A is-UP-monomE*(*1*) *monom-is-UP-monom*(*1*) **apply** *blast*
using *A is-UP-monomE*(*1*) *monom-is-UP-monom*(*1*) **by** *blast*
have *2*: $pderiv\ ((p \oplus_P q) \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1) = p \oplus_P pderiv\ p \otimes_P$

up-ring.monom P **1** *1* $\oplus_P$ (*q* $\oplus_P$ *pderiv q* $\otimes_P$ *up-ring.monom P* **1** *1*)
unfolding *1 A* **by** *blast*
have *3*: *pderiv* ((*p* $\oplus_P$ *q*) $\otimes_P$ *up-ring.monom P* **1** *1*) = *p* $\oplus_P$ *q* $\oplus_P$ (*pderiv p* $\otimes_P$ *up-ring.monom P* **1** *1* $\oplus_P$ *pderiv q* $\otimes_P$ *up-ring.monom P* **1** *1*)
unfolding *2*
using *A P.add.m-lcomm R.one-closed UP-a-assoc UP-a-closed UP-mult-closed is-UP-monomE*(*1*) *monom-is-UP-monom*(*1*) *pderiv-closed* **by** *presburger*
have *4*: *pderiv* ((*p* $\oplus_P$ *q*) $\otimes_P$ *up-ring.monom P* **1** *1*) = *p* $\oplus_P$ *q* $\oplus_P$ ((*pderiv p* $\oplus_P$ *pderiv q*) $\otimes_P$ *up-ring.monom P* **1** *1*)
unfolding *3* **using** *A P.l-distr R.one-closed is-UP-monomE*(*1*) *monom-is-UP-monom*(*1*) *pderiv-closed* **by** *presburger*
show *5*: *pderiv* ((*p* $\oplus_P$ *q*) $\otimes_P$ *up-ring.monom P* **1** *1*) = *p* $\oplus_P$ *q* $\oplus_P$ *pderiv* (*p* $\oplus_P$ *q*) $\otimes_P$ *up-ring.monom P* **1** *1*
unfolding *4* **using** *pderiv-add A* **by** *presburger*
qed
show $\bigwedge$*a n*. *a* $\in$ *carrier R* $\Longrightarrow$
pderiv (*up-ring.monom P a n* $\otimes_P$ *up-ring.monom P* **1** *1*) = *up-ring.monom P a n* $\oplus_P$ *pderiv* (*up-ring.monom P a n*) $\otimes_P$ *up-ring.monom P* **1** *1*
proof− **fix** *a n* **assume** *A*: *a* $\in$ *carrier R*
have *0*: *up-ring.monom P a n* $\otimes_P$ *up-ring.monom P* **1** *1* = *up-ring.monom P a* (*Suc n*)
using *A monom-Suc*(*2*) **by** *presburger*
have *1*: *pderiv* (*up-ring.monom P a n* $\otimes_P$ *up-ring.monom P* **1** *1*) = [(*Suc n*)] $\cdot_P$ (*up-ring.monom P a n*)
unfolding *0* **using** *A add-nat-pow-monom n-mult-monom pderiv-def poly-shift-monom*
by (*simp add*: *P-def*)
have *2*: *pderiv* (*up-ring.monom P a n* $\otimes_P$ *up-ring.monom P* **1** *1*) = (*up-ring.monom P a n*) $\oplus_P$ [*n*] $\cdot_P$ (*up-ring.monom P a n*)
unfolding *1* **using** *A P.add.nat-pow-Suc2 is-UP-monomE*(*1*) *monom-is-UP-monom*(*1*) **by** *blast*
have *3*: *pderiv* (*up-ring.monom P a n*) $\otimes_P$ *up-ring.monom P* **1** *1* = [*n*] $\cdot_P$ (*up-ring.monom P a n*)
apply(*cases n* = *0*)
using *A add-nat-pow-monom n-mult-monom pderiv-def poly-shift-monom pderiv-deg-0* **apply** *auto*[*1*]
using *monom-Suc*(*2*)[*of a n*−*1*] *A add-nat-pow-monom n-mult-monom pderiv-def poly-shift-monom*
by (*metis R.add.nat-pow-closed Suc-eq-plus1 add-eq-if monom-Suc*(*2*) *pderiv-monom*)
show *pderiv* (*up-ring.monom P a n* $\otimes_P$ *up-ring.monom P* **1** *1*) = *up-ring.monom P a n* $\oplus_P$ *pderiv* (*up-ring.monom P a n*) $\otimes_P$ *up-ring.monom P* **1** *1*
unfolding *2 3* **by** *blast*
qed
qed

lemma(**in** *UP-cring*) *deg-one-eval*:
assumes *g* $\in$ *carrier* (*UP R*)
assumes *deg R g* = *1*

shows $\bigwedge t.\ t \in \mathit{carrier}\ R \Longrightarrow \mathit{to\text{-}fun}\ g\ t = g\ 0 \oplus (g\ 1) \otimes t$
proof−
obtain h **where** *h-def*: $h = \mathit{ltrm}\ g$
by *blast*
have *0*: $\mathit{deg}\ R\ (g \ominus_{UP\ R} h) = 0$
using *assms* **unfolding** *h-def*
by (*metis ltrm-closed ltrm-eq-imp-deg-drop ltrm-monom P-def UP-car-memE*(*1*) *less-one*)
have *1*: $g \ominus_{UP\ R} h = \mathit{to\text{-}poly}\ (g\ 0)$
proof(*rule ext*) **fix** x **show** $(g \ominus_{UP\ R} h)\ x = \mathit{to\text{-}polynomial}\ R\ (g\ 0)\ x$
proof(*cases* $x = 0$)
case *True*
have *T0*: $h\ 0 = \mathbf{0}$
unfolding *h-def* **using** *assms UP-car-memE*(*1*) *cfs-monom* **by** *presburger*
have *T1*: $(g \ominus_{UP\ R} h)\ 0 = g\ 0 \ominus h\ 0$
using *ltrm-closed P-def assms*(*1*) *cfs-minus h-def* **by** *blast*
then show *?thesis* **using** *T0 assms*
by (*smt* (*verit*) *0 ltrm-closed ltrm-deg-0 P.minus-closed P-def UP-car-memE*(*1*) *UP-zero-closed zcf-def zcf-zero deg-zero degree-to-poly h-def to-poly-closed to-poly-inverse to-poly-minus trunc-simps*(*2*) *trunc-zero*)
next
case *False*
then have $x > 0$
by *presburger*
then show *?thesis*
by (*metis 0 ltrm-closed P.minus-closed P-def UP-car-memE*(*1*) *UP-cring.degree-to-poly UP-cring-axioms assms*(*1*) *deg-leE h-def to-poly-closed*)
qed
qed
have *2*: $g = (g \ominus_{UP\ R} h) \oplus_{UP\ R} h$
unfolding *h-def* **using** *assms*
by (*metis 1 P-def h-def lin-part-def lin-part-id to-polynomial-def trms-of-deg-leq-degree-f*)

fix t **assume** *A*: $t \in \mathit{carrier}\ R$
have *3*: $\mathit{to\text{-}fun}\ g\ t = \mathit{to\text{-}fun}\ (g \ominus_{UP\ R} h)\ t \oplus \mathit{to\text{-}fun}\ h\ t$
using *2*
by (*metis 1 A P-def UP-car-memE*(*1*) *assms*(*1*) *h-def monom-closed to-fun-plus to-polynomial-def*)
then show $\mathit{to\text{-}fun}\ g\ t = g\ 0 \oplus g\ 1 \otimes t$
unfolding *1 h-def*
using *A P-def UP-cring.lin-part-def UP-cring-axioms assms*(*1*) *assms*(*2*) *to-fun-lin-part trms-of-deg-leq-degree-f* **by** *fastforce*
qed

lemma *nmult-smult*:
assumes $a \in \mathit{carrier}\ R$
assumes $f \in \mathit{carrier}\ P$
shows $\mathit{n\text{-}mult}\ (a \odot_P f) = a \odot_P (\mathit{n\text{-}mult}\ f)$
apply(*rule poly-induct4*[*of f*])

apply (*simp add*: *assms*(*2*))
using *assms*(*1*) *n-mult-add n-mult-closed smult-closed smult-r-distr* **apply** *presburger*
using *assms* **apply**(*intro ext*, *metis* (*no-types*, *lifting*) *ctrm-smult ltrm-deg-0 P-def R.add.nat-pow-0 UP-cring.ctrm-degree UP-cring.n-mult-closed UP-cring.n-mult-def UP-cring-axioms UP-smult-closed UP-zero-closed zcf-degree-zero zcf-zero deg-const deg-zero le-0-eq monom-closed n-mult-degree-bound smult-r-null*)
using *monom-mult-smult n-mult-monom assms*
by (*smt* (*verit*) *lcf-monom*(*1*) *P-def R.add.nat-pow-closed R.add-pow-rdistr R.zero-closed UP-cring.to-poly-mult-simp*(*1*) *UP-cring-axioms UP-smult-closed cfs-closed cring-lcf-mult monom-closed to-polynomial-def*)

lemma *pderiv-smult*:
assumes $a \in$ *carrier R*
assumes $f \in$ *carrier P*
shows *pderiv* $(a \odot_P f) = a \odot_P$ (*pderiv f*)
unfolding *pderiv-def*
using *assms*
by (*simp add*: *n-mult-closed nmult-smult poly-shift-s-mult*)

lemma(**in** *UP-cring*) *pderiv-minus*:
assumes $a \in$ *carrier P*
assumes $b \in$ *carrier P*
shows *pderiv* $(a \ominus_P b) =$ *pderiv a* $\ominus_P$ *pderiv b*
proof−
have $\ominus_P b = (\ominus \mathbf{1}) \odot_P b$
using *R.one-closed UP-smult-one assms*(*2*) *smult-l-minus* **by** *presburger*
thus *?thesis* **unfolding** *a-minus-def* **using** *pderiv-add assms pderiv-smult*
by (*metis P.add.inv-closed R.add.inv-closed R.one-closed UP-smult-one pderiv-closed smult-l-minus*)
qed

lemma(**in** *UP-cring*) *pderiv-const*:
assumes $b \in$ *carrier R*
shows *pderiv* (*up-ring.monom P b 0*) $= \mathbf{0}_P$
using *assms pderiv-monom*[*of b 0*] *deg-const is-UP-monomE*(*1*) *monom-is-UP-monom*(*1*) *pderiv-deg-0*
by *blast*

lemma(**in** *UP-cring*) *pderiv-minus-const*:
assumes $a \in$ *carrier P*
assumes $b \in$ *carrier R*
shows *pderiv* $(a \ominus_P$ *up-ring.monom P b 0*$) =$ *pderiv a*
using *pderiv-minus*[*of a up-ring.monom P b 0*] *assms pderiv-const*[*of b*]
by (*smt* (*verit*) *P.l-zero P.minus-closed P-def UP-cring.pderiv-const UP-cring.pderiv-minus UP-cring.poly-shift-eq UP-cring-axioms cfs-closed monom-closed pderiv-add pderiv-closed poly-shift-id*)

lemma(**in** *UP-cring*) *monom-product-rule*:

assumes $f \in$ *carrier* P
assumes $a \in$ *carrier* R
shows *pderiv* $(f \otimes_P$ *up-ring.monom* $P\ a\ n) =\ f \otimes_P$ *pderiv* (*up-ring.monom* P $a\ n) \oplus_P$ *pderiv* $f \otimes_P$ *up-ring.monom* $P\ a\ n$
proof−
have $\forall f.\ f \in$ *carrier* $P \longrightarrow$ *pderiv* $(f \otimes_P$ *up-ring.monom* $P\ a\ n) =\ f \otimes_P$ *pderiv* (*up-ring.monom* $P\ a\ n) \oplus_P$ *pderiv* $f \otimes_P$ *up-ring.monom* $P\ a\ n$
proof(*induction* n)
case *0*
show *?case*
proof fix f **show** $f \in$ *carrier* $P \longrightarrow$ *pderiv* $(f \otimes_P$ *up-ring.monom* $P\ a\ 0) = f \otimes_P$ *pderiv* (*up-ring.monom* $P\ a\ 0) \oplus_P$ *pderiv* $f \otimes_P$ *up-ring.monom* $P\ a\ 0$
proof assume *A*: $f \in$ *carrier* P
have *0*: $f \otimes_P$ *up-ring.monom* $P\ a\ 0 = a \odot_P f$
using *assms A UP-m-comm is-UP-monomE*(*1*) *monom-is-UP-monom*(*1*) *monom-mult-is-smult* **by** *presburger*
have *1*: $f \otimes_P$ *pderiv* (*up-ring.monom* $P\ a\ 0) = \mathbf{0}_P$
using *A assms P.r-null pderiv-const* **by** *presburger*
have *2*: *pderiv* $f \otimes_P$ *up-ring.monom* $P\ a\ 0 = a \odot_P$ *pderiv* f
using *assms A UP-m-comm is-UP-monomE*(*1*) *monom-is-UP-monom*(*1*) *monom-mult-is-smult pderiv-closed* **by** *presburger*
show *pderiv* $(f \otimes_P$ *up-ring.monom* $P\ a\ 0) = f \otimes_P$ *pderiv* (*up-ring.monom* $P\ a\ 0) \oplus_P$ *pderiv* $f \otimes_P$ *up-ring.monom* $P\ a\ 0$
unfolding *0 1 2* **using** *A UP-l-zero UP-smult-closed assms*(*2*) *pderiv-closed pderiv-smult* **by** *presburger*
qed
qed
next
case (*Suc* n)
show $\forall f.\ f \in$ *carrier* $P \longrightarrow$
pderiv $(f \otimes_P$ *up-ring.monom* $P\ a$ (*Suc* $n)) = f \otimes_P$ *pderiv* (*up-ring.monom* $P\ a$ (*Suc* $n)) \oplus_P$ *pderiv* $f \otimes_P$ *up-ring.monom* $P\ a$ (*Suc* n)
proof fix f
show $f \in$ *carrier* $P \longrightarrow$
pderiv $(f \otimes_P$ *up-ring.monom* $P\ a$ (*Suc* $n)) = f \otimes_P$ *pderiv* (*up-ring.monom* $P\ a$ (*Suc* $n)) \oplus_P$ *pderiv* $f \otimes_P$ *up-ring.monom* $P\ a$ (*Suc* n)
proof
assume *A*: $f \in$ *carrier* P
show *pderiv* $(f \otimes_P$ *up-ring.monom* $P\ a$ (*Suc* $n)) = f \otimes_P$ *pderiv* (*up-ring.monom* $P\ a$ (*Suc* $n)) \oplus_P$ *pderiv* $f \otimes_P$ *up-ring.monom* $P\ a$ (*Suc* n)
proof(*cases* $n = 0$)
case *True*
have *0*: $(f \otimes_P$ *up-ring.monom* $P\ a$ (*Suc* $n)) = a \odot_P f \otimes_P$ *up-ring.monom* $P\ \mathbf{1}\ 1$
proof −
have $\forall n.$ *up-ring.monom* $P\ a\ n \in$ *carrier* P
using *assms*(*2*) *is-UP-monomE*(*1*) *monom-is-UP-monom*(*1*) **by** *presburger*
then show *?thesis*

by (*metis A P.m-assoc P.m-comm R.one-closed True assms*(2) *is-UP-monomE*(1) *monom-Suc*(2) *monom-is-UP-monom*(1) *monom-mult-is-smult*)
qed
have *1*: $f \otimes_P pderiv\ (up\text{-}ring.monom\ P\ a\ (Suc\ n)) = a \odot_P f$
using *assms True*
by (*metis A One-nat-def P.m-comm R.add.nat-pow-eone diff-Suc-1 is-UP-monomE*(1) *is-UP-monomI monom-mult-is-smult pderiv-monom*)
have *2*: $pderiv\ f \otimes_P up\text{-}ring.monom\ P\ a\ (Suc\ n) = a \odot_P (pderiv\ f \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1)$
using *A assms* **unfolding** *True*
by (*metis P.m-lcomm R.one-closed UP-mult-closed is-UP-monomE*(1) *monom-Suc*(2) *monom-is-UP-monom*(1) *monom-mult-is-smult pderiv-closed*)
have *3*: $a \odot_P f \oplus_P a \odot_P (pderiv\ f \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1) = a \odot_P (f \oplus_P (pderiv\ f \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1))$
using *assms A P.m-closed R.one-closed is-UP-monomE*(1) *monom-is-UP-monom*(1) *pderiv-closed smult-r-distr* **by** *presburger*
show *?thesis*
unfolding *0 1 2 3*
using *A times-x-product-rule P.m-closed R.one-closed UP-smult-assoc2 assms*(2) *is-UP-monomE*(1) *monom-is-UP-monom*(1) *pderiv-smult* **by** *presburger*
next
case *False*
have *IH*: $pderiv\ ((f \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1) \otimes_P up\text{-}ring.monom\ P\ a\ n) = (f \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1) \otimes_P pderiv\ (up\text{-}ring.monom\ P\ a\ n) \oplus_P pderiv\ (f \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1) \otimes_P up\text{-}ring.monom\ P\ a\ n$
using *Suc A P.m-closed R.one-closed is-UP-monomE*(1) *is-UP-monomI* **by** *presburger*
have *0*: $f \otimes_P up\text{-}ring.monom\ P\ a\ (Suc\ n) = (f \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1) \otimes_P up\text{-}ring.monom\ P\ a\ n$
using *A R.one-closed UP-m-assoc assms*(2) *is-UP-monomE*(1) *monom-Suc*(1) *monom-is-UP-monom*(1) **by** *presburger*
have *1*: $(f \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1) \otimes_P pderiv\ (up\text{-}ring.monom\ P\ a\ n) \oplus_P pderiv\ (f \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1) \otimes_P up\text{-}ring.monom\ P\ a\ n =$
$(f \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1) \otimes_P pderiv\ (up\text{-}ring.monom\ P\ a\ n) \oplus_P (f \oplus_P pderiv\ f \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1) \otimes_P up\text{-}ring.monom\ P\ a\ n$
using *A times-x-product-rule* **by** *presburger*
have *2*: $(f \otimes_P up\text{-}ring.monom\ P\ \mathbf{1}\ 1) \otimes_P pderiv\ (up\text{-}ring.monom\ P\ a\ n) = (f \otimes_P up\text{-}ring.monom\ P\ ([n]{\cdot}a)\ n)$
proof−
have *20*: $up\text{-}ring.monom\ P\ ([n] \cdot a)\ (n) = up\text{-}ring.monom\ P\ \mathbf{1}\ 1 \otimes_P up\text{-}ring.monom\ P\ ([n] \cdot a)\ (n - 1)$
using *A assms False monom-mult*[*of* **1** [*n*]·*a 1 n*−*1*]
by (*metis R.add.nat-pow-closed R.l-one R.one-closed Suc-eq-plus1 add.commute add-eq-if*)
show *?thesis* **unfolding** *20* **using** *assms A False pderiv-monom*[*of a n*]
using *P.m-assoc R.one-closed is-UP-monomE*(1) *monom-is-UP-monom*(1) **by** *simp*
qed
have *3*: $(f \otimes_P up\text{-}ring.monom\ P\ ([n]{\cdot}a)\ n) = [n]{\cdot}_P (f \otimes_P up\text{-}ring.monom\ P\ a$

n)
using *A assms* **by** (*metis P.add-pow-rdistr add-nat-pow-monom is-UP-monomE*(*1*) *monom-is-UP-monom*(*1*))
have *4*: *pderiv* (*f* ⊗*P up-ring.monom P* **1** *1*) = (*f* ⊕*P pderiv f* ⊗*P up-ring.monom P* **1** *1*)
using *times-x-product-rule A* **by** *blast*
have *5*: (*f* ⊕*P pderiv f* ⊗*P up-ring.monom P* **1** *1*) ⊗*P up-ring.monom P a n* =
(*f* ⊗*P up-ring.monom P a n*) ⊕*P* (*pderiv f* ⊗*P up-ring.monom P* **1** *1* ⊗*P up-ring.monom P a n*)
using *A assms* **by** (*meson P.l-distr P.m-closed R.one-closed is-UP-monomE*(*1*) *is-UP-monomI pderiv-closed*)
have *6*: (*f* ⊕*P pderiv f* ⊗*P up-ring.monom P* **1** *1*) ⊗*P up-ring.monom P a n* =
(*f* ⊗*P up-ring.monom P a n*) ⊕*P* (*pderiv f* ⊗*P up-ring.monom P* **1** *1* ⊗*P up-ring.monom P a n*)
using *A assms False 5* **by** *blast*
have *7*: (*f* ⊗*Pup-ring.monom P* **1** *1*) ⊗*P pderiv* (*up-ring.monom P a n*) ⊕*P pderiv* (*f* ⊗*Pup-ring.monom P* **1** *1*) ⊗*P up-ring.monom P a n* =
[(*Suc n*)] ·*P* (*f* ⊗*P up-ring.monom P a n*) ⊕*P pderiv f* ⊗*P up-ring.monom P* **1** *1* ⊗*P up-ring.monom P a n*
unfolding *2 3 5 6* **using** *assms A P.a-assoc*
by (*smt* (*verit*) *1 2 3 6 P.add.nat-pow-Suc P.m-closed R.one-closed is-UP-monomE*(*1*) *monom-is-UP-monom*(*1*) *pderiv-closed*)
have *8*: *pderiv* (*f* ⊗*P up-ring.monom P a* (*Suc n*)) = *pderiv* ((*f* ⊗*Pup-ring.monom P* **1** *1*) ⊗*P up-ring.monom P a n*)
using *A assms 0* **by** *presburger*
show *pderiv* (*f* ⊗*P up-ring.monom P a* (*Suc n*)) = *f* ⊗*P pderiv* (*up-ring.monom P a* (*Suc n*)) ⊕*P pderiv f* ⊗*P up-ring.monom P a* (*Suc n*)
unfolding *8 IH 0 1 2 3 4 5 6*
by (*smt* (*verit*) *2 4 6 7 A P.add-pow-rdistr R.one-closed UP-m-assoc add-nat-pow-monom assms*(*2*) *diff-Suc-1 is-UP-monomE*(*1*) *is-UP-monomI monom-Suc*(*1*) *pderiv-closed pderiv-monom*)
qed
qed
qed
qed
thus *?thesis* **using** *assms* **by** *blast*
qed

lemma(**in** *UP-cring*) *product-rule*:
assumes *f* ∈ *carrier* (*UP R*)
assumes *g* ∈ *carrier* (*UP R*)
shows *pderiv* (*f* ⊗*UP R g*) = (*pderiv f* ⊗*UP R g*) ⊕*UP R* (*f* ⊗*UP R pderiv g*)
proof(*rule poly-induct3*[*of f*])
show *f* ∈ *carrier P*
using *assms* **unfolding** *P-def* **by** *blast*
show ⋀*p q*. *q* ∈ *carrier P* ⟹
p ∈ *carrier P* ⟹

pderiv (*p* $\otimes_{UP\ R}$ *g*) = *pderiv p* $\otimes_{UP\ R}$ *g* $\oplus_{UP\ R}$ *p* $\otimes_{UP\ R}$ *pderiv g* $\Longrightarrow$
pderiv (*q* $\otimes_{UP\ R}$ *g*) = *pderiv q* $\otimes_{UP\ R}$ *g* $\oplus_{UP\ R}$ *q* $\otimes_{UP\ R}$ *pderiv g* $\Longrightarrow$
pderiv ((*p* $\oplus_P$ *q*) $\otimes_{UP\ R}$ *g*) = *pderiv* (*p* $\oplus_P$ *q*) $\otimes_{UP\ R}$ *g* $\oplus_{UP\ R}$ (*p* $\oplus_P$ *q*) $\otimes_{UP\ R}$ *pderiv g*

proof− **fix** *p q*

assume *A*: *q* ∈ *carrier P p* ∈ *carrier P*
pderiv (*p* $\otimes_{UP\ R}$ *g*) = *pderiv p* $\otimes_{UP\ R}$ *g* $\oplus_{UP\ R}$ *p* $\otimes_{UP\ R}$ *pderiv g*
pderiv (*q* $\otimes_{UP\ R}$ *g*) = *pderiv q* $\otimes_{UP\ R}$ *g* $\oplus_{UP\ R}$ *q* $\otimes_{UP\ R}$ *pderiv g*

have *0*: (*p* $\oplus_P$ *q*) $\otimes_{UP\ R}$ *g* = *p* $\otimes_{UP\ R}$ *g* $\oplus_{UP\ R}$ *q* $\otimes_{UP\ R}$ *g*
using *A assms* **unfolding** *P-def* **using** *P-def UP-l-distr* **by** *blast*

have *1*: *pderiv* ((*p* $\oplus_P$ *q*) $\otimes_{UP\ R}$ *g*) = *pderiv* (*p* $\otimes_{UP\ R}$ *g*) $\oplus_{UP\ R}$ *pderiv* (*q* $\otimes_{UP\ R}$ *g*)
unfolding *0* **using** *pderiv-add*[*of p* $\otimes_P$ *g q* $\otimes_P$ *g*] **unfolding** *P-def*
using *A*(*1*) *A*(*2*) *P-def UP-mult-closed assms*(*2*) **by** *blast*

have *2*: *pderiv* ((*p* $\oplus_P$ *q*) $\otimes_{UP\ R}$ *g*) = *pderiv p* $\otimes_{UP\ R}$ *g* $\oplus_{UP\ R}$ *p* $\otimes_{UP\ R}$ *pderiv g* $\oplus_{UP\ R}$ (*pderiv q* $\otimes_{UP\ R}$ *g* $\oplus_{UP\ R}$ *q* $\otimes_{UP\ R}$ *pderiv g*)
unfolding *1 A* **by** *blast*

have *3*: *pderiv* ((*p* $\oplus_P$ *q*) $\otimes_{UP\ R}$ *g*) = *pderiv p* $\otimes_{UP\ R}$ *g* $\oplus_{UP\ R}$ *pderiv q* $\otimes_{UP\ R}$ *g* $\oplus_{UP\ R}$ *p* $\otimes_{UP\ R}$ *pderiv g* $\oplus_{UP\ R}$ *q* $\otimes_{UP\ R}$ *pderiv g*
using *A assms* **by** (*metis 1 P.add.m-assoc P.add.m-lcomm P.m-closed P-def pderiv-closed*)

have *4*: *pderiv* ((*p* $\oplus_P$ *q*) $\otimes_{UP\ R}$ *g*) = (*pderiv p* $\otimes_{UP\ R}$ *g* $\oplus_{UP\ R}$ *pderiv q* $\otimes_{UP\ R}$ *g*) $\oplus_{UP\ R}$ (*p* $\otimes_{UP\ R}$ *pderiv g* $\oplus_{UP\ R}$ *q* $\otimes_{UP\ R}$ *pderiv g*)
unfolding *3* **using** *A assms P-def UP-a-assoc UP-a-closed UP-mult-closed pderiv-closed* **by** *auto*

have *5*: *pderiv* ((*p* $\oplus_P$ *q*) $\otimes_{UP\ R}$ *g*) = ((*pderiv p* $\oplus_{UP\ R}$ *pderiv q*) $\otimes_{UP\ R}$ *g*) $\oplus_{UP\ R}$ ((*p* $\oplus_{UP\ R}$ *q*) $\otimes_{UP\ R}$ *pderiv g*)
unfolding *4* **using** *A assms* **by** (*metis P.l-distr P-def pderiv-closed*)

have *6*: *pderiv* ((*p* $\oplus_P$ *q*) $\otimes_{UP\ R}$ *g*) = ((*pderiv* (*p* $\oplus_P$ *q*)) $\otimes_{UP\ R}$ *g*) $\oplus_{UP\ R}$ ((*p* $\oplus_{UP\ R}$ *q*) $\otimes_{UP\ R}$ *pderiv g*)
unfolding *5* **using** *A assms*
by (*metis P-def pderiv-add*)

show *pderiv* ((*p* $\oplus_P$ *q*) $\otimes_{UP\ R}$ *g*) = *pderiv* (*p* $\oplus_P$ *q*) $\otimes_{UP\ R}$ *g* $\oplus_{UP\ R}$ (*p* $\oplus_P$ *q*) $\otimes_{UP\ R}$ *pderiv g*
unfolding *6* **using** *A assms P-def* **by** *blast*

qed

show $\bigwedge$*a n*. *a* ∈ *carrier R* $\Longrightarrow$
pderiv (*up-ring.monom P a n* $\otimes_{UP\ R}$ *g*) = *pderiv* (*up-ring.monom P a n*) $\otimes_{UP\ R}$ *g* $\oplus_{UP\ R}$ *up-ring.monom P a n* $\otimes_{UP\ R}$ *pderiv g*
using *P-def UP-m-comm assms*(*2*) *is-UP-monomE*(*1*) *monom-is-UP-monom*(*1*) *monom-product-rule pderiv-closed* **by** *presburger*

qed

8.5 The Chain Rule

lemma(**in** *UP-cring*) *chain-rule*:
assumes *f* ∈ *carrier P*
assumes *g* ∈ *carrier P*
shows *pderiv* (*compose R f g*) = *compose R* (*pderiv f*) *g* $\otimes_{UP\ R}$ *pderiv g*

proof(*rule poly-induct3*[*of f*])
show $f \in$ *carrier P*
using *assms* **by** *blast*
show $\bigwedge p\ q.\ q \in$ *carrier P* $\Longrightarrow$
$p \in$ *carrier P* $\Longrightarrow$
pderiv (*Cring-Poly.compose R p g*) = *Cring-Poly.compose R* (*pderiv p*) *g* $\otimes_{UP\ R}$ *pderiv g* $\Longrightarrow$
pderiv (*Cring-Poly.compose R q g*) = *Cring-Poly.compose R* (*pderiv q*) *g* $\otimes_{UP\ R}$ *pderiv g* $\Longrightarrow$
pderiv (*Cring-Poly.compose R* (*p* $\oplus_P$ *q*) *g*) = *Cring-Poly.compose R* (*pderiv* (*p* $\oplus_P$ *q*)) *g* $\otimes_{UP\ R}$ *pderiv g*
using *pderiv-add sub-add*
by (*smt* (*verit*) *P-def UP-a-closed UP-m-comm UP-r-distr assms*(2) *pderiv-closed sub-closed*)
show $\bigwedge a\ n.\ a \in$ *carrier R* $\Longrightarrow$
pderiv (*compose R* (*up-ring.monom P a n*) *g*) = *compose R* (*pderiv* (*up-ring.monom P a n*)) *g* $\otimes_{UP\ R}$ *pderiv g*
proof−
fix *a n* **assume** *A*: $a \in$ *carrier R*
show *pderiv* (*compose R* (*up-ring.monom P a n*) *g*) = *compose R* (*pderiv* (*up-ring.monom P a n*)) *g* $\otimes_{UP\ R}$ *pderiv g*
proof(*induction n*)
case *0*
have *00*: (*compose R* (*up-ring.monom P a 0*) *g*) = (*up-ring.monom P a 0*)
using *A P-def assms*(2) *deg-const is-UP-monom-def monom-is-UP-monom*(1) *sub-const* **by** *presburger*
have *01*: *pderiv* (*up-ring.monom P a 0*) = $\mathbf{0}_P$
using *A pderiv-const* **by** *blast*
show *?case* **unfolding** *00 01*
by (*metis P.l-null P-def UP-zero-closed assms*(2) *deg-zero pderiv-closed sub-const*)
next
case (*Suc n*)
show *pderiv* (*Cring-Poly.compose R* (*up-ring.monom P a* (*Suc n*)) *g*) = *Cring-Poly.compose R* (*pderiv* (*up-ring.monom P a* (*Suc n*))) *g* $\otimes_{UP\ R}$ *pderiv g*
proof(*cases n* = *0*)
case *True*
have *0*: *compose R* (*up-ring.monom P a* (*Suc n*)) *g* = *a* $\odot_P$ *g*
using *A assms sub-monom-deg-one*[*of g a*] **unfolding** *True* **using** *One-nat-def*
by *presburger*
have *1*: (*pderiv* (*up-ring.monom P a* (*Suc n*))) = *up-ring.monom P a 0*
unfolding *True*
proof −
have *pderiv* (*up-ring.monom P a 0*) = $\mathbf{0}_P$
using *A pderiv-const* **by** *blast*
then show *pderiv* (*up-ring.monom P a* (*Suc 0*)) = *up-ring.monom P a 0*
using *A lcf-monom*(1) *P-def X-closed deg-const deg-nzero-nzero is-UP-monomE*(1) *monom-Suc*(2) *monom-is-UP-monom*(1) *monom-rep-X-pow pderiv-monom*

poly-shift-degree-zero poly-shift-eq sub-monom(2) sub-monom-deg-one to-poly-inverse to-poly-mult-simp(2)
by (*metis* (*no-types*, *lifting*) *P.l-null P.r-zero X-poly-def times-x-product-rule*)
qed
then show *?thesis* **unfolding** *0 1*
using *A P-def assms(2) deg-const is-UP-monomE(1) monom-is-UP-monom(1) monom-mult-is-smult pderiv-closed pderiv-smult sub-const*
by *presburger*
next
case *False*
have *0*: *compose R* (*up-ring.monom P a* (*Suc n*)) *g* = (*compose R* (*up-ring.monom P a n*) *g*) $\otimes_P$ (*compose R* (*up-ring.monom P* **1** *1*) *g*)
using *assms A* **by** (*metis R.one-closed monom-Suc(2) monom-closed sub-mult*)
have *1*: *compose R* (*up-ring.monom P a* (*Suc n*)) *g* = (*compose R* (*up-ring.monom P a n*) *g*) $\otimes_P$ *g*
unfolding *0* **using** *A assms*
by (*metis P-def R.one-closed UP-cring.lcf-monom(1) UP-cring.to-poly-inverse UP-cring-axioms UP-l-one UP-one-closed deg-one monom-one sub-monom-deg-one to-poly-mult-simp(1)*)
have *2*: *pderiv* (*compose R* (*up-ring.monom P a* (*Suc n*)) *g*) =
((*pderiv* (*compose R* (*up-ring.monom P a n*) *g*)) $\otimes_P$ *g*) $\oplus_P$ ((*compose R* (*up-ring.monom P a n*) *g*) $\otimes_P$ *pderiv g*)
unfolding *1* **unfolding** *P-def* **apply**(*rule product-rule*)
using *A assms* **unfolding** *P-def* **using** *P-def is-UP-monomE(1) is-UP-monomI rev-sub-closed sub-rev-sub* **apply** *presburger*
using *assms* **unfolding** *P-def* **by** *blast*
have *3*: *pderiv* (*compose R* (*up-ring.monom P a* (*Suc n*)) *g*) =
(*compose R* (*pderiv* (*up-ring.monom P a n*)) *g* $\otimes_{UP\ R}$ *pderiv g* $\otimes_P$ *g*) $\oplus_P$ ((*compose R* (*up-ring.monom P a n*) *g*) $\otimes_P$ *pderiv g*)
unfolding *2 Suc* **by** *blast*
have *4*: *pderiv* (*compose R* (*up-ring.monom P a* (*Suc n*)) *g*) =
((*compose R* (*pderiv* (*up-ring.monom P a n*)) *g* $\otimes_P$ *g*) $\otimes_{UP\ R}$ *pderiv g*) $\oplus_P$ ((*compose R* (*up-ring.monom P a n*) *g*) $\otimes_P$ *pderiv g*)
unfolding *3* **using** *A assms m-assoc m-comm*
by (*smt* (*verit*) *P-def monom-closed monom-rep-X-pow pderiv-closed sub-closed*)
have *5*: *pderiv* (*compose R* (*up-ring.monom P a* (*Suc n*)) *g*) =
((*compose R* (*pderiv* (*up-ring.monom P a n*)) *g* $\otimes_P$ *g*) $\oplus_P$ (*compose R* (*up-ring.monom P a n*) *g*)) $\otimes_P$ *pderiv g*
unfolding *4* **using** *A assms*
by (*metis P.l-distr P.m-closed P-def UP-cring.pderiv-closed UP-cring-axioms monom-closed sub-closed*)
have *6*: *compose R* (*pderiv* (*up-ring.monom P a n*)) *g* $\otimes_P$ *g* = $[n]\cdot_P$*compose R* ((*up-ring.monom P a n*)) *g*
proof−
have *60*: (*pderiv* (*up-ring.monom P a n*)) = (*up-ring.monom P* ($[n]\cdot a$) (*n*−*1*))
using *A assms pderiv-monom* **by** *blast*
have *61*: *compose R* (*pderiv* (*up-ring.monom P a n*)) *g* $\otimes_P$ *g* = *compose R*

((*up-ring.monom P* ([*n*]·*a*) (*n*−*1*))) *g* ⊗$_P$ (*compose R* (*up-ring.monom P* **1** *1*) *g*)
unfolding *60* **using** *A assms sub-monom-deg-one*[*of g* **1**] *R.one-closed smult-one* **by** *presburger*
have *62*: *compose R* (*pderiv* (*up-ring.monom P a n*)) *g* ⊗$_P$ *g* = *compose R* (*up-ring.monom P* ([*n*]·*a*) *n*) *g*
unfolding *61* **using** *False A assms sub-mult*[*of g up-ring.monom P* ([*n*] · *a*) (*n* − *1*) *up-ring.monom P* **1** *1*] *monom-mult*[*of* [*n*]·*a* **1** *n*−*1 1*]
by (*metis Nat.add-0-right R.add.nat-pow-closed R.one-closed R.r-one Suc-eq-plus1 add-eq-if monom-closed*)
have *63*: ⋀*k*::*nat*. *Cring-Poly.compose R* (*up-ring.monom P* ([*k*] · *a*) *n*) *g* = [*k*] ·$_P$*Cring-Poly.compose R* (*up-ring.monom P a n*) *g*
proof− **fix** *k*::*nat* **show** *Cring-Poly.compose R* (*up-ring.monom P* ([*k*] · *a*) *n*) *g* = [*k*] ·$_P$*Cring-Poly.compose R* (*up-ring.monom P a n*) *g*
apply(*induction k*)
using *UP-zero-closed assms*(*2*) *deg-zero monom-zero sub-const*
apply (*metis A P.add.nat-pow-0 add-nat-pow-monom*)
proof−
fix *k*::*nat*
assume *a*: *Cring-Poly.compose R* (*monom P* ([*k*] · *a*) *n*) *g* = [*k*] ·$_P$ *Cring-Poly.compose R* (*monom P a n*) *g*
have *0*: (*monom P* ([*Suc k*] · *a*) *n*) = [*Suc k*] · *a* ⊙$_P$(*monom P* **1** *n*)
by (*simp add*: *A monic-monom-smult*)
have *1*: (*monom P* ([*Suc k*] · *a*) *n*) = [*k*] · *a* ⊙$_P$(*monom P* **1** *n*) ⊕$_P$*a* ⊙$_P$(*monom P* **1** *n*)
unfolding *0*
by (*simp add*: *A UP-smult-l-distr*)
show *Cring-Poly.compose R* (*monom P* ([*Suc k*] · *a*) *n*) *g* = [*Suc k*] ·$_P$ (*Cring-Poly.compose R* (*monom P a n*) *g*)
unfolding *1*
by (*simp add*: *A a assms*(*2*) *monic-monom-smult sub-add*)
qed
qed
have *64*: *Cring-Poly.compose R* (*up-ring.monom P* ([*n*] · *a*) *n*) *g* = [*n*] ·$_P$*Cring-Poly.compose R* (*up-ring.monom P a n*) *g*
using *63* **by** *blast*
show *?thesis* **unfolding** *62 64* **by** *blast*
qed
have *63*: ⋀*k*::*nat*. *Cring-Poly.compose R* (*up-ring.monom P* ([*k*] · *a*) *n*) *g* = [*k*] ·$_P$*Cring-Poly.compose R* (*up-ring.monom P a n*) *g*
proof− **fix** *k*::*nat* **show** *Cring-Poly.compose R* (*up-ring.monom P* ([*k*] · *a*) *n*) *g* = [*k*] ·$_P$*Cring-Poly.compose R* (*up-ring.monom P a n*) *g*
apply(*induction k*)
using *UP-zero-closed assms*(*2*) *deg-zero monom-zero sub-const*
apply (*metis A P.add.nat-pow-0 add-nat-pow-monom*)
using *A P.add.nat-pow-Suc add-nat-pow-monom assms*(*2*) *is-UP-monomE*(*1*) *monom-is-UP-monom*(*1*) *rev-sub-add sub-rev-sub*
by (*metis P.add.nat-pow-closed*)
qed
have *7*: ([*n*] ·$_P$ *Cring-Poly.compose R* (*up-ring.monom P a n*) *g* ⊕$_P$ *Cring-Poly.compose*

R (*up-ring.monom P a n*) *g*) =
[*Suc n*] ·$_P$ (*Cring-Poly.compose R* (*up-ring.monom P a n*) *g*)
using *A assms P.add.nat-pow-Suc* **by** *presburger*
have *8*: [*Suc n*] ·$_P$ *Cring-Poly.compose R* (*up-ring.monom P a n*) *g* ⊗$_P$ *pderiv g* = *Cring-Poly.compose R* (*up-ring.monom P* ([*Suc n*] · *a*) *n*) *g* ⊗$_P$ *pderiv g*
unfolding *63*[*of Suc n*] **by** *blast*
show *?thesis* **unfolding** *5 6 7 8* **using** *A assms pderiv-monom*[*of a Suc n*]
using *P-def diff-Suc-1* **by** *metis*
qed
qed
qed
qed

lemma *deriv-prod-rule-times-monom*:
assumes *a* ∈ *carrier R*
assumes *b* ∈ *carrier R*
assumes *q* ∈ *carrier P*
shows *deriv* ((*monom P a n*) ⊗$_P$ *q*) *b* = (*deriv* (*monom P a n*) *b*) ⊗ (*to-fun q b*) ⊕ (*to-fun* (*monom P a n*) *b*) ⊗ *deriv q b*
proof(*rule poly-induct3*[*of q*])
show *q* ∈ *carrier P*
using *assms* **by** *simp*
show ⋀*p q*. *q* ∈ *carrier P* ⟹
p ∈ *carrier P* ⟹
deriv (*monom P a n* ⊗$_P$ *p*) *b* = *deriv* (*monom P a n*) *b* ⊗ *to-fun p b* ⊕ *to-fun* (*monom P a n*) *b* ⊗ *deriv p b* ⟹
deriv (*monom P a n* ⊗$_P$ *q*) *b* = *deriv* (*monom P a n*) *b* ⊗ *to-fun q b* ⊕ *to-fun* (*monom P a n*) *b* ⊗ *deriv q b* ⟹
deriv (*monom P a n* ⊗$_P$ (*p* ⊕$_P$ *q*)) *b* = *deriv* (*monom P a n*) *b* ⊗ *to-fun* (*p* ⊕$_P$ *q*) *b* ⊕ *to-fun* (*monom P a n*) *b* ⊗ *deriv* (*p* ⊕$_P$ *q*) *b*
proof− **fix** *p q* **assume** *A*: *q* ∈ *carrier P* *p* ∈ *carrier P*
deriv (*monom P a n* ⊗$_P$ *p*) *b* = *deriv* (*monom P a n*) *b* ⊗ *to-fun p b* ⊕ *to-fun* (*monom P a n*) *b* ⊗ *deriv p b*
deriv (*monom P a n* ⊗$_P$ *q*) *b* = *deriv* (*monom P a n*) *b* ⊗ *to-fun q b* ⊕ *to-fun* (*monom P a n*) *b* ⊗ *deriv q b*
have *deriv* (*monom P a n* ⊗$_P$ (*p* ⊕$_P$ *q*)) *b* = *deriv* (*monom P a n*) *b* ⊗ *to-fun p b* ⊕ *to-fun* (*monom P a n*) *b* ⊗ *deriv p b*
⊕*deriv* (*monom P a n*) *b* ⊗ *to-fun q b* ⊕ *to-fun* (*monom P a n*) *b* ⊗ *deriv q b*
using *A assms*
by (*simp add*: *P.r-distr R.add.m-assoc deriv-add deriv-closed to-fun-closed*)

hence *deriv* (*monom P a n* ⊗$_P$ (*p* ⊕$_P$ *q*)) *b* = *deriv* (*monom P a n*) *b* ⊗ *to-fun p b* ⊕*deriv* (*monom P a n*) *b* ⊗ *to-fun q b*
⊕ *to-fun* (*monom P a n*) *b* ⊗ *deriv p b* ⊕ *to-fun* (*monom P a n*) *b* ⊗ *deriv q b*
using *A*(*1*) *A*(*2*) *R.add.m-assoc R.add.m-comm assms*(*1*) *assms*(*2*) *deriv-closed to-fun-closed* **by** *auto*

hence *deriv* (*monom P a n* $\otimes_P$ (*p* $\oplus_P$ *q*)) *b* = *deriv* (*monom P a n*) *b* $\otimes$ (*to-fun p b* $\oplus$ *to-fun q b*)
$\oplus$ *to-fun* (*monom P a n*) *b* $\otimes$ (*deriv p b* $\oplus$ *deriv q b*)
by (*simp add*: *A*(*1*) *A*(*2*) *R.add.m-assoc R.r-distr assms*(*1*) *assms*(*2*) *deriv-closed to-fun-closed*)
thus *deriv* (*monom P a n* $\otimes_P$ (*p* $\oplus_P$ *q*)) *b* = *deriv* (*monom P a n*) *b* $\otimes$ *to-fun* (*p* $\oplus_P$ *q*) *b* $\oplus$ *to-fun* (*monom P a n*) *b* $\otimes$ *deriv* (*p* $\oplus_P$ *q*) *b*
by (*simp add*: *A*(*1*) *A*(*2*) *assms*(*2*) *deriv-add to-fun-plus*)
qed
show $\bigwedge$*c m*. *c* $\in$ *carrier R* $\Longrightarrow$ *deriv* (*monom P a n* $\otimes_P$ *monom P c m*) *b* = *deriv* (*monom P a n*) *b* $\otimes$ *to-fun* (*monom P c m*) *b* $\oplus$ *to-fun* (*monom P a n*) *b* $\otimes$ *deriv* (*monom P c m*) *b*
proof− **fix** *c m* **assume** *A*: *c* $\in$ *carrier R*
show *deriv* (*monom P a n* $\otimes_P$ *monom P c m*) *b* = *deriv* (*monom P a n*) *b* $\otimes$ *to-fun* (*monom P c m*) *b* $\oplus$ *to-fun* (*monom P a n*) *b* $\otimes$ *deriv* (*monom P c m*) *b*
proof(*cases n* = *0*)
case *True*
have *LHS*: *deriv* (*monom P a n* $\otimes_P$ *monom P c m*) *b* = *deriv* (*monom P* (*a* $\otimes$ *c*) *m*) *b*
by (*metis A True add.left-neutral assms*(*1*) *monom-mult*)
have *RHS*: *deriv* (*monom P a n*) *b* $\otimes$ *to-fun* (*monom P c m*) *b* $\oplus$ *to-fun* (*monom P a n*) *b* $\otimes$ *deriv* (*monom P c m*) *b*
= *a* $\otimes$ *deriv* (*monom P c m*) *b*
using *deriv-const to-fun-monom A True assms*(*1*) *assms*(*2*) *deriv-closed* **by** *auto*
show *?thesis* **using** *A assms LHS RHS deriv-monom*
by (*smt* (*verit*) *R.add.nat-pow-closed R.add-pow-rdistr R.m-assoc R.m-closed R.nat-pow-closed*)
next
case *False*
show *?thesis*
proof(*cases m* = *0*)
case *True*
have *LHS*: *deriv* (*monom P a n* $\otimes_P$ *monom P c m*) *b* = *deriv* (*monom P* (*a* $\otimes$ *c*) *n*) *b*
by (*metis A True add.comm-neutral assms*(*1*) *monom-mult*)
have *RHS*: *deriv* (*monom P a n*) *b* $\otimes$ *to-fun* (*monom P c m*) *b* $\oplus$ *to-fun* (*monom P a n*) *b* $\otimes$ *deriv* (*monom P c m*) *b*
= *c* $\otimes$ *deriv* (*monom P a n*) *b*
by (*metis* (*no-types*, *lifting*) *A lcf-monom*(*1*) *P-def R.m-closed R.m-comm R.r-null*
R.r-zero True UP-cring.to-fun-ctrm UP-cring-axioms assms(*1*) *assms*(*2*) *deg-const*
deriv-closed deriv-const to-fun-closed monom-closed)
show *?thesis* **using** *LHS RHS deriv-monom A assms*
by (*smt* (*verit*) *R.add.nat-pow-closed R.add-pow-ldistr R.m-assoc R.m-closed R.m-comm R.nat-pow-closed*)
next
case *F*: *False*

have *pos*: $n > 0$ $m > 0$
using *F False* **by** *auto*
have *RHS*: *deriv* (*monom P a n* $\otimes_P$ *monom P c m*) *b* = $[(n + m)] \cdot (a \otimes c) \otimes b\ [\frown]\ (n + m - 1)$
using *deriv-monom*[*of a* $\otimes$ *c b n* + *m*] *monom-mult*[*of a c n m*]
by (*simp add*: *A assms*(*1*) *assms*(*2*))
have *LHS*: *deriv* (*monom P a n*) *b* $\otimes$ *to-fun* (*monom P c m*) *b* $\oplus$ *to-fun* (*monom P a n*) *b* $\otimes$ *deriv* (*monom P c m*) *b*
$= [n]\cdot a \otimes (b[\frown](n-1)) \otimes c \otimes b[\frown]m \oplus a \otimes b[\frown]n \otimes [m]\cdot c \otimes (b[\frown](m-1))$
using *deriv-monom*[*of a b n*] *to-fun-monom*[*of a b n*]
deriv-monom[*of c b m*] *to-fun-monom*[*of c b m*] *A assms*
by (*simp add*: *R.m-assoc*)
have *0*: $[n]\cdot a \otimes (b[\frown](n-1)) \otimes c \otimes b[\frown]m = [n]\cdot a \otimes c \otimes b[\frown](n + m - 1)$
proof−
have $[n]\cdot a \otimes (b[\frown](n-1)) \otimes c \otimes b[\frown]m = [n]\cdot a \otimes c \otimes (b[\frown](n-1)) \otimes b[\frown]m$
by (*simp add*: *A R.m-lcomm R.semiring-axioms assms*(*1*) *assms*(*2*) *semiring.semiring-simprules*(*8*))
hence $[n]\cdot a \otimes (b[\frown](n-1)) \otimes c \otimes b[\frown]m = [n]\cdot a \otimes c \otimes ((b[\frown](n-1)) \otimes b[\frown]m)$
by (*simp add*: *A R.m-assoc assms*(*1*) *assms*(*2*))
thus *?thesis*
by (*simp add*: *False R.nat-pow-mult add-eq-if assms*(*2*))
qed
have *1*: $a \otimes b[\frown]n \otimes [m]\cdot c \otimes (b[\frown](m-1)) = a \otimes [m]\cdot c \otimes b[\frown](n + m - 1)$
proof−
have $a \otimes b[\frown]n \otimes [m]\cdot c \otimes (b[\frown](m-1)) = a \otimes [m]\cdot c \otimes b[\frown]n \otimes (b[\frown](m-1))$
using *A R.m-comm R.m-lcomm assms*(*1*) *assms*(*2*) **by** *auto*
hence $a \otimes b[\frown]n \otimes [m]\cdot c \otimes (b[\frown](m-1)) = a \otimes [m]\cdot c \otimes (b[\frown]n \otimes (b[\frown](m-1)))$
by (*simp add*: *A R.m-assoc assms*(*1*) *assms*(*2*))
thus *?thesis*
by (*simp add*: *F R.nat-pow-mult add.commute add-eq-if assms*(*2*))
qed
have *LHS*: *deriv* (*monom P a n*) *b* $\otimes$ *to-fun* (*monom P c m*) *b* $\oplus$ *to-fun* (*monom P a n*) *b* $\otimes$ *deriv* (*monom P c m*) *b*
$= [n]\cdot a \otimes c \otimes b[\frown](n + m - 1) \oplus a \otimes [m]\cdot c \otimes b[\frown](n + m - 1)$
using *LHS 0 1*
by *simp*
hence *LHS*: *deriv* (*monom P a n*) *b* $\otimes$ *to-fun* (*monom P c m*) *b* $\oplus$ *to-fun* (*monom P a n*) *b* $\otimes$ *deriv* (*monom P c m*) *b*
$= [n]\cdot (a \otimes c \otimes b[\frown](n + m - 1)) \oplus [m]\cdot (a \otimes c \otimes b[\frown](n + m - 1))$
by (*simp add*: *A R.add-pow-ldistr R.add-pow-rdistr assms*(*1*) *assms*(*2*))
show *?thesis* **using** *LHS RHS*
by (*simp add*: *A R.add.nat-pow-mult R.add-pow-ldistr assms*(*1*) *assms*(*2*))
qed

qed
qed
qed

lemma *deriv-prod-rule*:
assumes $p \in$ *carrier* P
assumes $q \in$ *carrier* P
assumes $a \in$ *carrier* R
shows *deriv* $(p \otimes_P q)$ a = *deriv* p $a \otimes$ (*to-fun* q a) $\oplus$ (*to-fun* p a) $\otimes$ *deriv* q a
proof(*rule poly-induct3*[*of* p])
show $p \in$ *carrier* P
using *assms*(*1*) **by** *simp*
show $\bigwedge p\ qa.$
$qa \in$ *carrier* $P \Longrightarrow$
$p \in$ *carrier* $P \Longrightarrow$
deriv $(p \otimes_P q)$ a = *deriv* p $a \otimes$ *to-fun* q $a \oplus$ *to-fun* p $a \otimes$ *deriv* q $a \Longrightarrow$
deriv $(qa \otimes_P q)$ a = *deriv* qa $a \otimes$ *to-fun* q $a \oplus$ *to-fun* qa $a \otimes$ *deriv* q $a \Longrightarrow$
deriv $((p \oplus_P qa) \otimes_P q)$ a = *deriv* $(p \oplus_P qa)$ $a \otimes$ *to-fun* q $a \oplus$ *to-fun* $(p \oplus_P qa)$ $a \otimes$ *deriv* q a
proof− **fix** f g **assume** A: $f \in$ *carrier* P $g \in$ *carrier* P
deriv $(f \otimes_P q)$ a = *deriv* f $a \otimes$ *to-fun* q $a \oplus$ *to-fun* f $a \otimes$ *deriv* q a
deriv $(g \otimes_P q)$ a = *deriv* g $a \otimes$ *to-fun* q $a \oplus$ *to-fun* g $a \otimes$ *deriv* q a
have *deriv* $((f \oplus_P g) \otimes_P q)$ a = *deriv* f $a \otimes$ *to-fun* q $a \oplus$ *to-fun* f $a \otimes$ *deriv* q $a \oplus$
deriv g $a \otimes$ *to-fun* q $a \oplus$ *to-fun* g $a \otimes$ *deriv* q a
using A *deriv-add*
by (*simp add*: *P.l-distr R.add.m-assoc assms*(*2*) *assms*(*3*) *deriv-closed to-fun-closed*)
hence *deriv* $((f \oplus_P g) \otimes_P q)$ a = *deriv* f $a \otimes$ *to-fun* q $a \oplus$ *deriv* g $a \otimes$ *to-fun* q $a \oplus$
to-fun f $a \otimes$ *deriv* q $a \oplus$ *to-fun* g $a \otimes$ *deriv* q a
using *R.a-comm R.a-assoc deriv-closed to-fun-closed assms*
by (*simp add*: $A(1)$ $A(2)$)
hence *deriv* $((f \oplus_P g) \otimes_P q)$ a = (*deriv* f $a \otimes$ *to-fun* q $a \oplus$ *deriv* g $a \otimes$ *to-fun* q a) $\oplus$
(*to-fun* f $a \otimes$ *deriv* q $a \oplus$ *to-fun* g $a \otimes$ *deriv* q a)
by (*simp add*: $A(1)$ $A(2)$ *R.add.m-assoc assms*(*2*) *assms*(*3*) *deriv-closed to-fun-closed*)
thus *deriv* $((f \oplus_P g) \otimes_P q)$ a = *deriv* $(f \oplus_P g)$ $a \otimes$ *to-fun* q $a \oplus$ *to-fun* $(f \oplus_P g)$ $a \otimes$ *deriv* q a
by (*simp add*: $A(1)$ $A(2)$ *R.l-distr assms*(*2*) *assms*(*3*) *deriv-add deriv-closed to-fun-closed to-fun-plus*)
qed
show $\bigwedge aa\ n.$ $aa \in$ *carrier* $R \Longrightarrow$ *deriv* (*monom* P aa $n \otimes_P q$) a = *deriv* (*monom* P aa n) $a \otimes$ *to-fun* q $a \oplus$ *to-fun* (*monom* P aa n) $a \otimes$ *deriv* q a
using *deriv-prod-rule-times-monom*
by (*simp add*: *assms*(*2*) *assms*(*3*))
qed

lemma *pderiv-eval-deriv-monom*:

assumes $a \in carrier\ R$
assumes $b \in carrier\ R$
shows *to-fun* (*pderiv* (*monom P a n*)) b = *deriv* (*monom P a n*) b
using *deriv-monom assms pderiv-monom*
by (*simp add*: *P-def UP-cring.to-fun-monom UP-cring-axioms*)

lemma *pderiv-eval-deriv*:
assumes $f \in carrier\ P$
assumes $a \in carrier\ R$
shows *deriv f a* = *to-fun* (*pderiv f*) *a*
apply(*rule poly-induct3*[*of f*])
apply (*simp add*: *assms*(*1*))
using *assms*(*2*) *deriv-add to-fun-plus pderiv-add pderiv-closed* **apply** *presburger*
using *assms*(*2*) *pderiv-eval-deriv-monom*
by *presburger*

Taking taylor expansions commutes with taking derivatives:

lemma(**in** *UP-cring*) *taylor-expansion-pderiv-comm*:
assumes $f \in carrier\ (UP\ R)$
assumes $c \in carrier\ R$
shows *pderiv* (*taylor-expansion R c f*) = *taylor-expansion R c* (*pderiv f*)
apply(*rule poly-induct3*[*of f*])
using *assms* **unfolding** *P-def* **apply** *blast*
proof−
fix *p q* **assume** *A*: $q \in carrier\ (UP\ R)$ $p \in carrier\ (UP\ R)$
pderiv (*taylor-expansion R c p*) = *taylor-expansion R c* (*pderiv p*)
pderiv (*taylor-expansion R c q*) = *taylor-expansion R c* (*pderiv q*)
have *0*: *pderiv* (*taylor-expansion R c* ($p \oplus_{UP\ R} q$)) = *pderiv* (*taylor-expansion R c p* $\oplus_{UP\ R}$ *taylor-expansion R c q*)
using *A P-def taylor-expansion-add assms*(*2*) **by** *presburger*
show *pderiv* (*taylor-expansion R c* ($p \oplus_{UP\ R} q$)) = *taylor-expansion R c* (*pderiv* ($p \oplus_{UP\ R} q$))
unfolding *0*
using *A*(*1*) *A*(*2*) *A*(*3*) *A*(*4*) *taylor-def UP-cring.taylor-closed UP-cring.taylor-expansion-add UP-cring.pderiv-add UP-cring.pderiv-closed UP-cring-axioms assms*(*2*) **by** *fastforce*
next
fix *a n* **assume** *A*: $a \in carrier\ R$
show *pderiv* (*taylor-expansion R c* (*up-ring.monom* (*UP R*) *a n*)) = *taylor-expansion R c* (*pderiv* (*up-ring.monom* (*UP R*) *a n*))
proof(*cases n* = *0*)
case *True*
have *0*: *deg R* (*taylor-expansion R c* (*up-ring.monom* (*UP R*) *a n*)) = *0*
unfolding *True*
using *P-def A assms taylor-def taylor-deg deg-const is-UP-monomE*(*1*) *monom-is-UP-monom*(*2*) **by** *presburger*
have *1*: (*pderiv* (*up-ring.monom* (*UP R*) *a n*)) = $\mathbf{0}_P$
unfolding *True* **using** *P-def A assms pderiv-const* **by** *blast*
show *?thesis* **unfolding** *1* **using** *0 A assms P-def*
by (*metis P.add.right-cancel taylor-closed taylor-def taylor-expansion-add*

UP-l-zero UP-zero-closed monom-closed pderiv-deg-0)
next
case *False*
have *0*: *pderiv* (*up-ring.monom* (*UP R*) *a n*) = (*up-ring.monom* (*UP R*) ([*n*]·*a*) (*n*−*1*))
using *A*
by (*simp add*: *UP-cring.pderiv-monom UP-cring-axioms*)
have *1*: *pderiv* (*taylor-expansion R c* (*up-ring.monom* (*UP R*) *a n*)) = (*Cring-Poly.compose R* (*up-ring.monom* (*UP R*) ([*n*]·*a*) (*n*−*1*)) (*X-plus c*)) ⊗$_P$ *pderiv* (*X-plus c*)
using *chain-rule*[*of up-ring.monom* (*UP R*) *a n X-plus c*] **unfolding** *0 taylor-expansion-def*
using *A P-def X-plus-closed assms*(*2*) *is-UP-monom-def monom-is-UP-monom*(*1*)
by *presburger*
have *2*: *pderiv* (*X-plus c*) = **1**$_P$
using *pderiv-add*[*of X-poly R to-poly c*] *P.l-null P.l-one P.r-zero P-def R.one-closed X-closed*
X-poly-def X-poly-plus-def assms(*2*) *monom-one pderiv-const to-poly-closed to-polynomial-def*
by (*metis times-x-product-rule*)
show *?thesis* **unfolding** *1 0 2 taylor-expansion-def*
by (*metis 1 2 A P.l-one P-def R.add.nat-pow-closed UP-m-comm UP-one-closed X-plus-closed assms*(*2*) *monom-closed sub-closed taylor-expansion-def*)
qed
qed

8.6 Linear Substitutions

lemma(**in** *UP-ring*) *lcoeff-Lcf*:
assumes *f* ∈ *carrier P*
shows *lcoeff f* = *lcf f*
unfolding *P-def*
using *assms coeff-simp*[*of f*] **by** *metis*

lemma(**in** *UP-cring*) *linear-sub-cfs*:
assumes *f* ∈ *carrier* (*UP R*)
assumes *d* ∈ *carrier R*
assumes *g* = *compose R f* (*up-ring.monom* (*UP R*) *d 1*)
shows *g i* = *d*[^]*i* ⊗ *f i*
proof−
have *0*: (*up-ring.monom* (*UP R*) *d 1*) ∈ *carrier* (*UP R*)
using *assms* **by** (*meson R.ring-axioms UP-ring.intro UP-ring.monom-closed*)
have *1*: (∀ *i*. *compose R f* (*up-ring.monom* (*UP R*) *d 1*) *i* = *d*[^]*i* ⊗ *f i*)
apply(*rule poly-induct3*[*of f*])
using *assms* **unfolding** *P-def* **apply** *blast*
proof−
show ⋀*p q*. *q* ∈ *carrier* (*UP R*) ⟹
p ∈ *carrier* (*UP R*) ⟹
∀ *i*. *Cring-Poly.compose R p* (*up-ring.monom* (*UP R*) *d 1*) *i* = *d* [^] *i* ⊗ *p i* ⟹

$\forall i.$ *Cring-Poly.compose* R q (*up-ring.monom* (*UP* R) d *1*) $i = d$ [^] $i \otimes$ q $i \Longrightarrow$
$\forall i.$ *Cring-Poly.compose* R ($p \oplus_{UP\ R} q$) (*up-ring.monom* (*UP* R) d *1*) i $= d$ [^] $i \otimes (p \oplus_{UP\ R} q)$ i
proof
fix p q i
assume *A*: $q \in$ *carrier* (*UP* R)
$p \in$ *carrier* (*UP* R)
$\forall i.$ *Cring-Poly.compose* R p (*up-ring.monom* (*UP* R) d *1*) $i = d$ [^] $i \otimes$ p i
$\forall i.$ *Cring-Poly.compose* R q (*up-ring.monom* (*UP* R) d *1*) $i = d$ [^] $i \otimes$ q i
show *Cring-Poly.compose* R ($p \oplus_{UP\ R} q$) (*up-ring.monom* (*UP* R) d *1*) $i = d$ [^] $i \otimes (p \oplus_{UP\ R} q)$ i
proof−
have *1*: *Cring-Poly.compose* R ($p \oplus_{UP\ R} q$) (*up-ring.monom* (*UP* R) d *1*) =
Cring-Poly.compose R p (*up-ring.monom* (*UP* R) d *1*) $\oplus_{UP\ R}$ *Cring-Poly.compose* R q (*up-ring.monom* (*UP* R) d *1*)
using *A*(*1*) *A*(*2*) *sub-add*[*of up-ring.monom* (*UP* R) d *1* q p] **unfolding** *P-def*
using *0 P-def sub-add* **by** *blast*
have *2*: *Cring-Poly.compose* R ($p \oplus_{UP\ R} q$) (*up-ring.monom* (*UP* R) d *1*) i =
Cring-Poly.compose R p (*up-ring.monom* (*UP* R) d *1*) $i \oplus$ *Cring-Poly.compose* R q (*up-ring.monom* (*UP* R) d *1*) i
using *1* **by** (*metis* (*no-types*, *lifting*) *0 A*(*1*) *A*(*2*) *P-def cfs-add sub-closed*)
have *3*: *Cring-Poly.compose* R ($p \oplus_{UP\ R} q$) (*up-ring.monom* (*UP* R) d *1*) i = d [^] $i \otimes p\ i \oplus d$ [^] $i \otimes q\ i$
unfolding *2* **using** *A* **by** *presburger*
have *4*: *Cring-Poly.compose* R ($p \oplus_{UP\ R} q$) (*up-ring.monom* (*UP* R) d *1*) i = d [^] $i \otimes (p\ i \oplus\ q\ i)$
using *3 A*(*1*) *A*(*2*) *R.nat-pow-closed R.r-distr UP-car-memE*(*1*) *assms*(*2*) **by** *presburger*
thus *Cring-Poly.compose* R ($p \oplus_{UP\ R} q$) (*up-ring.monom* (*UP* R) d *1*) $i = d$ [^] $i \otimes (p \oplus_{UP\ R} q)$ i
unfolding *4* **using** *A*(*1*) *A*(*2*) *P-def cfs-add* **by** *presburger*
qed
qed
show $\bigwedge a\ n.\ a \in$ *carrier* $R \Longrightarrow$
$\forall i.$ *Cring-Poly.compose* R (*up-ring.monom* (*UP* R) a n) (*up-ring.monom* (*UP* R) d *1*) $i = d$ [^] $i \otimes$ *up-ring.monom* (*UP* R) a n i
proof fix a n i **assume** *A*: $a \in$ *carrier* R
have *0*: *Cring-Poly.compose* R (*up-ring.monom* (*UP* R) a n) (*up-ring.monom* (*UP* R) d *1*) =
$a \odot_{UP\ R}$(*up-ring.monom* (*UP* R) d *1*)[^]$_{UP\ R} n$
using *assms A 0 P-def monom-sub* **by** *blast*
have *1*: *Cring-Poly.compose* R (*up-ring.monom* (*UP* R) a n) (*up-ring.monom* (*UP* R) d *1*) =

$a \odot_{UP\ R} (d[\widehat{}]n \odot_{UP\ R} (up\text{-}ring.monom\ (UP\ R)\ \mathbf{1}\ n))$
unfolding *0* **using** *A assms*
by (*metis P-def R.nat-pow-closed monic-monom-smult monom-pow mult.left-neutral*)
have *2*: *Cring-Poly.compose R* (*up-ring.monom* (*UP R*) *a n*) (*up-ring.monom* (*UP R*) *d 1*) =
$(a \otimes d[\widehat{}]n) \odot_{UP\ R} (up\text{-}ring.monom\ (UP\ R)\ \mathbf{1}\ n)$
unfolding *1* **using** *A assms*
by (*metis R.nat-pow-closed R.one-closed R.ring-axioms UP-ring.UP-smult-assoc1 UP-ring.intro UP-ring.monom-closed*)
show *Cring-Poly.compose R* (*up-ring.monom* (*UP R*) *a n*) (*up-ring.monom* (*UP R*) *d 1*) *i* = *d* [^] *i* ⊗ *up-ring.monom* (*UP R*) *a n i*
apply(*cases i* = *n*)
unfolding *2* **using** *A P-def R.m-closed R.m-comm R.nat-pow-closed assms*(*2*) *cfs-monom monic-monom-smult* **apply** *presburger*
using *A P-def R.m-closed R.nat-pow-closed R.r-null assms*(*2*) *cfs-monom monic-monom-smult* **by** *presburger*
qed
qed
show *?thesis* **using** *1* **unfolding** *assms*
by *blast*
qed

lemma(**in** *UP-cring*) *linear-sub-deriv*:
assumes *f* ∈ *carrier* (*UP R*)
assumes *d* ∈ *carrier R*
assumes *g* = *compose R f* (*up-ring.monom* (*UP R*) *d 1*)
assumes *c* ∈ *carrier R*
shows *pderiv g* = *d* $\odot_{UP\ R}$ *compose R* (*pderiv f*) (*up-ring.monom* (*UP R*) *d 1*)
unfolding *assms*
proof(*rule poly-induct3*[*of f*])
show *f* ∈ *carrier P*
using *assms* **unfolding** *P-def* **by** *blast*
show ⋀ *p q*. *q* ∈ *carrier P* ⟹
p ∈ *carrier P* ⟹
pderiv (*Cring-Poly.compose R p* (*up-ring.monom* (*UP R*) *d 1*)) = *d* $\odot_{UP\ R}$ *Cring-Poly.compose R* (*pderiv p*) (*up-ring.monom* (*UP R*) *d 1*) ⟹
pderiv (*Cring-Poly.compose R q* (*up-ring.monom* (*UP R*) *d 1*)) = *d* $\odot_{UP\ R}$ *Cring-Poly.compose R* (*pderiv q*) (*up-ring.monom* (*UP R*) *d 1*) ⟹
pderiv (*Cring-Poly.compose R* (*p* $\oplus_P$ *q*) (*up-ring.monom* (*UP R*) *d 1*)) =
d $\odot_{UP\ R}$ *Cring-Poly.compose R* (*pderiv* (*p* $\oplus_P$ *q*)) (*up-ring.monom* (*UP R*) *d 1*)
proof− **fix** *p q* **assume** *A*: *q* ∈ *carrier P p* ∈ *carrier P*
pderiv (*Cring-Poly.compose R p* (*up-ring.monom* (*UP R*) *d 1*)) = *d* $\odot_{UP\ R}$ *Cring-Poly.compose R* (*pderiv p*) (*up-ring.monom* (*UP R*) *d 1*)
pderiv (*Cring-Poly.compose R q* (*up-ring.monom* (*UP R*) *d 1*)) = *d* $\odot_{UP\ R}$ *Cring-Poly.compose R* (*pderiv q*) (*up-ring.monom* (*UP R*) *d 1*)
show *pderiv* (*Cring-Poly.compose R* (*p* $\oplus_P$ *q*) (*up-ring.monom* (*UP R*) *d 1*)) =
d $\odot_{UP\ R}$ *Cring-Poly.compose R* (*pderiv* (*p* $\oplus_P$ *q*)) (*up-ring.monom* (*UP*

R) *d 1*)
using *A assms P-def monom-closed pderiv-add pderiv-closed smult-r-distr sub-add sub-closed* **by** *force*
qed
show ⋀*a n*. *a* ∈ *carrier R* ⟹
pderiv (*Cring-Poly.compose R* (*up-ring.monom P a n*) (*up-ring.monom* (*UP R*) *d 1*)) =
d ⊙$_{UP\ R}$ *Cring-Poly.compose R* (*pderiv* (*up-ring.monom P a n*)) (*up-ring.monom* (*UP R*) *d 1*)
proof− **fix** *a n* **assume** *A*: *a* ∈ *carrier R*
have (*Cring-Poly.compose R* (*up-ring.monom P a n*) (*up-ring.monom* (*UP R*) *d 1*)) = *a* ⊙$_{UP\ R}$ (*up-ring.monom P d 1*)[^]$_{UP\ R}$ *n*
using *A assms sub-monom*(*2*) *P-def is-UP-monomE*(*1*) *monom-is-UP-monom*(*1*) **by** *blast*
hence *0*: (*Cring-Poly.compose R* (*up-ring.monom P a n*) (*up-ring.monom* (*UP R*) *d 1*)) = *a* ⊙$_{UP\ R}$ (*up-ring.monom P* (*d*[^]*n*) *n*)
using *A assms P-def monom-pow nat-mult-1* **by** *metis*
show *pderiv* (*Cring-Poly.compose R* (*up-ring.monom P a n*) (*up-ring.monom* (*UP R*) *d 1*)) =
d ⊙$_{UP\ R}$ *Cring-Poly.compose R* (*pderiv* (*up-ring.monom P a n*)) (*up-ring.monom* (*UP R*) *d 1*)
proof(*cases n* = *0*)
case *True*
have *T0*: *pderiv* (*up-ring.monom P a n*) = **0**$_{UP\ R}$ **unfolding** *True*
using *A P-def pderiv-const* **by** *blast*
have *T1*: (*Cring-Poly.compose R* (*up-ring.monom P a n*) (*up-ring.monom* (*UP R*) *d 1*)) = *up-ring.monom P a n*
unfolding *True*
using *A assms P-def deg-const is-UP-monomE*(*1*) *monom-is-UP-monom*(*1*) *sub-const* **by** *presburger*
thus *?thesis* **unfolding** *T0 T1*
by (*metis P.nat-pow-eone P-def UP-smult-closed UP-zero-closed X-closed assms*(*2*) *deg-zero monom-rep-X-pow smult-r-null sub-const*)
next
case *False*
have *F0*: *pderiv* (*Cring-Poly.compose R* (*up-ring.monom P a n*) (*up-ring.monom* (*UP R*) *d 1*)) = (*a* ⊙$_{UP\ R}$ (*up-ring.monom P* ([*n*]·$_R$(*d*[^]*n*))(*n*−*1*)))
using *A assms pderiv-monom* **unfolding** *0*
using *P-def R.nat-pow-closed is-UP-monomE*(*1*) *monom-is-UP-monom*(*1*) *pderiv-smult* **by** *metis*
have *F1*: (*pderiv* (*up-ring.monom P a n*)) = *up-ring.monom P* ([*n*] · *a*) (*n* − *1*)
using *A assms pderiv-monom*[*of a n*] **by** *blast*
hence *F2*: (*pderiv* (*up-ring.monom P a n*)) = ([*n*] · *a*) ⊙$_{UP\ R}$*up-ring.monom P* **1** (*n* − *1*)
using *A P-def monic-monom-smult* **by** *auto*
have *F3*: *Cring-Poly.compose R* ((([*n*] · *a*) ⊙$_{UP\ R}$ (*up-ring.monom P* **1** (*n* − *1*)))) (*up-ring.monom* (*UP R*) *d 1*) =
([*n*] · *a*) ⊙$_{UP\ R}$ ((*up-ring.monom* (*UP R*) *d 1*)[^]$_{UP\ R}$(*n*−*1*))

using *A F1 F2 P-def assms(2) monom-closed sub-monom(2)* **by** *fastforce*
have *F4*: *Cring-Poly.compose R ((([n] · a) ⊙$_{UP\ R}$ (up-ring.monom P* **1** *(n − 1)))) (up-ring.monom (UP R) d 1) =*
([n] · a) ⊙$_{UP\ R}$ ((up-ring.monom (UP R) (d[^](n−1)) (n−1)))
by *(metis F3 P-def assms(2) monom-pow nat-mult-1)*
have *F5*: *d ⊙$_{UP\ R}$ (Cring-Poly.compose R (pderiv (up-ring.monom P a n)) (up-ring.monom (UP R) d 1)) =*
(d ⊗([n] · a)) ⊙$_{UP\ R}$ up-ring.monom (UP R) (d [^] (n − 1)) (n − 1)
unfolding *F4 F2*
using *A P-def assms(2) monom-closed smult-assoc1* **by** *auto*
have *F6*: *d ⊙$_{UP\ R}$ (Cring-Poly.compose R (pderiv (up-ring.monom P a n)) (up-ring.monom (UP R) d 1)) =*
(d ⊗ d[^](n−1) ⊗[n] · a) ⊙$_{UP\ R}$ ((up-ring.monom (UP R) **1** *(n−1)))*
unfolding *F5* **using** *False A assms P-def R.m-assoc R.m-closed R.m-comm R.nat-pow-closed monic-monom-smult monom-mult-smult*
by *(smt (verit) R.add.nat-pow-closed)*
have *F7*: *pderiv (Cring-Poly.compose R (up-ring.monom P a n) (up-ring.monom (UP R) d 1)) = (a ⊗ ([n]·$_R$(d[^]n)) ⊙$_{UP\ R}$ (up-ring.monom P* **1** *(n−1)))*
unfolding *F0* **using** *A assms P-def R.m-closed R.nat-pow-closed monic-monom-smult monom-mult-smult*
by *simp*
have *F8*: *a ⊗ [n] · (d [^] n) = d ⊗ d [^] (n − 1) ⊗ [n] · a*
proof−
have *F80*: *d ⊗ d [^] (n − 1) ⊗ [n] · a = d [^] n ⊗ [n] · a*
using *A assms False* **by** *(metis R.nat-pow-Suc2 add.right-neutral add-eq-if)*
show *?thesis* **unfolding** *F80*
using *A R.add-pow-rdistr R.m-comm R.nat-pow-closed assms(2)* **by** *presburger*
qed
show *?thesis* **unfolding** *F6 F7* **unfolding** *F8 P-def* **by** *blast*
qed
qed
qed

lemma(**in** *UP-cring*) *linear-sub-deriv′*:
assumes *f ∈ carrier (UP R)*
assumes *d ∈ carrier R*
assumes *g = compose R f (up-ring.monom (UP R) d 1)*
assumes *c ∈ carrier R*
shows *pderiv g = compose R (d ⊙$_{UP\ R}$ pderiv f) (up-ring.monom (UP R) d 1)*
using *assms linear-sub-deriv[of f d g c] P-def is-UP-monomE(1) is-UP-monomI pderiv-closed sub-smult* **by** *metis*

lemma(**in** *UP-cring*) *linear-sub-inv*:
assumes *f ∈ carrier (UP R)*
assumes *d ∈ Units R*
assumes *g = compose R f (up-ring.monom (UP R) d 1)*
shows *compose R g (up-ring.monom (UP R) (inv d) 1) = f*
unfolding *assms*

proof fix *x*
have *0*: *Cring-Poly.compose R* (*Cring-Poly.compose R f* (*up-ring.monom* (*UP R*) *d 1*)) (*up-ring.monom* (*UP R*) (*inv d*) *1*) *x* =
(*inv d*)[^]*x* ⊗ ((*Cring-Poly.compose R f* (*up-ring.monom* (*UP R*) *d 1*)) *x*)
apply(*rule linear-sub-cfs*)
using *P-def R.Units-closed assms*(*1*) *assms*(*2*) *monom-closed sub-closed* **apply** *auto*[*1*]
apply (*simp add*: *assms*(*2*))
by *blast*
show *Cring-Poly.compose R* (*Cring-Poly.compose R f* (*up-ring.monom* (*UP R*) *d 1*)) (*up-ring.monom* (*UP R*) (*inv d*) *1*) *x* = *f x*
unfolding *0* **using** *linear-sub-cfs*[*of f d Cring-Poly.compose R f* (*up-ring.monom* (*UP R*) *d 1*) *x*]
assms
by (*smt* (*verit*) *R.Units-closed R.Units-inv-closed R.Units-l-inv R.m-assoc R.m-comm R.nat-pow-closed R.nat-pow-distrib R.nat-pow-one R.r-one UP-car-memE*(*1*))
qed

lemma(**in** *UP-cring*) *linear-sub-deg*:
assumes *f* ∈ *carrier* (*UP R*)
assumes *d* ∈ *Units R*
assumes *g* = *compose R f* (*up-ring.monom* (*UP R*) *d 1*)
shows *deg R g* = *deg R f*
proof(*cases deg R f* = *0*)
case *True*
show *?thesis* **using** *assms*
unfolding *True assms* **using** *P-def True monom-closed*
by (*simp add*: *R.Units-closed sub-const*)
next
case *False*
have *0*: *monom* (*UP R*) *d 1* (*deg R* (*monom* (*UP R*) *d 1*)) = *d*
using *assms lcf-monom*(*2*) **by** *blast*
have *1*: *d*[^](*deg R f*) ∈ *Units R*
using *assms*(*2*)
by (*metis Group.comm-monoid.axioms*(*1*) *R.units-comm-group R.units-of-pow comm-group-def monoid.nat-pow-closed units-of-carrier*)
have *2*: *f* (*deg R f*) ≠ **0**
using *assms False P-def UP-cring.ltrm-rep-X-pow UP-cring-axioms deg-ltrm degree-monom* **by** *fastforce*
have *deg R g* = *deg R f* ∗ *deg R* (*up-ring.monom* (*UP R*) *d 1*)
unfolding *assms*
apply(*rule cring-sub-deg*[*of up-ring.monom* (*UP R*) *d 1 f*])
using *assms P-def monom-closed* **apply** *blast*
unfolding *P-def* **apply**(*rule assms*)
unfolding *0* **using** *2 1*
by (*metis R.Units-closed R.Units-l-cancel R.m-comm R.r-null R.zero-closed UP-car-memE*(*1*) *assms*(*1*))
thus *?thesis* **using** *assms* **unfolding** *assms*
by (*metis* (*no-types*, *lifting*) *P-def R.Units-closed deg-monom deg-zero is-UP-monomE*(*1*)

linear-sub-inv monom-is-UP-monom(*2*) *monom-zero mult.right-neutral mult-0-right sub-closed sub-const*)
qed

end

9 Lemmas About Polynomial Division

context *UP-cring*
begin

9.1 Division by Linear Terms

definition *UP-root-div* **where**
UP-root-div f a = (*poly-shift* (*T*$_a$ *f*)) *of* (*X-minus a*)

definition *UP-root-rem* **where**
UP-root-rem f a = *ctrm* (*T*$_a$ *f*)

lemma *UP-root-div-closed*:
assumes *f* ∈ *carrier P*
assumes *a* ∈ *carrier R*
shows *UP-root-div f a* ∈ *carrier P*
using *assms*
unfolding *UP-root-div-def*
by (*simp add*: *taylor-closed X-minus-closed poly-shift-closed sub-closed*)

lemma *rem-closed*:
assumes *f* ∈ *carrier P*
assumes *a* ∈ *carrier R*
shows *UP-root-rem f a* ∈ *carrier P*
using *assms*
unfolding *UP-root-rem-def*
by (*simp add*: *taylor-closed ctrm-is-poly*)

lemma *rem-deg*:
assumes *f* ∈ *carrier P*
assumes *a* ∈ *carrier R*
shows *degree* (*UP-root-rem f a*) = *0*
by (*simp add*: *taylor-closed assms*(*1*) *assms*(*2*) *ctrm-degree UP-root-rem-def*)

lemma *remainder-theorem*:
assumes *f* ∈ *carrier P*
assumes *a* ∈ *carrier R*
assumes *g* = *UP-root-div f a*
assumes *r* = *UP-root-rem f a*
shows *f* = *r* ⊕$_P$ (*X-minus a*) ⊗$_P$ *g*
proof−
have *T*$_a$*f* = (*ctrm* (*T*$_a$*f*)) ⊕$_P$ *X* ⊗$_P$ *poly-shift* (*T*$_a$*f*)

using *poly-shift-eq*[*of* $T_a f$] *assms taylor-closed*
by *blast*
hence *1*: $T_a f$ *of* (*X-minus a*) = (*ctrm* ($T_a f$)) $\oplus_P$ (*X-minus a*) $\otimes_P$ (*poly-shift* ($T_a f$) *of* (*X-minus a*))
using *assms taylor-closed*[*of f a*] *X-minus-closed*[*of a*] *X-closed*
sub-add[*of X-minus a ctrm* ($T_a f$) *X* $\otimes_P$ *poly-shift* ($T_a f$)]
sub-const[*of X-minus a*] *sub-mult*[*of X-minus a X poly-shift* ($T_a f$)]
ctrm-degree ctrm-is-poly P.m-closed poly-shift-closed sub-X
by *presburger*
have *2*: *f* = (*ctrm* ($T_a f$)) $\oplus_P$ (*X-minus a*) $\otimes_P$ (*poly-shift* ($T_a f$) *of* (*X-minus a*))
using *1 taylor-id*[*of a f*] *assms*
by *simp*
then show *?thesis*
using *assms*
unfolding *UP-root-div-def UP-root-rem-def*
by *auto*
qed

lemma *remainder-theorem'*:
assumes *f* ∈ *carrier P*
assumes *a* ∈ *carrier R*
shows *f* = *UP-root-rem f a* $\oplus_P$ (*X-minus a*) $\otimes_P$ *UP-root-div f a*
using *assms remainder-theorem* **by** *auto*

lemma *factor-theorem*:
assumes *f* ∈ *carrier P*
assumes *a* ∈ *carrier R*
assumes *g* = *UP-root-div f a*
assumes *to-fun f a* = **0**
shows *f* = (*X-minus a*) $\otimes_P$ *g*
using *remainder-theorem*[*of f a g* -] *assms*
unfolding *UP-root-rem-def*
by (*simp add*: *ctrm-zcf taylor-zcf taylor-closed UP-root-div-closed X-minus-closed*)

lemma *factor-theorem'*:
assumes *f* ∈ *carrier P*
assumes *a* ∈ *carrier R*
assumes *to-fun f a* = **0**
shows *f* = (*X-minus a*) $\otimes_P$ *UP-root-div f a*
by (*simp add*: *assms*(*1*) *assms*(*2*) *assms*(*3*) *factor-theorem*)

9.2 Geometric Sums

lemma *geom-quot*:
assumes *a* ∈ *carrier R*
assumes *b* ∈ *carrier R*
assumes *p* = *monom P* **1** (*Suc n*) $\ominus_P$ *monom P* (*b*[^](*Suc n*)) *0*
assumes *g* = *UP-root-div p b*

shows $a[\hat{}](Suc\ n) \ominus b[\hat{}]\ (Suc\ n) = (a \ominus b) \otimes (to\text{-}fun\ g\ a)$
proof−
have *root*: $to\text{-}fun\ p\ b = \mathbf{0}$
using *assms to-fun-const*[*of b*[^](*Suc n*) *b*] *to-fun-monic-monom*[*of b Suc n*] *R.nat-pow-closed*[*of b Suc n*]
to-fun-diff[*of monom P* **1** (*Suc n*) *monom P* (*b*[^](*Suc n*)) *0 b*] *monom-closed*

by (*metis P.minus-closed P-def R.one-closed R.zero-closed UP-cring.f-minus-ctrm*

UP-cring.to-fun-diff UP-cring-axioms zcf-to-fun cfs-monom to-fun-const)
have *LHS*: $to\text{-}fun\ p\ a = a[\hat{}](Suc\ n) \ominus b[\hat{}]\ (Suc\ n)$
using *assms to-fun-const to-fun-monic-monom to-fun-diff*
by *auto*
have *RHS*: $to\text{-}fun\ ((X\text{-}minus\ b) \otimes_P g)\ a = (a \ominus b) \otimes (to\text{-}fun\ g\ a)$
using *to-fun-mult*[*of g X-minus b*] *assms X-minus-closed*
by (*metis P.minus-closed P-def R.nat-pow-closed R.one-closed UP-cring.UP-root-div-closed UP-cring-axioms to-fun-X-minus monom-closed*)
show *?thesis*
using *RHS LHS root factor-theorem' assms*(*2*) *assms*(*3*) *assms*(*4*)
by *auto*
qed

end

context *UP-cring*
begin

definition *geometric-series* **where**
$geometric\text{-}series\ n\ a\ b = to\text{-}fun\ (UP\text{-}root\text{-}div\ (monom\ P\ \mathbf{1}\ (Suc\ n) \ominus_{UP\ R} (monom\ P\ (b[\hat{}](Suc\ n))\ 0))\ b)\ a$

lemma *geometric-series-id*:
assumes $a \in carrier\ R$
assumes $b \in carrier\ R$
shows $a[\hat{}](Suc\ n) \ominus b[\hat{}]\ (Suc\ n) = (a \ominus b) \otimes (geometric\text{-}series\ n\ a\ b)$
using *assms geom-quot*
by (*simp add*: *P-def geometric-series-def*)

lemma *geometric-series-closed*:
assumes $a \in carrier\ R$
assumes $b \in carrier\ R$
shows $(geometric\text{-}series\ n\ a\ b) \in carrier\ R$
unfolding *geometric-series-def*
using *assms P.minus-closed P-def UP-root-div-closed to-fun-closed monom-closed*

by *auto*

Shows that $a^n - b^n$ has $a - b$ as a factor:

lemma *to-fun-monic-monom-diff*:

assumes $a \in$ *carrier* R
assumes $b \in$ *carrier* R
shows $\exists\, c.\ c \in$ *carrier* $R \wedge$ *to-fun* (*monom* P **1** n) $a \ominus$ *to-fun* (*monom* P **1** n) $b = (a \ominus b) \otimes c$
proof(*cases* $n = 0$)
case *True*
have *to-fun* (*monom* P **1** *0*) $a \ominus$ *to-fun* (*monom* P **1** *0*) $b = (a \ominus b) \otimes$ **0**
unfolding *a-minus-def* **using** *to-fun-const*[*of* **1**] *assms*
by (*simp add*: *R.r-neg*)
then show *?thesis*
using *True* **by** *blast*
next
case *False*
then show *?thesis*
using *Suc-diff-1*[*of n*] *geometric-series-id*[*of a b n−1*] *geometric-series-closed*[*of a b n−1*]
assms(*1*) *assms*(*2*) *to-fun-monic-monom*
by *auto*
qed

lemma *to-fun-diff-factor*:
assumes $a \in$ *carrier* R
assumes $b \in$ *carrier* R
assumes $f \in$ *carrier* P
shows $\exists\, c.\ c \in$ *carrier* $R \wedge$(*to-fun* f a) $\ominus$ (*to-fun* f b) = $(a \ominus b)\otimes c$
proof(*rule poly-induct5*[*of f*])
show $f \in$ *carrier* P **using** *assms* **by** *simp*
show $\bigwedge p\ q.\ q \in$ *carrier* $P \Longrightarrow$
$p \in$ *carrier* $P \Longrightarrow$
$\exists\, c.\ c \in$ *carrier* $R \wedge$ *to-fun* p $a \ominus$ *to-fun* p $b = (a \ominus b) \otimes c \Longrightarrow$
$\exists\, c.\ c \in$ *carrier* $R \wedge$ *to-fun* q $a \ominus$ *to-fun* q $b = (a \ominus b) \otimes c \Longrightarrow$
$\exists\, c.\ c \in$ *carrier* $R \wedge$ *to-fun* $(p \oplus_P q)$ $a \ominus$ *to-fun* $(p \oplus_P q)$ $b = (a \ominus b) \otimes c$
proof− **fix** p q **assume** *A*: $q \in$ *carrier* P $p \in$ *carrier* P
$\exists\, c.\ c \in$ *carrier* $R \wedge$ *to-fun* p $a \ominus$ *to-fun* p $b = (a \ominus b) \otimes c$
$\exists\, c.\ c \in$ *carrier* $R \wedge$ *to-fun* q $a \ominus$ *to-fun* q $b = (a \ominus b) \otimes c$
obtain c **where** *c-def*: $c \in$ *carrier* $R \wedge$ *to-fun* p $a \ominus$ *to-fun* p $b = (a \ominus b) \otimes c$
using *A* **by** *blast*
obtain c' **where** *c'-def*: $c' \in$ *carrier* $R \wedge$ *to-fun* q $a \ominus$ *to-fun* q $b = (a \ominus b) \otimes c'$
using *A* **by** *blast*
have *0*: $(a \ominus b) \otimes c \oplus (a \ominus b) \otimes c' = (a \ominus b)\otimes(c \oplus c')$
using *assms c-def c'-def* **unfolding** *a-minus-def*
by (*simp add*: *R.r-distr R.r-minus*)
have *1*: *to-fun* $(p \oplus_P q)$ $a \ominus$ *to-fun* $(p \oplus_P q)$ b = *to-fun* p $a \oplus$ *to-fun* q $a \ominus$ *to-fun* p $b \ominus$ *to-fun* q b
using *A to-fun-plus*[*of p q a*] *to-fun-plus*[*of p q b*] *assms to-fun-closed*
R.ring-simprules(*19*)[*of to-fun p b to-fun q b*]
by (*simp add*: *R.add.m-assoc R.minus-eq to-fun-plus*)
hence *to-fun* $(p \oplus_P q)$ $a \ominus$ *to-fun* $(p \oplus_P q)$ b = *to-fun* p $a \ominus$ *to-fun* p $b \oplus$

to-fun q a ⊖ *to-fun q b*
using *0 A assms R.ring-simprules to-fun-closed a-assoc a-comm*
unfolding *a-minus-def* **by** (*smt* (*verit*, *del-insts*))
hence *to-fun* (*p* ⊕$_P$*q*) *a* ⊖ *to-fun* (*p* ⊕$_P$ *q*) *b* = *to-fun p a* ⊖ *to-fun p b* ⊕ (*to-fun q a* ⊖ *to-fun q b*)
using *0 A assms R.ring-simprules to-fun-closed*
unfolding *a-minus-def* **by** *metis*
hence *to-fun* (*p* ⊕$_P$*q*) *a* ⊖ *to-fun* (*p* ⊕$_P$ *q*) *b* = (*a* ⊖ *b*)⊗(*c* ⊕ *c*′)
using *0 A c-def c*′*-def*
by *simp*
thus ∃ *c*. *c* ∈ *carrier R* ∧ *to-fun* (*p* ⊕$_P$ *q*) *a* ⊖ *to-fun* (*p* ⊕$_P$ *q*) *b* = (*a* ⊖ *b*) ⊗ *c*
using *R.add.m-closed c*′*-def c-def* **by** *blast*
qed
show ⋀*n*. ∃ *c*. *c* ∈ *carrier R* ∧ *to-fun* (*monom P* **1** *n*) *a* ⊖ *to-fun* (*monom P* **1** *n*) *b* = (*a* ⊖ *b*) ⊗ *c*
by (*simp add*: *assms*(*1*) *assms*(*2*) *to-fun-monic-monom-diff*)
show ⋀*p aa*.
aa ∈ *carrier R* ⟹
p ∈ *carrier P* ⟹ ∃ *c*. *c* ∈ *carrier R* ∧ *to-fun p a* ⊖ *to-fun p b* = (*a* ⊖ *b*) ⊗ *c* ⟹ ∃ *c*. *c* ∈ *carrier R* ∧ *to-fun* (*aa* ⊙$_P$ *p*) *a* ⊖ *to-fun* (*aa* ⊙$_P$ *p*) *b* = (*a* ⊖ *b*) ⊗ *c*
proof− **fix** *p c* **assume** *A*: *c* ∈ *carrier R* *p* ∈ *carrier P*
∃ *e*. *e* ∈ *carrier R* ∧ *to-fun p a* ⊖ *to-fun p b* = (*a* ⊖ *b*) ⊗ *e*
then obtain *d* **where** *d-def*: *d* ∈ *carrier R* ∧ *to-fun p a* ⊖ *to-fun p b* = (*a* ⊖ *b*) ⊗ *d*
by *blast*
have *to-fun* (*c* ⊙$_P$ *p*) *a* ⊖ *to-fun* (*c* ⊙$_P$ *p*) *b* = *c* ⊗ (*to-fun p a* ⊖ *to-fun p b*)
using *A d-def assms to-fun-smult*[*of p a c*] *to-fun-smult*[*of p b c*]
to-fun-closed[*of p a*] *to-fun-closed*[*of p b*] *R.ring-simprules*
by *presburger*
hence *c*⊗*d* ∈ *carrier R* ∧ *to-fun* (*c* ⊙$_P$ *p*) *a* ⊖ *to-fun* (*c* ⊙$_P$ *p*) *b* = (*a* ⊖ *b*) ⊗ (*c* ⊗*d*)
by (*simp add*: *A*(*1*) *R.m-lcomm assms*(*1*) *assms*(*2*) *d-def*)
thus ∃ *e*. *e* ∈ *carrier R* ∧ *to-fun* (*c* ⊙$_P$ *p*) *a* ⊖ *to-fun* (*c* ⊙$_P$ *p*) *b* = (*a* ⊖ *b*) ⊗ *e*
by *blast*
qed
qed

Any finite set over a domain is the zero set of a polynomial:

lemma(**in** *UP-domain*) *fin-set-poly-roots*:
assumes *F* ⊆ *carrier R*
assumes *finite F*
shows ∃ *P* ∈ *carrier* (*UP R*). ∀ *x* ∈ *carrier R*. *to-fun P x* = **0** ⟷ *x* ∈ *F*
apply(*rule finite.induct*)
apply (*simp add*: *assms*(*2*))
proof−
show ∃ *P*∈*carrier* (*UP R*). ∀ *x*∈*carrier R*. (*to-fun P x* = **0**) = (*x* ∈ {})
proof−

have $\forall x{\in}carrier\ R.\ (to\text{-}fun\ (\mathbf{1}_{UP\ R})\ x = \mathbf{0}) = (x \in \{\})$
proof
fix x
assume A: $x \in carrier\ R$
then have $(to\text{-}fun\ (\mathbf{1}_{UP\ R}))\ x = \mathbf{1}$
by (*metis P-def R.one-closed UP-cring.to-fun-to-poly UP-cring-axioms ring-hom-one to-poly-is-ring-hom*)
then show $(to\text{-}fun\ \mathbf{1}_{UP\ R}\ x = \mathbf{0}) = (x \in \{\})$
by *simp*
qed
then show *?thesis*
using *P-def UP-one-closed*
by *blast*
qed
show $\bigwedge A\ a.\ finite\ A \Longrightarrow$
$\exists P{\in}carrier\ (UP\ R).\ \forall x{\in}carrier\ R.\ (to\text{-}fun\ P\ x = \mathbf{0}) = (x \in A) \Longrightarrow \exists P{\in}carrier\ (UP\ R).\ \forall x{\in}carrier\ R.\ (to\text{-}fun\ P\ x = \mathbf{0}) = (x \in insert\ a\ A)$
proof−
fix $A :: 'a\ set$ **fix** a
assume *fin-A*: *finite A*
assume *IH*: $\exists P{\in}carrier\ (UP\ R).\ \forall x{\in}carrier\ R.\ (to\text{-}fun\ P\ x = \mathbf{0}) = (x \in A)$
then obtain p **where** *p-def*: $p \in carrier\ (UP\ R) \wedge (\forall x{\in}carrier\ R.\ (to\text{-}fun\ p\ x = \mathbf{0}) = (x \in A))$
by *blast*
show $\exists P{\in}carrier\ (UP\ R).\ \forall x{\in}carrier\ R.\ (to\text{-}fun\ P\ x = \mathbf{0}) = (x \in insert\ a\ A)$
proof(*cases* $a \in carrier\ R$)
case *True*
obtain Q **where** *Q-def*: $Q = p \otimes_{UP\ R} (X \ominus_{UP\ R} to\text{-}poly\ a)$
by *blast*
have $\forall x{\in}carrier\ R.\ (to\text{-}fun\ Q\ x = \mathbf{0}) = (x \in insert\ a\ A)$
proof fix x
assume *P*: $x \in carrier\ R$
have *P0*: $to\text{-}fun\ (X \ominus_{UP\ R} to\text{-}poly\ a)\ x = x \ominus a$
using *to-fun-plus*[*of* $X \ominus_{UP\ R} to\text{-}poly\ a\ x$] *True P*
unfolding *a-minus-def*
by (*metis X-poly-minus-def a-minus-def to-fun-X-minus*)
then have $to\text{-}fun\ Q\ x = (to\text{-}fun\ p\ x) \otimes (x \ominus a)$
proof−
have *0*: $p \in carrier\ P$
by (*simp add*: *P-def p-def*)
have *1*: $X \ominus_{UP\ R} to\text{-}poly\ a \in carrier\ P$
using *P.minus-closed P-def True X-closed to-poly-closed* **by** *auto*
have *2*: $x \in carrier\ R$
by (*simp add*: *P*)
then show *?thesis*
using *to-fun-mult*[*of* $p\ (X \ominus_{UP\ R} to\text{-}poly\ a)\ x$] *P0 0 1 2 Q-def True P-def to-fun-mult*
by *auto*
qed

then show (*to-fun Q x* = **0**) = (*x* ∈ *insert a A*)
using *p-def*
by (*metis P R.add.inv-closed R.integral-iff R.l-neg R.minus-closed R.minus-unique True UP-cring.to-fun-closed UP-cring-axioms a-minus-def insert-iff*)
qed
then have *Q* ∈ *carrier* (*UP R*) ∧ (∀ *x*∈*carrier R*. (*to-fun Q x* = **0**) = (*x* ∈ *insert a A*))
using *P.minus-closed P-def Q-def True UP-mult-closed X-closed p-def to-poly-closed*
by *auto*
then show *?thesis*
by *blast*
next
case *False*
then show *?thesis*
using *IH subsetD* **by** *auto*
qed
qed
qed

9.3 Polynomial Evaluation at Multiplicative Inverses

For every polynomial $p(x)$ of degree n, there is a unique polynomial $q(x)$ which satisfies the equation $q(x) = x^n p(1/x)$. This section defines this polynomial and proves this identity.

definition(**in** *UP-cring*) *one-over-poly* **where**
one-over-poly p = (λ *n*. *if n* ≤ *degree p then p* ((*degree p*) − *n*) *else* **0**)

lemma(**in** *UP-cring*) *one-over-poly-closed*:
assumes *p* ∈ *carrier P*
shows *one-over-poly p* ∈ *carrier P*
apply(*rule UP-car-memI*[*of degree p*])
unfolding *one-over-poly-def* **using** *assms* **apply** *simp*
by (*simp add*: *assms cfs-closed*)

lemma(**in** *UP-cring*) *one-over-poly-monom*:
assumes *a* ∈ *carrier R*
shows *one-over-poly* (*monom P a n*) = *monom P a 0*
apply(*rule ext*)
unfolding *one-over-poly-def* **using** *assms*
by (*metis cfs-monom deg-monom diff-diff-cancel diff-is-0-eq diff-self-eq-0 zero-diff*)

lemma(**in** *UP-cring*) *one-over-poly-monom-add*:
assumes *a* ∈ *carrier R*
assumes *a* ≠ **0**
assumes *p* ∈ *carrier P*
assumes *degree p* < *n*
shows *one-over-poly* (*p* ⊕$_P$ *monom P a n*) = *monom P a 0* ⊕$_P$ *monom P* **1** (*n* − *degree p*) ⊗$_P$ *one-over-poly p*
proof−

have *0*: *degree* $(p \oplus_P$ *monom P a n*$) = n$
by (*simp add*: *assms(1) assms(2) assms(3) assms(4) equal-deg-sum*)
show *?thesis*
proof(*rule ext*) **fix** x **show** *one-over-poly* $(p \oplus_P$ *monom P a n*$)\ x =$
$(monom\ P\ a\ 0 \oplus_P monom\ P\ \mathbf{1}\ (n - deg\ R\ p) \otimes_P$ *one-over-poly p*$)\ x$
proof(*cases* $x = 0$)
case *T*: *True*
have *T0*: *one-over-poly* $(p \oplus_P$ *monom P a n*$)\ 0 = a$
unfolding *one-over-poly-def*
by (*metis lcf-eq lcf-monom(1) ltrm-of-sum-diff-deg P.add.m-closed assms(1) assms(2) assms(3) assms(4) diff-zero le0 monom-closed*)
have *T1*: $(monom\ P\ a\ 0 \oplus_P monom\ P\ \mathbf{1}\ (n - degree\ p) \otimes_P$ *one-over-poly p*$)\ 0 = a$
using *one-over-poly-closed*
by (*metis* (*no-types*, *lifting*) *lcf-monom(1) R.one-closed R.r-zero UP-m-comm UP-mult-closed assms(1) assms(3) assms(4) cfs-add cfs-monom-mult deg-const monom-closed zero-less-diff*)
show *?thesis* **using** *T0 T1 T* **by** *auto*
next
case *F*: *False*
show *?thesis*
proof(*cases* $x < n - degree\ p$)
case *True*
then have *T0*: *degree* $p < n - x \wedge n - x < n$
using *F* **by** *auto*
then have *T1*: *one-over-poly* $(p \oplus_P$ *monom P a n*$)\ x = \mathbf{0}$
using *True F 0* **unfolding** *one-over-poly-def*
using *assms(1) assms(3) coeff-of-sum-diff-degree0*
by (*metis ltrm-cfs ltrm-of-sum-diff-deg P.add.m-closed P.add.m-comm assms(2) assms(4) monom-closed nat-neq-iff*)
have $(monom\ P\ a\ 0 \oplus_P monom\ P\ \mathbf{1}\ (n - degree\ p) \otimes_P$ *one-over-poly p*$)\ x = \mathbf{0}$
using *True F 0 one-over-poly-def one-over-poly-closed*
by (*metis* (*no-types*, *lifting*) *P.add.m-comm P.m-closed R.one-closed UP-m-comm assms(1)*
assms(3) cfs-monom-mult coeff-of-sum-diff-degree0 deg-const monom-closed neq0-conv)
then show *?thesis* **using** *T1* **by** *auto*
next
case *False*
then have $n - degree\ p \leq x$
by *auto*
then obtain k **where** *k-def*: $k + (n - degree\ p) = x$
using *le-Suc-ex diff-add*
by *blast*
have *F0*: $(monom\ P\ a\ 0 \oplus_P monom\ P\ \mathbf{1}\ (n - deg\ R\ p) \otimes_P$ *one-over-poly p*$)\ x$
$= one\text{-}over\text{-}poly\ p\ k$

```
        using k-def one-over-poly-closed assms
              times-X-pow-coeff[of one-over-poly p n − deg R p k]
              P.m-closed
          by (metis (no-types, lifting) P.add.m-comm R.one-closed add-gr-0 co-
eff-of-sum-diff-degree0 deg-const monom-closed zero-less-diff)
      show ?thesis
      proof(cases x ≤ n)
        case True
        have T0: n − x = degree p − k
          using assms(4) k-def by linarith
        have T1: n − x < n
          using True F
          by linarith
        then have F1: (p ⊕_P monom P a n) (n − x) = p (degree p − k)
          using True False F0 0 k-def cfs-add
         by (simp add: F0 T0 assms(1) assms(3) cfs-closed cfs-monom)
        then show ?thesis
          using 0 F0 assms(1) assms(2) assms(3) degree-of-sum-diff-degree k-def
one-over-poly-def
          by auto
      next
        case False
        then show ?thesis
          using 0 F0 assms(1) assms(2) assms(3) degree-of-sum-diff-degree k-def
one-over-poly-def
          by auto
      qed
    qed
  qed
 qed
qed

lemma( in UP-cring) one-over-poly-eval:
  assumes p ∈ carrier P
  assumes x ∈ carrier R
  assumes x ∈ Units R
  shows to-fun (one-over-poly p) x = (x[^](degree p)) ⊗ (to-fun p (inv_R x))
proof(rule poly-induct6[of p])
  show  p ∈ carrier P
    using assms by simp
  show ⋀a n. a ∈ carrier R ⟹
        to-fun (one-over-poly (monom P a 0)) x = x [^] deg R (monom P a 0) ⊗
to-fun (monom P a 0) (inv x)
    using assms to-fun-const one-over-poly-monom by auto
  show ⋀a n p.
       a ∈ carrier R ⟹
       a ≠ 0 ⟹
       p ∈ carrier P ⟹
       deg R p < n ⟹
```

to-fun (*one-over-poly p*) *x* = *x* [^] *deg R p* ⊗ *to-fun p* (*inv x*) ⟹
to-fun (*one-over-poly* (*p* ⊕$_P$ *monom P a n*)) *x* = *x* [^] *deg R* (*p* ⊕$_P$ *monom P a n*) ⊗ *to-fun* (*p* ⊕$_P$ *monom P a n*) (*inv x*)
proof− **fix** *a n p* **assume** *A*: *a* ∈ *carrier R a* ≠ **0** *p* ∈ *carrier P deg R p* < *n*
to-fun (*one-over-poly p*) *x* = *x* [^] *deg R p* ⊗ *to-fun p* (*inv x*)
have *one-over-poly* (*p* ⊕$_P$ *monom P a n*) = *monom P a 0* ⊕$_P$ *monom P* **1** (*n* − *degree p*) ⊗$_P$ *one-over-poly p*
using *A* **by** (*simp add*: *one-over-poly-monom-add*)
hence *to-fun* (*one-over-poly* (*p* ⊕$_P$ *monom P a n*)) *x* =
a ⊕ *to-fun* (*monom P* **1** (*n* − *degree p*) ⊗$_P$ *one-over-poly p*) *x*
using *A to-fun-plus one-over-poly-closed cfs-add*
by (*simp add*: *assms*(*2*) *to-fun-const*)
hence *to-fun* (*one-over-poly* (*p* ⊕$_P$ *monom P a n*)) *x* = *a* ⊕ *x*[^](*n* − *degree p*) ⊗ *x* [^] *degree p* ⊗ *to-fun p* (*inv x*)
by (*simp add*: *A*(*3*) *A*(*5*) *R.m-assoc assms*(*2*) *assms*(*3*) *to-fun-closed to-fun-monic-monom to-fun-mult one-over-poly-closed*)
hence *0*:*to-fun* (*one-over-poly* (*p* ⊕$_P$ *monom P a n*)) *x* = *a* ⊕ *x*[^]*n* ⊗ *to-fun p* (*inv x*)
using *A R.nat-pow-mult assms*(*2*)
by *auto*
have *1*: *to-fun* (*one-over-poly* (*p* ⊕$_P$ *monom P a n*)) *x* = *x*[^]*n* ⊗ (*a* ⊗ *inv x* [^]*n* ⊕ *to-fun p* (*inv x*))
proof−
have *x*[^]*n* ⊗ *a* ⊗ *inv x* [^]*n* = *a*
by (*metis* (*no-types*, *opaque-lifting*) *A*(*1*) *R.Units-inv-closed R.Units-r-inv R.m-assoc*
R.m-comm R.nat-pow-closed R.nat-pow-distrib R.nat-pow-one R.r-one assms(*2*) *assms*(*3*))
thus *?thesis*
using *A R.ring-simprules*(*23*)[*of* - - *x*[^]*n*] *0 R.m-assoc assms*(*2*) *assms*(*3*) *to-fun-closed*
by *auto*
qed
have *2*: *degree* (*p* ⊕$_P$ *monom P a n*) = *n*
by (*simp add*: *A*(*1*) *A*(*2*) *A*(*3*) *A*(*4*) *equal-deg-sum*)
show *to-fun* (*one-over-poly* (*p* ⊕$_P$ *monom P a n*)) *x* = *x* [^] *deg R* (*p* ⊕$_P$ *monom P a n*) ⊗ *to-fun* (*p* ⊕$_P$ *monom P a n*) (*inv x*)
using *1 2*
by (*metis* (*no-types*, *lifting*) *A*(*1*) *A*(*3*) *P-def R.Units-inv-closed R.add.m-comm*

UP-cring.to-fun-monom UP-cring-axioms assms(*3*) *to-fun-closed to-fun-plus monom-closed*)
qed
qed

end

10 Lifting Homomorphisms of Rings to Polynomial Rings by Application to Coefficients

definition *poly-lift-hom* **where**
poly-lift-hom R S φ = eval R (UP S) (to-polynomial S ∘ φ) (X-poly S)

context *UP-ring*
begin

lemma(**in** *UP-cring*) *pre-poly-lift-hom-is-hom*:
assumes *cring S*
assumes *φ ∈ ring-hom R S*
shows *ring-hom-ring R (UP S) (to-polynomial S ∘ φ)*
apply(*rule ring-hom-ringI*)
apply (*simp add*: *R.ring-axioms*)
apply (*simp add*: *UP-ring.UP-ring UP-ring.intro assms*(*1*) *cring.axioms*(*1*))
using *UP-cring.intro UP-cring.to-poly-closed assms*(*1*) *assms*(*2*) *ring-hom-closed*
apply *fastforce*
using *assms UP-cring.to-poly-closed*[*of S*] *ring-hom-closed*[*of φ R S*] *comp-apply*[*of to-polynomial S φ*]
unfolding *UP-cring-def*
apply (*metis UP-cring.to-poly-mult UP-cring-def ring-hom-mult*)
using *assms UP-cring.to-poly-closed*[*of S*] *ring-hom-closed*[*of φ R S*] *comp-apply*[*of to-polynomial S φ*]
unfolding *UP-cring-def*
apply (*metis UP-cring.to-poly-add UP-cring-def ring-hom-add*)
using *assms UP-cring.to-poly-closed*[*of S*] *ring-hom-one*[*of φ R S*] *comp-apply*[*of to-polynomial S φ*]
unfolding *UP-cring-def*
by (*simp add*: ‹*φ ∈ ring-hom R S ⟹ φ* **1** = **1**$_S$›
UP-cring.intro UP-cring.to-poly-is-ring-hom ring-hom-one)

lemma(**in** *UP-cring*) *poly-lift-hom-is-hom*:
assumes *cring S*
assumes *φ ∈ ring-hom R S*
shows *poly-lift-hom R S φ ∈ ring-hom (UP R) (UP S)*
unfolding *poly-lift-hom-def*
apply(*rule UP-pre-univ-prop.eval-ring-hom*[*of R UP S*])
unfolding *UP-pre-univ-prop-def*
apply (*simp add*: *R-cring RingHom.ring-hom-cringI UP-cring.UP-cring UP-cring-def assms*(*1*) *assms*(*2*) *pre-poly-lift-hom-is-hom*)
by (*simp add*: *UP-cring.X-closed UP-cring.intro assms*(*1*))

lemma(**in** *UP-cring*) *poly-lift-hom-closed*:
assumes *cring S*
assumes *φ ∈ ring-hom R S*
assumes *p ∈ carrier (UP R)*
shows *poly-lift-hom R S φ p ∈ carrier (UP S)*
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *poly-lift-hom-is-hom ring-hom-closed*)

lemma(**in** *UP-cring*) *poly-lift-hom-add*:
assumes *cring S*
assumes $\varphi \in$ *ring-hom R S*
assumes $p \in$ *carrier* (*UP R*)
assumes $q \in$ *carrier* (*UP R*)
shows *poly-lift-hom R S* φ ($p \oplus_{UP\ R} q$) = *poly-lift-hom R S* φ *p* $\oplus_{UP\ S}$ *poly-lift-hom R S* φ *q*
using *assms poly-lift-hom-is-hom*[*of S* φ] *ring-hom-add*[*of poly-lift-hom R S* φ *UP R UP S p q*]
by *blast*

lemma(**in** *UP-cring*) *poly-lift-hom-mult*:
assumes *cring S*
assumes $\varphi \in$ *ring-hom R S*
assumes $p \in$ *carrier* (*UP R*)
assumes $q \in$ *carrier* (*UP R*)
shows *poly-lift-hom R S* φ ($p \otimes_{UP\ R} q$) = *poly-lift-hom R S* φ *p* $\otimes_{UP\ S}$ *poly-lift-hom R S* φ *q*
using *assms poly-lift-hom-is-hom*[*of S* φ] *ring-hom-mult*[*of poly-lift-hom R S* φ *UP R UP S p q*]
by *blast*

lemma(**in** *UP-cring*) *poly-lift-hom-extends-hom*:
assumes *cring S*
assumes $\varphi \in$ *ring-hom R S*
assumes $r \in$ *carrier R*
shows *poly-lift-hom R S* φ (*to-polynomial R r*) = *to-polynomial S* (φ *r*)
using *UP-pre-univ-prop.eval-const*[*of R UP S to-polynomial S* $\circ$ φ *X-poly S r*] *assms*
comp-apply[*of* λa. *monom* (*UP S*) *a 0* φ *r*] *pre-poly-lift-hom-is-hom*[*of S* φ]
unfolding *poly-lift-hom-def to-polynomial-def UP-pre-univ-prop-def*
by (*simp add*: *R-cring RingHom.ring-hom-cringI UP-cring.UP-cring UP-cring.X-closed UP-cring.intro*)

lemma(**in** *UP-cring*) *poly-lift-hom-extends-hom*$'$:
assumes *cring S*
assumes $\varphi \in$ *ring-hom R S*
assumes $r \in$ *carrier R*
shows *poly-lift-hom R S* φ (*monom P r 0*) = *monom* (*UP S*) (φ *r*) *0*
using *poly-lift-hom-extends-hom*[*of S* φ *r*] *assms*
unfolding *to-polynomial-def P-def*
by *blast*

lemma(**in** *UP-cring*) *poly-lift-hom-smult*:
assumes *cring S*
assumes $\varphi \in$ *ring-hom R S*
assumes $p \in$ *carrier* (*UP R*)

assumes $a \in$ *carrier* R
shows *poly-lift-hom* R S φ $(a \odot_{UP\ R} p) = \varphi\ a \odot_{UP\ S}$ (*poly-lift-hom* R S φ p)
using *assms poly-lift-hom-is-hom*[*of* S φ] *poly-lift-hom-extends-hom'*[*of* S φ a]
poly-lift-hom-mult[*of* S φ *monom* P a 0 p] *ring-hom-closed*[*of* φ R S a]
UP-ring.monom-mult-is-smult[*of* S φ a *poly-lift-hom* R S φ p]
monom-mult-is-smult[*of* a p] *monom-closed*[*of* a 0] *poly-lift-hom-closed*[*of* S φ p]
unfolding *to-polynomial-def UP-ring-def P-def cring-def*
by *simp*

lemma(**in** *UP-cring*) *poly-lift-hom-monom*:
assumes *cring* S
assumes $\varphi \in$ *ring-hom* R S
assumes $r \in$ *carrier* R
shows *poly-lift-hom* R S φ (*monom* (*UP* R) r n) = (*monom* (*UP* S) (φ r) n)
proof−
have *eval* R (*UP* S) (*to-polynomial* $S \circ \varphi$) (*X-poly* S) (*monom* (*UP* R) r n) = (*to-polynomial* $S \circ \varphi$) $r \otimes_{UP\ S}$ *X-poly* S $\lceil\rceil_{UP\ S}$ n
using *assms UP-pre-univ-prop.eval-monom*[*of* R *UP* S *to-polynomial* $S \circ \varphi$ r *X-poly* S n]
unfolding *UP-pre-univ-prop-def UP-cring-def ring-hom-cring-def*
by (*meson UP-cring.UP-cring UP-cring.X-closed UP-cring.pre-poly-lift-hom-is-hom UP-cring-axioms*
UP-cring-def ring-hom-cring-axioms.intro ring-hom-ring.homh)
then have *eval* R (*UP* S) (*to-polynomial* $S \circ \varphi$) (*X-poly* S) (*monom* (*UP* R) r n) = (*to-polynomial* S (φ r)) $\otimes_{UP\ S}$ *X-poly* S $\lceil\rceil_{UP\ S}$ n
by *simp*
then show *?thesis*
unfolding *poly-lift-hom-def*
using *assms UP-cring.monom-rep-X-pow*[*of* S φ r n] *ring-hom-closed*[*of* φ R S r]
by (*metis UP-cring.X-closed UP-cring.intro UP-cring.monom-sub UP-cring.sub-monom*(*1*))
qed

lemma(**in** *UP-cring*) *poly-lift-hom-X-var*:
assumes *cring* S
assumes $\varphi \in$ *ring-hom* R S
shows *poly-lift-hom* R S φ (*monom* (*UP* R) $\mathbf{1}_R$ 1) = (*monom* (*UP* S) $\mathbf{1}_S$ 1)
using *assms*(*1*) *assms*(*2*) *poly-lift-hom-monom ring-hom-one* **by** *fastforce*

lemma(**in** *UP-cring*) *poly-lift-hom-X-var'*:
assumes *cring* S
assumes $\varphi \in$ *ring-hom* R S
shows *poly-lift-hom* R S φ (*X-poly* R) = (*X-poly* S)
unfolding *X-poly-def*
using *assms*(*1*) *assms*(*2*) *poly-lift-hom-X-var* **by** *blast*

lemma(**in** *UP-cring*) *poly-lift-hom-X-var''*:
assumes *cring* S

assumes $\varphi \in$ *ring-hom R S*
shows *poly-lift-hom R S* φ (*monom* (*UP R*) $\mathbf{1}_R$ *n*) = (*monom* (*UP S*) $\mathbf{1}_S$ *n*)
using *assms*(*1*) *assms*(*2*) *poly-lift-hom-monom ring-hom-one* **by** *fastforce*

lemma(**in** *UP-cring*) *poly-lift-hom-X-var'''*:
assumes *cring S*
assumes $\varphi \in$ *ring-hom R S*
shows *poly-lift-hom R S* φ (*X-poly R* [^]$_{UP\ R}$ (*n*::*nat*)) = (*X-poly S*) [^]$_{UP\ S}$ (*n*::*nat*)
using *assms*
by (*smt* (*verit*) *ltrm-of-X P.nat-pow-closed P-def R.ring-axioms UP-cring.to-fun-closed UP-cring.intro*
UP-cring.monom-pow UP-cring.poly-lift-hom-monom UP-cring-axioms X-closed cfs-closed
cring.axioms(*1*) *to-fun-X-pow poly-lift-hom-X-var' ring-hom-closed ring-hom-nat-pow*)

lemma(**in** *UP-cring*) *poly-lift-hom-X-plus*:
assumes *cring S*
assumes $\varphi \in$ *ring-hom R S*
assumes *a* $\in$ *carrier R*
shows *poly-lift-hom R S* φ (*X-poly-plus R a*) = *X-poly-plus S* (φ *a*)
using *ring-hom-add*
unfolding *X-poly-plus-def*
using *P-def X-closed assms*(*1*) *assms*(*2*) *assms*(*3*) *poly-lift-hom-X-var' poly-lift-hom-add poly-lift-hom-extends-hom to-poly-closed* **by** *fastforce*

lemma(**in** *UP-cring*) *poly-lift-hom-X-plus-nat-pow*:
assumes *cring S*
assumes $\varphi \in$ *ring-hom R S*
assumes *a* $\in$ *carrier R*
shows *poly-lift-hom R S* φ (*X-poly-plus R a* [^]$_{UP\ R}$ (*n*::*nat*)) = *X-poly-plus S* (φ *a*) [^]$_{UP\ S}$ (*n*::*nat*)
using *assms poly-lift-hom-X-plus*[*of S* φ *a*]
ring-hom-nat-pow[*of UP R UP S poly-lift-hom R S* φ *X-poly-plus R a n*]
poly-lift-hom-is-hom[*of S* φ] *X-plus-closed*[*of a*] *UP-ring.UP-ring*[*of S*]
unfolding *P-def cring-def UP-cring-def*
using *P-def UP-ring UP-ring.intro*
by (*simp add*: *UP-ring.intro*)

lemma(**in** *UP-cring*) *X-poly-plus-nat-pow-closed*:
assumes *a* $\in$ *carrier R*
shows *X-poly-plus R a* [^]$_{UP\ R}$ (*n*::*nat*) $\in$ *carrier* (*UP R*)
using *assms P.nat-pow-closed P-def X-plus-closed* **by** *auto*

lemma(**in** *UP-cring*) *poly-lift-hom-X-plus-nat-pow-smult*:
assumes *cring S*
assumes $\varphi \in$ *ring-hom R S*
assumes *a* $\in$ *carrier R*

assumes $b \in carrier\ R$
shows *poly-lift-hom R S φ (b* $\odot_{UP\ R}$ *X-poly-plus R a* $[\widehat{}]_{UP\ R}$ *(n::nat)) = φ b* $\odot_{UP\ S}$ *X-poly-plus S (φ a)* $[\widehat{}]_{UP\ S}$ *(n::nat)*
by (*simp add*: *X-poly-plus-nat-pow-closed assms(1) assms(2) assms(3) assms(4) poly-lift-hom-X-plus-nat-pow poly-lift-hom-smult*)

lemma(**in** *UP-cring*) *poly-lift-hom-X-minus*:
assumes *cring S*
assumes $\varphi \in ring\text{-}hom\ R\ S$
assumes $a \in carrier\ R$
shows *poly-lift-hom R S φ (X-poly-minus R a) = X-poly-minus S (φ a)*
using *poly-lift-hom-X-plus*[*of S φ* $\ominus$ *a*] *X-minus-plus*[*of a*] *UP-cring.X-minus-plus*[*of S φ a*]
R.ring-hom-a-inv[*of S φ a*]
unfolding *UP-cring-def P-def*
by (*metis R.add.inv-closed assms(1) assms(2) assms(3) cring.axioms(1) ring-hom-closed*)

lemma(**in** *UP-cring*) *poly-lift-hom-X-minus-nat-pow*:
assumes *cring S*
assumes $\varphi \in ring\text{-}hom\ R\ S$
assumes $a \in carrier\ R$
shows *poly-lift-hom R S φ (X-poly-minus R a* $[\widehat{}]_{UP\ R}$ *(n::nat)) = X-poly-minus S (φ a)* $[\widehat{}]_{UP\ S}$ *(n::nat)*
using *assms poly-lift-hom-X-minus ring-hom-nat-pow X-minus-plus UP-cring.X-minus-plus*

poly-lift-hom-X-plus poly-lift-hom-X-plus-nat-pow **by** *fastforce*

lemma(**in** *UP-cring*) *X-poly-minus-nat-pow-closed*:
assumes $a \in carrier\ R$
shows *X-poly-minus R a* $[\widehat{}]_{UP\ R}$ *(n::nat)* $\in$ *carrier (UP R)*
using *assms monoid.nat-pow-closed*[*of UP R X-poly-minus R a n*]
P.nat-pow-closed P-def X-minus-closed **by** *auto*

lemma(**in** *UP-cring*) *poly-lift-hom-X-minus-nat-pow-smult*:
assumes *cring S*
assumes $\varphi \in ring\text{-}hom\ R\ S$
assumes $a \in carrier\ R$
assumes $b \in carrier\ R$
shows *poly-lift-hom R S φ (b* $\odot_{UP\ R}$ *X-poly-minus R a* $[\widehat{}]_{UP\ R}$ *(n::nat)) = φ b* $\odot_{UP\ S}$ *X-poly-minus S (φ a)* $[\widehat{}]_{UP\ S}$ *(n::nat)*
by (*simp add*: *X-poly-minus-nat-pow-closed assms(1) assms(2) assms(3) assms(4) poly-lift-hom-X-minus-nat-pow poly-lift-hom-smult*)

lemma(**in** *UP-cring*) *poly-lift-hom-cf*:
assumes *cring S*
assumes $\varphi \in ring\text{-}hom\ R\ S$
assumes $p \in carrier\ P$
shows *poly-lift-hom R S φ p k = φ (p k)*
apply(*rule poly-induct3*[*of p*])

apply (*simp add*: *assms*(*3*))
proof−
show ⋀*p q*. *q* ∈ *carrier P* ⟹
p ∈ *carrier P* ⟹
poly-lift-hom R S φ p k = *φ* (*p k*) ⟹ *poly-lift-hom R S φ q k* = *φ* (*q k*)
⟹ *poly-lift-hom R S φ* (*p* ⊕*P q*) *k* = *φ* ((*p* ⊕*P q*) *k*)
proof− **fix** *p q* **assume** *A*: *p* ∈ *carrier P q* ∈ *carrier P*
poly-lift-hom R S φ p k = *φ* (*p k*) *poly-lift-hom R S φ q k* = *φ* (*q k*)
show *poly-lift-hom R S φ q k* = *φ* (*q k*) ⟹ *poly-lift-hom R S φ* (*p* ⊕*P q*) *k* =
φ ((*p* ⊕*P q*) *k*)
using *A assms poly-lift-hom-add*[*of S φ p q*]
poly-lift-hom-closed[*of S φ p*] *poly-lift-hom-closed*[*of S φ q*]
UP-ring.cfs-closed[*of S poly-lift-hom R S φ q k*] *UP-ring.cfs-closed*[*of S*
poly-lift-hom R S φ p k]
UP-ring.cfs-add[*of S poly-lift-hom R S φ p poly-lift-hom R S φ q k*]
unfolding *P-def UP-ring-def*
by (*metis* (*full-types*) *P-def cfs-add cfs-closed cring.axioms*(*1*) *ring-hom-add*)
qed
show ⋀*a n*. *a* ∈ *carrier R* ⟹ *poly-lift-hom R S φ* (*monom P a n*) *k* = *φ*
(*monom P a n k*)
proof− **fix** *a m* **assume** *A*: *a* ∈ *carrier R*
show *poly-lift-hom R S φ* (*monom P a m*) *k* = *φ* (*monom P a m k*)
apply(*cases m* = *k*)
using *cfs-monom*[*of a m k*] *assms poly-lift-hom-monom*[*of S φ a m*] *UP-ring.cfs-monom*[*of*
S φ a m k]
unfolding *P-def UP-ring-def*
apply (*simp add*: *A cring.axioms*(*1*) *ring-hom-closed*)
using *cfs-monom*[*of a m k*] *assms poly-lift-hom-monom*[*of S φ a m*]
UP-ring.cfs-monom[*of S φ a m k*]
unfolding *P-def UP-ring-def*
by (*metis A P-def R.ring-axioms cring.axioms*(*1*) *ring-hom-closed ring-hom-zero*)

qed
qed

lemma(**in** *ring*) *ring-hom-monom-term*:
assumes *a* ∈ *carrier R*
assumes *c* ∈ *carrier R*
assumes *ring S*
assumes *h* ∈ *ring-hom R S*
shows *h* (*a* ⊗ *c*[^](*n*::*nat*)) = *h a* ⊗*S* (*h c*)[^]*Sn*
apply(*induction n*)
using *assms ringE*(*2*) *ring-hom-closed* **apply** *fastforce*
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *local.ring-axioms nat-pow-closed*
ring-hom-mult ring-hom-nat-pow)

lemma(**in** *UP-cring*) *poly-lift-hom-eval*:
assumes *cring S*
assumes *φ* ∈ *ring-hom R S*

assumes $p \in$ *carrier* P
assumes $a \in$ *carrier* R
shows *UP-cring.to-fun S* (*poly-lift-hom R S* φ p) (φ a) = φ (*to-fun* p a)
apply(*rule poly-induct3*[*of* p])
apply (*simp add*: *assms*(*3*))
proof−
show $\bigwedge p\ q.\ q \in$ *carrier* $P \Longrightarrow$
$p \in$ *carrier* $P \Longrightarrow$
UP-cring.to-fun S (*poly-lift-hom R S* φ p) (φ a) = φ (*to-fun* p a) $\Longrightarrow$
UP-cring.to-fun S (*poly-lift-hom R S* φ q) (φ a) = φ (*to-fun* q a) $\Longrightarrow$
UP-cring.to-fun S (*poly-lift-hom R S* φ ($p \oplus_P q$)) (φ a) = φ (*to-fun* ($p \oplus_P q$) a)
proof− **fix** p q **assume** *A*: $q \in$ *carrier* P $p \in$ *carrier* P
UP-cring.to-fun S (*poly-lift-hom R S* φ p) (φ a) = φ (*to-fun* p a)
UP-cring.to-fun S (*poly-lift-hom R S* φ q) (φ a) = φ (*to-fun* q a)
have (*poly-lift-hom R S* φ ($p \oplus_P q$)) = *poly-lift-hom R S* φ $p \oplus_{UP\ S}$ *poly-lift-hom R S* φ q
using *A*(*1*) *A*(*2*) *P-def assms*(*1*) *assms*(*2*) *poly-lift-hom-add* **by** *auto*
hence *UP-cring.to-fun S* (*poly-lift-hom R S* φ ($p \oplus_P q$)) (φ a) =
UP-cring.to-fun S (*poly-lift-hom R S* φ p) (φ a) $\oplus_S$ *UP-cring.to-fun S* (*poly-lift-hom R S* φ q) (φ a)
using *UP-cring.to-fun-plus*[*of S*] *assms*
unfolding *UP-cring-def*
by (*metis* (*no-types*, *lifting*) *A*(*1*) *A*(*2*) *P-def poly-lift-hom-closed ring-hom-closed*)
thus *UP-cring.to-fun S* (*poly-lift-hom R S* φ ($p \oplus_P q$)) (φ a) = φ (*to-fun* ($p \oplus_P q$) a)
using *A to-fun-plus assms ring-hom-add*[*of* φ *R S*]
poly-lift-hom-closed[*of S* φ] *UP-cring.to-fun-def*[*of S*] *to-fun-def*
unfolding *P-def UP-cring-def*
using *UP-cring.to-fun-closed UP-cring-axioms*
by *metis*
qed
show $\bigwedge c\ n.\ c \in$ *carrier* $R \Longrightarrow$ *UP-cring.to-fun S* (*poly-lift-hom R S* φ (*monom P c n*)) (φ a) = φ (*to-fun* (*monom P c n*) a)
unfolding *P-def*
proof − **fix** c n **assume** *A*: $c \in$ *carrier* R
have *0*: φ (a [^]$_R$ (n::*nat*)) = φ a [^]$_S$ n
using *assms ring-hom-nat-pow*[*of R S* φ a n]
unfolding *cring-def*
using *R.ring-axioms* **by** *blast*
have *1*: φ ($c \otimes_R a$ [^]$_R$ n) = φ $c \otimes_S \varphi$ a [^]$_S$ n
using *ring-hom-mult*[*of* φ *R S c a* [^]$_R$ *n*] *0 assms A monoid.nat-pow-closed* [*of R a n*]
by (*simp add*: *cring.axioms*(*1*) *ringE*(*2*))
show *UP-cring.to-fun S* (*poly-lift-hom R S* φ (*monom* (*UP R*) c n)) (φ a) = φ (*to-fun*(*monom* (*UP R*) c n) a)
using *assms A poly-lift-hom-monom*[*of S* φ c n] *UP-cring.to-fun-monom*[*of S* φ c φ a n]
to-fun-monom[*of c a n*] *0 1 ring-hom-closed*[*of* φ *R S*] **unfolding**

UP-cring-def
by (*simp add*: *P-def to-fun-def*)
qed
qed

lemma(**in** *UP-cring*) *poly-lift-hom-sub*:
assumes *cring S*
assumes $\varphi \in$ *ring-hom R S*
assumes $p \in$ *carrier P*
assumes $q \in$ *carrier P*
shows *poly-lift-hom R S* φ (*compose R p q*) = *compose S* (*poly-lift-hom R S* φ *p*) (*poly-lift-hom R S* φ *q*)
apply(*rule poly-induct3*[*of p*])
apply (*simp add*: *assms*(*3*))
proof−
show $\bigwedge p\ qa.$
qa ∈ *carrier P* ⟹
p ∈ *carrier P* ⟹
poly-lift-hom R S φ (*Cring-Poly.compose R p q*) = *Cring-Poly.compose S* (*poly-lift-hom R S* φ *p*) (*poly-lift-hom R S* φ *q*) ⟹
poly-lift-hom R S φ (*Cring-Poly.compose R qa q*) = *Cring-Poly.compose S* (*poly-lift-hom R S* φ *qa*) (*poly-lift-hom R S* φ *q*) ⟹
poly-lift-hom R S φ (*Cring-Poly.compose R* (*p* ⊕$_P$ *qa*) *q*) = *Cring-Poly.compose S* (*poly-lift-hom R S* φ (*p* ⊕$_P$ *qa*)) (*poly-lift-hom R S* φ *q*)
proof− **fix** *a b* **assume** *A*: *a* ∈ *carrier P*
b ∈ *carrier P*
poly-lift-hom R S φ (*Cring-Poly.compose R a q*) = *Cring-Poly.compose S* (*poly-lift-hom R S* φ *a*) (*poly-lift-hom R S* φ *q*)
poly-lift-hom R S φ (*Cring-Poly.compose R b q*) = *Cring-Poly.compose S* (*poly-lift-hom R S* φ *b*) (*poly-lift-hom R S* φ *q*)
show *poly-lift-hom R S* φ (*Cring-Poly.compose R* (*a* ⊕$_P$ *b*) *q*) = *Cring-Poly.compose S* (*poly-lift-hom R S* φ (*a* ⊕$_P$ *b*)) (*poly-lift-hom R S* φ *q*)
using *assms UP-cring.sub-add*[*of R q a b*] *UP-cring.sub-add*[*of S*]
unfolding *UP-cring-def*
by (*metis A*(*1*) *A*(*2*) *A*(*3*) *A*(*4*) *P-def R-cring UP-cring.sub-closed UP-cring-axioms poly-lift-hom-add poly-lift-hom-closed*)
qed
show $\bigwedge a\ n.$ *a* ∈ *carrier R* ⟹
poly-lift-hom R S φ (*Cring-Poly.compose R* (*monom P a n*) *q*) =
Cring-Poly.compose S (*poly-lift-hom R S* φ (*monom P a n*)) (*poly-lift-hom R S* φ *q*)
proof−
fix *a n* **assume** *A*: *a* ∈ *carrier R*
have *0*: (*poly-lift-hom R S* φ (*monom* (*UP R*) *a n*)) = *monom* (*UP S*) (φ *a*) *n*
by (*simp add*: *A assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *poly-lift-hom-monom*)
have *1*: *q* [^]$_{UP\ R}$ *n* ∈ *carrier* (*UP R*)
using *monoid.nat-pow-closed*[*of UP R q n*] *UP-ring.UP-ring UP-ring.intro assms*(*1*) *assms*
P.monoid-axioms P-def **by** *blast*

have *2*: *poly-lift-hom R S φ* (*to-polynomial R a* $\otimes_{UP\ R}$ *q* [^]$_{UP\ R}$ *n*) =
to-polynomial S (*φ a*) $\otimes_{UP\ S}$ (*poly-lift-hom R S φ q*) [^]$_{UP\ S}$ *n*
using *poly-lift-hom-mult*[*of S φ to-polynomial R a q* [^]$_{UP\ R}$ *n*] *poly-lift-hom-is-hom*[*of S φ*]
ring-hom-nat-pow[*of P UP S poly-lift-hom R S φ q n*] *UP-cring.UP-cring*[*of S*]
UP-cring poly-lift-hom-monom[*of S φ a 0*] *ring-hom-closed*[*of φ R S a*]
monom-closed[*of a 0*] *nat-pow-closed*[*of q n*] *assms A*
unfolding *to-polynomial-def P-def UP-cring-def cring-def*
by *auto*
have *3*: *poly-lift-hom R S φ* (*Cring-Poly.compose R* (*monom* (*UP R*) *a n*) *q*) = *to-polynomial S* (*φ a*) $\otimes_{UP\ S}$ (*poly-lift-hom R S φ q*) [^]$_{UP\ S}$ *n*
using *2 A P-def assms*(*4*) *sub-monom*(*1*) **by** *auto*
have *4*: *Cring-Poly.compose S* (*poly-lift-hom R S φ* (*monom* (*UP R*) *a n*)) (*poly-lift-hom R S φ q*)
= *Cring-Poly.compose S* (*monom* (*UP S*) (*φ a*) *n*) (*poly-lift-hom R S φ q*)
by (*simp add*: *0*)
have *poly-lift-hom R S φ q* ∈ *carrier* (*UP S*)
using *P-def UP-cring.poly-lift-hom-closed UP-cring-axioms assms*(*1*) *assms*(*2*) *assms*(*4*) **by** *blast*
then have *5*: *Cring-Poly.compose S* (*poly-lift-hom R S φ* (*monom* (*UP R*) *a n*)) (*poly-lift-hom R S φ q*)
= *to-polynomial S* (*φ a*) $\otimes_{UP\ S}$ (*poly-lift-hom R S φ q*) [^]$_{UP\ S}$ *n*
using *4 UP-cring.sub-monom*[*of S poly-lift-hom R S φ q φ a n*] *assms*
unfolding *UP-cring-def*
by (*simp add*: *A ring-hom-closed*)
thus *poly-lift-hom R S φ* (*Cring-Poly.compose R* (*monom P a n*) *q*) =
Cring-Poly.compose S (*poly-lift-hom R S φ* (*monom P a n*)) (*poly-lift-hom R S φ q*)
using *0 1 2 3 4 assms A*
by (*simp add*: *P-def*)
qed
qed

lemma(**in** *UP-cring*) *poly-lift-hom-comm-taylor-expansion*:
assumes *cring S*
assumes *φ* ∈ *ring-hom R S*
assumes *p* ∈ *carrier P*
assumes *a* ∈ *carrier R*
shows *poly-lift-hom R S φ* (*taylor-expansion R a p*) = *taylor-expansion S* (*φ a*) (*poly-lift-hom R S φ p*)
unfolding *taylor-expansion-def*
using *poly-lift-hom-sub*[*of S φ p* (*X-poly-plus R a*)] *poly-lift-hom-X-plus*[*of S φ a*] *assms*
by (*simp add*: *P-def UP-cring.X-plus-closed UP-cring-axioms*)

lemma(**in** *UP-cring*) *poly-lift-hom-comm-taylor-expansion-cf*:

assumes *cring S*
assumes $\varphi \in$ *ring-hom R S*
assumes $p \in$ *carrier* (*UP R*)
assumes $a \in$ *carrier R*
shows φ (*taylor-expansion R a p i*) = *taylor-expansion S* (φ *a*) (*poly-lift-hom R S* φ *p*) *i*
using *poly-lift-hom-cf assms poly-lift-hom-comm-taylor-expansion P-def taylor-def UP-cring.taylor-closed UP-cring-axioms* **by** *fastforce*

lemma(**in** *UP-cring*) *taylor-expansion-cf-closed*:
assumes $p \in$ *carrier P*
assumes $a \in$ *carrier R*
shows *taylor-expansion R a p i* $\in$ *carrier R*
using *assms taylor-closed*
by (*simp add*: *taylor-def cfs-closed*)

lemma(**in** *UP-cring*) *poly-lift-hom-comm-taylor-term*:
assumes *cring S*
assumes $\varphi \in$ *ring-hom R S*
assumes $p \in$ *carrier* (*UP R*)
assumes $a \in$ *carrier R*
shows *poly-lift-hom R S* φ (*taylor-term a p i*) = *UP-cring.taylor-term S* (φ *a*) (*poly-lift-hom R S* φ *p*) *i*
using *poly-lift-hom-X-minus-nat-pow-smult*[*of S* φ *a taylor-expansion R a p i i*]
poly-lift-hom-comm-taylor-expansion[*of S* φ *p a*]
poly-lift-hom-comm-taylor-expansion-cf[*of S* φ *p a i*]
assms UP-cring.taylor-term-def[*of S*]
unfolding *taylor-term-def UP-cring-def P-def*
by (*simp add*: *UP-cring.taylor-expansion-cf-closed UP-cring-axioms*)

lemma(**in** *UP-cring*) *poly-lift-hom-degree-bound*:
assumes *cring S*
assumes $h \in$ *ring-hom R S*
assumes $f \in$ *carrier* (*UP R*)
shows *deg S* (*poly-lift-hom R S h f*) $\leq$ *deg R f*
using *poly-lift-hom-closed*[*of S h f*] *UP-cring.deg-leqI*[*of S poly-lift-hom R S h f deg R f*] *assms ring-hom-zero*[*of h R S*] *deg-aboveD*[*of f*] *coeff-simp*[*of f*]
unfolding *P-def UP-cring-def*
by (*simp add*: *P-def R.ring-axioms cring.axioms*(*1*) *poly-lift-hom-cf*)

lemma(**in** *UP-cring*) *deg-eqI*:
assumes $f \in$ *carrier* (*UP R*)
assumes *deg R f* $\leq n$
assumes *f n* $\neq \mathbf{0}$
shows *deg R f* = *n*
using *assms coeff-simp*[*of f*] *P-def deg-leE le-neq-implies-less* **by** *blast*

lemma(**in** *UP-cring*) *poly-lift-hom-degree-eq*:
assumes *cring S*

assumes $h \in$ *ring-hom R S*
assumes h (*lcf f*) $\neq \mathbf{0}_S$
assumes $f \in$ *carrier* (*UP R*)
shows *deg S* (*poly-lift-hom R S h f*) = *deg R f*
apply(*rule UP-cring.deg-eqI*)
using *assms* **unfolding** *UP-cring-def* **apply** *blast*
using *poly-lift-hom-closed*[*of S h f*] *assms* **apply** *blast*
using *poly-lift-hom-degree-bound*[*of S h f*] *assms* **apply** *blast*
using *assms poly-lift-hom-cf*[*of S h f*]
by (*metis P-def*)

lemma(**in** *UP-cring*) *poly-lift-hom-lcoeff*:
assumes *cring S*
assumes $h \in$ *ring-hom R S*
assumes h (*lcf f*) $\neq \mathbf{0}_S$
assumes $f \in$ *carrier* (*UP R*)
shows *UP-ring.lcf S* (*poly-lift-hom R S h f*) = h (*lcf f*)
using *poly-lift-hom-degree-eq*[*of S h f*] *assms*
by (*simp add*: *P-def poly-lift-hom-cf*)

end

11 Coefficient List Constructor for Polynomials

definition *list-to-poly* **where**
list-to-poly R as n = (*if n* < *length as then as*!*n else* $\mathbf{0}_R$)

context *UP-ring*
begin

lemma(**in** *UP-ring*) *list-to-poly-closed*:
assumes *set as* $\subseteq$ *carrier R*
shows *list-to-poly R as* $\in$ *carrier P*
apply(*rule UP-car-memI*[*of length as*])
apply (*simp add*: *list-to-poly-def*)
by (*metis R.zero-closed assms in-mono list-to-poly-def nth-mem*)

lemma(**in** *UP-ring*) *list-to-poly-zero*[*simp*]:
list-to-poly R [] = $\mathbf{0}_{UP\ R}$
unfolding *list-to-poly-def*
apply *auto*
by(*simp add*: *UP-def*)

lemma(**in** *UP-domain*) *list-to-poly-singleton*:
assumes *a* $\in$ *carrier R*
shows *list-to-poly R* [*a*] = *monom P a 0*
apply(*rule ext*)
unfolding *list-to-poly-def* **using** *assms*
by (*simp add*: *cfs-monom*)

end

definition *cf-list* **where**
cf-list R p = *map p* [(*0*::*nat*)..< *Suc* (*deg R p*)]

lemma *cf-list-length*:
length (*cf-list R p*) = *Suc* (*deg R p*)
unfolding *cf-list-def*
by *simp*

lemma *cf-list-entries*:
assumes $i \leq$ *deg R p*
shows (*cf-list R p*)!*i* = *p i*
unfolding *cf-list-def*
by (*metis add.left-neutral assms diff-zero less-Suc-eq-le map-eq-map-tailrec nth-map-upt*)

lemma(**in** *UP-ring*) *list-to-poly-cf-list-inv*:
assumes *p* ∈ *carrier* (*UP R*)
shows *list-to-poly R* (*cf-list R p*) = *p*
proof
fix *x*
show *list-to-poly R* (*cf-list R p*) *x* = *p x*
apply(*cases x* < *degree p*)
unfolding *list-to-poly-def*
using *assms cf-list-length*[*of R p*] *cf-list-entries*[*of - R p*]
apply *simp*
by (*metis P-def UP-ring.coeff-simp UP-ring-axioms* ‹⋀*i*. *i* ≤ *deg R p* ⟹ *cf-list R p* ! *i* = *p i*› ‹*length* (*cf-list R p*) = *Suc* (*deg R p*)› *assms deg-belowI less-Suc-eq-le*)
qed

12 Polynomial Rings over a Subring

12.1 Characterizing the Carrier of a Polynomial Ring over a Subring

lemma(**in** *ring*) *carrier-update*:
carrier (*R*(|*carrier* := *S*|)) = *S*
$\mathbf{0}_{(R(|carrier := S|))} = \mathbf{0}$
$\mathbf{1}_{(R(|carrier := S|))} = \mathbf{1}$
$(\oplus_{(R(|carrier := S|))}) = (\oplus)$
$(\otimes_{(R(|carrier := S|))}) = (\otimes)$
by *auto*

lemma(**in** *UP-cring*) *poly-cfs-subring*:
assumes *subring S R*
assumes *g* ∈ *carrier* (*UP R*)
assumes ⋀*n*. *g n* ∈ *S*

shows $g \in carrier\ (UP\ (R\ (\!|\ carrier := S\ |\!)))$
apply(*rule UP-cring.UP-car-memI'*)
using *R.subcringI' R.subcring-iff UP-cring.intro assms(1) subringE(1)* **apply** *blast*
proof−
have $carrier\ (R(\!|carrier := S|\!)) = S$
using *ring.carrier-update* **by** *simp*
then show *0*: $\bigwedge x.\ g\ x \in carrier\ (R(\!|carrier := S|\!))$
using *assms* **by** *blast*
have *0*: $\mathbf{0}_{R(\!|carrier := S|\!)} = \mathbf{0}$
using *R.carrier-update(2)* **by** *blast*
then show $\bigwedge x.\ (deg\ R\ g) < x \Longrightarrow g\ x = \mathbf{0}_{R(\!|carrier := S|\!)}$
using *UP-car-memE assms(2)* **by** *presburger*
qed

lemma(**in** *UP-cring*) *UP-ring-subring*:
assumes *subring S R*
shows *UP-cring* $(R\ (\!|\ carrier := S\ |\!))$ *UP-ring* $(R\ (\!|\ carrier := S\ |\!))$
using *assms* **unfolding** *UP-cring-def*
using *R.subcringI' R.subcring-iff subringE(1)* **apply** *blast*
using *assms* **unfolding** *UP-ring-def*
using *R.subcringI' R.subcring-iff subringE(1)*
by (*simp add*: *R.subring-is-ring*)

lemma(**in** *UP-cring*) *UP-ring-subring-is-ring*:
assumes *subring S R*
shows *cring* $(UP\ (R\ (\!|\ carrier := S\ |\!)))$
using *assms UP-ring-subring*[*of S*] *UP-cring.UP-cring*[*of* $R(\!|carrier := S|\!)$]
by *blast*

lemma(**in** *UP-cring*) *UP-ring-subring-add-closed*:
assumes *subring S R*
assumes $g \in carrier\ (UP\ (R\ (\!|\ carrier := S\ |\!)))$
assumes $f \in carrier\ (UP\ (R\ (\!|\ carrier := S\ |\!)))$
shows $f \oplus_{UP\ (R\ (\!|\ carrier := S\ |\!))} g \in carrier\ (UP\ (R\ (\!|\ carrier := S\ |\!)))$
using *assms UP-ring-subring-is-ring*[*of S*]
by (*meson cring.cring-simprules(1)*)

lemma(**in** *UP-cring*) *UP-ring-subring-mult-closed*:
assumes *subring S R*
assumes $g \in carrier\ (UP\ (R\ (\!|\ carrier := S\ |\!)))$
assumes $f \in carrier\ (UP\ (R\ (\!|\ carrier := S\ |\!)))$
shows $f \otimes_{UP\ (R\ (\!|\ carrier := S\ |\!))} g \in carrier\ (UP\ (R\ (\!|\ carrier := S\ |\!)))$
using *assms UP-ring-subring-is-ring*[*of S*]
by (*meson cring.carrier-is-subcring subcringE(6)*)

lemma(**in** *UP-cring*) *UP-ring-subring-car*:
assumes *subring S R*
shows $carrier\ (UP\ (R\ (\!|\ carrier := S\ |\!))) = \{h \in carrier\ (UP\ R).\ \forall n.\ h\ n \in S\}$

proof
show *carrier* (*UP* (*R*(|*carrier* := *S*|))) ⊆ {*h* ∈ *carrier* (*UP R*). ∀ *n*. *h n* ∈ *S*}
proof
fix *h* **assume** *A*: *h* ∈ *carrier* (*UP* (*R*(|*carrier* := *S*|)))
have *h* ∈ *carrier P*
apply(*rule UP-car-memI*[*of deg* (*R*(|*carrier* := *S*|)) *h*]) **unfolding** *P-def*
using *UP-cring.UP-car-memE*[*of R*(|*carrier* := *S*|) *h*] *R.carrier-update*[*of S*]
assms UP-ring-subring A **apply** *presburger*
using *UP-cring.UP-car-memE*[*of R*(|*carrier* := *S*|) *h*] *assms*
by (*metis A R.ring-axioms UP-cring-def* ‹*carrier* (*R*(|*carrier* := *S*|)) = *S*› *cring.subcringI' is-UP-cring ring.subcring-iff subringE*(*1*) *subsetD*)
then show *h* ∈ {*h* ∈ *carrier* (*UP R*). ∀ *n*. *h n* ∈ *S*}
unfolding *P-def*
using *assms A UP-cring.UP-car-memE*[*of R*(|*carrier* := *S*|) *h*] *R.carrier-update*[*of S*]
UP-ring-subring **by** *blast*
qed
show {*h* ∈ *carrier* (*UP R*). ∀ *n*. *h n* ∈ *S*} ⊆ *carrier* (*UP* (*R*(|*carrier* := *S*|)))
proof fix *h* **assume** *A*: *h* ∈ {*h* ∈ *carrier* (*UP R*). ∀ *n*. *h n* ∈ *S*}
have *0*: *h* ∈ *carrier* (*UP R*)
using *A* **by** *blast*
have *1*: ⋀*n*. *h n* ∈ *S*
using *A* **by** *blast*
show *h* ∈ *carrier* (*UP* (*R*(|*carrier* := *S*|)))
apply(*rule UP-ring.UP-car-memI*[*of - deg R h*])
using *assms UP-ring-subring*[*of S*] *UP-cring.axioms UP-ring.intro cring.axioms*(*1*)
apply *blast*
using *UP-car-memE*[*of h*] *carrier-update 0 R.carrier-update*(*2*) **apply** *presburger*
using *assms 1 R.carrier-update*(*1*) **by** *blast*
qed
qed

lemma(**in** *UP-cring*) *UP-ring-subring-car-subset*:
assumes *subring S R*
shows *carrier* (*UP* (*R* (| *carrier* := *S* |))) ⊆ *carrier* (*UP R*)
proof fix *h* **assume** *h* ∈ *carrier* (*UP* (*R* (| *carrier* := *S* |)))
then show *h* ∈ *carrier* (*UP R*)
using *assms UP-ring-subring-car*[*of S*] **by** *blast*
qed

lemma(**in** *UP-cring*) *UP-ring-subring-car-subset'*:
assumes *subring S R*
assumes *h* ∈ *carrier* (*UP* (*R* (| *carrier* := *S* |)))
shows *h* ∈ *carrier* (*UP R*)
using *assms UP-ring-subring-car-subset*[*of S*] **by** *blast*

lemma(**in** *UP-cring*) *UP-ring-subring-add*:
assumes *subring S R*

assumes $g \in carrier\ (UP\ (R\ (\!|\ carrier := S\ |\!)))$
assumes $f \in carrier\ (UP\ (R\ (\!|\ carrier := S\ |\!)))$
shows $g \oplus_{UP\ R} f = g \oplus_{UP\ (R\ (\!|\ carrier := S\ |\!))} f$
proof(*rule ext*) **fix** x **show** $(g \oplus_{UP\ R} f)\ x = (g \oplus_{UP\ (R(\!|carrier := S|\!))} f)\ x$
proof−
have *0*: $(g \oplus_P f)\ x = g\ x \oplus f\ x$
using *assms cfs-add*[*of g f x*] **unfolding** *P-def*
using *UP-ring-subring-car-subset'* **by** *blast*
have *1*: $(g \oplus_{UP\ (R(\!|carrier := S|\!))} f)\ x = g\ x \oplus_{R(\!|carrier := S|\!)} f\ x$
using *UP-ring.cfs-add*[*of R (| carrier := S |) g f x*] *UP-ring-subring*[*of S*] *assms*
unfolding *UP-ring-def UP-cring-def*
using *R.subring-is-ring* **by** *blast*
show *?thesis* **using** *0 1 R.carrier-update*(*4*)[*of S*]
by (*simp add*: *P-def*)
qed
qed

lemma(**in** *UP-cring*) *UP-ring-subring-deg*:
assumes *subring S R*
assumes $g \in carrier\ (UP\ (R\ (\!|\ carrier := S\ |\!)))$
shows $deg\ R\ g = deg\ (R\ (\!|\ carrier := S\ |\!))\ g$
proof−
have *0*: $g \in carrier\ (UP\ R)$
using *assms UP-ring-subring-car*[*of S*] **by** *blast*
have *1*: $deg\ R\ g \leq deg\ (R\ (\!|\ carrier := S\ |\!))\ g$
using *0 assms UP-cring.UP-car-memE*[*of R (| carrier := S |) g*] *UP-car-memE*[*of g*] *P-def R.carrier-update*(*2*) *UP-ring-subring deg-leqI*
by *presburger*
have *2*: $deg\ (R\ (\!|\ carrier := S\ |\!))\ g \leq deg\ R\ g$
using *0 assms UP-cring.UP-car-memE*[*of R (| carrier := S |) g*] *UP-car-memE*[*of g*] *P-def R.carrier-update*(*2*) *UP-ring-subring UP-cring.deg-leqI*

by *metis*

show *?thesis* **using** *1 2* **by** *presburger*
qed

lemma(**in** *UP-cring*) *UP-subring-monom*:
assumes *subring S R*
assumes $a \in S$
shows $up\text{-}ring.monom\ (UP\ R)\ a\ n = up\text{-}ring.monom\ (UP\ (R\ (\!|\ carrier := S\ |\!)))\ a\ n$
proof fix x
have *0*: $a \in carrier\ R$
using *assms subringE*(*1*) **by** *blast*
have *1*: $a \in carrier\ (R(\!|carrier := S|\!))$
using *assms* **by** (*simp add*: *assms*(*2*))

have *2*: *up-ring.monom* (*UP* (*R*(|*carrier* := *S*|))) *a n x* = (*if n* = *x then a else* $\mathbf{0}_{R(|carrier := S|)}$)
using *1 assms UP-ring-subring*[*of S*] *UP-ring.cfs-monom*[*of R*(|*carrier* := *S*|) *a n x*] *UP-cring.axioms UP-ring.intro cring.axioms*(*1*)
by *blast*
show *up-ring.monom* (*UP R*) *a n x* = *up-ring.monom* (*UP* (*R*(|*carrier* := *S*|))) *a n x*
using *0 1 2 cfs-monom*[*of a n x*] *R.carrier-update*(*2*)[*of S*] **unfolding** *P-def*
by *presburger*
qed

lemma(**in** *UP-cring*) *UP-ring-subring-mult*:
assumes *subring S R*
assumes *g* ∈ *carrier* (*UP* (*R* (| *carrier* := *S* |)))
assumes *f* ∈ *carrier* (*UP* (*R* (| *carrier* := *S* |)))
shows *g* ⊗$_{UP\ R}$ *f* = *g* ⊗$_{UP}$ (*R* (| *carrier* := *S* |)) *f*
proof(*rule UP-ring.poly-induct3*[*of R* (| *carrier* := *S* |) *f*])
show *UP-ring* (*R*(|*carrier* := *S*|))
by (*simp add*: *UP-ring-subring*(*2*) *assms*(*1*))
show *f* ∈ *carrier* (*UP* (*R*(|*carrier* := *S*|)))
by (*simp add*: *assms*(*3*))
show ⋀*p q*. *q* ∈ *carrier* (*UP* (*R*(|*carrier* := *S*|))) ⟹
p ∈ *carrier* (*UP* (*R*(|*carrier* := *S*|))) ⟹
g ⊗$_{UP\ R}$ *p* = *g* ⊗$_{UP}$ (*R*(|*carrier* := *S*|)) *p* ⟹
g ⊗$_{UP\ R}$ *q* = *g* ⊗$_{UP}$ (*R*(|*carrier* := *S*|)) *q* ⟹ *g* ⊗$_{UP\ R}$ (*p* ⊕$_{UP}$ (*R*(|*carrier* := *S*|)) *q*) = *g* ⊗$_{UP}$ (*R*(|*carrier* := *S*|)) (*p* ⊕$_{UP}$ (*R*(|*carrier* := *S*|)) *q*)
proof− **fix** *p q*
assume *A*: *q* ∈ *carrier* (*UP* (*R*(|*carrier* := *S*|)))
p ∈ *carrier* (*UP* (*R*(|*carrier* := *S*|)))
g ⊗$_{UP\ R}$ *p* = *g* ⊗$_{UP}$ (*R*(|*carrier* := *S*|)) *p*
g ⊗$_{UP\ R}$ *q* = *g* ⊗$_{UP}$ (*R*(|*carrier* := *S*|)) *q*
have *0*: *p* ⊕$_{UP}$ (*R*(|*carrier* := *S*|)) *q* = *p* ⊕$_{UP\ R}$ *q*
using *A UP-ring-subring-add*[*of S p q*]
by (*simp add*: *assms*(*1*))
have *1*: *g* ⊗$_{UP\ R}$ (*p* ⊕$_{UP\ R}$ *q*) = *g* ⊗$_{UP\ R}$ *p* ⊕$_{UP\ R}$ *g* ⊗$_{UP\ R}$ *q*
using *0 A assms P.r-distr P-def UP-ring-subring-car-subset'* **by** *auto*
hence *2*:*g* ⊗$_{UP\ R}$ (*p* ⊕$_{UP}$ (*R*(|*carrier* := *S*|)) *q*) = *g* ⊗$_{UP\ R}$ *p* ⊕$_{UP\ R}$ *g* ⊗$_{UP\ R}$ *q*
using *0* **by** *simp*
have *3*: *g* ⊗$_{UP}$ (*R*(|*carrier* := *S*|)) (*p* ⊕$_{UP}$ (*R*(|*carrier* := *S*|)) *q*) =
g ⊗$_{UP}$ (*R*(|*carrier* := *S*|)) *p* ⊕$_{UP}$ (*R*(|*carrier* := *S*|)) *g* ⊗$_{UP}$ (*R*(|*carrier* := *S*|)) *q*
using *0 A assms semiring.r-distr*[*of UP* (*R*(|*carrier* := *S*|))] *UP-ring-subring-car-subset'*
using *UP-ring.UP-r-distr* ‹*UP-ring* (*R*(|*carrier* := *S*|))› **by** *blast*
hence *4*: *g* ⊗$_{UP}$ (*R*(|*carrier* := *S*|)) (*p* ⊕$_{UP}$ (*R*(|*carrier* := *S*|)) *q*) =
g ⊗$_{UP\ R}$ *p* ⊕$_{UP}$ (*R*(|*carrier* := *S*|)) *g* ⊗$_{UP\ R}$ *q*
using *A* **by** *simp*

hence *5*: $g \otimes_{UP\ (R\langle\langle carrier := S\rangle\rangle)} (p \oplus_{UP\ (R\langle\langle carrier := S\rangle\rangle)} q) =$
$g \otimes_{UP\ R} p \oplus_{UP\ R} g \otimes_{UP\ R} q$
using *UP-ring-subring-add*[*of S*]
by (*simp add*: *A*(*1*) *A*(*2*) *A*(*3*) *A*(*4*) *UP-ring.UP-mult-closed* ‹*UP-ring* ($R\langle\langle carrier := S\rangle\rangle$)› *assms*(*1*) *assms*(*2*))
show $g \otimes_{UP\ R} (p \oplus_{UP\ (R\langle\langle carrier := S\rangle\rangle)} q) = g \otimes_{UP\ (R\langle\langle carrier := S\rangle\rangle)} (p \oplus_{UP\ (R\langle\langle carrier := S\rangle\rangle)} q)$
by (*simp add*: *2 5*)
qed
show $\bigwedge a\ n.\ a \in carrier\ (R\langle\langle carrier := S\rangle\rangle) \Longrightarrow g \otimes_{UP\ R} monom\ (UP\ (R\langle\langle carrier := S\rangle\rangle))\ a\ n = g \otimes_{UP\ (R\langle\langle carrier := S\rangle\rangle)} monom\ (UP\ (R\langle\langle carrier := S\rangle\rangle))\ a\ n$
proof fix *a n x* **assume** *A*: $a \in carrier\ (R\langle\langle carrier := S\rangle\rangle)$
have *0*: $monom\ (UP\ (R\langle\langle carrier := S\rangle\rangle))\ a\ n = monom\ (UP\ R)\ a\ n$
using *A UP-subring-monom assms*(*1*) **by** *auto*
have *1*: $g \in carrier\ (UP\ R)$
using *assms UP-ring-subring-car-subset'* **by** *blast*
have *2*: $a \in carrier\ R$
using *A assms subringE*(*1*)[*of S R*] *R.carrier-update*[*of S*] **by** *blast*
show $(g \otimes_{UP\ R} monom\ (UP\ (R\langle\langle carrier := S\rangle\rangle))\ a\ n)\ x = (g \otimes_{UP\ (R\langle\langle carrier := S\rangle\rangle)} monom\ (UP\ (R\langle\langle carrier := S\rangle\rangle))\ a\ n)\ x$
proof(*cases* $x < n$)
case *True*
have *T0*: $(g \otimes_{UP\ R} monom\ (UP\ R)\ a\ n)\ x = \mathbf{0}$
using *1 2 True cfs-monom-mult*[*of g a x n*] *A assms* **unfolding** *P-def* **by** *blast*
then show *?thesis* **using** *UP-cring.cfs-monom-mult*[*of* $R\langle\langle carrier := S\rangle\rangle$ *g a x n*] *0 A True*
UP-ring-subring(*1*) *assms*(*1*) *assms*(*2*) **by** *auto*
next
case *False*
have *F0*: $(g \otimes_{UP\ R} monom\ (UP\ R)\ a\ n)\ x = a \otimes (g\ (x - n))$
using *1 2 False cfs-monom-mult-l*[*of g a n x − n*] *A assms* **unfolding** *P-def* **by** *simp*
have *F1*: $(g \otimes_{UP\ (R\langle\langle carrier := S\rangle\rangle)} monom\ (UP\ (R\langle\langle carrier := S\rangle\rangle))\ a\ n)\ (x - n + n) = a \otimes_{R\langle\langle carrier := S\rangle\rangle} g\ (x - n)$
using *1 2 False UP-cring.cfs-monom-mult-l*[*of* $R\langle\langle carrier := S\rangle\rangle$ *g a n x − n*] *A assms*
UP-ring-subring(*1*) **by** *blast*
hence *F2*: $(g \otimes_{UP\ (R\langle\langle carrier := S\rangle\rangle)} monom\ (UP\ R)\ a\ n)\ (x - n + n) = a \otimes g\ (x - n)$
using *UP-subring-monom*[*of S a n*] *R.carrier-update*[*of S*] *assms 0* **by** *metis*
show *?thesis* **using** *F0 F1 1 2 assms*
by (*simp add*: *0 False add.commute add-diff-inverse-nat*)
qed
qed
qed

lemma(**in** *UP-cring*) *UP-ring-subring-one*:
assumes *subring S R*
shows $\mathbf{1}_{UP\ R} = \mathbf{1}_{UP\ (R\ (\!|\ carrier := S\ |\!))}$
using *UP-subring-monom*[*of S* **1** *0*] *assms P-def R.subcringI' UP-ring.monom-one UP-ring-subring*(*2*) *monom-one subcringE*(*3*) **by** *force*

lemma(**in** *UP-cring*) *UP-ring-subring-zero*:
assumes *subring S R*
shows $\mathbf{0}_{UP\ R} = \mathbf{0}_{UP\ (R\ (\!|\ carrier := S\ |\!))}$
using *UP-subring-monom*[*of S* **0** *0*] *UP-ring.monom-zero*[*of R* (|| *carrier* := *S* ||) *0*] *assms monom-zero*[*of 0*]
UP-ring-subring[*of S*] *subringE*(*2*)[*of S R*]
unfolding *P-def*
by (*simp add*: *P-def R.carrier-update*(*2*))

lemma(**in** *UP-cring*) *UP-ring-subring-nat-pow*:
assumes *subring S R*
assumes $g \in carrier\ (UP\ (R\ (\!|\ carrier := S\ |\!)))$
shows $g[\hat{}]_{UP\ R} n = g[\hat{}]_{UP\ (R\ (\!|\ carrier := S\ |\!))}(n::nat)$
apply(*induction n*)
using *assms* **apply** (*simp add*: *UP-ring-subring-one*)
proof−
fix *n*::*nat*
assume *A*: $g\ [\hat{}]_{UP\ R}\ n = g\ [\hat{}]_{UP\ (R(\!|carrier := S|\!))}\ n$
have *Group.monoid* (*UP* (*R*(||*carrier* := *S*||)))
using *assms UP-ring-subring*[*of S*] *UP-ring.UP-ring*[*of R*(||*carrier* := *S*||)] *ring.is-monoid* **by** *blast*
hence *0* : $g\ [\hat{}]_{UP\ (R(\!|carrier := S|\!))}\ n \in carrier\ (UP\ (R(\!|carrier := S|\!)))$
using *monoid.nat-pow-closed*[*of UP* (*R* (|| *carrier* := *S* ||) *g n*] *assms UP-ring-subring*
unfolding *UP-ring-def ring-def* **by** *blast*
have *1*: $g\ [\hat{}]_{UP\ R}\ n \in carrier\ (UP\ R)$
using *0 assms UP-ring-subring-car-subset'*[*of S*] **by** (*simp add*: *A*)
then have *2*: $g\ [\hat{}]_{UP\ R}\ n \otimes_{UP\ R} g = g\ [\hat{}]_{UP\ (R(\!|carrier := S|\!))}\ n \otimes_{UP\ (R(\!|carrier := S|\!))} g$
using *assms UP-ring-subring-mult*[*of S g* $[\hat{}]_{UP\ R}$ *n g*]
by (*simp add*: *0 A*)
then show $g\ [\hat{}]_{UP\ R}\ Suc\ n = g\ [\hat{}]_{UP\ (R(\!|carrier := S|\!))}\ Suc\ n$
by *simp*
qed

lemma(**in** *UP-cring*) *UP-subring-compose-monom*:
assumes *subring S R*
assumes $g \in carrier\ (UP\ (R\ (\!|\ carrier := S\ |\!)))$
assumes $a \in S$
shows *compose R* (*up-ring.monom* (*UP R*) *a n*) *g* = *compose* (*R* (|| *carrier* := *S* ||) (*up-ring.monom* (*UP* (*R* (|| *carrier* := *S* ||))) *a n*) *g*
proof−
have *g-closed*: $g \in carrier\ (UP\ R)$

```
    using assms UP-ring-subring-car by blast
  have 0: a ∈ carrier R
    using assms subringE(1) by blast
  have 1: compose R (up-ring.monom (UP R) a n) g = a ⊙_UP R (g[^]_UP R^n)
    using monom-sub[of a g n] unfolding P-def
    using 0 assms(2) g-closed by blast
  have 2: compose (R⦇carrier := S⦈) (up-ring.monom (UP (R⦇carrier := S⦈)) a
n) g = a ⊙_UP (R⦇carrier := S⦈) g [^]_UP (R⦇carrier := S⦈) n
   using assms UP-cring.monom-sub[of R ⦇ carrier := S ⦈ a g n] UP-ring-subring[of
S] R.carrier-update[of S]
    by blast
  have 3:  g [^]_UP (R⦇carrier := S⦈) n = g[^]_UP R^n
    using UP-ring-subring-nat-pow[of S g n]
    by (simp add: assms(1) assms(2))
 have 4: a ⊙_UP R (g[^]_UP R^n) = a ⊙_UP (R⦇carrier := S⦈) g [^]_UP (R⦇carrier := S⦈)
n
  proof fix x
   show (a ⊙_UP R g [^]_UP R n) x = (a ⊙_UP (R⦇carrier := S⦈) g [^]_UP (R⦇carrier := S⦈)
n) x
    proof−
      have LHS: (a ⊙_UP R g [^]_UP R n) x = a ⊗ ((g [^]_UP R n) x)
        using 0 P.nat-pow-closed P-def cfs-smult g-closed by auto
       have RHS: (a ⊙_UP (R⦇carrier := S⦈) g [^]_UP (R⦇carrier := S⦈) n) x = a
⊗_R⦇carrier := S⦈ ((g [^]_UP (R⦇carrier := S⦈) n) x)
      proof−
        have Group.monoid (UP (R⦇carrier := S⦈))
          using assms UP-ring-subring[of S] UP-ring.UP-ring[of R⦇carrier := S⦈]
ring.is-monoid by blast
        hence 0 :  g [^]_UP (R⦇carrier := S⦈) n ∈ carrier (UP (R⦇carrier := S⦈))
            using monoid.nat-pow-closed[of UP (R ⦇ carrier := S ⦈) g n] assms
UP-ring-subring
          unfolding UP-ring-def ring-def by blast
        have 1: g [^]_UP (R⦇carrier := S⦈) n ∈ carrier (UP (R⦇carrier := S⦈))
          using assms UP-ring-subring[of S] R.carrier-update[of S] 0 by blast
        then show ?thesis using UP-ring.cfs-smult UP-ring-subring assms
          by (simp add: UP-ring.cfs-smult)
      qed
      show ?thesis using R.carrier-update RHS LHS 3 assms
        by simp
    qed
  qed
  show ?thesis using 0 1 2 3 4
    by simp
qed

lemma(in UP-cring) UP-subring-compose:
  assumes subring S R
  assumes g ∈ carrier (UP R)
```

assumes $f \in carrier\ (UP\ R)$
assumes $\bigwedge n.\ g\ n \in S$
assumes $\bigwedge n.\ f\ n \in S$
shows $compose\ R\ f\ g = compose\ (R\ (\!|\ carrier := S\ |\!))\ f\ g$
proof−
have *g-closed*: $g \in carrier\ (UP\ (R\ (\!|\ carrier := S\ |\!)))$
using *assms poly-cfs-subring* **by** *blast*
have *0*: $\bigwedge n.\ (\forall\ h.\ h \in carrier\ (UP\ R) \wedge deg\ R\ h \leq n \wedge h \in carrier\ (UP\ (R\ (\!|\ carrier := S\ |\!))) \longrightarrow compose\ R\ h\ g = compose\ (R\ (\!|\ carrier := S\ |\!))\ h\ g)$
proof− **fix** *n* **show** $(\forall\ h.\ h \in carrier\ (UP\ R) \wedge deg\ R\ h \leq n \wedge h \in carrier\ (UP\ (R\ (\!|\ carrier := S\ |\!))) \longrightarrow compose\ R\ h\ g = compose\ (R\ (\!|\ carrier := S\ |\!))\ h\ g)$
proof(*induction n*)
show $\forall h.\ h \in carrier\ (UP\ R) \wedge deg\ R\ h \leq 0 \wedge h \in carrier\ (UP\ (R(\!|carrier := S|\!))) \longrightarrow Cring\text{-}Poly.compose\ R\ h\ g = Cring\text{-}Poly.compose\ (R(\!|carrier := S|\!))\ h\ g$
proof fix *h*
show $h \in carrier\ (UP\ R) \wedge deg\ R\ h \leq 0 \wedge h \in carrier\ (UP\ (R(\!|carrier := S|\!))) \longrightarrow Cring\text{-}Poly.compose\ R\ h\ g = Cring\text{-}Poly.compose\ (R(\!|carrier := S|\!))\ h\ g$
proof
assume *A*: $h \in carrier\ (UP\ R) \wedge deg\ R\ h \leq 0 \wedge h \in carrier\ (UP\ (R(\!|carrier := S|\!)))$
then have *0*: $deg\ R\ h = 0$
by *linarith*
then have *1*: $deg\ (R\ (\!|\ carrier := S\ |\!))\ h = 0$
using *A assms UP-ring-subring-deg*[*of S h*]
by *linarith*
show $Cring\text{-}Poly.compose\ R\ h\ g = Cring\text{-}Poly.compose\ (R(\!|carrier := S|\!))\ h\ g$
using *0 1 g-closed assms sub-const*[*of g h*] *UP-cring.sub-const*[*of R*(|*carrier* := *S*|) *g h*] *A P-def UP-ring-subring*
by *presburger*
qed
qed
show $\bigwedge n.\ \forall h.\ h \in carrier\ (UP\ R) \wedge deg\ R\ h \leq n \wedge h \in carrier\ (UP\ (R(\!|carrier := S|\!))) \longrightarrow Cring\text{-}Poly.compose\ R\ h\ g = Cring\text{-}Poly.compose\ (R(\!|carrier := S|\!))\ h\ g \Longrightarrow \forall h.\ h \in carrier\ (UP\ R) \wedge deg\ R\ h \leq Suc\ n \wedge h \in carrier\ (UP\ (R(\!|carrier := S|\!))) \longrightarrow Cring\text{-}Poly.compose\ R\ h\ g = Cring\text{-}Poly.compose\ (R(\!|carrier := S|\!))\ h\ g$
proof fix *n h*
assume *IH*: $\forall h.\ h \in carrier\ (UP\ R) \wedge deg\ R\ h \leq n \wedge h \in carrier\ (UP\ (R(\!|carrier := S|\!))) \longrightarrow Cring\text{-}Poly.compose\ R\ h\ g = Cring\text{-}Poly.compose\ (R(\!|carrier := S|\!))\ h\ g$
show $h \in carrier\ (UP\ R) \wedge deg\ R\ h \leq Suc\ n \wedge h \in carrier\ (UP\ (R(\!|carrier := S|\!))) \longrightarrow Cring\text{-}Poly.compose\ R\ h\ g = Cring\text{-}Poly.compose\ (R(\!|carrier := S|\!))\ h\ g$

proof assume *A*: $h \in carrier\ (UP\ R) \wedge deg\ R\ h \leq Suc\ n \wedge h \in carrier\ (UP\ (R(\!|carrier := S|\!)))$
show *Cring-Poly.compose R h g = Cring-Poly.compose* $(R(\!|carrier := S|\!))$ *h g*
proof(*cases deg R h* $\leq$ *n*)
case *True*
then show *?thesis* **using** *A IH* **by** *blast*
next
case *False*
then have *F0*: *deg R h = Suc n*
using *A* **by** (*simp add*: *A le-Suc-eq*)
then have *F1*: *deg* $(R(\!|carrier := S|\!))$ *h = Suc n*
using *UP-ring-subring-deg*[*of S h*] *A*
by (*simp add*: ‹$h \in carrier\ (UP\ R) \wedge deg\ R\ h \leq Suc\ n \wedge h \in carrier\ (UP\ (R(\!|carrier := S|\!)))$› *assms*(*1*))
obtain *j* **where** *j-def*: $j \in carrier\ (UP\ (R(\!|carrier := S|\!))) \wedge h = j \oplus_{UP\ (R(\!|carrier := S|\!))} up\text{-}ring.monom\ (UP\ (R(\!|carrier := S|\!)))\ (h\ (deg\ (R(\!|carrier := S|\!))\ h))\ (deg\ (R(\!|carrier := S|\!))\ h) \wedge deg\ (R(\!|carrier := S|\!))\ j < deg\ (R(\!|carrier := S|\!))\ h$
using *A UP-ring.ltrm-decomp*[*of* $R(\!|carrier := S|\!)$ *h*] *assms UP-ring-subring*[*of S*]
F1 **by** (*metis* (*mono-tags*, *lifting*) *F0 False zero-less-Suc*)
have *j-closed*: $j \in carrier\ (UP\ R)$
using *j-def assms UP-ring-subring-car-subset* **by** *blast*
have *F2*: *deg R j < deg R h*
using *j-def assms*
by (*metis* (*no-types*, *lifting*) *F0 F1 UP-ring-subring-deg*)
have *F3*: $(deg\ (R(\!|carrier := S|\!))\ h) = deg\ R\ h$
by (*simp add*: *F0 F1*)
have *F30*: $h\ (deg\ (R(\!|carrier := S|\!))\ h) \in S$
using *A UP-cring.UP-car-memE*[*of* $R(\!|carrier := S|\!)$ *h deg* $(R(\!|carrier := S|\!))$ *h*]
by (*metis R.carrier-update*(*1*) *UP-ring-subring*(*1*) *assms*(*1*))
hence *F4*: *up-ring.monom P* $(h\ (deg\ R\ h))\ (deg\ R\ h) =$
$up\text{-}ring.monom\ (UP\ (R(\!|carrier := S|\!)))\ (h\ (deg\ (R(\!|carrier := S|\!))\ h))\ (deg\ (R(\!|carrier := S|\!))\ h)$
using *F3 g-closed j-def UP-subring-monom*[*of S h* $(deg\ (R(\!|carrier := S|\!))\ h)$] *assms*
unfolding *P-def* **by** *metis*
have *F5*: *compose R* $(up\text{-}ring.monom\ (UP\ R)\ (h\ (deg\ R\ h))\ (deg\ R\ h))\ g =$
$compose\ (R\ (\!|\ carrier := S\ |\!))\ (up\text{-}ring.monom\ (UP\ (R\ (\!|\ carrier := S\ |\!)))\ (h\ (deg\ (R(\!|carrier := S|\!))\ h))\ (deg\ (R(\!|carrier := S|\!))\ h))\ g$
using *F0 F1 F2 F3 F4 UP-subring-compose-monom*[*of S*] *assms P-def* ‹$h\ (deg\ (R(\!|carrier := S|\!))\ h) \in S$›
by (*metis g-closed*)
have *F5*: *compose R j g* = *compose* $(R\ (\!|\ carrier := S\ |\!))\ j\ g$
using *F0 F2 IH UP-ring-subring-car-subset′ assms*(*1*) *j-def* **by** *auto*
have *F6*: $h = j \oplus_{UP\ R} monom\ (UP\ R)\ (h\ (deg\ R\ h))\ (deg\ R\ h)$

using *j-def F4 UP-ring-subring-add*[*of S j up-ring.monom* (*UP* (*R*(|*carrier* := *S*|))) (*h* (*deg* (*R*(|*carrier* := *S*|)) *h*)) (*deg* (*R*(|*carrier* := *S*|)) *h*)]
UP-ring.monom-closed[*of R*(|*carrier* := *S*|) *h* (*deg* (*R*(|*carrier* := *S*|)) *h*) *deg* (*R*(|*carrier* := *S*|)) *h*]
using *P-def UP-ring-subring*(*2*) ‹*h* (*deg* (*R*(|*carrier* := *S*|)) *h*) ∈ *S*› *assms*(*1*) **by** *auto*
have *F7*: *compose R h g* =*compose R j g* ⊕$_{UP\ R}$
compose R (*up-ring.monom* (*UP R*) (*h* (*deg R h*)) (*deg R h*)) *g*
proof−
show *?thesis*
using *assms*(*2*) *j-closed F5 sub-add*[*of g j up-ring.monom P* (*h* (*deg R h*)) (*deg R h*)]
F4 F3 F2 F1 g-closed **unfolding** *P-def*
by (*metis A F6 ltrm-closed P-def*)
qed
have *F8*: *compose* (*R* (| *carrier* := *S* |)) *h g* = *compose* (*R* (| *carrier* := *S* |)) *j g* ⊕$_{UP}$ (*R* (| *carrier* := *S* |))
compose (*R* (| *carrier* := *S* |)) (*up-ring.monom* (*UP* (*R* (| *carrier* := *S* |))) (*h* (*deg* (*R* (| *carrier* := *S* |)) *h*)) (*deg* (*R* (| *carrier* := *S* |)) *h*)) *g*
proof−
have *0*: *UP-cring* (*R*(|*carrier* := *S*|))
by (*simp add*: *UP-ring-subring*(*1*) *assms*(*1*))
have *1*: *monom* (*UP* (*R*(|*carrier* := *S*|))) (*h* (*deg R h*)) (*deg R h*) ∈ *carrier* (*UP* (*R*(|*carrier* := *S*|)))
using *assms 0 F30 UP-ring.monom-closed*[*of R*(|*carrier* := *S*|) *h* (*deg R h*) *deg R h*] *R.carrier-update*[*of S*]
unfolding *UP-ring-def UP-cring-def*
by (*simp add*: *F3 cring.axioms*(*1*))
show *?thesis*
using *0 1 g-closed j-def UP-cring.sub-add*[*of R* (| *carrier* := *S* |) *g j monom* (*UP* (*R*(|*carrier* := *S*|))) (*h* (*deg R h*)) (*deg R h*)]
using *F3* **by** *auto*
qed
have *F9*: *compose R j g* ∈ *carrier* (*UP R*)
by (*simp add*: *UP-cring.sub-closed assms*(*2*) *is-UP-cring j-closed*)
have *F10*: *compose* (*R* (| *carrier* := *S* |)) *j g* ∈ *carrier* (*UP* (*R* (| *carrier* := *S* |)))
using *assms j-def UP-cring.sub-closed*[*of R* (| *carrier* := *S* |)] *UP-ring-subring*(*1*) *g-closed* **by** *blast*
have *F11*: *compose R* (*up-ring.monom* (*UP R*) (*h* (*deg R h*)) (*deg R h*)) *g* ∈ *carrier* (*UP R*)
using *assms j-def UP-cring.sub-closed*[*of R* (| *carrier* := *S* |)]
UP-ring.monom-closed[*of R* (| *carrier* := *S* |)]
by (*simp add*: *A UP-car-memE*(*1*) *UP-cring.rev-sub-closed UP-ring.monom-closed is-UP-cring is-UP-ring sub-rev-sub*)
have *F12*: *compose* (*R* (| *carrier* := *S* |)) (*up-ring.monom* (*UP* (*R* (| *carrier* := *S* |))) (*h* (*deg* (*R* (| *carrier* := *S* |)) *h*)) (*deg* (*R* (| *carrier* := *S* |)) *h*)) *g* ∈ *carrier* (*UP* (*R* (| *carrier* := *S* |)))

using *assms j-def UP-cring.sub-closed*[*of R* (| *carrier* := *S* |)]
UP-ring.monom-closed[*of R* (| *carrier* := *S* |)] *UP-ring-subring*[*of S*]
using *A UP-ring.ltrm-closed g-closed* **by** *fastforce*
show *?thesis* **using** *F9 F10 F11 F12 F7 F8 F5 UP-ring-subring-add*[*of S compose R j g compose R* (*up-ring.monom* (*UP R*) (*h* (*deg R h*)) (*deg R h*)) *g*]
assms
using *F3 F30 UP-subring-compose-monom g-closed* **by** *auto*
qed
qed
qed
qed
qed
show *?thesis* **using** *0*[*of deg R f*]
by (*simp add*: *assms*(*1*) *assms*(*3*) *assms*(*5*) *poly-cfs-subring*)
qed

12.2 Evaluation over a Subring

lemma(**in** *UP-cring*) *UP-subring-eval*:
assumes *subring S R*
assumes *g* ∈ *carrier* (*UP* (*R* (| *carrier* := *S* |)))
assumes *a* ∈ *S*
shows *to-function R g a* = *to-function* (*R* (| *carrier* := *S* |)) *g a*
apply(*rule UP-ring.poly-induct3*[*of R* (| *carrier* := *S* |) *g*])
apply (*simp add*: *UP-ring-subring*(*2*) *assms*(*1*))
apply (*simp add*: *assms*(*2*))
proof−
show ⋀*p q*. *q* ∈ *carrier* (*UP* (*R*(|*carrier* := *S*|))) ⟹
p ∈ *carrier* (*UP* (*R*(|*carrier* := *S*|))) ⟹
to-function R p a = *to-function* (*R*(|*carrier* := *S*|)) *p a* ⟹
to-function R q a = *to-function* (*R*(|*carrier* := *S*|)) *q a* ⟹
to-function R (*p* ⊕$_{UP}$ (*R*(|*carrier* := *S*|)) *q*) *a* = *to-function* (*R*(|*carrier* := *S*|)) (*p* ⊕$_{UP}$ (*R*(|*carrier* := *S*|)) *q*) *a*
proof− **fix** *p q* **assume** *A*: *q* ∈ *carrier* (*UP* (*R*(|*carrier* := *S*|)))
p ∈ *carrier* (*UP* (*R*(|*carrier* := *S*|)))
to-function R p a = *to-function* (*R*(|*carrier* := *S*|)) *p a*
to-function R q a = *to-function* (*R*(|*carrier* := *S*|)) *q a*
have *a-closed*: *a* ∈ *carrier R*
using *assms R.carrier-update*[*of S*] *subringE*(*1*) **by** *blast*
have *0*: *UP-cring* (*R*(|*carrier* := *S*|))
using *assms* **by** (*simp add*: *UP-ring-subring*(*1*))
have *1*: *to-function* (*R*(|*carrier* := *S*|)) *p a* ∈ *S*
using *A 0 UP-cring.to-fun-closed*[*of R*(|*carrier* := *S*|)]
by (*simp add*: *UP-cring.to-fun-def assms*(*3*))
have *2*: *to-function* (*R*(|*carrier* := *S*|)) *q a* ∈ *S*
using *A 0 UP-cring.to-fun-closed*[*of R*(|*carrier* := *S*|)]
by (*simp add*: *UP-cring.to-fun-def assms*(*3*))
have *3*: *p* ∈ *carrier* (*UP R*)

using *A assms 0 UP-ring-subring-car-subset'* **by** *blast*
have *4*: $q \in carrier\ (UP\ R)$
using *A assms 0 UP-ring-subring-car-subset'* **by** *blast*
have *5*: *to-fun p a* $\oplus$ *to-fun q a* = *UP-cring.to-fun* $(R\langle\!|carrier := S|\!\rangle)$ *p a* $\oplus_{R\langle\!|carrier := S|\!\rangle}$ *UP-cring.to-fun* $(R\langle\!|carrier := S|\!\rangle)$ *q a*
using *1 2 A R.carrier-update*[*of S*] *assms* **by** (*simp add*: *0 UP-cring.to-fun-def to-fun-def*)
have *6*: *UP-cring.to-fun* $(R\langle\!|carrier := S|\!\rangle)$ $(p \oplus_{UP} (R\langle\!|carrier := S|\!\rangle)\ q)$ *a* = *UP-cring.to-fun* $(R\langle\!|carrier := S|\!\rangle)$ *p a* $\oplus_{R\langle\!|carrier := S|\!\rangle}$ *UP-cring.to-fun* $(R\langle\!|carrier := S|\!\rangle)$ *q a*
using *UP-cring.to-fun-plus*[*of R* $\langle\!|$ *carrier* := *S* $|\!\rangle$ *q p a*]
by (*simp add*: *0 A*(*1*) *A*(*2*) *assms*(*3*))
have *7*: *to-fun* $(p \oplus_P q)$ *a* = *to-fun p a* $\oplus$ *to-fun q a*
using *to-fun-plus*[*of q p a*] *3 4 a-closed* **by** (*simp add*: *P-def*)
have *8*: $p \oplus_{UP} (R\langle\!|carrier := S|\!\rangle)\ q = p \oplus_P q$
unfolding *P-def* **using** *assms A R.carrier-update*[*of S*] *UP-ring-subring-add*[*of S p q*] **by** *simp*
show *to-function R* $(p \oplus_{UP} (R\langle\!|carrier := S|\!\rangle)\ q)$ *a* = *to-function* $(R\langle\!|carrier := S|\!\rangle)$ $(p \oplus_{UP} (R\langle\!|carrier := S|\!\rangle)\ q)$ *a*
using *UP-ring-subring-car-subset'*[*of S*] *0 1 2 3 4 5 6 7 8 A R.carrier-update*[*of S*]
unfolding *P-def* **by** (*simp add*: *UP-cring.to-fun-def to-fun-def*)
qed
show $\bigwedge b\ n.$
$b \in carrier\ (R\langle\!|carrier := S|\!\rangle) \Longrightarrow$
to-function R (*monom* (*UP* $(R\langle\!|carrier := S|\!\rangle)$) *b n*) *a* = *to-function* $(R\langle\!|carrier := S|\!\rangle)$ (*monom* (*UP* $(R\langle\!|carrier := S|\!\rangle)$) *b n*) *a*
proof− **fix** *b n* **assume** *A*: $b \in carrier\ (R\langle\!|carrier := S|\!\rangle)$
have *0*: *UP-cring* $(R\langle\!|carrier := S|\!\rangle)$
by (*simp add*: *UP-ring-subring*(*1*) *assms*(*1*))
have *a-closed*: $a \in carrier\ R$
using *assms subringE* **by** *blast*
have *1*: *UP-cring.to-fun* $(R\langle\!|carrier := S|\!\rangle)$ (*monom* (*UP* $(R\langle\!|carrier := S|\!\rangle)$) *b n*) *a* = *b* $\otimes_{R\langle\!|carrier := S|\!\rangle}$ *a* $[\widehat{}]_{R\langle\!|carrier := S|\!\rangle}$ *n*
using *assms A UP-cring.to-fun-monom*[*of R*$\langle\!|$*carrier* := *S*$|\!\rangle$ *b a n*]
by (*simp add*: *0*)
have *2*: *UP-cring.to-fun* $(R\langle\!|carrier := S|\!\rangle)$ (*monom* (*UP* $(R\langle\!|carrier := S|\!\rangle)$) *b n*) ≡ *to-function* $(R\langle\!|carrier := S|\!\rangle)$ (*monom* (*UP* $(R\langle\!|carrier := S|\!\rangle)$) *b n*)
using *UP-cring.to-fun-def*[*of R*$\langle\!|$*carrier* := *S*$|\!\rangle$ *monom* (*UP* $(R\langle\!|carrier := S|\!\rangle)$) *b n*] *0* **by** *linarith*
have *3*: (*monom* (*UP* $(R\langle\!|carrier := S|\!\rangle)$) *b n*) = *monom P b n*
using *A assms* **unfolding** *P-def* **using** *UP-subring-monom* **by** *auto*
have *4*: *b* $\otimes$ *a* $[\widehat{}]$ *n* = *b* $\otimes_{R\langle\!|carrier := S|\!\rangle}$ *a* $[\widehat{}]_{R\langle\!|carrier := S|\!\rangle}$ *n*
apply(*induction n*) **using** *R.carrier-update*[*of S*] **apply** *simp*
using *R.carrier-update*[*of S*] *R.nat-pow-consistent* **by** *auto*
hence *5*: *to-function R* (*monom* (*UP* $(R\langle\!|carrier := S|\!\rangle)$) *b n*) *a* = *b* $\otimes_{R\langle\!|carrier := S|\!\rangle}$ *a*$[\widehat{}]_{R\langle\!|carrier := S|\!\rangle}$*n*
using *0 1 2 3 assms A UP-cring.to-fun-monom*[*of R*$\langle\!|$*carrier* := *S*$|\!\rangle$ *b a n*]

UP-cring.to-fun-def[*of* $R(\!|carrier := S|\!)$ *monom* (*UP* ($R(\!|carrier := S|\!)$)) *b n*]
R.carrier-update[*of S*] *subringE*[*of S R*] *a-closed UP-ring.monom-closed*[*of* $R(\!|carrier := S|\!)$ *a n*]
to-fun-monom[*of b a n*]
unfolding *P-def UP-cring.to-fun-def to-fun-def* **by** (*metis subsetD*)
thus *to-function R* (*monom* (*UP* ($R(\!|carrier := S|\!)$)) *b n*) *a* = *to-function* ($R(\!|carrier := S|\!)$) (*monom* (*UP* ($R(\!|carrier := S|\!)$)) *b n*) *a*
using *1 2* **by** *auto*
qed
qed

lemma(**in** *UP-cring*) *UP-subring-eval'*:
assumes *subring S R*
assumes *g* ∈ *carrier* (*UP* (*R* (| *carrier* := *S* |)))
assumes *a* ∈ *S*
shows *to-fun g a* = *to-function* (*R* (| *carrier* := *S* |)) *g a*
unfolding *to-fun-def* **using** *assms*
by (*simp add*: *UP-subring-eval*)

lemma(**in** *UP-cring*) *UP-subring-eval-closed*:
assumes *subring S R*
assumes *g* ∈ *carrier* (*UP* (*R* (| *carrier* := *S* |)))
assumes *a* ∈ *S*
shows *to-fun g a* ∈ *S*
using *assms UP-subring-eval'*[*of S g a*] *UP-cring.to-fun-closed UP-cring.to-fun-def R.carrier-update*(*1*) *UP-ring-subring*(*1*) **by** *fastforce*

12.3 Derivatives and Taylor Expansions over a Subring

lemma(**in** *UP-cring*) *UP-subring-taylor*:
assumes *subring S R*
assumes *g* ∈ *carrier* (*UP R*)
assumes ⋀*n*. *g n* ∈ *S*
assumes *a* ∈ *S*
shows *taylor-expansion R a g* = *taylor-expansion* (*R* (| *carrier* := *S* |)) *a g*
proof−
have *a-closed*: *a* ∈ *carrier R*
using *assms subringE* **by** *blast*
have *0*: *X-plus a* ∈ *carrier* (*UP R*)
using *assms X-plus-closed* **unfolding** *P-def*
using *local.a-closed* **by** *auto*
have *1*: ⋀*n*. *X-plus a n* ∈ *S*
proof− **fix** *n*
have *X-plus a n* = (*if n* = *0 then a else*
(*if n* = *1 then* **1** *else* **0**))
using *a-closed*
by (*simp add*: *cfs-X-plus*)
then show *X-plus a n* ∈ *S* **using** *subringE assms*
by (*simp add*: *subringE*(*2*) *subringE*(*3*))

qed
have *2*: (*X-poly-plus* (*R*(|*carrier* := *S*|)) *a*) = *X-plus a*
proof−
have *20*: (*X-poly-plus* (*R*(|*carrier* := *S*|)) *a*) = (λ*k*. *if k* = (*0*::*nat*) *then a else*
(*if k* = *1 then* **1** *else* **0**))
using *a-closed assms UP-cring.cfs-X-plus*[*of R*(|*carrier* := *S*|) *a*] *R.carrier-update*

UP-ring-subring(*1*) **by** *auto*
have *21*: *X-plus a* = (λ*k*. *if k* = (*0*::*nat*) *then a else*
(*if k* = *1 then* **1** *else* **0**))
using *cfs-X-plus*[*of a*] *a-closed*
by *blast*
show *?thesis* **apply**(*rule ext*) **using** *20 21*
by *auto*
qed
show *?thesis*
unfolding *taylor-expansion-def* **using** *0 1 2 assms UP-subring-compose*[*of S g X-plus a*]
by (*simp add*: *UP-subring-compose*)
qed

lemma(**in** *UP-cring*) *UP-subring-taylor-closed*:
assumes *subring S R*
assumes *g* ∈ *carrier* (*UP R*)
assumes ⋀*n*. *g n* ∈ *S*
assumes *a* ∈ *S*
shows *taylor-expansion R a g* ∈ *carrier* (*UP* (*R* (| *carrier* := *S* |)))
proof−
have *g* ∈ *carrier* (*UP* (*R*(|*carrier* := *S*|)))
by (*metis P-def R.carrier-update*(*1*) *R.carrier-update*(*2*) *UP-cring.UP-car-memI'* *UP-ring-subring*(*1*) *assms*(*1*) *assms*(*2*) *assms*(*3*) *deg-leE*)
then show *?thesis*
using *assms UP-cring.taylor-def*[*of R*(|*carrier* := *S*|)] *UP-subring-taylor*[*of S g a*]
UP-cring.taylor-closed[*of R* (| *carrier* := *S* |) *g a*] *UP-ring-subring*(*1*)[*of S*]
by *simp*
qed

lemma(**in** *UP-cring*) *UP-subring-taylor-closed'*:
assumes *subring S R*
assumes *g* ∈ *carrier* (*UP* (*R* (| *carrier* := *S* |)))
assumes *a* ∈ *S*
shows *taylor-expansion R a g* ∈ *carrier* (*UP* (*R* (| *carrier* := *S* |)))
using *UP-subring-taylor-closed assms UP-cring.UP-car-memE*[*of R* (| *carrier* := *S* |) *g*] *R.carrier-update*[*of S*]
UP-ring-subring(*1*) *UP-ring-subring-car-subset'* **by** *auto*

lemma(**in** *UP-cring*) *UP-subring-taylor'*:
assumes *subring S R*

assumes $g \in carrier\ (UP\ R)$
assumes $\bigwedge n.\ g\ n \in S$
assumes $a \in S$
shows *taylor-expansion R a g n* $\in S$
using *assms UP-subring-taylor R.carrier-update*[*of S*] *UP-cring.taylor-closed*[*of R* (| *carrier* := *S* |)]
using *UP-cring.taylor-expansion-cf-closed UP-ring-subring*(*1*) *poly-cfs-subring*
by *metis*

lemma(**in** *UP-cring*) *UP-subring-deriv*:
assumes *subring S R*
assumes $g \in carrier\ (UP\ (R\ (|\ carrier := S\ |)))$
assumes $a \in S$
shows *deriv g a*= *UP-cring.deriv* (*R* (| *carrier* := *S* |)) *g a*
proof−
have *0*: $(\bigwedge n.\ g\ n \in S)$
using *assms UP-ring-subring-car* **by** *blast*
thus *?thesis*
unfolding *derivative-def* **using** *0 UP-ring-subring-car-subset*[*of S*] *assms UP-subring-taylor*[*of S g a*]
by (*simp add*: *subset-iff*)
qed

lemma(**in** *UP-cring*) *UP-subring-deriv-closed*:
assumes *subring S R*
assumes $g \in carrier\ (UP\ (R\ (|\ carrier := S\ |)))$
assumes $a \in S$
shows *deriv g a* $\in S$
using *assms UP-cring.deriv-closed*[*of R* (| *carrier* := *S* |) *g a*] *UP-subring-deriv*[*of S g a*]
UP-ring-subring-car-subset[*of S*] *UP-ring-subring*[*of S*]
by *simp*

lemma(**in** *UP-cring*) *poly-shift-subring-closed*:
assumes *subring S R*
assumes $g \in carrier\ (UP\ (R\ (|\ carrier := S\ |)))$
shows *poly-shift g* $\in carrier\ (UP\ (R\ (|\ carrier := S\ |)))$
using *UP-cring.poly-shift-closed*[*of R* (| *carrier* := *S* |) *g*] *assms UP-ring-subring*[*of S*]
by *simp*

lemma(**in** *UP-cring*) *UP-subring-taylor-appr*:
assumes *subring S R*
assumes $g \in carrier\ (UP\ (R\ (|\ carrier := S\ |)))$
assumes $a \in S$
assumes $b \in S$
shows $\exists c \in S.$ *to-fun g a*= *to-fun g b* $\oplus$ (*deriv g b*)$\otimes$ $(a \ominus b) \oplus$ $(c \otimes (a \ominus b)[\hat{}](2::nat))$

proof−
have *a-closed*: $a \in$ *carrier* R
using *assms subringE* **by** *blast*
have *b-closed*: $b \in$ *carrier* R
using *assms subringE* **by** *blast*
have *g-closed*: $g \in$ *carrier* (*UP* R)
using *UP-ring-subring-car-subset*[*of* S] *assms* **by** *blast*
have *0*: *to-fun* (*shift* 2 (T_b g)) ($a \ominus b$) = *to-fun* (*shift* 2 (T_b g)) ($a \ominus b$)
by *simp*
have *1*: *to-fun* g b = *to-fun* g b
by *simp*
have *2*: *deriv* g b = *deriv* g b
by *simp*
have *3*: *to-fun* g a = *to-fun* g b ⊕ *deriv* g b ⊗ ($a \ominus b$) ⊕ *to-fun* (*shift* 2 (T_b g)) ($a \ominus b$) ⊗ ($a \ominus b$) [^] (2::*nat*)
using *taylor-deg-1-expansion*[*of* g b a *to-fun* (*shift* 2 (T_b g)) ($a \ominus b$) *to-fun* g b *deriv* g b]
assms a-closed b-closed g-closed 0 1 2 **unfolding** *P-def* **by** *blast*
have *4*: *to-fun* (*shift* 2 (T_b g)) ($a \ominus b$) ∈ S
proof−
have *0*: (2::*nat*) = *Suc* (*Suc* 0)
by *simp*
have *1*: $a \ominus b \in S$
using *assms* **unfolding** *a-minus-def*
by (*simp add*: *subringE*(5) *subringE*(7))
have *2*: *poly-shift* (T_b g) ∈ *carrier* (*UP* (*R*(|*carrier* := S|)))
using *poly-shift-subring-closed*[*of* S *taylor-expansion* R b g] *UP-ring-subring*[*of* S]
UP-subring-taylor-closed'[*of* S g b] *assms* **unfolding** *taylor-def*
by *blast*
hence *3*: *poly-shift* (*poly-shift* (T_b g)) ∈ *carrier* (*UP* (*R*(|*carrier* := S|)))
using *UP-cring.poly-shift-closed*[*of* *R*(|*carrier* := S|) (*poly-shift* (T_b g))]
unfolding *taylor-def*
using *assms*(1) *poly-shift-subring-closed* **by** *blast*
have *4*: *to-fun* (*poly-shift* (*poly-shift* (T_b g))) ($a \ominus b$) ∈ S
using *1 2 3 0 UP-subring-eval-closed*[*of* S *poly-shift* (*poly-shift* (T_b g)) $a \ominus b$]
UP-cring.poly-shift-closed[*of* *R*(|*carrier* := S|)] *assms*
by *blast*
then show *?thesis*
by (*simp add*: *numeral-2-eq-2*)
qed
obtain c **where** *c-def*: c = *to-fun* (*shift* 2 (T_b g)) ($a \ominus b$)
by *blast*
have *5*: $c \in S \wedge$ *to-fun* g a = *to-fun* g b ⊕ *deriv* g b ⊗ ($a \ominus b$) ⊕ c ⊗ ($a \ominus b$) [^] (2::*nat*)
unfolding *c-def* **using** *3 4* **by** *blast*
thus *?thesis* **using** *c-def 4* **by** *blast*

qed

lemma(**in** *UP-cring*) *UP-subring-taylor-appr'*:
assumes *subring S R*
assumes $g \in$ *carrier* (*UP* (*R* (| *carrier* := *S* |)))
assumes $a \in S$
assumes $b \in S$
shows $\exists\, c\ c'\ c''.\ c \in S \wedge c' \in S \wedge c'' \in S \wedge$ *to-fun g a*$= c \oplus c' \otimes (a \ominus b) \oplus (c'' \otimes (a \ominus b)[\widehat{\ }](2::nat))$
using *UP-subring-taylor-appr*[*of S g a b*] *assms UP-subring-deriv-closed*[*of S g b*] *UP-subring-eval-closed*[*of S g b*]
by *blast*

lemma (**in** *UP-cring*) *pderiv-cfs*:
assumes*g* $\in$ *carrier* (*UP R*)
shows *pderiv g n* $= [Suc\ n] \cdot (g\ (Suc\ n))$
unfolding *pderiv-def*
using *n-mult-closed*[*of g*] *assms poly-shift-cfs*[*of n-mult g n*]
unfolding *P-def n-mult-def* **by** *blast*

lemma(**in** *ring*) *subring-add-pow*:
assumes *subring S R*
assumes $a \in S$
shows $[(n::nat)]\ \cdot_{R(|carrier := S|)}\ a = [(n::nat)]\ \cdot a$
proof−
have *0*: $a \in$ *carrier R*
using *assms*(*1*) *assms*(*2*) *subringE*(*1*) **by** *blast*
have *1*: $a \in$ *carrier* (*R*(|*carrier* := *S*|))
by (*simp add*: *assms*(*2*))
show *?thesis*
apply(*induction n*)
using *assms 0 1 carrier-update*[*of S*]
apply (*simp add*: *add-pow-def*)
using *assms 0 1 carrier-update*[*of S*]
by (*simp add*: *add-pow-def*)
qed

lemma(**in** *UP-cring*) *UP-subring-pderiv-equal*:
assumes *subring S R*
assumes $g \in$ *carrier* (*UP* (*R* (| *carrier* := *S* |)))
shows *pderiv g* = *UP-cring.pderiv* (*R*(|*carrier* := *S*|)) *g*
proof fix *n*
show *pderiv g n* = *UP-cring.pderiv* (*R*(|*carrier* := *S*|)) *g n*
using *UP-cring.pderiv-cfs*[*of R* (| *carrier* := *S* |) *g n*] *pderiv-cfs*[*of g n*]
assms R.subring-add-pow[*of S g* (*Suc n*) *Suc n*]
by (*simp add*: *UP-ring-subring*(*1*) *UP-ring-subring-car*)
qed

lemma(**in** *UP-cring*) *UP-subring-pderiv-closed*:

assumes *subring S R*
assumes $g \in$ *carrier* (*UP* (*R* (| *carrier* := *S* |)))
shows *pderiv g* $\in$ *carrier* (*UP* (*R* (| *carrier* := *S* |)))
using *assms UP-cring.pderiv-closed*[*of R* (| *carrier* := *S* |) *g*] *R.carrier-update*(*1*)
UP-ring-subring(*1*)
UP-subring-pderiv-equal **by** *auto*

lemma(**in** *UP-cring*) *UP-subring-pderiv-closed'*:
assumes *subring S R*
assumes *g* $\in$ *carrier* (*UP R*)
assumes $\bigwedge n.\ g\ n \in S$
shows $\bigwedge n.$ *pderiv g n* $\in S$
using *assms UP-subring-pderiv-closed*[*of S g*] *poly-cfs-subring*[*of S g*] *UP-ring-subring-car*
by *blast*

lemma(**in** *UP-cring*) *taylor-deg-one-expansion-subring*:
assumes *f* $\in$ *carrier* (*UP R*)
assumes *subring S R*
assumes $\bigwedge i.\ f\ i \in S$
assumes *a* $\in S$
assumes *b* $\in S$
shows $\exists\, c \in S.$ *to-fun f b* = (*to-fun f a*) $\oplus$ (*deriv f a*) $\otimes$ (*b* $\ominus$ *a*) $\oplus$ (*c* $\otimes$ (*b* $\ominus$ *a*)[^](*2*::*nat*))
apply(*rule UP-subring-taylor-appr*, *rule assms*)
using *assms poly-cfs-subring* **apply** *blast*
by(*rule assms*, *rule assms*)

lemma(**in** *UP-cring*) *taylor-deg-one-expansion-subring'*:
assumes *f* $\in$ *carrier* (*UP R*)
assumes *subring S R*
assumes $\bigwedge i.\ f\ i \in S$
assumes *a* $\in S$
assumes *b* $\in S$
shows $\exists\, c \in S.$ *to-fun f b* = (*to-fun f a*) $\oplus$ (*to-fun* (*pderiv f*) *a*) $\otimes$ (*b* $\ominus$ *a*) $\oplus$ (*c* $\otimes$ (*b* $\ominus$ *a*)[^](*2*::*nat*))
proof−
have *S* $\subseteq$ *carrier R*
using *assms subringE*(*1*) **by** *blast*
hence *0*: *deriv f a* = *to-fun* (*pderiv f*) *a*
using *assms pderiv-eval-deriv*[*of f a*] **unfolding** *P-def* **by** *blast*
show *?thesis*
using *assms taylor-deg-one-expansion-subring*[*of f S a b*]
unfolding *0* **by** *blast*
qed

end
theory *Supplementary-Ring-Facts*
imports *HOL*−*Algebra.Ring*
HOL−*Algebra.UnivPoly*

begin

13 Supplementary Ring Facts

The nonzero elements of a ring.

definition *nonzero* :: ('*a*, '*b*) *ring-scheme* ⇒ '*a set* **where**
nonzero R = {*a* ∈ *carrier R*. *a* ≠ **0**$_R$}

lemma *zero-not-in-nonzero*:
0$_R$ ∉ *nonzero R*
unfolding *nonzero-def* **by** *blast*

lemma(**in** *domain*) *nonzero-memI*:
assumes *a* ∈ *carrier R*
assumes *a* ≠ **0**
shows *a* ∈ *nonzero R*
using *assms* **by**(*simp add*: *nonzero-def*)

lemma(**in** *domain*) *nonzero-memE*:
assumes *a* ∈ *nonzero R*
shows *a* ∈ *carrier R a* ≠**0**
using *assms* **by**(*auto simp*: *nonzero-def*)

lemma(**in** *domain*) *not-nonzero-memE*:
assumes *a* ∉ *nonzero R*
assumes *a* ∈ *carrier R*
shows *a* = **0**
using *assms*
by (*simp add*: *nonzero-def*)

lemma(**in** *domain*) *not-nonzero-memI*:
assumes *a* = **0**
shows *a* ∉ *nonzero R*
using *assms nonzero-memE*(*2*) **by** *auto*

lemma(**in** *domain*) *nonzero-closed*:
assumes *a* ∈ *nonzero R*
shows *a* ∈ *carrier R*
using *assms*
by (*simp add*: *nonzero-def*)

lemma(**in** *domain*) *nonzero-mult-in-car*:
assumes *a* ∈ *nonzero R*
assumes *b* ∈ *nonzero R*
shows *a* ⊗ *b* ∈ *carrier R*

using *assms*
by (*simp add*: *nonzero-def*)

lemma(**in** *domain*) *nonzero-mult-closed*:
assumes $a \in$ *nonzero R*
assumes $b \in$ *nonzero R*
shows $a \otimes b \in$ *nonzero R*
apply(*rule nonzero-memI*)
using *assms nonzero-memE* **apply** *blast*
using *assms nonzero-memE*
by (*simp add*: *integral-iff*)

lemma(**in** *domain*) *nonzero-one-closed*:
$\mathbf{1} \in$ *nonzero R*
by (*simp add*: *nonzero-def*)

lemma(**in** *domain*) *one-nonzero*:
$\mathbf{1} \in$ *nonzero R*
by (*simp add*: *nonzero-one-closed*)

lemma(**in** *domain*) *nat-pow-nonzero*:
assumes $x \in$*nonzero R*
shows $x[\hat{}](n::nat) \in$ *nonzero R*
unfolding *nonzero-def*
apply(*induction n*)
using *assms integral-iff nonzero-closed zero-not-in-nonzero* **by** *auto*

lemma(**in** *monoid*) *Units-int-pow-closed*:
assumes $x \in$ *Units G*
shows $x[\hat{}](n::int) \in$ *Units G*
by (*metis Units-pow-closed assms int-pow-def2 monoid.Units-inv-Units monoid-axioms*)

lemma(**in** *comm-monoid*) *UnitsI*:
assumes $a \in$ *carrier G*
assumes $b \in$ *carrier G*
assumes $a \otimes b = \mathbf{1}$
shows $a \in$ *Units G* $b \in$ *Units G*
unfolding *Units-def* **using** *comm-monoid-axioms-def assms m-comm*[*of a b*]
by *auto*

lemma(**in** *comm-monoid*) *is-invI*:
assumes $a \in$ *carrier G*
assumes $b \in$ *carrier G*
assumes $a \otimes b = \mathbf{1}$
shows $inv_G\ b = a$ $inv_G\ a = b$
using *assms inv-char m-comm*
by *auto*

lemma(**in** *ring*) *ring-in-Units-imp-not-zero*:

assumes $\mathbf{1} \neq \mathbf{0}$
assumes $a \in$ *Units* R
shows $a \neq \mathbf{0}$
using *assms monoid.Units-l-cancel*
by (*metis l-null monoid-axioms one-closed zero-closed*)

lemma(**in** *ring*) *Units-nonzero*:
assumes $u \in$ *Units* R
assumes $\mathbf{1}_R \neq \mathbf{0}_R$
shows $u \in$ *nonzero* R
proof−
have $u \in$*carrier* R
using *Units-closed assms* **by** *auto*
have $u \neq$**0**
using *Units-r-inv-ex assms*(*1*) *assms*(*2*)
by *force*
thus *?thesis*
by (*simp add*: ‹$u \in$ *carrier* R› *nonzero-def*)
qed

lemma(**in** *ring*) *Units-inverse*:
assumes $u \in$ *Units* R
shows *inv* $u \in$ *Units* R
by (*simp add*: *assms*)

lemma(**in** *cring*) *invI*:
assumes $x \in$ *carrier* R
assumes $y \in$ *carrier* R
assumes $x \otimes_R y = \mathbf{1}_R$
shows $y =$ *inv* $_R\ x$
$x =$ *inv* $_R\ y$
using *assms*(*1*) *assms*(*2*) *assms*(*3*) *is-invI*
by *auto*

lemma(**in** *cring*) *inv-cancelR*:
assumes $x \in$ *Units* R
assumes $y \in$ *carrier* R
assumes $z \in$ *carrier* R
assumes $y = x \otimes_R z$
shows $inv_R\ x \otimes_R y = z$
$y \otimes_R (inv_R\ x) = z$
apply (*metis Units-closed assms*(*1*) *assms*(*3*) *assms*(*4*) *cring.cring-simprules*(*12*)

is-cring m-assoc monoid.Units-inv-closed monoid.Units-l-inv monoid-axioms)
by (*metis Units-closed assms*(*1*) *assms*(*3*) *assms*(*4*) *m-assoc m-comm monoid.Units-inv-closed*

monoid.Units-r-inv monoid.r-one monoid-axioms)

lemma(**in** *cring*) *inv-cancelL*:
assumes $x \in$ *Units* R
assumes $y \in$ *carrier* R
assumes $z \in$ *carrier* R
assumes $y = z \otimes_R x$
shows $inv_R\ x \otimes_R y = z$
$y \otimes_R (inv_R\ x) = z$
apply (*simp add*: *Units-closed assms*(*1*) *assms*(*3*) *assms*(*4*) *m-lcomm*)
by (*simp add*: *Units-closed assms*(*1*) *assms*(*3*) *assms*(*4*) *m-assoc*)

end

14 Extended integers (i.e. with infinity)

This section formalizes the extended integers, which serve as the codomain for the p-adic valuation. The element ∞ is added to the integers to serve as a maximxal element in the order, which is the valuation of 0.

theory *Extended-Int*
imports *Main HOL−Library.Countable HOL−Library.Order-Continuity HOL−Library.Extended-Nat*
begin

The following is based very closely on the theory *HOL−Library.Extended-Nat* from the standard Isabelle distribution, with adaptations made to formalize the integers extended with an element for infinity. This is the standard range for the valuation function on a discretely valued ring such as the field of p-adic numbers, such as in [4].

context
fixes f :: *nat* ⇒ 'a::{*canonically-ordered-monoid-add*, *linorder-topology*, *complete-linorder*}
begin

lemma *sums-SUP*[*simp*, *intro*]: f *sums* (*SUP n.* $\sum i<n.\ f\ i$)
unfolding *sums-def* **by** (*intro LIMSEQ-SUP monoI sum-mono2 zero-le*) *auto*

lemma *suminf-eq-SUP*: *suminf* f = (*SUP n.* $\sum i<n.\ f\ i$)
using *sums-SUP* **by** (*rule sums-unique*[*symmetric*])

end

14.1 Type definition

We extend the standard natural numbers by a special value indicating infinity.

typedef *eint* = *UNIV* :: *int option set* **..**

definition *eint* :: *int* ⇒ *eint* **where**
eint n = *Abs-eint* (*Some n*)

instantiation *eint* :: *infinity*
begin

definition ∞ = *Abs-eint None*
instance ..

end

fun *int-option-enumeration* :: *int option* ⇒ *nat* **where**
int-option-enumeration (*Some n*) = (*if n* ≥ *0 then nat* (*2*∗(*n* + *1*)) *else nat* (*2*∗(−*n*) + *1*))|
int-option-enumeration None = (*0*::*nat*)

lemma *int-option-enumeration-inj*:
inj int-option-enumeration
proof
have *pos-even*: ⋀*n*::*int*. *n* ≥ *0* ⟹ *even* (*int-option-enumeration* (*Some n*)) ∧ (*int-option-enumeration* (*Some n*))> *0*
proof−
fix *n*::*int* **assume** *n* ≥*0* **then have** (*int-option-enumeration* (*Some n*)) = *nat* (*2*∗(*n* + *1*))
by *simp*
then show *even* (*int-option-enumeration* (*Some n*)) ∧ *0* < *int-option-enumeration* (*Some n*)
using ‹*0* ≤ *n*› *even-nat-iff* **by** *force*
qed
have *neg-odd*: ⋀*n*::*int*. *n* < *0* ⟹ *odd* (*int-option-enumeration* (*Some n*))
by (*simp add*: *even-nat-iff*)
fix *x y* **assume** *A*: *x* ∈ *UNIV y* ∈ *UNIV int-option-enumeration x* = *int-option-enumeration y*
show *x* = *y*
apply(*cases x* = *None*)
using *A pos-even neg-odd*
apply (*metis dvd-0-right int-option-enumeration.elims int-option-enumeration.simps*(*2*) *not-gr0 not-le*)
proof−
assume *x* ≠*None*
then obtain *n* **where** *n-def*: *x* = *Some n*
by *auto*
then have *y-not-None*: *y* ≠ *None*
using *A*
by (*metis* ‹⋀*thesis*. (⋀*n*. *x* = *Some n* ⟹ *thesis*) ⟹ *thesis*›
add-cancel-right-right even-add int-option-enumeration.simps(*2*)
linorder-not-less neg-odd neq0-conv pos-even)
then obtain *m* **where** *m-def*: *y* = *Some m*

```
    by blast
  show ?thesis
  proof(cases n ≥0)
    case True
    then show ?thesis
      using n-def A neg-odd pos-even m-def int-option-enumeration.simps
      by (smt (verit) eq-nat-nat-iff)
  next
    case False
    then show ?thesis
      using n-def A neg-odd pos-even m-def int-option-enumeration.simps(1)
      by (smt (verit) eq-nat-nat-iff)
  qed
 qed
qed

definition eint-enumeration where
eint-enumeration = int-option-enumeration ∘ Rep-eint

lemma eint-enumeration-inj:
inj eint-enumeration
 unfolding eint-enumeration-def
 using int-option-enumeration-inj Rep-eint-inject
 by (metis (mono-tags, lifting) comp-apply injD inj-on-def)

instance eint :: countable
proof
 show ∃ to-int::eint ⇒ nat. inj to-int
  using eint-enumeration-inj by blast
qed

old-rep-datatype eint ∞ :: eint
proof −
 fix P i assume ⋀j. P (eint j) P ∞
 then show P i
 proof induct
  case (Abs-eint y) then show ?case
   by (cases y rule: option.exhaust)
     (auto simp: eint-def infinity-eint-def)
 qed
qed (auto simp add: eint-def infinity-eint-def Abs-eint-inject)

declare [[coercion eint::int⇒eint]]

lemmas eint2-cases = eint.exhaust[case-product eint.exhaust]
lemmas eint3-cases = eint.exhaust[case-product eint.exhaust eint.exhaust]

lemma not-infinity-eq [iff]: (x ≠ ∞) = (∃ i. x = eint i)
 by (cases x) auto
```

lemma *not-eint-eq* [*iff*]: $(\forall\, y.\ x \neq \mathit{eint}\ y) = (x = \infty)$
by (*cases x*) *auto*

lemma *eint-ex-split*: $(\exists\, c{::}\mathit{eint}.\ P\ c) \longleftrightarrow P\ \infty \vee (\exists\, c{::}\mathit{int}.\ P\ c)$
by (*metis eint.exhaust*)

primrec *the-eint* :: *eint* ⇒ *int*
where *the-eint* (*eint n*) = *n*

14.2 Constructors and numbers

instantiation *eint* :: *zero-neq-one*
begin

definition
0 = *eint 0*

definition
1 = *eint 1*

instance
proof qed (*simp add*: *zero-eint-def one-eint-def*)

end

lemma *eint-0* [*code-post*]: *eint 0* = *0*
by (*simp add*: *zero-eint-def*)

lemma *eint-1* [*code-post*]: *eint 1* = *1*
by (*simp add*: *one-eint-def*)

lemma *eint-0-iff*: *eint x* = *0* ⟷ *x* = *0* *0* = *eint x* ⟷ *x* = *0*
by (*auto simp add*: *zero-eint-def*)

lemma *eint-1-iff*: *eint x* = *1* ⟷ *x* = *1* *1* = *eint x* ⟷ *x* = *1*
by (*auto simp add*: *one-eint-def*)

lemma *infinity-ne-i0* [*simp*]: $(\infty{::}\mathit{eint}) \neq 0$
by (*simp add*: *zero-eint-def*)

lemma *i0-ne-infinity* [*simp*]: $0 \neq (\infty{::}\mathit{eint})$
by (*simp add*: *zero-eint-def*)

lemma *zero-one-eint-neq*:
¬ *0* = (*1*::*eint*)
¬ *1* = (*0*::*eint*)
unfolding *zero-eint-def one-eint-def* **by** *simp-all*

lemma *infinity-ne-i1* [*simp*]: $(\infty::eint) \neq 1$
by (*simp add*: *one-eint-def*)

lemma *i1-ne-infinity* [*simp*]: $1 \neq (\infty::eint)$
by (*simp add*: *one-eint-def*)

14.3 Addition

instantiation *eint* :: *comm-monoid-add*
begin

definition [*nitpick-simp*]:
$m + n = (case\ m\ of\ \infty \Rightarrow \infty \mid eint\ m \Rightarrow (case\ n\ of\ \infty \Rightarrow \infty \mid eint\ n \Rightarrow eint\ (m + n)))$

lemma *plus-eint-simps* [*simp*, *code*]:
fixes q :: *eint*
shows $eint\ m + eint\ n = eint\ (m + n)$
and $\infty + q = \infty$
and $q + \infty = \infty$
by (*simp-all add*: *plus-eint-def split*: *eint.splits*)

instance
proof
fix $n\ m\ q$ :: *eint*
show $n + m + q = n + (m + q)$
by (*cases n m q rule*: *eint3-cases*) *auto*
show $n + m = m + n$
by (*cases n m rule*: *eint2-cases*) *auto*
show $0 + n = n$
by (*cases n*) (*simp-all add*: *zero-eint-def*)
qed

end

lemma *eSuc-eint*: $(eint\ n) + 1 = eint\ (n + 1)$
by (*simp add*: *one-eint-def*)

lemma *eSuc-infinity* [*simp*]: $\infty + (1::eint) = \infty$
unfolding *plus-eint-def*
by *auto*

lemma *eSuc-inject* [*simp*]: $m + (1::eint) = n + 1 \longleftrightarrow m = n$
unfolding *plus-eint-def*
apply(*cases* $m = \infty$)
apply (*metis* (*no-types*, *lifting*) *eSuc-eint*
eint.distinct(*2*) *eint.exhaust eint.simps*(*4*) *eint.simps*(*5*) *plus-eint-def*)
apply(*cases* $n = \infty$)

using *eSuc-eint plus-eint-def* **apply** *auto*[*1*]
unfolding *one-eint-def*
using *add.commute eSuc-eint*
by *auto*

lemma *eSuc-eint-iff*: $x + 1 = eint\ y \longleftrightarrow (\exists n.\ y = n + 1 \wedge x = eint\ n)$
apply(*cases* $x = \infty$)
apply *simp*
unfolding *plus-eint-def one-eint-def*
using *eSuc-eint*
by *auto*

lemma *enat-eSuc-iff*: $eint\ y = x + 1 \longleftrightarrow (\exists n.\ y = n + 1 \wedge eint\ n = x)$
using *eSuc-eint-iff*
by *metis*

lemma *iadd-Suc*: $((m::eint) + 1) + n = (m + n) + 1$
by (*metis ab-semigroup-add-class.add-ac*(*1*) *add.assoc add.commute*)

lemma *iadd-Suc-right*: $(m::eint) + (n + 1) = (m + n) + 1$
using *add.assoc*[*of m n 1*] **by** *auto*

14.4 Multiplication

instantiation *eint* :: {*comm-semiring*}
begin

definition *times-eint-def* [*nitpick-simp*]:
$m * n = (case\ m\ of\ \infty \Rightarrow\ \infty \mid eint\ m \Rightarrow$
$(case\ n\ of\ \infty \Rightarrow \infty \mid eint\ n \Rightarrow eint\ (m * n)))$

lemma *times-eint-simps* [*simp*, *code*]:
$eint\ m * eint\ n = eint\ (m * n)$
$\infty * \infty = (\infty::eint)$
$\infty * eint\ n = \infty$
$eint\ m * \infty = \infty$
unfolding *times-eint-def zero-eint-def*
by (*simp-all split*: *eint.split*)

lemma *sum-infinity-imp-summand-infinity*:
assumes $a + b = (\infty::eint)$
shows $a = \infty \vee b = \infty$
using *assms*
by (*metis not-eint-eq plus-eint-simps*(*1*))

lemma *sum-finite-imp-summands-finite*:
assumes $a + b \neq (\infty::eint)$
shows $a \neq \infty\ b \neq \infty$

```
  using assms eint.simps(5) apply fastforce
  using assms eint.simps(5) by fastforce


instance
proof
  fix a b c :: eint
  show (a * b) * c = a * (b * c)
    unfolding times-eint-def zero-eint-def
    by (simp split: eint.split)
  show comm: a * b = b * a
    unfolding times-eint-def zero-eint-def
    by (simp split: eint.split)
  show distr: (a + b) * c = a * c + b * c
    unfolding times-eint-def plus-eint-def
    apply(cases a + b = ∞)
    apply(cases a = ∞)
    apply simp
    using sum-infinity-imp-summand-infinity[of a b]
    apply (metis eint.simps(5) plus-eint-def plus-eint-simps(3))
    apply(cases c = ∞)
   apply (metis eint.exhaust plus-eint-def plus-eint-simps(3) times-eint-def times-eint-simps(4))
    using sum-finite-imp-summands-finite[of a b]
    apply auto
    by (simp add: semiring-normalization-rules(1))
qed


end

lemma mult-one-right[simp]:
(n::eint)*1 = n
  apply(cases n = ∞)
  apply (simp add: one-eint-def)
  by (metis eint2-cases mult-cancel-left2 one-eint-def times-eint-simps(1))

lemma mult-one-left[simp]:
1*(n::eint) = n
  by (metis mult.commute mult-one-right)

lemma mult-eSuc: ((m::eint) + 1) * n = m * n + n
  by (simp add: distrib-right)

lemma mult-eSuc': ((m::eint) + 1) * n = n + m * n
  using mult-eSuc add.commute by simp

lemma mult-eSuc-right: (m::eint) * (n + 1) =  m * n +  m
  by(simp add: distrib-left)
```

lemma *mult-eSuc-right'*: $(m::eint) * (n + 1) = m + m * n$
using *mult-eSuc-right add.commute* **by** *simp*

14.5 Numerals

lemma *numeral-eq-eint*:
$numeral\ k = eint\ (numeral\ k)$
by *simp*

lemma *eint-numeral* [*code-abbrev*]:
$eint\ (numeral\ k) = numeral\ k$
using *numeral-eq-eint* **..**

lemma *infinity-ne-numeral* [*simp*]: $(\infty::eint) \neq numeral\ k$
by *auto*

lemma *numeral-ne-infinity* [*simp*]: $numeral\ k \neq (\infty::eint)$
by *simp*

14.6 Subtraction

instantiation *eint* :: *minus*
begin

definition *diff-eint-def*:
$a - b = (case\ a\ of\ (eint\ x) \Rightarrow (case\ b\ of\ (eint\ y) \Rightarrow eint\ (x - y) \mid \infty \Rightarrow \infty)$
$\quad \mid \infty \Rightarrow \infty)$

instance **..**

end

lemma *idiff-eint-eint* [*simp*, *code*]: $eint\ a - eint\ b = eint\ (a - b)$
by (*simp add*: *diff-eint-def*)

lemma *idiff-infinity* [*simp*, *code*]: $\infty - n = (\infty::eint)$
by (*simp add*: *diff-eint-def*)

lemma *idiff-infinity-right* [*simp*, *code*]: $eint\ a - \infty = \infty$
by (*simp add*: *diff-eint-def*)

lemma *idiff-0* [*simp*]: $(0::eint) - n = -n$
by (*cases n*, *simp-all add*: *zero-eint-def*)

lemmas *idiff-eint-0* [*simp*] = *idiff-0* [*unfolded zero-eint-def*]

lemma *idiff-0-right* [*simp*]: $(n::eint) - 0 = n$

by (*cases n*) (*simp-all add*: *zero-eint-def*)

lemmas *idiff-eint-0-right* [*simp*] = *idiff-0-right* [*unfolded zero-eint-def*]

lemma *idiff-self* [*simp*]: $n \neq \infty \Longrightarrow (n::eint) - n = 0$
by (*auto simp*: *zero-eint-def*)

lemma *eSuc-minus-eSuc* [*simp*]: $((n::eint) + 1) - (m + 1) = n - m$
apply(*cases* $n = \infty$)
apply *simp*
apply(*cases* $m = \infty$)
apply (*metis eSuc-infinity eint.exhaust idiff-infinity-right infinity-ne-i1 sum-infinity-imp-summand-infinity*)
proof−
assume *A*: $n \neq \infty$ $m \neq \infty$
obtain *a* **where** *a-def*: $n = eint\ a$
using *A*
by *auto*
obtain *b* **where** *b-def*: $m = eint\ b$
using *A*
by *auto*
show *?thesis*
using *idiff-eint-eint*[*of* $a + 1$ $b + 1$]
idiff-eint-eint[*of a b*]
by (*simp add*: *a-def b-def eSuc-eint*)
qed

lemma *eSuc-minus-1* [*simp*]: $((n::eint) + 1) - 1 = n$
using *eSuc-minus-eSuc*[*of n 0*]
by *auto*

14.7 Ordering

instantiation *eint* :: *linordered-ab-semigroup-add*
begin

definition [*nitpick-simp*]:
$m \leq n = (case\ n\ of\ eint\ n1 \Rightarrow (case\ m\ of\ eint\ m1 \Rightarrow m1 \leq n1 \mid \infty \Rightarrow False) \mid \infty \Rightarrow True)$

definition [*nitpick-simp*]:
$m < n = (case\ m\ of\ eint\ m1 \Rightarrow (case\ n\ of\ eint\ n1 \Rightarrow m1 < n1 \mid \infty \Rightarrow True) \mid \infty \Rightarrow False)$

lemma *eint-ord-simps* [*simp*]:
$eint\ m \leq eint\ n \longleftrightarrow m \leq n$
$eint\ m < eint\ n \longleftrightarrow m < n$
$q \leq (\infty::eint)$
$q < (\infty::eint) \longleftrightarrow q \neq \infty$

$(\infty::eint) \leq q \longleftrightarrow q = \infty$
$(\infty::eint) < q \longleftrightarrow False$
by (*simp-all add*: *less-eq-eint-def less-eint-def split*: *eint.splits*)

lemma *numeral-le-eint-iff*[*simp*]:
shows *numeral m* $\leq$ *eint n* $\longleftrightarrow$ *numeral m* $\leq$ *n*
by *auto*

lemma *numeral-less-eint-iff*[*simp*]:
shows *numeral m* $<$ *eint n* $\longleftrightarrow$ *numeral m* $<$ *n*
by *simp*

lemma *eint-ord-code* [*code*]:
eint m $\leq$ *eint n* $\longleftrightarrow$ *m* $\leq$ *n*
eint m $<$ *eint n* $\longleftrightarrow$ *m* $<$ *n*
q $\leq (\infty::eint) \longleftrightarrow True$
eint m $< \infty \longleftrightarrow True$
$\infty \leq$ *eint n* $\longleftrightarrow False$
$(\infty::eint) < q \longleftrightarrow False$
by *simp-all*

lemma *eint-ord-plus-one*[*simp*]:
assumes *eint n* $\leq$ *x*
assumes *x* $<$ *y*
shows *eint* (*n* + *1*) $\leq$ *y*
proof−
obtain *m* **where** *x* = *eint m*
using *assms*(*2*)
by *fastforce*
show *?thesis* **apply**(*cases y* $= \infty$)
apply *simp*
using ‹*x* = *eint m*› *assms*(*1*) *assms*(*2*)
by *force*
qed

instance
by *standard* (*auto simp add*: *less-eq-eint-def less-eint-def plus-eint-def split*: *eint.splits*)

end

instance *eint* :: {*strict-ordered-comm-monoid-add*}
proof
show *a* $<$ *b* $\Longrightarrow$ *c* $<$ *d* $\Longrightarrow$ *a* + *c* $<$ *b* + *d* **for** *a b c d* :: *eint*
by (*cases a b c d rule*: *eint2-cases*[*case-product eint2-cases*]) *auto*
qed

lemma *add-diff-assoc-eint*: *z* $\leq$ *y* $\Longrightarrow$ *x* + (*y* − *z*) = *x* + *y* − (*z*::*eint*)

by(*cases x*)(*auto simp add*: *diff-eint-def split*: *eint.split*)

lemma *eint-ord-number* [*simp*]:
(*numeral m* :: *eint*) ≤ *numeral n* ⟷ (*numeral m* :: *nat*) ≤ *numeral n*
(*numeral m* :: *eint*) < *numeral n* ⟷ (*numeral m* :: *nat*) < *numeral n*
apply *simp*
by *simp*

lemma *infinity-ileE* [*elim*!]: ∞ ≤ *eint m* ⟹ *R*
by *simp*

lemma *infinity-ilessE* [*elim*!]: ∞ < *eint m* ⟹ *R*
by *simp*

lemma *imult-infinity*: (*0*::*eint*) < *n* ⟹ ∞ ∗ *n* = ∞
by (*simp add*: *zero-eint-def less-eint-def split*: *eint.splits*)

lemma *imult-infinity-right*: (*0*::*eint*) < *n* ⟹ *n* ∗ ∞ = ∞
by (*simp add*: *zero-eint-def less-eint-def split*: *eint.splits*)

lemma *min-eint-simps* [*simp*]:
min (*eint m*) (*eint n*) = *eint* (*min m n*)
min q (∞::*eint*) = *q*
min (∞::*eint*) *q* = *q*
by (*auto simp add*: *min-def*)

lemma *max-eint-simps* [*simp*]:
max (*eint m*) (*eint n*) = *eint* (*max m n*)
max q ∞ = (∞::*eint*)
max ∞ *q* = (∞::*eint*)
by (*simp-all add*: *max-def*)

lemma *eint-ile*: *n* ≤ *eint m* ⟹ ∃ *k*. *n* = *eint k*
by (*cases n*) *simp-all*

lemma *eint-iless*: *n* < *eint m* ⟹ ∃ *k*. *n* = *eint k*
by (*cases n*) *simp-all*

lemma *iadd-le-eint-iff*:
x + *y* ≤ *eint n* ⟷ (∃ *y' x'*. *x* = *eint x'* ∧ *y* = *eint y'* ∧ *x'* + *y'* ≤ *n*)
by(*cases x y rule*: *eint.exhaust*[*case-product eint.exhaust*]) *simp-all*

lemma *chain-incr*: ∀ *i*. ∃ *j*. *Y i* < *Y j* ⟹ ∃ *j*. *eint k* < *Y j*
proof−
assume *A*: ∀ *i*. ∃ *j*. *Y i* < *Y j*
then have ∀ *i*. ∃ *n*::*int*. *Y i* = *eint n*
by (*metis eint.exhaust eint-ord-simps*(*6*))
then obtain *i n* **where** *in-def*: *Y* (*i*::*'a*) = *eint n*
by *blast*

```
  show ∃j. eint k < Y j
  proof(rule ccontr)
    assume C: ¬(∃j. eint k < Y j)
    then have C':∀j. Y j ≤ eint k
      using le-less-linear
      by blast
    then have Y (i::'a) ≤ eint k
      by simp
    have ⋀m::nat. ∃j::'a. Y j ≥ eint (n + int m)
    proof− fix m
      show ∃j::'a. Y j ≥ eint (n + int m)
        apply(induction m)
         apply (metis in-def int-ops(1) order-refl plus-int-code(1))
      proof− fix m
        assume ∃j. eint (n + int m) ≤ Y j
        then obtain j where j-def: eint (n + int m) ≤ Y j
          by blast
        obtain j' where j'-def: Y j < Y j'
          using A by blast
        have eint (n + int (Suc m)) = eint (n + m + 1)
          by auto
        then have eint (n + int (Suc m)) ≤ Y j'
          using j-def j'-def  eint-ord-plus-one[of n + m Y j Y j']
          by presburger
        then show ∃j. eint (n + int (Suc m)) ≤ Y j
          by blast
      qed
    qed
    then show False
      by (metis A C ‹Y i ≤ eint k›   eint-ord-simps(1) in-def
          order.not-eq-order-implies-strict zle-iff-zadd)
  qed
qed

lemma eint-ord-Suc:
  assumes (x::eint) < y
  shows x + 1 < y + 1
  apply(cases y = ∞)
  using assms i1-ne-infinity sum-infinity-imp-summand-infinity
   apply fastforce
  by (metis add-mono-thms-linordered-semiring(3) assms eSuc-inject order-less-le)

lemma eSuc-ile-mono [simp]: (n::eint) + 1 ≤ m+ 1 ⟷ n ≤ m
  by (meson add-mono-thms-linordered-semiring(3) eint-ord-Suc linorder-not-le)

lemma eSuc-mono [simp]: (n::eint) + 1 < m+ 1 ⟷ n < m
  by (meson add-mono-thms-linordered-semiring(3) eint-ord-Suc linorder-not-le)

lemma ile-eSuc [simp]: (n::eint) ≤ n + 1
```

by (*metis add.right-neutral add-left-mono eint-1-iff*(*2*) *eint-ord-code*(*1*) *linear not-one-le-zero zero-eint-def*)

lemma *ileI1*: (*m*::*eint*) < *n* ⟹ *m* + *1* ≤ *n*
by (*metis eSuc-eint eint.exhaust eint-ex-split eint-iless eint-ord-Suc eint-ord-code*(*6*)

eint-ord-plus-one eint-ord-simps(*3*) *less-le-trans linear*)

lemma *Suc-ile-eq*: *eint* (*m* +*1*) ≤ *n* ⟷ *eint m* < *n*
by (*cases n*) *auto*

lemma *iless-Suc-eq* [*simp*]: *eint m* < *n* + *1* ⟷ *eint m* ≤ *n*
by (*metis Suc-ile-eq eSuc-eint eSuc-ile-mono*)

lemma *eSuc-max*: (*max* (*x*::*eint*) *y*) + *1* = *max* (*x*+*1*) (*y*+*1*)
by (*simp add*: *max-def*)

lemma *eSuc-Max*:
assumes *finite A A* ≠ ({}::*eint set*)
shows (*Max A*) + *1* = *Max* ((+)*1* ‘ *A*)
using *assms* **proof** *induction*
case (*insert x A*)
thus *?case*
using *Max-insert*[*of A x*] *Max-singleton*[*of x*] *add.commute*[*of 1*] *eSuc-max finite-imageI*
image-insert image-is-empty
by (*simp add*: *add.commute hom-Max-commute*)
qed *simp*

instantiation *eint* :: {*order-top*}
begin

definition *top-eint* :: *eint* **where** *top-eint* = ∞

instance
by *standard* (*simp add*: *top-eint-def*)

end

lemma *finite-eint-bounded*:
assumes *le-fin*: ⋀*y*. *y* ∈ *A* ⟹ *eint m* ≤ *y* ∧ *y* ≤ *eint n*
shows *finite A*
proof (*rule finite-subset*)
show *finite* (*eint* ‘ {*m*..*n*}) **by** *blast*
have *A* ⊆ {*eint m*..*eint n*} **using** *le-fin* **by** *fastforce*
also have ... ⊆ *eint* ‘ {*m*..*n*}
apply (*rule subsetI*)
subgoal for *x* **by** (*cases x*) *auto*
done

finally show $A \subseteq eint\ `\ \{m..n\}$ **.**
qed

14.8 Cancellation simprocs

lemma *add-diff-cancel-eint*[*simp*]: $x \neq \infty \Longrightarrow x + y - x = (y::eint)$
by (*metis add.commute add.right-neutral add-diff-assoc-eint idiff-self order-refl*)

lemma *eint-add-left-cancel*: $a + b = a + c \longleftrightarrow a = (\infty::eint) \vee b = c$
unfolding *plus-eint-def* **by** (*simp split*: *eint.split*)

lemma *eint-add-left-cancel-le*: $a + b \leq a + c \longleftrightarrow a = (\infty::eint) \vee b \leq c$
unfolding *plus-eint-def* **by** (*simp split*: *eint.split*)

lemma *eint-add-left-cancel-less*: $a + b < a + c \longleftrightarrow a \neq (\infty::eint) \wedge b < c$
unfolding *plus-eint-def* **by** (*simp split*: *eint.split*)

lemma *plus-eq-infty-iff-eint*: $(m::eint) + n = \infty \longleftrightarrow m{=}\infty \vee n{=}\infty$
using *eint-add-left-cancel* **by** *fastforce*

ML ‹
structure Cancel-Enat-Common =
struct
(∗ copied from src/HOL/Tools/nat-numeral-simprocs.ML ∗)
fun find-first-t - - [] = raise TERM(find-first-t, [])
| find-first-t past u (t::terms) =
if u aconv t then (rev past @ terms)
else find-first-t (t::past) u terms

*fun dest-summing (Const (**const-name**‹Groups.plus›, -) $ t $ u, ts) =*
dest-summing (t, dest-summing (u, ts))
| dest-summing (t, ts) = t :: ts

val mk-sum = Arith-Data.long-mk-sum
fun dest-sum t = dest-summing (t, [])
val find-first = find-first-t []
val trans-tac = Numeral-Simprocs.trans-tac
val norm-ss =
*simpset-of (put-simpset HOL-basic-ss **context***
addsimps @{thms ac-simps add-0-left add-0-right})
fun norm-tac ctxt = ALLGOALS (simp-tac (put-simpset norm-ss ctxt))
fun simplify-meta-eq ctxt cancel-th th =
Arith-Data.simplify-meta-eq [] ctxt
([th, cancel-th] MRS trans)
fun mk-eq (a, b) = HOLogic.mk-Trueprop (HOLogic.mk-eq (a, b))
end

structure Eq-Enat-Cancel = ExtractCommonTermFun
(open Cancel-Enat-Common

```
  val mk-bal = HOLogic.mk-eq
  val dest-bal = HOLogic.dest-bin const-name‹HOL.eq› typ‹eint›
  fun simp-conv - - = SOME @{thm eint-add-left-cancel}
)

structure Le-Enat-Cancel = ExtractCommonTermFun
(open Cancel-Enat-Common
  val mk-bal = HOLogic.mk-binrel const-name‹Orderings.less-eq›
  val dest-bal = HOLogic.dest-bin const-name‹Orderings.less-eq› typ‹eint›
  fun simp-conv - - = SOME @{thm eint-add-left-cancel-le}
)

structure Less-Enat-Cancel = ExtractCommonTermFun
(open Cancel-Enat-Common
  val mk-bal = HOLogic.mk-binrel const-name‹Orderings.less›
  val dest-bal = HOLogic.dest-bin const-name‹Orderings.less› typ‹eint›
  fun simp-conv - - = SOME @{thm eint-add-left-cancel-less}
)
›

simproc-setup eint-eq-cancel
  ((l::eint) + m = n | (l::eint) = m + n) =
  ‹fn phi => fn ctxt => fn ct => Eq-Enat-Cancel.proc ctxt (Thm.term-of ct)›

simproc-setup eint-le-cancel
  ((l::eint) + m ≤ n | (l::eint) ≤ m + n) =
  ‹fn phi => fn ctxt => fn ct => Le-Enat-Cancel.proc ctxt (Thm.term-of ct)›

simproc-setup eint-less-cancel
  ((l::eint) + m < n | (l::eint) < m + n) =
  ‹fn phi => fn ctxt => fn ct => Less-Enat-Cancel.proc ctxt (Thm.term-of ct)›
```

TODO: add regression tests for these simprocs

TODO: add simprocs for combining and cancelling numerals

14.9 Well-ordering

```
lemma less-eintE:
  [| n < eint m; !!k. n = eint k ==> k < m ==> P |] ==> P
by (induct n) auto

lemma less-infinityE:
  [| n < ∞; !!k. n = eint k ==> P |] ==> P
by (induct n) auto
```

14.10 Traditional theorem names

```
lemmas eint-defs = zero-eint-def one-eint-def
  plus-eint-def less-eq-eint-def less-eint-def
```

instantiation *eint* :: *uminus*
begin

definition
$-\ b = (case\ b\ of\ \infty \Rightarrow \infty \mid eint\ m \Rightarrow eint\ (-m))$

lemma *eint-uminus-eq*:
$(a::eint) + (-a) = a - a$
apply(*induction a*)
apply (*simp add*: *uminus-eint-def*)
by *simp*

instance..
end

15 Additional Lemmas (Useful for the Proof of Hensel's Lemma)

lemma *eint-mult-mono*:
assumes $(c::eint) > 0 \wedge c \neq \infty$
assumes $k > n$
shows $k*c > n*c$
using *assms* **apply**(*induction k*, *induction n*, *induction c*)
by(*auto simp add*: *zero-eint-def*)

lemma *eint-mult-mono'*:
assumes $(c::eint) \geq 0 \wedge c \neq \infty$
assumes $k > n$
shows $k*c \geq n*c$
apply(*cases c = 0*)
apply (*metis add.right-neutral assms*(*2*) *eint-add-left-cancel eint-ord-code*(*3*)
eint-ord-simps(*4*) *eq-iff less-le-trans mult.commute mult-eSuc-right'*
mult-one-right not-less times-eint-simps(*4*) *zero-eint-def*)
using *assms eint-mult-mono*
by (*simp add*: *le-less*)

lemma *eint-minus-le*:
assumes $(b::eint) < c$
shows $c - b > 0$
using *assms* **apply**(*induction b*, *induction c*)
by (*auto simp add*: *zero-eint-def*)

lemma *eint-nat-times*:
assumes $(c::eint) > 0$

shows (*Suc n*)∗(*c*::*eint*) > *0*
using *assms* **apply**(*induction c*)
apply (*simp add*: *zero-eint-def*)
by *simp*

lemma *eint-pos-times-is-pos*:
assumes (*c*::*eint*) > *0*
assumes *b* > *0*
shows *b*∗*c* > *0*
using *assms* **apply**(*induction c*, *induction b*)
by(*auto simp add*: *zero-eint-def imult-infinity-right*)

lemma *eint-nat-is-pos*:
eint (*Suc n*) > *0*
by (*simp add*: *zero-eint-def*)

lemma *eint-pow-int-is-pos*:
assumes *n* > *0*
shows *eint n* > *0*
using *assms* **by** (*simp add*: *zero-eint-def*)

lemma *eint-nat-times′*:
assumes (*c*::*eint*) ≥ *0*
shows (*Suc n*)∗*c* ≥ *0*
using *assms zero-eint-def* **by** *fastforce*

lemma *eint-pos-int-times-ge*:
assumes (*c*::*eint*) ≥ *0*
assumes *n* > *0*
shows *eint n* ∗ *c* ≥ *c*
using *assms* **apply**(*induction c*)
apply (*metis Extended-Int.ileI1 Groups.mult-ac*(*2*) *add-0-left eint-mult-mono′*
eint-pow-int-is-pos
mult-one-right nless-le not-less times-eint-simps(*4*))
apply *simp*
done

lemma *eint-pos-int-times-gt*:
assumes (*c*::*eint*) > *0*
assumes *c* ≠∞
assumes *n* > *1*
shows *eint n* ∗ *c* > *c*
using *assms eint-mult-mono*[*of c 1 eint n*]
by (*metis eint-ord-simps*(*2*) *mult-one-left one-eint-def*)

lemma *eint-add-cancel-fact*[*simp*]:
assumes (*c*::*eint*) ≠ ∞
shows *c* + (*b* − *c*) = *b*
using *assms* **apply**(*induction c*, *induction b*)

by *auto*

lemma *nat-mult-not-infty*[*simp*]:
assumes $c \neq \infty$
shows $(eint\ n) * c \neq \infty$
using *assms* **by** *auto*

lemma *eint-minus-distl*:
assumes $(b::eint) \neq d$
shows $b*c - d*c = (b-d)*c$
using *assms* **apply**(*induction c*, *induction b*, *induction d*)
apply (*metis add-diff-cancel-eint distrib-right eint.distinct*(*2*) *eint-add-cancel-fact nat-mult-not-infty*)
apply *simp*
apply *simp*
by (*simp add*: *mult.commute times-eint-def*)

lemma *eint-minus-distr*:
assumes $(b::eint) \neq d$
shows $c*(b - d) = c*b - c*d$
by (*metis assms eint-minus-distl mult.commute*)

lemma *eint-int-minus-distr*:
$(eint\ n)*c - (eint\ m)*c = eint\ (n - m) * c$
by (*metis add.right-neutral distrib-right eint-add-left-cancel eint-minus-distl idiff-eint-eint idiff-infinity idiff-self infinity-ne-i0 nat-mult-not-infty not-eint-eq times-eint-simps*(*4*))

lemma *eint-2-minus-1-mult*[*simp*]:
$2*(b::eint) - b = b$
proof −
have $\forall e.\ (\infty::eint) * e = \infty$
by (*simp add*: *times-eint-def*)
then show *?thesis*
by (*metis add-diff-cancel-eint idiff-infinity mult.commute mult-eSuc-right' mult-one-right one-add-one one-eint-def plus-eint-simps*(*1*))
qed

lemma *eint-minus-comm*:
$(d::eint) + b - c = d - c + b$
apply(*induction c*)
apply (*metis add.assoc add-diff-cancel-eint eint.distinct*(*2*) *eint-add-cancel-fact*)
apply(*induction d*)
apply (*metis distrib-left eint2-cases eint-minus-distl i1-ne-infinity idiff-infinity-right mult-one-left plus-eq-infty-iff-eint sum-infinity-imp-summand-infinity times-eint-simps*(*3*))
apply(*induction b*)
apply *simp*

by *simp*

lemma *ge-plus-pos-imp-gt*:
assumes (*c*::*eint*) ≠∞
assumes (*b*::*eint*) > *0*
assumes *d* ≥ *c* + *b*
shows *d* > *c*
using *assms* **apply**(*induction d*, *induction c*)
apply (*metis add.comm-neutral assms*(*2*) *eint-add-left-cancel-less less-le-trans*)
apply *blast*
by *simp*

lemma *eint-minus-ineq*:
assumes (*c*::*eint*) ≠∞
assumes *b* ≥ *d*
shows *b* − *c* ≥ *d* − *c*
by (*metis add-left-mono antisym assms*(*1*) *assms*(*2*) *eint-add-cancel-fact linear*)

lemma *eint-minus-ineq′*:
assumes (*c*::*eint*) ≠∞
assumes *b* ≥ *d*
assumes (*e*::*eint*) > *0*
assumes *e* ≠ ∞
shows *e*∗(*b* − *c*) ≥ *e*∗(*d* − *c*)
using *assms eint-minus-ineq*
by (*metis eint-mult-mono′ eq-iff less-le mult.commute*)

lemma *eint-minus-ineq′′*:
assumes (*c*::*eint*) ≠∞
assumes *b* ≥ *d*
assumes (*e*::*eint*) > *0*
assumes *e* ≠ ∞
shows *e*∗(*b* − *c*) ≥ *e*∗*d* − *e*∗*c*
using *assms eint-minus-ineq′*
proof −
have ∀ *e*. (*0*::*eint*) + *e* = *e*
by *simp*
then have *f1*: *e* ∗ *0* = *0*
by (*metis add-diff-cancel-eint assms*(*4*) *idiff-self mult-eSuc-right′ mult-one-right*)
have ∞ ≠ *c* ∗ *e*
using *assms*(*1*) *assms*(*4*) *eint-pos-times-is-pos* **by** *auto*
then show *?thesis*
using *f1* **by** (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *eint-minus-distl eint-minus-ineq′ idiff-self mult.commute*)
qed

lemma *eint-min-ineq*:
assumes (*b*::*eint*) ≥ *min c d*
assumes *c* > *e*

assumes $d > e$
shows $b > e$
by (*meson assms*(*1*) *assms*(*2*) *assms*(*3*) *less-le-trans min-le-iff-disj*)

lemma *eint-plus-times*:
assumes (d::*eint*) $\geq$ *0*
assumes (b::*eint*) $\geq$ c + (*eint* k)∗d
assumes $k \geq l$
shows $b \geq c + l*d$
proof−
have $k*d \geq l*d$
by (*smt* (*verit*) *assms*(*1*) *assms*(*3*) *eint-mult-mono*′ *eint-ord-simps*(*2*) *eq-iff times-eint-simps*(*4*))
thus *?thesis*
by (*meson add-mono-thms-linordered-semiring*(*2*) *assms*(*2*) *order-subst2*)
qed
end
theory *Padic-Construction*
imports *HOL*−*Number-Theory.Residues HOL*−*Algebra.RingHom HOL*−*Algebra.IntRing*
begin

type-synonym *padic-int* = *nat* ⇒ *int*

16 Inverse Limit Construction of the p-adic Integers

This section formalizes the standard construction of the p-adic integers as the inverse limit of the finite rings $\mathbb{Z}/p^n\mathbb{Z}$ along the residue maps $\mathbb{Z}/p^n\mathbb{Z} \mapsto \mathbb{Z}/p^n\mathbb{Z}$ defined by $x \mapsto x \mod p^m$ when $n \geq m$. This is exposited, for example, in section 7.6 of [3]. The other main route for formalization is to first define the p-adic absolute value $|\cdot|_p$ on the rational numbers, and then define the field $\mathbb{Q}_p$ of p-adic numbers as the completion of the rationals under this absolute value. One can then define the ring of p-adic integers $\mathbb{Z}_p$ as the unit ball in $\mathbb{Q}_p$ using the unique extension of $|\cdot|_p$. There exist advantages and disadvantages to both approaches. The primary advantage to the absolute value approach is that the construction can be done generically using existing libraries for completions of normed fields. There are difficulties associated with performing such a construction in Isabelle using existing HOL formalizations. The chief issue is that the tools in HOL-Analysis require that a metric space be a type. If one then wanted to construct the fields $\mathbb{Q}_p$ as metric spaces, one would have to circumvent the apparent dependence on the parameter p, as Isabelle does not support dependent types. A workaround to this proposed by José Manuel Rodríguez Caballero on the Isabelle mailing list is to define a typeclass for fields $\mathbb{Q}_p$ as the completions of the rational numbers with a non-Archimedean absolute value. By Ostrowski's Theorem,

any such absolute value must be a p-adic absolute value. We can recover the parameter p from a completion under one of these absolute values as the cardinality of the residue field.

Our approach uses HOL-Algebra, where algebraic structures are constructed as records which carry the data of the underlying carrier set plus other algebraic operations, and assumptions about these structures can be organized into locales. This approach is practical for abstract algebraic reasoning where definitions of structures which are dependent on object-level parameters are ubiquitous. Using this approach, we define $\mathbb{Z}_p$ directly as an inverse limit of rings, from which $\mathbb{Q}_p$ can later be defined as the field of fractions.

16.1 Canonical Projection Maps Between Residue Rings

definition *residue* :: *int* ⇒ *int* ⇒ *int* **where**
residue n m = *m mod n*

lemma *residue-is-hom-0*:
assumes *n* > *1*
shows *residue n* ∈ *ring-hom* $\mathcal{Z}$ (*residue-ring n*)
proof(*rule ring-hom-memI*)
have *R*: *residues n*
by (*simp add*: *assms residues-def*)
show ⋀*x*. *x* ∈ *carrier* $\mathcal{Z}$ ⟹ *residue n x* ∈ *carrier* (*residue-ring n*)
using *assms residue-def residues.mod-in-carrier residues-def* **by** *auto*
show ⋀*x y*. *x* ∈ *carrier* $\mathcal{Z}$ ⟹ *y* ∈ *carrier* $\mathcal{Z}$ ⟹
residue n (*x* ⊗$_{\mathcal{Z}}$ *y*) = *residue n x* ⊗$_{\text{residue-ring } n}$ *residue n y*
by (*simp add*: *R residue-def residues.mult-cong*)
show ⋀*x y*. *x* ∈ *carrier* $\mathcal{Z}$ ⟹
y ∈ *carrier* $\mathcal{Z}$ ⟹
residue n (*x* ⊕$_{\mathcal{Z}}$ *y*) = *residue n x* ⊕$_{\text{residue-ring } n}$ *residue n y*
by (*simp add*: *R residue-def residues.res-to-cong-simps*(*1*))
show *residue n* **1**$_{\mathcal{Z}}$ = **1**$_{\text{residue-ring } n}$
by (*simp add*: *R residue-def residues.res-to-cong-simps*(*4*))
qed

The residue map is a ring homomorphism from $\mathbb{Z}/m\mathbb{Z} \to \mathbb{Z}/n\mathbb{Z}$ when n divides m

lemma *residue-is-hom-1*:
assumes *n* > *1*
assumes *m* > *1*
assumes *n dvd m*
shows *residue n* ∈ *ring-hom* (*residue-ring m*) (*residue-ring n*)
proof(*rule ring-hom-memI*)
have *0*: *residues n*
by (*simp add*: *assms*(*1*) *residues-def*)
have *1*: *residues m*
by (*simp add*: *assms*(*2*) *residues-def*)

show $\bigwedge x.\ x \in carrier\ (residue\text{-}ring\ m) \Longrightarrow residue\ n\ x \in carrier\ (residue\text{-}ring\ n)$
using *assms*(*1*) *residue-def residue-ring-def* **by** *auto*
show $\bigwedge x\ y.\ x \in carrier\ (residue\text{-}ring\ m) \Longrightarrow y \in carrier\ (residue\text{-}ring\ m) \Longrightarrow residue\ n\ (x \otimes_{residue\text{-}ring\ m} y) = residue\ n\ x \otimes_{residue\text{-}ring\ n} residue\ n\ y$
using *0 1 assms* **by** (*metis mod-mod-cancel residue-def residues.mult-cong residues.res-mult-eq*)
show $\bigwedge x\ y.\ x \in carrier\ (residue\text{-}ring\ m) \Longrightarrow y \in carrier\ (residue\text{-}ring\ m) \Longrightarrow residue\ n\ (x \oplus_{residue\text{-}ring\ m} y) = residue\ n\ x \oplus_{residue\text{-}ring\ n} residue\ n\ y$
using *0 1 assms* **by** (*metis mod-mod-cancel residue-def residues.add-cong residues.res-add-eq*)

show $residue\ n\ \mathbf{1}_{residue\text{-}ring\ m} = \mathbf{1}_{residue\text{-}ring\ n}$
by (*simp add*: *assms*(*1*) *residue-def residue-ring-def*)
qed

lemma *residue-id*:
assumes $x \in carrier\ (residue\text{-}ring\ n)$
assumes $n \geq 0$
shows $residue\ n\ x = x$
proof(*cases n=0*)
case *True*
then show *?thesis*
by (*simp add*: *residue-def*)
next
case *False*
have *0*: $x \geq 0$
using *assms*(*1*) **by** (*simp add*: *residue-ring-def*)
have *1*: $x < n$
using *assms*(*1*) *residue-ring-def* **by** *auto*
have $x\ mod\ n = x$
using *0 1* **by** *simp*
then show *?thesis*
using *residue-def* **by** *auto*
qed

The residue map is a ring homomorphism from $\mathbb{Z}/p^n\mathbb{Z} \to \mathbb{Z}/p^m\mathbb{Z}$ when $n \geq m$:

lemma *residue-hom-p*:
assumes $(n::nat) \geq m$
assumes $m > 0$
assumes $prime\ (p::int)$
shows $residue\ (p\hat{}m) \in ring\text{-}hom\ (residue\text{-}ring\ (p\hat{}n))\ (residue\text{-}ring\ (p\hat{}m))$
proof(*rule residue-is-hom-1*)
show $1 < p\hat{}n$ **using** *assms*
using *prime-gt-1-int* **by** *auto*
show $1 < p\hat{}m$

by (*simp add*: *assms*(*2*) *assms*(*3*) *prime-gt-1-int*)
show $p \mathbin{\hat{}} m$ *dvd* $p \mathbin{\hat{}} n$ **using** *assms*(*1*)
by (*simp add*: *dvd-power-le*)
qed

16.2 Defining the Set of p-adic Integers

The set of p-adic integers is the set of all maps $f : \mathbb{N} \to \mathbb{Z}$ which maps $n \to \{0, ..., p^n - 1\}$ such that $fm \mod p^n = fn$ when $m \geq n$. A p-adic integer x consists of the data of a residue map $x \mapsto x \mod p^n$ which commutes with further reduction $\mod p^m$. This formalization is specialized to just the p-adics, but this definition would work essentially as-is for any family of rings and residue maps indexed by a partially ordered type.

definition *padic-set* :: *int* ⇒ *padic-int set* **where**
padic-set p = {*f*::*nat* ⇒ *int* .(∀ *m*::*nat*. (*f m*) ∈ *carrier* (*residue-ring* (*p*^*m*)))

∧(∀ (*n*::*nat*) (*m*::*nat*). *n* > *m* ⟶ *residue* (*p*^*m*) (*f n*) = (*f m*)) }

lemma *padic-set-res-closed*:
assumes *f* ∈ *padic-set p*
shows (*f m*) ∈ (*carrier* (*residue-ring* (*p*^*m*)))
using *assms padic-set-def* **by** *auto*

lemma *padic-set-res-coherent*:
assumes *f* ∈ *padic-set p*
assumes *n* ≥ *m*
assumes *prime p*
shows *residue* (*p*^*m*) (*f n*) = (*f m*)
proof(*cases n*=*m*)
case *True*
have (*f m*) ∈ *carrier* (*residue-ring* (*p*^*m*))
using *assms padic-set-res-closed* **by** *blast*
then have *residue* (*p*^*m*) (*f m*) = (*f m*)
by (*simp add*: *residue-def residue-ring-def*)
then show *?thesis*
using *True* **by** *blast*
next
case *False*
then show *?thesis*
using *assms*(*1*) *assms*(*2*) *padic-set-def* **by** *auto*
qed

A consequence of this formalization is that each p-adic number is trivially defined to take a value of 0 at 0:

lemma *padic-set-zero-res*:
assumes *prime p*
assumes *f* ∈ (*padic-set p*)

shows *f 0 = 0*
proof−
have *f 0 ∈ carrier (residue-ring 1)*
using *assms(1) padic-set-res-closed*
by (*metis assms(2) power-0*)
then show *?thesis*
using *residue-ring-def* **by** *simp*
qed

lemma *padic-set-memI*:
fixes *f* :: *padic-int*
assumes ⋀*m*. (*f m*) ∈ (*carrier* (*residue-ring* (*p*^*m*)))
assumes (⋀(*m*::*nat*) *n*. (*n* > *m* ⟹ (*residue* (*p*^*m*) (*f n*) = (*f m*))))
shows *f* ∈ *padic-set* (*p*::*int*)
by (*simp add*: *assms(1) assms(2) padic-set-def*)

lemma *padic-set-memI′*:
fixes *f* :: *padic-int*
assumes ⋀*m*. (*f m*) ∈ {*0*..<*p*^*m*}
assumes ⋀(*m*::*nat*) *n*. *n* > *m* ⟹ (*f n*) *mod p*^*m* = (*f m*)
shows *f* ∈ *padic-set* (*p*::*int*)
apply(*rule padic-set-memI*)
using *assms(1) residue-ring-def* **apply** *auto*[*1*]
by (*simp add*: *assms(2) residue-def*)

17 The standard operations on the p-adic integers

17.1 Addition

Addition and multiplication are defined componentwise on residue rings:

definition *padic-add* :: *int* ⇒ *padic-int* ⇒ *padic-int* ⇒ *padic-int*
where *padic-add p f g* ≡ (λ *n*. (*f n*) ⊕$_{(residue\text{-}ring\ (p\hat{}n))}$ (*g n*))

lemma *padic-add-res*:
(*padic-add p f g*) *n* = (*f n*) ⊕$_{(residue\text{-}ring\ (p\hat{}n))}$ (*g n*)
by (*simp add*: *padic-add-def*)

Definition of the p-adic additive unit:

definition *padic-zero* :: *int* ⇒ *padic-int* **where**
padic-zero p ≡ (λ*n*. *0*)

lemma *padic-zero-simp*:
padic-zero p n = **0**$_{residue\text{-}ring\ (p\hat{}n)}$
padic-zero p n = *0*
apply (*simp add*: *padic-zero-def residue-ring-def*)
using *padic-zero-def* **by** *auto*

lemma *padic-zero-in-padic-set*:

assumes $p > 0$
shows *padic-zero p* $\in$ *padic-set p*
apply(*rule padic-set-memI*)
by(*auto simp*: *assms padic-zero-def residue-def residue-ring-def*)

p-adic additive inverses:

definition *padic-a-inv* :: *int* $\Rightarrow$ *padic-int* $\Rightarrow$ *padic-int* **where**
padic-a-inv p f $\equiv \lambda$ *n*. $\ominus_{\textit{residue-ring}\ (p\hat{}n)}$ (*f n*)

lemma *padic-a-inv-simp*:
padic-a-inv p f n$\equiv \ominus_{\textit{residue-ring}\ (p\hat{}n)}$ (*f n*)
by (*simp add*: *padic-a-inv-def*)

lemma *padic-a-inv-simp'*:
assumes *prime p*
assumes *f* $\in$ *padic-set p*
assumes *n* >0
shows *padic-a-inv p f n* = (*if n*=*0 then 0 else* (− (*f n*)) *mod* (*p*$\hat{}$*n*))
proof−
have *residues* (*p*$\hat{}$*n*)
by (*simp add*: *assms*(*1*) *assms*(*3*) *prime-gt-1-int residues.intro*)
then show *?thesis*
using *residue-ring-def padic-a-inv-def residues.res-neg-eq*
by *auto*
qed

We show that *padic-set* is closed under additive inverses. Note that we have to treat the case of residues at 0 separately.

lemma *residue-1-prop*:
$\ominus_{\textit{residue-ring 1}}\ \mathbf{0}_{\textit{residue-ring 1}} = \ \mathbf{0}_{\textit{residue-ring 1}}$
proof−
let *?x* = $\mathbf{0}_{\textit{residue-ring 1}}$
let *?y* = $\ominus_{\textit{residue-ring 1}}\ \mathbf{0}_{\textit{residue-ring 1}}$
let *?G* = *add-monoid* (*residue-ring 1*)
have *P0*: *?x* $\oplus_{\textit{residue-ring 1}}$ *?x* = *?x*
by (*simp add*: *residue-ring-def*)
have *P1*: *?x* $\in$ *carrier* (*residue-ring 1*)
by (*simp add*: *residue-ring-def*)
have *?x* $\in$ *carrier ?G* $\wedge$ *?x* $\otimes_{?G}$ *?x* = $\mathbf{1}_{?G}$ $\wedge$ *?x* $\otimes_{?G}$ *?x* = $\mathbf{1}_{?G}$
using *P0 P1* **by** *auto*
then show *?thesis*
by (*simp add*: *m-inv-def a-inv-def residue-ring-def*)
qed

lemma *residue-1-zero*:
residue 1 n = *0*
by (*simp add*: *residue-def*)

lemma *padic-a-inv-in-padic-set*:

assumes $f \in$ *padic-set p*
assumes *prime* (*p*::*int*)
shows (*padic-a-inv p f*) $\in$ *padic-set p*
proof(*rule padic-set-memI*)
show $\bigwedge m$. *padic-a-inv p f m* $\in$ *carrier* (*residue-ring* (*p* ^ *m*))
proof−
fix *m*
show *padic-a-inv p f m* $\in$ *carrier* (*residue-ring* (*p* ^ *m*))
proof−
have *P0*: *padic-a-inv p f m* = $\ominus_{\textit{residue-ring}\ (p\hat{}m)}$ (*f m*)
using *padic-a-inv-def* **by** *simp*
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *assms*(*1*) *assms*(*2*) *cring.cring-simprules*(*3*) *neq0-conv*
one-less-power padic-set-res-closed padic-set-zero-res power-0 prime-gt-1-int residue-1-prop
residue-ring-def residues.cring residues.intro ring.simps(*1*))
qed
qed
show $\bigwedge m\ n$. $m < n \Longrightarrow$ *residue* (*p* ^ *m*) (*padic-a-inv p f n*) = *padic-a-inv p f m*
proof−
fix *m n*::*nat*
assume $m < n$
show *residue* (*p* ^ *m*) (*padic-a-inv p f n*) = *padic-a-inv p f m*
proof(*cases m*=*0*)
case *True*
then have *0*: *residue* (*p* ^ *m*) (*padic-a-inv p f n*) = *0* **using** *residue-1-zero*
by *simp*
have *f m* = *0*
using *assms True padic-set-def residue-ring-def padic-set-zero-res*
by *auto*
then have *1*: *padic-a-inv p f m* = *0* **using** *residue-1-prop assms*
by (*simp add*: *True padic-a-inv-def residue-ring-def*)
then show *?thesis* **using** *0 1*
by *simp*
next
case *False*
have *0*: *f n* $\in$ *carrier* (*residue-ring* (*p*^*n*))
using *assms*(*1*) *padic-set-res-closed* **by** *auto*
have *1*: *padic-a-inv p f n* = $\ominus_{\textit{residue-ring}\ (p\hat{}n)}$ (*f n*) **using** *padic-a-inv-def*
by *simp*
have *2*: *padic-a-inv p f m* = $\ominus_{\textit{residue-ring}\ (p\hat{}m)}$ (*f m*) **using** *False padic-a-inv-def*
by *simp*
have *3*: *residue* (*p* ^ *m*) $\in$ *ring-hom* (*residue-ring* (*p* ^ *n*)) (*residue-ring* (*p* ^ *m*))
using *residue-hom-p False* ‹*m* < *n*› *assms*(*2*) **by** *auto*
have *4*: *cring* (*residue-ring* (*p* ^ *n*))
using ‹*m* < *n*› *assms*(*2*) *prime-gt-1-int residues.cring residues.intro* **by**

auto
have *5*: *cring* (*residue-ring* (*p* ^ *m*))
using *False assms(2) prime-gt-1-int residues.cring residues.intro* **by** *auto*
have *ring-hom-cring* (*residue-ring* (*p* ^ *n*)) (*residue-ring* (*p* ^ *m*)) (*residue* (*p* ^ *m*))
using *3 4 5 UnivPoly.ring-hom-cringI* **by** *blast*
then show *?thesis* **using** *0 1 2 ring-hom-cring.hom-a-inv*
by (*metis* ‹*m* < *n*› *assms(1) assms(2) less-imp-le-nat padic-set-res-coherent*)

qed
qed
qed

17.2 Multiplication

definition *padic-mult* :: *int* ⇒ *padic-int* ⇒ *padic-int* ⇒ *padic-int*
where *padic-mult p f g* ≡ (λ *n*. (*f n*) ⊗$_{(residue\text{-}ring\ (p\hat{}n))}$ (*g n*))

lemma *padic-mult-res*:
(*padic-mult p f g*) *n* = (*f n*) ⊗$_{(residue\text{-}ring\ (p\hat{}n))}$ (*g n*)
by (*simp add*: *padic-mult-def*)

Definition of the p-adic multiplicative unit:

definition *padic-one* :: *int* ⇒ *padic-int* **where**
padic-one p ≡ (λ*n*.(*if n=0 then 0 else 1*))

lemma *padic-one-simp*:
assumes *n >0*
shows *padic-one p n* = **1**$_{residue\text{-}ring\ (p\hat{}n)}$
padic-one p n = 1
apply (*simp add*: *assms padic-one-def residue-ring-def*)
using *assms padic-one-def* **by** *auto*

lemma *padic-one-in-padic-set*:
assumes *prime p*
shows *padic-one p* ∈ *padic-set p*
apply(*rule padic-set-memI*)
by(*auto simp* : *assms padic-one-def prime-gt-1-int residue-def residue-ring-def*)

lemma *padic-simps*:
padic-zero p n = **0**$_{residue\text{-}ring\ (p\hat{}n)}$
padic-a-inv p f n ≡ ⊖$_{residue\text{-}ring\ (p\hat{}n)}$ (*f n*)
(*padic-mult p f g*) *n* = (*f n*) ⊗$_{(residue\text{-}ring\ (p\hat{}n))}$ (*g n*)
(*padic-add p f g*) *n* = (*f n*) ⊕$_{(residue\text{-}ring\ (p\hat{}n))}$ (*g n*)
n>0 ⟹*padic-one p n* = **1**$_{residue\text{-}ring\ (p\hat{}n)}$
apply (*simp add*: *padic-zero-simp*)
apply (*simp add*: *padic-a-inv-simp*)
apply (*simp add*: *padic-mult-def*)

apply (*simp add*: *padic-add-res*)
using *padic-one-simp* **by** *auto*

lemma *residue-1-mult*:
assumes $x \in$ *carrier* (*residue-ring 1*)
assumes $y \in$ *carrier* (*residue-ring 1*)
shows $x \otimes_{\textit{residue-ring 1}} y = 0$
by (*simp add*: *residue-ring-def*)

lemma *padic-mult-in-padic-set*:
assumes $f \in$ (*padic-set p*)
assumes $g \in$ (*padic-set p*)
assumes *prime p*
shows (*padic-mult p f g*)$\in$ (*padic-set p*)
proof(*rule padic-set-memI'*)
show $\bigwedge m$. *padic-mult p f g m* $\in \{0..<p \hat{} m\}$
unfolding *padic-mult-def*
using *assms residue-ring-def*
by (*simp add*: *prime-gt-0-int*)
show $\bigwedge m\ n.\ m < n \Longrightarrow$ *padic-mult p f g n mod p* $\hat{}\ m$ = *padic-mult p f g m*
proof−
fix *m n*::*nat*
assume *A*: $m < n$
then show *padic-mult p f g n mod p* $\hat{}\ m$ = *padic-mult p f g m*
proof(*cases m=0*)
case *True*
then show *?thesis*
by (*metis assms*(*1*) *assms*(*2*) *mod-by-1 padic-mult-def padic-set-res-closed power-0 residue-1-mult*)
next
case *False*
have *0*:*residue* (*p* $\hat{}$ *m*) $\in$ *ring-hom* (*residue-ring* (*p*$\hat{}$*n*)) (*residue-ring* (*p*$\hat{}$*m*))
using *A residue-hom-p assms False* **by** *auto*
have *1*:*f n* $\in$ *carrier* (*residue-ring* (*p*$\hat{}$*n*))
using *assms*(*1*) *padic-set-res-closed* **by** *auto*
have *2*:*g n* $\in$ *carrier* (*residue-ring* (*p*$\hat{}$*n*))
using *assms*(*2*) *padic-set-res-closed* **by** *auto*
have *3*: *residue* (*p*$\hat{}$*m*) (*f n* $\otimes_{\textit{residue-ring}\ (p\hat{}n)}$ *g n*)
= *f m* $\otimes_{\textit{residue-ring}\ (p\hat{}m)}$ *g m*
using *0 1 2 A assms*(*1*) *assms*(*2*) *assms*(*3*) *less-imp-le of-nat-power padic-set-res-coherent*
by (*simp add*: *assms*(*2*) *ring-hom-mult*)
then show *?thesis*
using *ring-hom-mult padic-simps*[*simp*] *residue-def*
by *auto*
qed
qed
qed

18 The p-adic Valuation

This section defines the integer-valued p-adic valuation. Maps 0 to -1 for now, otherwise is correct. We want the valuation to be integer-valued, but in practice we know it will always be positive. When we extend the valuation from the p-adic integers to the p-adic field we will have elements of negative valuation.

definition *padic-val* :: *int* ⇒ *padic-int* ⇒ *int* **where**
padic-val p f ≡ *if* (*f* = *padic-zero p*) *then* −*1 else int* (*LEAST k*::*nat*. (*f* (*Suc k*)) ≠ $\mathbf{0}_{\textit{residue-ring}\ (p\hat{}(\textit{Suc}\ k))}$)

Characterization of *padic_val* on nonzero elements

lemma *val-of-nonzero*:
assumes *f* ∈ *padic-set p*
assumes *f* ≠ *padic-zero p*
assumes *prime p*
shows *f* (*nat* (*padic-val p f*) + *1*) ≠ $\mathbf{0}_{\textit{residue-ring}\ (p\hat{}((\textit{nat}\ (\textit{padic-val}\ p\ f) + 1)))}$
f (*nat* (*padic-val p f*)) = $\mathbf{0}_{\textit{residue-ring}\ (p\hat{}((\textit{nat}\ (\textit{padic-val}\ p\ f))))}$
f (*nat* (*padic-val p f*) + *1*) ≠ *0*
f (*nat* (*padic-val p f*)) = *0*
proof−
let *?vf* = *padic-val p f*
have *0*: *?vf* =*int* (*LEAST k*::*nat*. (*f* (*Suc k*)) ≠ $\mathbf{0}_{\textit{residue-ring}\ (p\hat{}(\textit{Suc}\ k))}$)
using *assms*(*2*) *padic-val-def* **by** *auto*
have *1*: (∃ *k*::*nat*. (*f* (*Suc k*)) ≠ $\mathbf{0}_{\textit{residue-ring}\ (p\hat{}(\textit{Suc}\ k))}$)
proof−
obtain *k* **where** *1*: (*f k*) ≠ (*padic-zero p k*)
using *assms*(*2*) **by** (*meson ext*)
have *2*: *k* ≠ *0*
proof
assume *k*=*0*
then have *f k* = *0*
using *assms padic-set-zero-res* **by** *blast*
then show *False*
using *padic-zero-def 1* **by** *simp*
qed
then obtain *m* **where** *k* = *Suc m*
by (*meson lessI less-Suc-eq-0-disj*)
then have (*f* (*Suc m*)) ≠ $\mathbf{0}_{\textit{residue-ring}\ (p\hat{}(\textit{Suc}\ m))}$
using *1 padic-zero-simp* **by** *simp*
then show *?thesis*
by *auto*
qed
then have (*f* (*Suc* (*nat ?vf*))) ≠ $\mathbf{0}_{\textit{residue-ring}\ (p\hat{}(\textit{Suc}\ (\textit{nat}\ ?vf)))}$
using *0* **by** (*metis* (*mono-tags*, *lifting*) *LeastI-ex nat-int*)
then show *C0*: *f* (*nat* (*padic-val p f*) + *1*) ≠ $\mathbf{0}_{\textit{residue-ring}\ (p\hat{}((\textit{nat}\ (\textit{padic-val}\ p\ f) + 1)))}$

```
    using 0 1 by simp
  show C1: f (nat (padic-val p f)) =  0_{residue-ring (p^(nat (padic-val p f)))}
  proof(cases (padic-val p f) = 0)
    case True
    then show ?thesis
      using assms(1) assms(3) padic-set-zero-res residue-ring-def by auto
  next
    case False
    have ¬ f (nat (padic-val p f)) ≠ 0_{residue-ring (p ^ nat (padic-val p f))}
    proof
      assume f (nat (padic-val p f)) ≠ 0_{residue-ring (p ^ nat (padic-val p f))}
      obtain k where  (Suc k) = (nat (padic-val p f)) using False
        using 0 gr0-conv-Suc by auto
     then have ?vf ≠ int (LEAST k::nat. (f (Suc k)) ≠ 0_{residue-ring (p^(Suc k))})
        using False by (metis (mono-tags, lifting) Least-le
          ‹f (nat (padic-val p f)) ≠ 0_{residue-ring (p ^ nat (padic-val p f))}›
            add-le-same-cancel2 nat-int not-one-le-zero plus-1-eq-Suc)
      then show False  using 0 by blast
    qed
    then show f (nat (padic-val p f)) = 0_{residue-ring (p ^ nat (padic-val p f))} by
auto
  qed
  show  f (nat (padic-val p f) + 1) ≠ 0
    using C0  residue-ring-def
    by auto
  show  f (nat (padic-val p f)) =  0
    by (simp add: C1 residue-ring-def)
qed
```

If $x \mod p^{n+1} \neq 0$, then $n \geq valx$.

```
lemma below-val-zero:
  assumes prime p
  assumes x ∈ (padic-set p)
  assumes x (Suc n) ≠  0_{residue-ring (p^(Suc n))}
  shows  int n ≥ (padic-val p x )
proof(cases x = padic-zero p)
  case True
  then show  ?thesis
    using assms(3) padic-zero-simp by blast
next
  case False
 then have padic-val p x = int (LEAST k::nat. x (Suc k) ≠ 0_{residue-ring (p ^ Suc k)})
    using padic-val-def by auto
  then show of-nat n ≥ (padic-val p x )
    by (metis (mono-tags, lifting) Least-le assms(3) nat-int nat-le-iff)
qed
```

If $n < valx$ then $x \mod p^n = 0$:

lemma *zero-below-val*:
assumes *prime p*
assumes $x \in$ *padic-set p*
assumes $n \leq$ *padic-val p x*
shows $x\ n = \mathbf{0}_{\textit{residue-ring}\ (p\hat{}n)}$
$x\ n = 0$
proof−
show $x\ n = \mathbf{0}_{\textit{residue-ring}\ (p\ \hat{}\ n)}$
proof(*cases n=0*)
case *True*
then have $x\ 0 \in$*carrier* (*residue-ring* $(p\hat{}0)$)
using *assms*(*2*) *padic-set-res-closed* **by** *blast*
then show *?thesis*
by (*simp add*: *True residue-ring-def*)
next
case *False*
show *?thesis*
proof(*cases x = padic-zero p*)
case *True*
then show *?thesis*
by (*simp add*: *padic-zero-simp*)
next
case *F*: *False*
then have *A*: *padic-val p x* = *int* (*LEAST k*::*nat*. $(x\ (Suc\ k)) \neq \mathbf{0}_{\textit{residue-ring}\ (p\hat{}(Suc\ k))}$)
using *padic-val-def* **by** *auto*
have $\neg\ (x\ n) \neq \mathbf{0}_{\textit{residue-ring}\ (p\hat{}n)}$
proof
assume $(x\ n) \neq \mathbf{0}_{\textit{residue-ring}\ (p\hat{}n)}$
obtain *k* **where** *n* = *Suc k*
using *False old.nat.exhaust* **by** *auto*
then have $k \geq$ *padic-val p x* **using** *A*
using ‹$x\ n \neq \mathbf{0}_{\textit{residue-ring}\ (p\ \hat{}\ n)}$› *assms*(*1*) *assms*(*2*) *below-val-zero* **by**
blast
then have *n* > *padic-val p x*
using ‹*n* = *Suc k*› **by** *linarith*
then show *False* **using** *assms*(*3*)
by *linarith*
qed
then show *?thesis*
by *simp*
qed
qed
show *x n* = *0*
by (*simp add*: ‹$x\ n = \mathbf{0}_{\textit{residue-ring}\ (p\ \hat{}\ n)}$› *residue-ring-def*)
qed

Zero is the only element with valuation equal to −1:

lemma *val-zero*:

```
 assumes P: f ∈ (padic-set p)
 shows padic-val p f = −1 ⟷ (f = (padic-zero p))
proof
  show padic-val p f = −1 ⟹ (f = (padic-zero p))
  proof
    assume A:padic-val p f = −1
    fix k
    show f k = padic-zero p k
    proof−
      have  f k ≠ padic-zero p k ⟹ False
      proof−
        assume A0:  f k ≠ padic-zero p k
        have False
        proof−
          have f 0 ∈ carrier (residue-ring 1) using P padic-set-def
            by (metis (no-types, lifting) mem-Collect-eq power-0)
          then have f 0 = 0_residue-ring (p^0)
            by (simp add: residue-ring-def)
          then have k>0
            using A0 gr0I padic-zero-def
            by (metis padic-zero-simp)
          then have (LEAST k. 0 < k ∧ f (Suc k) ≠ padic-zero p (Suc k)) ≥0
            by simp
          then have padic-val p f ≥0
            using A0 padic-val-def by auto
          then show ?thesis  using A0 by (simp add: A)
        qed
        then show ?thesis by blast
      qed
      then show ?thesis
        by blast
    qed
  qed
  assume B: f = padic-zero p
  then show padic-val p f = −1
  using padic-val-def by simp
qed
```

The valuation turns multiplication into integer addition on nonzero elements. Note that this i the first instance where we need to explicity use the fact that p is a prime.

```
lemma val-prod:
  assumes prime p
  assumes f ∈ (padic-set p)
  assumes g ∈ (padic-set p)
  assumes f ≠ padic-zero p
  assumes g ≠ padic-zero p
  shows padic-val p (padic-mult p f g) = padic-val p f + padic-val p g
proof−
```

let *?vp* = *padic-val p* (*padic-mult p f g*)
let *?vf* = *padic-val p f*
let *?vg* = *padic-val p g*
have *0*: *f* (*nat ?vf* + *1*) ≠ $\mathbf{0}_{\textit{residue-ring}\ (p\hat{}(\textit{nat}\ ?vf + 1))}$
using *assms*(*2*) *assms*(*4*) *val-of-nonzero* *assms*(*1*) **by** *blast*
have *1*: *g* (*nat ?vg* + *1*) ≠ $\mathbf{0}_{\textit{residue-ring}\ (p\hat{}(\textit{nat}\ ?vg + 1))}$
using *assms*(*3*) *assms*(*5*) *val-of-nonzero assms*(*1*) **by** *blast*
have *2*: *f* (*nat ?vf*) = $\mathbf{0}_{\textit{residue-ring}\ (p\hat{}(\textit{nat}\ ?vf))}$
using *assms*(*1*) *assms*(*2*) *assms*(*4*) *val-of-nonzero*(*2*) **by** *blast*
have *3*: *g* (*nat ?vg*) = $\mathbf{0}_{\textit{residue-ring}\ (p\hat{}(\textit{nat}\ ?vg))}$
using *assms*(*1*) *assms*(*3*) *assms*(*5*) *val-of-nonzero*(*2*) **by** *blast*
let *?nm* = ((*padic-mult p f g*) (*Suc* (*nat* (*?vf* + *?vg*))))
let *?n* = (*f* (*Suc* (*nat* (*?vf* + *?vg*))))
let *?m* = (*g* (*Suc* (*nat* (*?vf* + *?vg*))))
have *A*: *?nm* = *?n* $\otimes_{\textit{residue-ring}\ (p\hat{}((\textit{Suc}\ (\textit{nat}\ (?vf + ?vg)))))}$ *?m*
using *padic-mult-def* **by** *simp*
have *5*: *f* (*nat ?vf* + *1*) = *residue* (*p*^(*nat ?vf* + *1*)) *?n*
proof−
have (*Suc* (*nat* (*?vf* + *?vg*))) ≥ (*nat ?vf* + *1*)
by (*simp add*: *assms*(*5*) *padic-val-def*)
then have *f* (*nat ?vf* + *1*) = *residue* (*p*^(*nat ?vf* + *1*)) (*f* (*Suc* (*nat* (*?vf* + *?vg*))))
using *assms*(*1*) *assms*(*2*) *padic-set-res-coherent* **by** *presburger*
then show *?thesis* **by** *auto*
qed
have *6*: *f* (*nat ?vf*) = *residue* (*p*^(*nat ?vf*)) *?n*
using *add.commute assms*(*1*) *assms*(*2*) *assms*(*5*) *int-nat-eq nat-int*
nat-le-iff not-less-eq-eq padic-set-res-coherent padic-val-def plus-1-eq-Suc **by** *auto*
have *7*: *g* (*nat ?vg* + *1*) = *residue* (*p*^(*nat ?vg* + *1*)) *?m*
proof−
have (*Suc* (*nat* (*?vf* + *?vg*))) ≥ (*nat ?vg* + *1*)
by (*simp add*: *assms*(*4*) *padic-val-def*)
then have *g* (*nat ?vg* + *1*) = *residue* (*p*^(*nat ?vg* + *1*)) (*g* (*Suc* (*nat* (*?vf* + *?vg*))))
using *assms*(*1*) *assms*(*3*) *padic-set-res-coherent* **by** *presburger*
then show *?thesis* **by** *auto*
qed
have *8*: *g* (*nat ?vg*) = *residue* (*p*^(*nat ?vg*)) *?m*
proof−
have (*Suc* (*nat* (*?vf* + *?vg*))) ≥ (*nat ?vg*)
by (*simp add*: *assms*(*4*) *padic-val-def*)
then have *g* (*nat ?vg*) = *residue* (*p*^(*nat ?vg*)) (*g* (*Suc* (*nat* (*?vf* + *?vg*))))
using *assms*(*1*) *assms*(*3*) *padic-set-res-coherent* **by** *presburger*
then show *?thesis* **by** *auto*
qed
have *9*: *f* (*nat ?vf*) = *0*
by (*simp add*: *2 residue-ring-def*)

have *10*: *g* (*nat ?vg*) = *0*
by (*simp add*: *3 residue-ring-def*)
have *11*: *f* (*nat ?vf* + *1*) ≠ *0*
using *0 residue-ring-def* **by** *auto*
have *12*: *g* (*nat ?vg* + *1*) ≠*0*
using *1 residue-ring-def* **by** *auto*
have *13*:∃ *i*. *?n* = *i*∗*p*^(*nat ?vf*) ∧ ¬ *p dvd* (*nat i*)
proof−
have *residue* (*p*^(*nat ?vf*)) (*?n*) = *f* (*nat ?vf*)
by (*simp add*: *6*)
then have *P0*: *residue* (*p*^(*nat ?vf*)) (*?n*) = *0*
using *9* **by** *linarith*
have *residue* (*p*^(*nat ?vf* + *1*)) (*?n*) = *f* (*nat ?vf* + *1*)
using *5* **by** *linarith*
then have *P1*: *residue* (*p*^(*nat ?vf* + *1*)) (*?n*) ≠ *0*
using *11* **by** *linarith*
have *P2*: *?n mod* (*p*^(*nat ?vf*)) = *0*
using *P0 residue-def* **by** *auto*
have *P3*: *?n mod* (*p*^(*nat ?vf* + *1*)) ≠ *0*
using *P1 residue-def* **by** *auto*
have *p*^(*nat ?vf*) *dvd ?n*
using *P2* **by** *auto*
then obtain *i* **where** *A0*:*?n* = *i*∗(*p*^(*nat ?vf*))
by *fastforce*
have *?n* ∈ *carrier* (*residue-ring* (*p*^(*Suc* (*nat* (*?vf* + *?vg*)))))
using *assms*(*2*) *padic-set-res-closed* **by** *blast*
then have *?n* ≥*0*
by (*simp add*: *residue-ring-def*)
then have *NN*:*i* ≥ *0*
proof−
have *S0*:*?n* ≥*0*
using ‹*0* ≤ *f* (*Suc* (*nat* (*padic-val p f* + *padic-val p g*)))› **by** *blast*
have *S1*:(*p*^(*nat ?vf*)) > *0*
using *assms*(*1*) *prime-gt-0-int zero-less-power* **by** *blast*
have ¬ *i*<*0*
proof
assume *i* < *0*
then have *?n* < *0*
using *S1 A0* **by** (*metis mult.commute times-int-code*(*1*) *zmult-zless-mono2*)
then show *False*
using *S0* **by** *linarith*
qed
then show *?thesis* **by** *auto*
qed
have *A1*: ¬ *p dvd* (*nat i*)
proof
assume *p dvd nat i*
then obtain *j* **where** *nat i* = *j*∗*p*
by *fastforce*

```
    then have ?n = j*p*(p^(nat ?vf)) using A0 NN
      by simp
    then show False
      using P3 by auto
  qed
  then show ?thesis
    using A0 by blast
qed
have 14:∃ i. ?m = i*p^(nat ?vg) ∧ ¬ p dvd (nat i)
proof−
  have  residue (p^(nat ?vg)) (?m) = g (nat ?vg)
    by (simp add: 8)
  then have P0: residue (p^(nat ?vg)) (?m) = 0
    using 10 by linarith
  have residue (p^(nat ?vg + 1)) (?m) = g (nat ?vg + 1)
    using 7 by auto
  then have P1: residue (p^(nat ?vg + 1)) (?m) ≠ 0
    using 12 by linarith
  have P2: ?m mod (p^(nat ?vg)) = 0
    using P0 residue-def by auto
  have P3: ?m mod (p^(nat ?vg + 1)) ≠  0
    using P1 residue-def by auto
  have p^(nat ?vg) dvd ?m
    using P2 by auto
  then obtain i where A0:?m = i*(p^(nat ?vg))
    by fastforce
  have ?m ∈ carrier (residue-ring (p^(Suc (nat (?vf + ?vg)))))
    using assms(3) padic-set-res-closed by blast
  then have S0: ?m ≥0
    by (simp add: residue-ring-def)
  then have NN:i ≥ 0
    using 0 assms(1) prime-gt-0-int[of p] zero-le-mult-iff zero-less-power[of p]
    by (metis A0 linorder-not-less)
  have A1: ¬ p dvd (nat i)
  proof
    assume p dvd nat i
    then obtain j where nat i = j*p
      by fastforce
    then have ?m = j*p*(p^(nat ?vg)) using A0 NN
      by (metis int-nat-eq )
    then show False
      using P3 by auto
  qed
  then show ?thesis
    by (metis (no-types, lifting) A0)
qed
obtain i where I:?n = i*p^(nat ?vf) ∧ ¬ p dvd (nat i)
  using 13 by blast
obtain j where J:?m = j*p^(nat ?vg) ∧ ¬ p dvd (nat j)
```

```
    using 14 by blast
  let ?i = (p^(Suc (nat (?vf + ?vg))))
  have P:?nm mod ?i = ?n*?m mod ?i
  proof−
    have P1:?nm = (?n ⊗residue-ring ?i  ?m)
      using A by simp
   have P2:(?n ⊗residue-ring ?i  ?m) = (residue ?i (?n)) ⊗residue-ring ?i  (residue
?i (?m))
    using assms(1) assms(2) assms(3) padic-set-res-closed prime-ge-0-int residue-id
by presburger
    then have P3:(?n ⊗residue-ring ?i  ?m) = (residue ?i (?n*?m))
      by (metis monoid.simps(1) residue-def residue-ring-def)
    then show ?thesis
      by (simp add: P1 residue-def)
  qed
  then have 15: ?nm mod ?i =  i*j*p^((nat ?vf) +(nat ?vg)) mod ?i
    by (simp add: I J mult.assoc mult.left-commute power-add)
  have 16: ¬ p dvd (i*j) using 13 14
    using I J assms(1) prime-dvd-mult-iff
    by (metis dvd-0-right int-nat-eq)
  have 17: ((nat ?vf) +(nat ?vg)) < (Suc (nat (?vf + ?vg)))
    by (simp add: assms(4) assms(5) nat-add-distrib padic-val-def)
  have 18:?nm mod ?i ≠0
  proof−
    have A0:¬  p^((Suc (nat (?vf + ?vg)))) dvd p^((nat ?vf) +(nat ?vg))
      using 17
     by (metis 16 assms(1) dvd-power-iff dvd-trans less-int-code(1) linorder-not-less
one-dvd prime-gt-0-int)
    then have A1: p^((nat ?vf) +(nat ?vg)) mod ?i ≠ 0
      using dvd-eq-mod-eq-0
      by auto
    have ¬  p^((Suc (nat (?vf + ?vg)))) dvd i*j*p^((nat ?vf) +(nat ?vg))
      using 16 A0 assms(1) assms(4) assms(5) nat-int-add padic-val-def by auto

    then show ?thesis
      using 15 by force
  qed
  have 19: (?nm mod ?i ) mod (p^(nat ?vf + nat ?vg)) = i*j*p^((nat ?vf) +(nat
?vg)) mod (p^(nat ?vf + nat ?vg))
    using 15 by (simp add: assms(4) assms(5) nat-add-distrib padic-val-def)
  have 20: ?nm mod (p^(nat ?vf + nat ?vg)) = 0
  proof−
    have (?nm mod ?i ) mod (p^(nat ?vf + nat ?vg)) = 0
      using 19
      by simp
    then show ?thesis
      using 17 assms(1) int-nat-eq mod-mod-cancel[of p^(nat ?vf + nat ?vg) ?i]
          mod-pos-pos-trivial
      by (metis le-imp-power-dvd less-imp-le-nat)
```

```
  qed
  have 21: (padic-mult p f g) ≠ padic-zero p
  proof
    assume (padic-mult p f g) =  padic-zero p
     then have (padic-mult p f g) (Suc (nat (padic-val p f + padic-val p g))) =
padic-zero p (Suc (nat (padic-val p f + padic-val p g)))
      by simp
    then have ?nm  = (padic-zero p (Suc (nat (padic-val p f + padic-val p g))))
      by blast
    then have ?nm = 0
      by (simp add: padic-zero-def)
    then show False
      using 18 by auto
  qed
  have 22: (padic-mult p f g)∈ (padic-set p)
    using assms(1) assms(2) assms(3) padic-mult-in-padic-set by blast
  have 23: ⋀ j. j < Suc (nat (padic-val p f + padic-val p g)) ⟹ (padic-mult p f
g) j = 0_residue-ring (p^j)
  proof−
    fix k
    let ?j = Suc (nat (padic-val p f + padic-val p g))
    assume P: k < ?j
    show (padic-mult p f g) k = 0_residue-ring (p^k)
      proof−
     have P0: (padic-mult p f g) (nat ?vf + nat ?vg) = 0_residue-ring (p^(nat ?vf + nat ?vg))
        proof−
          let ?k = (nat ?vf + nat ?vg)
          have ((padic-mult p f g) ?k) = residue (p^?k) ((padic-mult p f g) ?k)
          using P 22 padic-set-res-coherent by (simp add: assms(1) prime-gt-0-nat)
          then have ((padic-mult p f g) ?k) = residue (p^?k) ?nm
            using 17 22 assms(1) padic-set-res-coherent by fastforce
          then have ((padic-mult p f g) ?k) = residue (p^?k) ?nm
            by (simp add: residue-def)
          then have ((padic-mult p f g) ?k) = residue (p^?k) 0
            using 20 residue-def by auto
          then show ?thesis
            by (simp add: residue-def residue-ring-def)
        qed
      then show ?thesis
      proof(cases k = (nat ?vf + nat ?vg))
      case True then show ?thesis
        using P0 by blast
    next
      case B: False
      then show ?thesis
      proof(cases k=0)
        case True
        then show ?thesis
          using 22 assms(1) padic-set-zero-res residue-ring-def by auto
```

```
      next
        case C: False
        then have ((padic-mult p f g) k) = residue (p^k) ((padic-mult p f g) (nat
?vf + nat ?vg))
          using B P 22 padic-set-res-coherent by (simp add: assms(1) assms(4)
assms(5) padic-val-def prime-gt-0-nat)
      then have S: ((padic-mult p f g) k) = residue (p^k) 0_residue-ring (p^((nat ?vf + nat ?vg)))

          by (simp add: P0)
         have residue (p^k) ∈ ring-hom (residue-ring (p^((nat ?vf + nat ?vg))))
(residue-ring (p^k))
          using B P C residue-hom-p
       using assms(1) assms(4) assms(5) less-Suc0 nat-int not-less-eq of-nat-power
padic-val-def prime-nat-int-transfer by auto
        then show ?thesis using S
          using P0 padic-zero-def padic-zero-simp residue-def by auto
      qed
    qed
  qed
qed
 have 24: (padic-mult p f g) (Suc (nat ?vf + nat ?vg)) ≠ 0_residue-ring ((p ^ Suc (nat (padic-val p f + padic-val p

   by (metis (no-types, lifting) 18 A P assms(4) assms(5) monoid.simps(1) nat-int
nat-int-add padic-val-def residue-ring-def ring.simps(1))
  have 25: padic-val p (padic-mult p f g) = int (LEAST k::nat. ((padic-mult p f g)
(Suc k)) ≠ 0_residue-ring (p^(Suc k)))
    using padic-val-def 21 by auto
  have 26:(nat (padic-val p f + padic-val p g)) ∈ {k::nat. ((padic-mult p f g) (Suc
k)) ≠ 0_residue-ring (p^(Suc k))} using 24
    using 18 assms(1) prime-gt-0-nat
   by (metis (mono-tags, lifting) mem-Collect-eq mod-0 residue-ring-def ring.simps(1))
  have 27: ⋀ j. j < (nat (padic-val p f + padic-val p g)) ⟹
     j ∉ {k::nat. ((padic-mult p f g) (Suc k)) ≠ 0_residue-ring (p^(Suc k))}
    by (simp add: 23)
  have (nat (padic-val p f + padic-val p g)) = (LEAST k::nat. ((padic-mult p f g)
(Suc k)) ≠ 0_residue-ring (p^(Suc k)))
  proof−
   obtain P where C0: P= (λ k. ((padic-mult p f g) (Suc k)) ≠ 0_residue-ring (p^(Suc k)))

     by simp
    obtain x where C1: x = (nat (padic-val p f + padic-val p g))
     by blast
    have C2: P x
     using 26 C0 C1  by blast
    have C3:⋀ j. j< x ⟹ ¬ P j
     using C0 C1 by (simp add: 23)
    have C4: ⋀ j. P j ⟹ x ≤j
     using C3 le-less-linear by blast
    have x = (LEAST k. P k)
```

using *C2 C4 Least-equality* **by** *auto*
then show *?thesis* **using** *C0 C1* **by** *auto*
qed
then have *padic-val p* (*padic-mult p f g*) = (*nat* (*padic-val p f* + *padic-val p g*))
using *25* **by** *linarith*
then show *?thesis*
by (*simp add*: *assms*(*4*) *assms*(*5*) *padic-val-def*)

qed

19 Defining the Ring of p-adic Integers:

definition *padic-int* :: *int* ⇒ *padic-int ring*
where *padic-int p* ≡ ⦇*carrier* = (*padic-set p*),
Group.monoid.mult = (*padic-mult p*), *one* = (*padic-one p*),
zero = (*padic-zero p*), *add* = (*padic-add p*)⦈

lemma *padic-int-simps*:
$\mathbf{1}_{padic\text{-}int\ p}$ = *padic-one p*
$\mathbf{0}_{padic\text{-}int\ p}$ = *padic-zero p*
$(\oplus_{padic\text{-}int\ p})$ = *padic-add p*
$(\otimes_{padic\text{-}int\ p})$ = *padic-mult p*
carrier (*padic-int p*) = *padic-set p*
unfolding *padic-int-def* **by** *auto*

lemma *residues-n*:
assumes *n* ≠ *0*
assumes *prime p*
shows *residues* (*p*^*n*)
proof
have *p* > *1* **using** *assms*(*2*)
using *prime-gt-1-int* **by** *auto*
then show *1* < *p* ^ *n*
using *assms*(*1*) **by** *auto*
qed

p-adic multiplication is associative

lemma *padic-mult-assoc*:
assumes *prime p*
shows ⋀*x y z*.
x ∈ *carrier* (*padic-int p*) ⟹
y ∈ *carrier* (*padic-int p*) ⟹
z ∈ *carrier* (*padic-int p*) ⟹
$x \otimes_{padic\text{-}int\ p} y \otimes_{padic\text{-}int\ p} z = x \otimes_{padic\text{-}int\ p} (y \otimes_{padic\text{-}int\ p} z)$
proof−
fix *x y z*
assume *Ax*: *x* ∈ *carrier* (*padic-int p*)
assume *Ay*: *y* ∈ *carrier* (*padic-int p*)

```
  assume Az:  z ∈ carrier (padic-int p)
  show x ⊗padic-int p y ⊗padic-int p z = x ⊗padic-int p (y ⊗padic-int p z)
  proof
    fix n
    show ((x ⊗padic-int p y) ⊗padic-int p z) n = (x ⊗padic-int p (y ⊗padic-int p z)) n
    proof(cases n=0)
      case True
      then show ?thesis using padic-int-simps
        by (metis Ax Ay Az assms padic-mult-in-padic-set padic-set-zero-res)
    next
      case False
      then have residues (p^n)
        by (simp add: assms residues-n)
      then show ?thesis
         using residues.cring padic-set-res-closed padic-mult-in-padic-set Ax Ay Az padic-mult-res
        by (simp add: cring.cring-simprules(11) padic-int-def)
    qed
  qed
qed
```

The p-adic integers are closed under addition:

```
lemma padic-add-closed:
  assumes prime p
  shows  ⋀x y.
         x ∈ carrier (padic-int p) ⟹
         y ∈ carrier (padic-int p) ⟹
         x ⊕(padic-int p) y ∈ carrier (padic-int p)
proof
  fix x::padic-int
  fix y::padic-int
  assume Px: x ∈carrier (padic-int p)
  assume Py: y ∈carrier (padic-int p)
  show x ⊕(padic-int p) y ∈ carrier (padic-int p)
  proof−
    let ?f = x ⊕(padic-int p) y
    have 0: (∀(m::nat). (?f m) ∈ (carrier (residue-ring (p^m))))
    proof fix m
      have A1 : ?f m = (x m) ⊕(residue-ring (p^m)) (y m)
        by (simp add: padic-int-def padic-add-def)
      have A2: (x m) ∈(carrier (residue-ring (p^m)))
        using Px by (simp add: padic-int-def padic-set-def)
      have A3: (y m) ∈(carrier (residue-ring (p^m)))
        using Py by (simp add: padic-int-def padic-set-def)
      then show (?f m) ∈ (carrier (residue-ring (p^m)))
        using A1 assms of-nat-0-less-iff prime-gt-0-nat residue-ring-def by force
    qed
    have 1: (∀(n::nat) (m::nat). (n > m ⟶ (residue (p^m) (?f n) = (?f m))))
```

proof
fix n::nat
show $(\forall (m::nat).\ (n > m \longrightarrow (residue\ (p\hat{}m)\ (?f\ n) = (?f\ m))))$
proof
fix m::nat
show $(n > m \longrightarrow (residue\ (p\hat{}m)\ (?f\ n) = (?f\ m)))$
proof
assume A: $m < n$
show $(residue\ (p\hat{}m)\ (?f\ n) = (?f\ m))$
proof$(cases\ m = 0)$
case $True$
then have $A0$: $(residue\ (p\hat{}m)\ (?f\ n)) = 0$
by $(simp\ add$: $residue\text{-}1\text{-}zero)$
have $A1$: $?f\ m = 0$ **using** $True$
by $(simp\ add$: $padic\text{-}add\text{-}res\ padic\text{-}int\text{-}simps(3)\ residue\text{-}ring\text{-}def)$
then show $?thesis$
using $A0$ **by** $linarith$
next
case $False$
then have $m \neq 0$ **using** A **by** $linarith$
have D: $p\hat{}n\ mod\ p\hat{}m = 0$ **using** A
by $(simp\ add$: $le\text{-}imp\text{-}power\text{-}dvd)$
let $?LHS = residue\ (p\ \hat{}\ m)\ ((x \oplus_{padic\text{-}int\ p} y)\ n)$
have $A0$: $?LHS = residue\ (p\ \hat{}\ m)\ ((x\ n) \oplus_{residue\text{-}ring\ (p\hat{}n)} (\ y\ n))$
by $(simp\ add$: $padic\text{-}int\text{-}def\ padic\text{-}add\text{-}def)$
have $residue\ (p\hat{}m) \in ring\text{-}hom\ (residue\text{-}ring\ ((p\hat{}n)))\ (residue\text{-}ring\ ((p\hat{}m)))$
using $A\ False\ assms\ residue\text{-}hom\text{-}p$ **by** $auto$
then have $residue\ (p\ \hat{}\ m)\ ((x\ n) \oplus_{residue\text{-}ring\ (p\hat{}n)} (\ y\ n)) = (residue\ (p\ \hat{}\ m)\ (x\ n)) \oplus_{residue\text{-}ring\ (p\hat{}m)} ((residue\ (p\ \hat{}\ m)\ (y\ n)))$
by $(metis\ (no\text{-}types,\ lifting)\ padic\text{-}int\text{-}simps(5)\ Px\ Py\ mem\text{-}Collect\text{-}eq\ padic\text{-}set\text{-}def\ ring\text{-}hom\text{-}add)$
then have $?LHS = (residue\ (p\ \hat{}\ m)\ (x\ n)) \oplus_{residue\text{-}ring\ (p\hat{}m)} ((residue\ (p\ \hat{}\ m)\ (y\ n)))$
using $A0$ **by** $force$
then show $?thesis$
using $A\ Px\ Py\ padic\text{-}set\text{-}def$ **by** $(simp\ add$: $padic\text{-}int\text{-}def\ padic\text{-}add\text{-}def)$

qed
qed
qed
qed
then show $?thesis$
using $0\ padic\text{-}set\text{-}memI\ padic\text{-}int\text{-}simps$ **by** $auto$
qed
then have $x \oplus_{padic\text{-}int\ p} y \in (padic\text{-}set\ p)$
by$(simp\ add$: $padic\text{-}int\text{-}def)$
then show $carrier\ (padic\text{-}int\ p) \subseteq carrier\ (padic\text{-}int\ p)$

by *blast*
qed

p-adic addition is associative:

lemma *padic-add-assoc*:
assumes *prime p*
shows $\bigwedge x\ y\ z.$
$x \in carrier\ (padic\text{-}int\ p) \Longrightarrow$
$y \in carrier\ (padic\text{-}int\ p) \Longrightarrow z \in carrier\ (padic\text{-}int\ p)$
$\Longrightarrow x \oplus_{padic\text{-}int\ p} y \oplus_{padic\text{-}int\ p} z = x \oplus_{padic\text{-}int\ p} (y \oplus_{padic\text{-}int\ p} z)$
proof−
fix *x y z*
assume *Ax*: $x \in carrier\ (padic\text{-}int\ p)$
assume *Ay*: $y \in carrier\ (padic\text{-}int\ p)$
assume *Az*: $z \in carrier\ (padic\text{-}int\ p)$
show $(x \oplus_{padic\text{-}int\ p} y) \oplus_{padic\text{-}int\ p} z = x \oplus_{padic\text{-}int\ p} (y \oplus_{padic\text{-}int\ p} z)$
proof
fix *n*
show $((x \oplus_{padic\text{-}int\ p} y) \oplus_{padic\text{-}int\ p} z)\ n = (x \oplus_{padic\text{-}int\ p} (y \oplus_{padic\text{-}int\ p} z))$
n
proof−
have *Ex*: $(x\ n) \in carrier\ (residue\text{-}ring\ (p\hat{}n))$
using *Ax padic-set-def padic-int-simps* **by** *auto*
have *Ey*: $(y\ n) \in carrier\ (residue\text{-}ring\ (p\hat{}n))$
using *Ay padic-set-def padic-int-simps* **by** *auto*
have *Ez*: $(z\ n) \in carrier\ (residue\text{-}ring\ (p\hat{}n))$
using *Az padic-set-def padic-int-simps* **by** *auto*
let *?x* $= (x\ n)$
let *?y* $= (y\ n)$
let *?z* $= (z\ n)$
have *P1*: $(?x \oplus_{residue\text{-}ring\ (p\hat{}n)} ?y) \oplus_{residue\text{-}ring\ (p\hat{}n)} ?z = (x\ n) \oplus_{residue\text{-}ring\ (p\hat{}n)}$
$((y \oplus_{padic\text{-}int\ p} z)\ n)$
proof(*cases n* $= 0$)
case *True*
then show *?thesis*
by (*simp add*: *residue-ring-def*)
next
case *False*
then have *residues* $(p\hat{}n)$
by (*simp add*: *assms residues-n*)
then show *?thesis*
using *Ex Ey Ez cring.cring-simprules*(7) *padic-add-res residues.cring*
padic-int-simps
by *fastforce*
qed
have $((y\ n)) \oplus_{residue\text{-}ring\ (p\hat{}n)} z\ n = ((y \oplus_{padic\text{-}int\ p} z)\ n)$
using *padic-add-def padic-int-simps* **by** *simp*
then have *P0*: $(x\ n) \oplus_{residue\text{-}ring\ (p\hat{}n)} ((y \oplus_{padic\text{-}int\ p} z)\ n) = ((x\ n)$
$\oplus_{residue\text{-}ring\ (p\hat{}n)} ((y\ n) \oplus_{residue\text{-}ring\ (p\hat{}n)} z\ n))$

using *padic-add-def padic-int-simps* **by** *simp*
have $((x \oplus_{padic\text{-}int\ p} y) \oplus_{padic\text{-}int\ p} z)\ n = ((x \oplus_{padic\text{-}int\ p} y)\ n) \oplus_{residue\text{-}ring\ (p\hat{}n)} z\ n$
using *padic-add-def padic-int-simps* **by** *simp*
then have $((x \oplus_{padic\text{-}int\ p} y) \oplus_{padic\text{-}int\ p} z)\ n = ((x\ n) \oplus_{residue\text{-}ring\ (p\hat{}n)} (y\ n)) \oplus_{residue\text{-}ring\ (p\hat{}n)} z\ n$
using *padic-add-def padic-int-simps* **by** *simp*
then have $((x \oplus_{padic\text{-}int\ p} y) \oplus_{padic\text{-}int\ p} z)\ n = ((x\ n) \oplus_{residue\text{-}ring\ (p\hat{}n)} ((y\ n) \oplus_{residue\text{-}ring\ (p\hat{}n)} z\ n))$
using *Ex Ey Ez P1 P0* **by** *linarith*
then have $((x \oplus_{padic\text{-}int\ p} y) \oplus_{padic\text{-}int\ p} z)\ n = (x\ n) \oplus_{residue\text{-}ring\ (p\hat{}n)} ((y \oplus_{padic\text{-}int\ p} z)\ n)$
using *P0* **by** *linarith*
then show *?thesis* **by** (*simp add*: *padic-int-def padic-add-def*)
qed
qed
qed

p-adic addition is commutative:

lemma *padic-add-comm*:
assumes *prime p*
shows $\bigwedge x\ y.$
$x \in carrier\ (padic\text{-}int\ p) \Longrightarrow$
$y \in carrier\ (padic\text{-}int\ p) \Longrightarrow$
$x \oplus_{padic\text{-}int\ p} y = y \oplus_{padic\text{-}int\ p} x$
proof−
fix *x y*
assume *Ax*: $x \in carrier\ (padic\text{-}int\ p)$ **assume** *Ay*:$y \in carrier\ (padic\text{-}int\ p)$
show $x \oplus_{padic\text{-}int\ p} y = y \oplus_{padic\text{-}int\ p} x$
proof fix *n*
show $(x \oplus_{padic\text{-}int\ p} y)\ n = (y \oplus_{padic\text{-}int\ p} x)\ n$
proof(*cases n=0*)
case *True*
then show *?thesis*
by (*metis Ax Ay assms padic-add-def padic-set-zero-res padic-int-simps*(*3*,*5*))

next
case *False*
have *LHS0*: $(x \oplus_{padic\text{-}int\ p} y)\ n = (x\ n) \oplus_{residue\text{-}ring\ (p\hat{}n)} (y\ n)$
by (*simp add*: *padic-int-simps padic-add-res*)
have *RHS0*: $(y \oplus_{padic\text{-}int\ p} x)\ n = (y\ n) \oplus_{residue\text{-}ring\ (p\hat{}n)} (x\ n)$
by (*simp add*: *padic-int-simps padic-add-res*)
have *Ex*: $(x\ n) \in carrier\ (residue\text{-}ring\ (p\hat{}n))$
using *Ax padic-set-res-closed padic-int-simps* **by** *auto*
have *Ey*: $(y\ n) \in carrier\ (residue\text{-}ring\ (p\hat{}n))$
using *Ay padic-set-res-closed padic-int-simps* **by** *auto*
have *LHS1*: $(x \oplus_{padic\text{-}int\ p} y)\ n = ((x\ n) + (y\ n))\ mod\ (p\hat{}n)$
using *LHS0 residue-ring-def* **by** *simp*

have *RHS1*: $(y \oplus_{padic\text{-}int\ p} x)\ n = ((y\ n) + (x\ n))\ mod\ (p \hat{} n)$
using *RHS0 residue-ring-def* **by** *simp*
then show *?thesis* **using** *LHS1 RHS1* **by** *presburger*
qed
qed
qed

padic_zero is an additive identity:

lemma *padic-add-zero*:
assumes *prime p*
shows $\bigwedge x.\ x \in carrier\ (padic\text{-}int\ p) \Longrightarrow \mathbf{0}_{padic\text{-}int\ p} \oplus_{padic\text{-}int\ p} x = x$
proof−
fix *x*
assume *Ax*: $x \in carrier\ (padic\text{-}int\ p)$
show $\mathbf{0}_{padic\text{-}int\ p} \oplus_{padic\text{-}int\ p} x = x$
proof fix *n*
have *A*: $(padic\text{-}zero\ p)\ n = 0$
by (*simp add*: *padic-zero-def*)
have $((padic\text{-}zero\ p) \oplus_{padic\text{-}int\ p} x)\ n = x\ n$
using *Ax padic-int-simps*(5) *padic-set-res-closed residue-ring-def*
by(*auto simp add*: *padic-zero-def padic-int-simps padic-add-res residue-ring-def*)
then show $(\mathbf{0}_{padic\text{-}int\ p} \oplus_{padic\text{-}int\ p} x)\ n = x\ n$
by (*simp add*: *padic-int-def*)
qed
qed

Closure under additive inverses:

lemma *padic-add-inv*:
assumes *prime p*
shows $\bigwedge x.\ x \in carrier\ (padic\text{-}int\ p) \Longrightarrow$
$\exists\, y \in carrier\ (padic\text{-}int\ p).\ y \oplus_{padic\text{-}int\ p} x = \mathbf{0}_{padic\text{-}int\ p}$
proof−
fix *x*
assume *Ax*: $x \in carrier\ (padic\text{-}int\ p)$
show $\exists\, y \in carrier\ (padic\text{-}int\ p).\ y \oplus_{padic\text{-}int\ p} x = \mathbf{0}_{padic\text{-}int\ p}$
proof
let *?y* = $(padic\text{-}a\text{-}inv\ p)\ x$
show $?y \oplus_{padic\text{-}int\ p} x = \mathbf{0}_{padic\text{-}int\ p}$
proof
fix *n*
show $(?y \oplus_{padic\text{-}int\ p} x)\ n = \mathbf{0}_{padic\text{-}int\ p}\ n$
proof(*cases n=0*)
case *True*
then show *?thesis*
using *Ax assms padic-add-closed padic-set-zero-res*
padic-a-inv-in-padic-set padic-zero-def padic-int-simps **by** *auto*
next
case *False*
have *C*: $(x\ n) \in carrier\ (residue\text{-}ring\ (p \hat{} n))$

```
      using Ax padic-set-res-closed padic-int-simps by auto
    have R: residues (p^n)
      using False  by (simp add: assms residues-n)
    have (?y ⊕padic-int p x) n = (?y n) ⊕residue-ring (p^n) x n
      by (simp add: padic-int-def padic-add-res)
    then have (?y ⊕padic-int p x) n = 0
      using C R residue-ring-def[simp] residues.cring
        by (metis (no-types, lifting) cring.cring-simprules(9) padic-a-inv-def
residues.zero-cong)
    then show ?thesis
      by (simp add: padic-int-def padic-zero-def)
   qed
  qed
 then show padic-a-inv p x ∈ carrier (padic-int p)
  using padic-a-inv-in-padic-set padic-int-simps
      Ax assms prime-gt-0-nat by auto
 qed
qed
```

The ring of padic integers forms an abelian group under addition:

```
lemma padic-is-abelian-group:
assumes prime p
shows abelian-group (padic-int p)
 proof (rule abelian-groupI)
  show 0: ⋀x y. x ∈ carrier (padic-int p) ⟹
            y ∈ carrier (padic-int p) ⟹
            x ⊕(padic-int p) y ∈ carrier (padic-int p)
    using padic-add-closed  by (simp add: assms)
  show zero: 0padic-int p ∈ carrier (padic-int p)
    by (metis 0 assms padic-add-inv padic-int-simps(5) padic-one-in-padic-set)

  show add-assoc: ⋀x y z.
                  x ∈ carrier (padic-int p) ⟹
                  y ∈ carrier (padic-int p) ⟹
                  z ∈ carrier (padic-int p) ⟹
                       x ⊕padic-int p y ⊕padic-int p z
                     = x ⊕padic-int p (y ⊕padic-int p z)
    using assms padic-add-assoc by auto
  show comm: ⋀x y.
                  x ∈ carrier (padic-int p) ⟹
                  y ∈ carrier (padic-int p) ⟹
                  x ⊕padic-int p y = y ⊕padic-int p x
    using assms padic-add-comm by blast
  show ⋀x. x ∈ carrier (padic-int p) ⟹ 0padic-int p ⊕padic-int p x = x
    using assms padic-add-zero by blast
  show ⋀x. x ∈ carrier (padic-int p) ⟹
       ∃y∈carrier (padic-int p). y ⊕padic-int p x = 0padic-int p
    using assms padic-add-inv by blast
 qed
```

One is a multiplicative identity:

lemma *padic-one-id*:
assumes *prime p*
assumes $x \in$ *carrier* (*padic-int p*)
shows $\mathbf{1}_{padic\text{-}int\ p} \otimes_{padic\text{-}int\ p} x = x$
proof
fix *n*
show $(\mathbf{1}_{padic\text{-}int\ p} \otimes_{padic\text{-}int\ p} x)\ n = x\ n$
proof(*cases n=0*)
case *True*
then show *?thesis*
by (*metis padic-int-simps*(*1*,*4*,*5*) *assms*(*1*) *assms*(*2*) *padic-mult-in-padic-set padic-one-in-padic-set padic-set-zero-res*)
next
case *False*
then have *residues* (*p*^*n*)
by (*simp add*: *assms*(*1*) *residues-n*)
then show *?thesis*
using *False assms*(*2*) *cring.cring-simprules*(*12*) *padic-int-simps padic-mult-res padic-one-simp padic-set-res-closed residues.cring* **by** *fastforce*
qed
qed

p-adic multiplication is commutative:

lemma *padic-mult-comm*:
assumes *prime p*
assumes $x \in$ *carrier* (*padic-int p*)
assumes $y \in$ *carrier* (*padic-int p*)
shows $x \otimes_{padic\text{-}int\ p} y = y \otimes_{padic\text{-}int\ p} x$
proof
fix *n*
have *Ax*: (*x n*) ∈ *carrier* (*residue-ring* (*p*^*n*))
using *padic-set-def assms*(*2*) *padic-int-simps* **by** *auto*
have *Ay*: (*y n*) ∈*carrier* (*residue-ring* (*p*^*n*))
using *padic-set-def assms*(*3*) *padic-set-res-closed padic-int-simps*
by *blast*
show $(x \otimes_{padic\text{-}int\ p} y)\ n = (y \otimes_{padic\text{-}int\ p} x)\ n$
proof(*cases n=0*)
case *True*
then show *?thesis*
by (*metis padic-int-simps*(*4*,*5*) *assms*(*1*) *assms*(*2*) *assms*(*3*) *padic-set-zero-res padic-simps*(*3*))
next
case *False*
have *LHS0*: $(x \otimes_{padic\text{-}int\ p} y)\ n = (x\ n) \otimes_{residue\text{-}ring\ (p\hat{}n)} (y\ n)$
by (*simp add*: *padic-int-def padic-mult-res*)
have *RHS0*: $(y \otimes_{padic\text{-}int\ p} x)\ n = (y\ n) \otimes_{residue\text{-}ring\ (p\hat{}n)} (x\ n)$
by (*simp add*: *padic-int-def padic-mult-res*)

```
    have Ex: (x n) ∈ carrier (residue-ring (p^n))
      using Ax padic-set-res-closed by auto
    have Ey: (y n) ∈ carrier (residue-ring (p^n))
      using Ay padic-set-res-closed by auto
    have LHS1: (x ⊗padic-int p y) n = ((x n) ∗(y n)) mod (p^n)
      using LHS0
      by (simp add: residue-ring-def)
    have RHS1: (y ⊗padic-int p x) n = ((y n) ∗(x n)) mod (p^n)
      using RHS0
      by (simp add: residue-ring-def)
    then show ?thesis using LHS1 RHS1
      by (simp add: mult.commute)
  qed
qed

lemma padic-is-comm-monoid:
assumes prime p
shows Group.comm-monoid (padic-int p)
proof(rule comm-monoidI)
  show  ⋀x y.
          x ∈ carrier (padic-int p) ⟹
          y ∈ carrier (padic-int p) ⟹
          x ⊗padic-int p y ∈ carrier (padic-int p)
    by (simp add: padic-int-def assms padic-mult-in-padic-set)
  show 1padic-int p ∈ carrier (padic-int p)
    by (metis padic-int-simps(1,5) assms  padic-one-in-padic-set)
  show ⋀x y z.
          x ∈ carrier (padic-int p) ⟹
          y ∈ carrier (padic-int p) ⟹
          z ∈ carrier (padic-int p) ⟹
          x ⊗padic-int p y ⊗padic-int p z = x ⊗padic-int p (y ⊗padic-int p z)
    using assms padic-mult-assoc by auto
  show ⋀x. x ∈ carrier (padic-int p) ⟹ 1padic-int p ⊗padic-int p x = x
    using assms  padic-one-id by blast
  show ⋀x y.
        x ∈ carrier (padic-int p) ⟹
        y ∈ carrier (padic-int p) ⟹
        x ⊗padic-int p y = y ⊗padic-int p x
    using padic-mult-comm  by (simp add: assms)
qed

lemma padic-int-is-cring:
  assumes prime p
  shows cring (padic-int p)
proof (rule cringI)
  show abelian-group (padic-int p)
    by (simp add: assms padic-is-abelian-group)
  show Group.comm-monoid (padic-int p)
    by (simp add: assms padic-is-comm-monoid)
```

show $\bigwedge x\ y\ z.$
$x \in carrier\ (padic\text{-}int\ p) \Longrightarrow$
$y \in carrier\ (padic\text{-}int\ p) \Longrightarrow$
$z \in carrier\ (padic\text{-}int\ p) \Longrightarrow$
$(x \oplus_{padic\text{-}int\ p}\ y) \otimes_{padic\text{-}int\ p}\ z =$
$x \otimes_{padic\text{-}int\ p}\ z \oplus_{padic\text{-}int\ p}\ y \otimes_{padic\text{-}int\ p}\ z$
proof−
fix $x\ y\ z$
assume Ax: $x \in carrier\ (padic\text{-}int\ p)$
assume Ay: $y \in carrier\ (padic\text{-}int\ p)$
assume Az: $z \in carrier\ (padic\text{-}int\ p)$
show $(x \oplus_{padic\text{-}int\ p}\ y) \otimes_{padic\text{-}int\ p}\ z$
$= x \otimes_{padic\text{-}int\ p}\ z \oplus_{padic\text{-}int\ p}\ y \otimes_{padic\text{-}int\ p}\ z$
proof
fix n
have Ex: $(x\ n) \in carrier\ (residue\text{-}ring\ (p\hat{}n))$
using Ax *padic-set-def padic-int-simps* **by** *auto*
have Ey: $(y\ n) \in carrier\ (residue\text{-}ring\ (p\hat{}n))$
using Ay *padic-set-def padic-int-simps* **by** *auto*
have Ez: $(z\ n) \in carrier\ (residue\text{-}ring\ (p\hat{}n))$
using Az *padic-set-def padic-int-simps* **by** *auto*
show $(\ (x \oplus_{padic\text{-}int\ p}\ y) \otimes_{padic\text{-}int\ p}\ z)\ n$
$= (x \otimes_{padic\text{-}int\ p}\ z \oplus_{padic\text{-}int\ p}\ y \otimes_{padic\text{-}int\ p}\ z)\ n$
proof(*cases n=0*)
case *True*
then show *?thesis*
by (*metis Ax Ay Az assms padic-add-closed padic-int-simps*(*4*) *padic-int-simps*(*5*) *padic-mult-in-padic-set padic-set-zero-res*)
next
case *False*
then have *residues* $(p\hat{}n)$
by (*simp add*: *assms residues-n*)
then show *?thesis*
using *Ex Ey Ez cring.cring-simprules*(*13*) *padic-add-res padic-int-simps padic-mult-res residues.cring* **by** *fastforce*
qed
qed
qed
qed

The p-adic ring has no nontrivial zero divisors. Note that this argument is short because we have proved that the valuation is multiplicative on nonzero elements, which is where the primality assumption is used.

lemma *padic-no-zero-divisors*:
assumes *prime p*
assumes $a \in carrier\ (padic\text{-}int\ p)$
assumes $b \in carrier\ (padic\text{-}int\ p)$
assumes $a \neq \mathbf{0}_{padic\text{-}int\ p}$
assumes $b \neq \mathbf{0}_{padic\text{-}int\ p}$

shows $a \otimes_{padic\text{-}int\ p} b \neq \mathbf{0}_{padic\text{-}int\ p}$
proof
  assume *C*: $a \otimes_{padic\text{-}int\ p} b = \mathbf{0}_{padic\text{-}int\ p}$
  show *False*
  proof−
    have *0*: $a = \mathbf{0}_{padic\text{-}int\ p} \vee b = \mathbf{0}_{padic\text{-}int\ p}$
    proof(*cases* $a = \mathbf{0}_{padic\text{-}int\ p}$)
      case *True*
      then show *?thesis* **by** *auto*
    next
      case *False*
      have $\neg\ b\ = \mathbf{0}_{padic\text{-}int\ p}$
      proof
        assume $b \neq \mathbf{0}_{padic\text{-}int\ p}$
        have *padic-val p* $(a \otimes_{padic\text{-}int\ p} b) = (padic\text{-}val\ p\ a) + (padic\text{-}val\ p\ b)$
        using *False assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*5*) *val-prod padic-int-simps*
by *auto*
          then have *padic-val p* $(a \otimes_{padic\text{-}int\ p} b) \neq -1$
          using *False* ‹$b \neq \mathbf{0}_{padic\text{-}int\ p}$› *padic-val-def padic-int-simps* **by** *auto*
        then show *False*
          using *C padic-val-def padic-int-simps* **by** *auto*
        qed
      then show *?thesis*
        by *blast*
    qed
    show *?thesis*
      using *0 assms*(*4*) *assms*(*5*) **by** *blast*
  qed
qed

lemma *padic-int-is-domain*:
  assumes *prime p*
  shows *domain* (*padic-int p*)
proof(*rule domainI*)
  show *cring* (*padic-int p*)
    using *padic-int-is-cring assms*(*1*) **by** *auto*
  show $\mathbf{1}_{padic\text{-}int\ p} \neq \mathbf{0}_{padic\text{-}int\ p}$
  proof
    assume $\mathbf{1}_{padic\text{-}int\ p} = \mathbf{0}_{padic\text{-}int\ p}$
    then have $(\mathbf{1}_{padic\text{-}int\ p})\ 1 = \mathbf{0}_{padic\text{-}int\ p}\ 1$ **by** *auto*
    then show *False*
      using *padic-int-simps*(*1*,*2*)
      unfolding *padic-one-def padic-zero-def* **by** *auto*
  qed
  show $\bigwedge a\ b.\ a \otimes_{padic\text{-}int\ p} b = \mathbf{0}_{padic\text{-}int\ p} \implies$
            $a \in carrier\ (padic\text{-}int\ p) \implies$
            $b \in carrier\ (padic\text{-}int\ p) \implies$
            $a = \mathbf{0}_{padic\text{-}int\ p} \vee b = \mathbf{0}_{padic\text{-}int\ p}$
    using *assms padic-no-zero-divisors*

by (*meson prime-nat-int-transfer*)
qed

20 The Ultrametric Inequality:

lemma *padic-val-ultrametric*:
assumes *prime p*
assumes $a \in$ *carrier* (*padic-int p*)
assumes $b \in$ *carrier* (*padic-int p*)
assumes $a \neq \mathbf{0}_{(padic\text{-}int\ p)}$
assumes $b \neq \mathbf{0}_{(padic\text{-}int\ p)}$
assumes $a \oplus_{(padic\text{-}int\ p)} b \neq \mathbf{0}_{(padic\text{-}int\ p)}$
shows *padic-val p* ($a \oplus_{(padic\text{-}int\ p)} b$) $\geq$ *min* (*padic-val p a*) (*padic-val p b*)
proof−
let *?va* = *nat* (*padic-val p a*)
let *?vb* = *nat* (*padic-val p b*)
let *?vab* = *nat* (*padic-val p* ($a \oplus_{(padic\text{-}int\ p)} b$))
have *P*: ¬ *?vab* < *min ?va ?vb*
proof
assume *P0*: *?vab* < *min ?va ?vb*
then have *Suc ?vab* ≤ *min ?va ?vb*
using *Suc-leI* **by** *blast*
have ($a \oplus_{(padic\text{-}int\ p)} b$) ∈ *carrier* (*padic-int p*)
using *assms*(*1*) *assms*(*2*) *assms*(*3*) *padic-add-closed* **by** *simp*
then have *C*: ($a \oplus_{(padic\text{-}int\ p)} b$) (*?vab* + *1*) ≠ $\mathbf{0}_{residue\text{-}ring\ (p\hat{}(?vab + 1))}$
using *val-of-nonzero*(*1*) *assms*(*6*)
by (*simp add*: *padic-int-def val-of-nonzero*(*1*) *assms*(*1*))
have *S*: ($a \oplus_{(padic\text{-}int\ p)} b$) (*?vab* + *1*) = (*a* (*?vab* + *1*)) $\oplus_{residue\text{-}ring\ (p\hat{}(?vab + 1))}$ (*b* ((*?vab* + *1*)))
by (*simp add*: *padic-int-def padic-add-def*)
have *int* (*?vab* + *1*) ≤ *padic-val p a*
using *P0* **using** *Suc-le-eq* **by** *auto*
then have *A*: (*a* (*?vab* + *1*)) = $\mathbf{0}_{residue\text{-}ring\ (p\hat{}(?vab + 1))}$
using *assms*(*1*) *assms*(*2*) *zero-below-val padic-int-simps residue-ring-def*
by *auto*
have *int* (*?vab* + *1*) ≤ *padic-val p b*
using *P0* **using** *Suc-le-eq* **by** *auto*
then have *B*: (*b* (*?vab* + *1*)) = $\mathbf{0}_{residue\text{-}ring\ (p\hat{}(?vab + 1))}$
using *assms*(*1*) *assms*(*3*) *zero-below-val*
by (*metis A* ‹*int* (*nat* (*padic-val p* ($a \oplus_{padic\text{-}int\ p} b$)) + *1*) ≤ *padic-val p a*›
assms(*2*) *padic-int-simps*(*3*,*5*))
have *p*^(*?vab* + *1*) > *1*
using *assms*(*1*) **by** (*metis add.commute plus-1-eq-Suc power-gt1 prime-gt-1-int*)
then have *residues* (*p*^(*?vab* + *1*))
using *less-imp-of-nat-less residues.intro* **by** *fastforce*
then have ($a \oplus_{(padic\text{-}int\ p)} b$) (*?vab* + *1*) = $\mathbf{0}_{residue\text{-}ring\ (p\hat{}(?vab + 1))}$
using *A B* **by** (*metis* (*no-types*, *lifting*) *S cring.cring-simprules*(*2*)

```
        cring.cring-simprules(8) residues.cring)
    then show False using C by auto
  qed
  have A0: (padic-val p a) ≥ 0
    using assms(4) padic-val-def by(auto simp: padic-int-def)
  have A1: (padic-val p b) ≥ 0
    using assms(5) padic-val-def by(auto simp: padic-int-def)
  have A2: padic-val p (a ⊕_(padic-int p) b) ≥ 0
    using assms(6) padic-val-def by(auto simp: padic-int-def)
  show ?thesis using P A0 A1 A2
    by linarith
qed

lemma padic-a-inv:
  assumes prime p
  assumes a ∈ carrier (padic-int p)
  shows ⊖_padic-int p a = (λ n. ⊖_residue-ring (p^n) (a n))
proof
  fix n
  show (⊖_padic-int p a) n = ⊖_residue-ring (p^n) a n
  proof(cases n=0)
    case True
    then show ?thesis
      by (metis (no-types, lifting) abelian-group.a-inv-closed assms(1) assms(2)
padic-int-simps(5)
      padic-is-abelian-group padic-set-zero-res power-0 residue-1-prop residue-ring-def
ring.simps(1))
  next
    case False
    then have R: residues (p^n)
      by (simp add: assms(1) residues-n)
    have (⊖_padic-int p a) ⊕_padic-int p a = 0_padic-int p
      by (simp add: abelian-group.l-neg assms(1) assms(2) padic-is-abelian-group)

    then have P: (⊖_padic-int p a) n ⊕_residue-ring (p^n) a n = 0
      by (metis padic-add-res padic-int-simps(2) padic-int-simps(3) padic-zero-def)

    have Q: (a n) ∈ carrier (residue-ring (p^n))
      using assms(2) padic-set-res-closed by(auto simp: padic-int-def)
    show ?thesis using R Q residues.cring
      by (metis P abelian-group.a-inv-closed abelian-group.minus-equality assms(1)
assms(2)
      padic-int-simps(5) padic-is-abelian-group padic-set-res-closed residues.abelian-group
        residues.res-zero-eq)
  qed
qed

lemma padic-val-a-inv:
  assumes prime p
```

assumes $a \in$ *carrier* (*padic-int p*)
shows *padic-val p a* = *padic-val p* ($\ominus_{\textit{padic-int } p}$ *a*)
proof(*cases* $a = \mathbf{0}_{\textit{padic-int } p}$)
case *True*
then show *?thesis*
by (*metis abelian-group.a-inv-closed abelian-group.r-neg abelian-groupE*(5) *assms*(1) *assms*(2) *padic-is-abelian-group*)
next
case *False*
have *0*: $\bigwedge n.\ (a\ n) = \mathbf{0}_{\textit{residue-ring } (p\hat{}n)} \Longrightarrow (\ominus_{\textit{padic-int } p}\ a)\ n = \mathbf{0}_{\textit{residue-ring } (p\hat{}n)}$
using *padic-a-inv*
by (*metis* (*no-types*, *lifting*) *assms*(1) *assms*(2) *cring.cring-simprules*(22) *power-0 residue-1-prop residues.cring residues-n*)
have *1*: $\bigwedge n.\ (a\ n) \neq \mathbf{0}_{\textit{residue-ring } (p\hat{}n)} \Longrightarrow (\ominus_{\textit{padic-int } p}\ a)\ n \neq \mathbf{0}_{\textit{residue-ring } (p\hat{}n)}$
using *padic-a-inv*
by (*metis* (*no-types*, *lifting*) *abelian-group.a-inv-closed abelian-group.minus-minus assms*(1)
assms(2) *cring.cring-simprules*(22) *padic-int-simps*(5) *padic-is-abelian-group padic-set-zero-res*
residues.cring residues-n)
have *A*:*padic-val p* ($\ominus_{\textit{padic-int } p}$ *a*) $\geq$ (*padic-val p a*)
proof−
have ¬ *padic-val p* ($\ominus_{\textit{padic-int } p}$ *a*) < (*padic-val p a*)
proof
assume *padic-val p* ($\ominus_{\textit{padic-int } p}$ *a*) < *padic-val p a*
let *?n* = *padic-val p* ($\ominus_{\textit{padic-int } p}$ *a*)
let *?m* = *padic-val p a*
have ($\ominus_{\textit{padic-int } p}$ *a*) ≠ (*padic-zero p*)
by (*metis False abelian-group.l-neg assms*(1) *assms*(2) *padic-add-zero padic-int-simps*(2) *padic-is-abelian-group*)
then have *P0*: *?n* ≥*0*
by (*simp add*: *padic-val-def*)
have *P1*: *?m* ≥*0* **using** *False*
using ‹*0* ≤ *padic-val p* ($\ominus_{\textit{padic-int } p}$ *a*)›
‹*padic-val p* ($\ominus_{\textit{padic-int } p}$ *a*) < *padic-val p a*› **by** *linarith*
have (*Suc* (*nat ?n*)) < *Suc* (*nat* (*padic-val p a*))
using *P0 P1* ‹*padic-val p* ($\ominus_{\textit{padic-int } p}$ *a*) < *padic-val p a*› **by** *linarith*
then have *int* (*Suc* (*nat ?n*)) ≤ (*padic-val p a*)
using *of-nat-less-iff* **by** *linarith*
then have *a* (*Suc* (*nat ?n*)) = $\mathbf{0}_{\textit{residue-ring } (p\ \hat{}\ ((\textit{Suc } (\textit{nat } ?n))))}$
using *assms*(1) *assms*(2) *zero-below-val residue-ring-def* **by**(*auto simp*: *padic-int-def*)
then have ($\ominus_{\textit{padic-int } p}$ *a*) (*Suc* (*nat ?n*)) = $\mathbf{0}_{\textit{residue-ring } (p\ \hat{}\ ((\textit{Suc } (\textit{nat } ?n))))}$
using *0* **by** *simp*
then show *False* **using** *below-val-zero assms*
by (*metis Suc-eq-plus1* ‹$\ominus_{\textit{padic-int } p}$ *a* ≠ *padic-zero p*› *abelian-group.a-inv-closed*

padic-int-simps(5) padic-is-abelian-group val-of-nonzero(1))
  qed
  then show *?thesis*
  by *linarith*
 qed
 have *B*: *padic-val p* (⊖$_{padic\text{-}int\ p}$ *a*) ≤ (*padic-val p a*)
 proof−
  let *?n* = *nat* (*padic-val p a*)
  have *a* (*Suc ?n*) ≠ **0**$_{residue\text{-}ring\ (p\hat{}(Suc\ ?n))}$
  using *False assms(2) val-of-nonzero(1)*
  by (*metis padic-int-simps(2,5) Suc-eq-plus1 assms(1)*)
  then have (⊖$_{padic\text{-}int\ p}$ *a*) (*Suc ?n*) ≠ **0**$_{residue\text{-}ring\ (p\hat{}(Suc\ ?n))}$
  using *1* **by** *blast*
  then have *padic-val p* (⊖$_{padic\text{-}int\ p}$ *a*) ≤ *int ?n* **using** *assms(1) assms(2) below-val-zero*
  by (*metis padic-int-simps(5) abelian-group.a-inv-closed padic-is-abelian-group*)

  then show *?thesis*
  using *False padic-val-def padic-int-simps* **by** *auto*
 qed
 then show *?thesis* **using** *A B* **by** *auto*
qed

end
theory *Padic-Integers*
 imports *Padic-Construction*
  Extended-Int
  Supplementary-Ring-Facts
  HOL−Algebra.Subrings
  HOL−Number-Theory.Residues
  HOL−Algebra.Multiplicative-Group

begin

In what follows we establish a locale for reasoning about the ring of p-adic integers for a fixed prime p. We will elaborate on the basic metric properties of $\mathbb{Z}_p$ and construct the angular component maps to the residue rings.

21 A Locale for p-adic Integer Rings

locale *padic-integers* =
fixes *Zp*:: - *ring* (**structure**)
fixes *p*
defines *Zp* ≡ *padic-int p*
assumes *prime*: *prime p*

sublocale *padic-integers* < *UPZ?*: *UP Zp UP Zp*
 by *simp*

sublocale *padic-integers* < *Zp?*:*UP-cring Zp UP Zp*
unfolding *UP-cring-def*
by(*auto simp add*: *Zp-def padic-int-is-cring prime*)

sublocale *padic-integers* < *Zp?*:*ring Zp*
using *Zp-def cring.axioms*(*1*) *padic-int-is-cring prime*
by *blast*

sublocale *padic-integers* < *Zp?*:*cring Zp*
by (*simp add*: *Zp-def padic-int-is-cring prime*)

sublocale *padic-integers* < *Zp?*:*domain Zp*
by (*simp add*: *Zp-def padic-int-is-domain padic-integers.prime padic-integers-axioms*)

context *padic-integers*
begin

lemma *Zp-defs*:
1 = *padic-one p*
0 = *padic-zero p*
carrier Zp = *padic-set p*
(⊗) = *padic-mult p*
(⊕) = *padic-add p*
unfolding *Zp-def* **using** *padic-int-simps* **by** *auto*

end

22 Residue Rings

lemma(**in** *field*) *field-inv*:
assumes *a* ∈ *carrier R*
assumes *a* ≠**0**
shows $inv_R\ a \otimes a = \mathbf{1}$
$a \otimes inv_R\ a = \mathbf{1}$
$inv_R\ a \in carrier\ R$
proof−
have *a* ∈ *Units R*
using *assms* **by** (*simp add*: *local.field-Units*)
then show $inv_R\ a \otimes a = \mathbf{1}$
by *simp*
show *a* ⊗ *inv a* = **1**
using ‹*a* ∈ *Units R*› **by** *auto*
show $inv_R\ a \in carrier\ R$
by (*simp add*: ‹*a* ∈ *Units R*›)
qed

$p_residue$ defines the standard projection maps between residue rings:

definition(**in** *padic-integers*) *p-residue*:: *nat* ⇒ *int* ⇒ - **where**
p-residue m n ≡ *residue* (*p*^*m*) *n*

lemma(**in** *padic-integers*) *p-residue-alt-def*:
p-residue m n = *n mod* (*p*^*m*)
using *residue-def*
by (*simp add*: *p-residue-def*)

lemma(**in** *padic-integers*) *p-residue-range*:
p-residue m n ∈ {*0*..<*p*^*m*}
using *prime* **by** (*simp add*: *p-residue-alt-def prime-gt-0-int*)

lemma(**in** *padic-integers*) *p-residue-mod*:
assumes *m* > *k*
shows *p-residue k* (*p-residue m n*) = *p-residue k n*
using *assms*
unfolding *p-residue-def residue-def*
by (*simp add*: *le-imp-power-dvd mod-mod-cancel*)

Compatibility of p_residue with elements of $\mathbb{Z}_p$:

lemma(**in** *padic-integers*) *p-residue-padic-int*:
assumes *x* ∈ *carrier Zp*
assumes *m* ≥ *k*
shows *p-residue k* (*x m*) = *x k*
using *Zp-def assms*(*1*) *assms*(*2*) *padic-set-res-coherent prime*
by (*simp add*: *p-residue-def padic-int-simps*(*5*))

Definition of residue rings:

abbreviation(**in** *padic-integers*) (*input*) *Zp-res-ring*:: *nat* ⇒ - *ring* **where**
(*Zp-res-ring n*) ≡ *residue-ring* (*p*^*n*)

lemma (**in** *padic-integers*) *p-res-ring-zero*:
0$_{Zp\text{-}res\text{-}ring\ k}$ = *0*
by (*simp add*: *residue-ring-def*)

lemma (**in** *padic-integers*) *p-res-ring-one*:
assumes *k* > *0*
shows **1**$_{Zp\text{-}res\text{-}ring\ k}$ = *1*
using *assms*
by (*simp add*: *residue-ring-def*)

lemma (**in** *padic-integers*) *p-res-ring-car*:
carrier (*Zp-res-ring k*) = {*0*..<*p*^*k*}
using *residue-ring-def*[*of p*^*k*]
by *auto*

lemma(**in** *padic-integers*) *p-residue-range'*:
p-residue m n ∈ *carrier* (*Zp-res-ring m*)
using *p-residue-range residue-ring-def prime prime-gt-0-nat p-residue-def*

by *fastforce*

First residue ring is a field:

lemma(**in** *padic-integers*) *p-res-ring-1-field*:
field (*Zp-res-ring 1*)
by (*metis int-nat-eq power-one-right prime prime-ge-0-int prime-nat-int-transfer residues-prime.intro residues-prime.is-field*)

0^{th} residue ring is the zero ring:

lemma(**in** *padic-integers*) *p-res-ring-0*:
carrier (*Zp-res-ring 0*) = {*0*}
by (*simp add*: *residue-ring-def*)

lemma(**in** *padic-integers*) *p-res-ring-0'*:
assumes *x* ∈ *carrier* (*Zp-res-ring 0*)
shows *x* = *0*
using *p-res-ring-0 assms* **by** *blast*

If $m > 0$ then Zp_res_ringm is an instance of the residues locale:

lemma(**in** *padic-integers*) *p-residues*:
assumes *m* >*0*
shows *residues* (*p*^*m*)
proof−
have *p*^*m* > *1*
using *assms*
by (*simp add*: *prime prime-gt-1-int*)
then show *residues* (*p*^*m*)
using *less-imp-of-nat-less residues.intro* **by** *fastforce*
qed

If $m > 0$ then Zp_res_ringm is a commutative ring:

lemma(**in** *padic-integers*) *R-cring*:
assumes *m* >*0*
shows *cring* (*Zp-res-ring m*)
using *p-residues assms residues.cring* **by** *auto*

lemma(**in** *padic-integers*) *R-comm-monoid*:
assumes *m* >*0*
shows *comm-monoid* (*Zp-res-ring m*)
by (*simp add*: *assms p-residues residues.comm-monoid*)

lemma(**in** *padic-integers*) *zero-rep*:
0 = (λ*m*. (*p-residue m 0*))
unfolding *p-residue-def*
using *Zp-defs*(*2*) *padic-zero-simp*(*1*) *residue-def residue-ring-def* **by** *auto*

The operations on residue rings are just the standard operations on the integers mod p^n. This means that the basic closure properties and algebraic

properties of operations on these rings hold for all integers, not just elements of the ring carrier:

lemma(**in** *padic-integers*) *residue-add*:
shows $(x \oplus_{Zp\text{-}res\text{-}ring\ k} y) = (x + y)\ mod\ p\hat{}k$
unfolding *residue-ring-def*
by *simp*

lemma(**in** *padic-integers*) *residue-add-closed*:
shows $(x \oplus_{Zp\text{-}res\text{-}ring\ k} y) \in carrier\ (Zp\text{-}res\text{-}ring\ k)$
using *p-residue-def p-residue-range residue-def residue-ring-def* **by** *auto*

lemma(**in** *padic-integers*) *residue-add-closed'*:
shows $(x \oplus_{Zp\text{-}res\text{-}ring\ k} y) \in \{0..<p\hat{}k\}$
using *residue-add-closed residue-ring-def* **by** *auto*

lemma(**in** *padic-integers*) *residue-mult*:
shows $(x \otimes_{Zp\text{-}res\text{-}ring\ k} y) = (x * y)\ mod\ p\hat{}k$
unfolding *residue-ring-def*
by *simp*

lemma(**in** *padic-integers*) *residue-mult-closed*:
shows $(x \otimes_{Zp\text{-}res\text{-}ring\ k} y) \in carrier\ (Zp\text{-}res\text{-}ring\ k)$
using *p-residue-def p-residue-range residue-def residue-ring-def* **by** *auto*

lemma(**in** *padic-integers*) *residue-mult-closed'*:
shows $(x \otimes_{Zp\text{-}res\text{-}ring\ k} y) \in \{0..<p\hat{}k\}$
using *residue-mult-closed residue-ring-def* **by** *auto*

lemma(**in** *padic-integers*) *residue-add-assoc*:
shows $(x \oplus_{Zp\text{-}res\text{-}ring\ k} y) \oplus_{Zp\text{-}res\text{-}ring\ k} z = x \oplus_{Zp\text{-}res\text{-}ring\ k} (y \oplus_{Zp\text{-}res\text{-}ring\ k} z)$
using *residue-add*
by (*simp add*: *add.commute add.left-commute mod-add-right-eq*)

lemma(**in** *padic-integers*) *residue-mult-comm*:
shows $x \otimes_{Zp\text{-}res\text{-}ring\ k} y = y \otimes_{Zp\text{-}res\text{-}ring\ k} x$
using *residue-mult*
by (*simp add*: *mult.commute*)

lemma(**in** *padic-integers*) *residue-mult-assoc*:
shows $(x \otimes_{Zp\text{-}res\text{-}ring\ k} y) \otimes_{Zp\text{-}res\text{-}ring\ k} z = x \otimes_{Zp\text{-}res\text{-}ring\ k} (y \otimes_{Zp\text{-}res\text{-}ring\ k} z)$
using *residue-mult*
by (*simp add*: *mod-mult-left-eq mod-mult-right-eq mult.assoc*)

lemma(**in** *padic-integers*) *residue-add-comm*:
shows $x \oplus_{Zp\text{-}res\text{-}ring\ k} y = y \oplus_{Zp\text{-}res\text{-}ring\ k} x$
using *residue-add*
by *presburger*

lemma(**in** *padic-integers*) *residue-minus-car*:
assumes $y \in$ *carrier* (*Zp-res-ring k*)
shows ($x \ominus_{Zp\text{-}res\text{-}ring\ k} y$) = ($x - y$) *mod* $p\hat{}k$
proof(*cases* $k = 0$)
case *True*
then show *?thesis*
using *residue-ring-def a-minus-def*
by(*simp add*: *a-minus-def residue-ring-def*)
next
case *False*
have ($x \ominus_{Zp\text{-}res\text{-}ring\ k} y$) $\oplus_{Zp\text{-}res\text{-}ring\ k} y = x \oplus_{Zp\text{-}res\text{-}ring\ k}$ ($\ominus_{Zp\text{-}res\text{-}ring\ k} y \oplus_{Zp\text{-}res\text{-}ring\ k} y$)
by (*simp add*: *a-minus-def residue-add-assoc*)
then have *0*: ($x \ominus_{Zp\text{-}res\text{-}ring\ k} y$) $\oplus_{Zp\text{-}res\text{-}ring\ k} y = x$ *mod* $p\hat{}k$
using *assms False*
by (*smt* (*verit*) *cring.cring-simprules*(*9*) *prime residue-add residues.cring residues.res-zero-eq residues-n*)
have *1*: x *mod* $p\ \hat{}\ k = (((x - y)$ *mod* $p\ \hat{}\ k + y)$ *mod* $p\ \hat{}\ k$
proof −
have *f1*: $x - y = x + -1 * y$
by *auto*
have $y + (x + -1 * y) = x$
by *simp*
then show *?thesis*
using *f1* **by** *presburger*
qed
have ($x \ominus_{Zp\text{-}res\text{-}ring\ k} y$) $\oplus_{Zp\text{-}res\text{-}ring\ k} y = (x - y)$ *mod* $p\hat{}k \oplus_{Zp\text{-}res\text{-}ring\ k} y$
using *residue-add*[*of k* ($x - y$) *mod* $p\hat{}k$ *y*] *0 1*
by *linarith*
then show *?thesis* **using** *assms residue-add-closed*
by (*metis False a-minus-def cring.cring-simprules*(*10*) *cring.cring-simprules*(*19*)

prime residues.cring residues.mod-in-carrier residues-n)
qed

lemma(**in** *padic-integers*) *residue-a-inv*:
shows $\ominus_{Zp\text{-}res\text{-}ring\ k} y = \ominus_{Zp\text{-}res\text{-}ring\ k}$ (y *mod* $p\hat{}k$)
proof−
have $y \oplus_{Zp\text{-}res\text{-}ring\ k}$ ($\ominus_{Zp\text{-}res\text{-}ring\ k}$ (y *mod* $p\hat{}k$)) = (y *mod* $p\hat{}k$) $\oplus_{Zp\text{-}res\text{-}ring\ k}$ ($\ominus_{Zp\text{-}res\text{-}ring\ k}$ (y *mod* $p\hat{}k$))
using *residue-minus-car*[*of* $\ominus_{Zp\text{-}res\text{-}ring\ k}$ (y *mod* $p\hat{}k$) *k y*] *residue-add*
by (*simp add*: *mod-add-left-eq*)
then have *0*: $y \oplus_{Zp\text{-}res\text{-}ring\ k}$ ($\ominus_{Zp\text{-}res\text{-}ring\ k}$ (y *mod* $p\hat{}k$)) = $\mathbf{0}_{Zp\text{-}res\text{-}ring\ k}$
by (*metis cring.cring-simprules*(*17*) *p-res-ring-zero padic-integers.p-res-ring-0'*

padic-integers-axioms prime residue-add-closed residues.cring residues.mod-in-carrier residues-n)
have *1*: ($\ominus_{Zp\text{-}res\text{-}ring\ k}$ (y *mod* $p\hat{}k$)) $\oplus_{Zp\text{-}res\text{-}ring\ k} y = \mathbf{0}_{Zp\text{-}res\text{-}ring\ k}$

using *residue-add-comm 0* **by** *auto*
have *2*: $\bigwedge x.\ x \in carrier\ (Zp\text{-}res\text{-}ring\ k) \wedge x \oplus_{Zp\text{-}res\text{-}ring\ k} y = \mathbf{0}_{Zp\text{-}res\text{-}ring\ k} \wedge y \oplus_{Zp\text{-}res\text{-}ring\ k} x = \mathbf{0}_{Zp\text{-}res\text{-}ring\ k} \Longrightarrow x = \ominus_{Zp\text{-}res\text{-}ring\ k}\ (y\ mod\ p\hat{}k)$
using *0 1*
by (*metis cring.cring-simprules*(*3*) *cring.cring-simprules*(*8*) *mod-by-1 padic-integers.p-res-ring-0'*
padic-integers.p-res-ring-zero padic-integers-axioms power-0 prime residue-1-prop

residue-add-assoc residues.cring residues.mod-in-carrier residues-n)
have *3*: *carrier* (*add-monoid* (*residue-ring* (*p* ^ *k*))) = *carrier* (*Zp-res-ring k*)
by *simp*
have *4*: $(\otimes_{add\text{-}monoid\ (residue\text{-}ring\ (p\ \hat{}\ k))}) = (\oplus_{Zp\text{-}res\text{-}ring\ k})$
by *simp*
have *5*: $\bigwedge x.\ x \in carrier\ (add\text{-}monoid\ (residue\text{-}ring\ (p\ \hat{}\ k))) \wedge x \otimes_{add\text{-}monoid\ (residue\text{-}ring\ (p\ \hat{}\ k))} y = \mathbf{1}_{add\text{-}monoid\ (residue\text{-}ring\ (p\ \hat{}\ k))} \wedge y \otimes_{add\text{-}monoid\ (residue\text{-}ring\ (p\ \hat{}\ k))} x = \mathbf{1}_{add\text{-}monoid\ (residue\text{-}ring\ (p\ \hat{}\ k))} \Longrightarrow x = \ominus_{Zp\text{-}res\text{-}ring\ k}\ (y\ mod\ p\hat{}k)$
using *0 1 2 3 4*
by *simp*
show *?thesis*
unfolding *a-inv-def m-inv-def*
apply(*rule the-equality*)
using *1 2 3 4 5* **unfolding** *a-inv-def m-inv-def*
apply (*metis* (*no-types*, *lifting*) *0 1 cring.cring-simprules*(*3*) *mod-by-1*
monoid.select-convs(*2*) *padic-integers.p-res-ring-zero padic-integers-axioms power-0 prime*
residue-1-prop residue-add-closed residues.cring residues.mod-in-carrier residues-n)
using *1 2 3 4 5* **unfolding** *a-inv-def m-inv-def*
by *blast*
qed

lemma(**in** *padic-integers*) *residue-a-inv-closed*:
$\ominus_{Zp\text{-}res\text{-}ring\ k}\ y \in carrier\ (Zp\text{-}res\text{-}ring\ k)$
apply(*cases k = 0*)
apply (*metis add.comm-neutral add.commute*
atLeastLessThanPlusOne-atLeastAtMost-int
insert-iff mod-by-1 p-res-ring-car p-res-ring-zero padic-integers.p-res-ring-0
padic-integers-axioms power-0 residue-1-prop residue-a-inv)
by (*simp add*: *prime residues.mod-in-carrier residues.res-neg-eq residues-n*)

lemma(**in** *padic-integers*) *residue-minus*:
$(x \ominus_{Zp\text{-}res\text{-}ring\ k} y) = (x - y)\ mod\ p\hat{}k$
using *residue-minus-car*[*of y mod p*^*k k x*] *residue-a-inv*[*of k y*] **unfolding** *a-minus-def*
by (*metis a-minus-def mod-diff-right-eq p-residue-alt-def p-residue-range'*)

lemma(**in** *padic-integers*) *residue-minus-closed*:
$(x \ominus_{Zp\text{-}res\text{-}ring\ k} y) \in carrier\ (Zp\text{-}res\text{-}ring\ k)$

by (*simp add*: *a-minus-def residue-add-closed*)

lemma (**in** *padic-integers*) *residue-plus-zero-r*:
$0 \oplus_{Zp\text{-}res\text{-}ring\ k} y = y \bmod p\hat{}k$
by (*simp add*: *residue-add*)

lemma (**in** *padic-integers*) *residue-plus-zero-l*:
$y \oplus_{Zp\text{-}res\text{-}ring\ k} 0 = y \bmod p\hat{}k$
by (*simp add*: *residue-add*)

lemma (**in** *padic-integers*) *residue-times-zero-r*:
$0 \otimes_{Zp\text{-}res\text{-}ring\ k} y = 0$
by (*simp add*: *residue-mult*)

lemma (**in** *padic-integers*) *residue-times-zero-l*:
$y \otimes_{Zp\text{-}res\text{-}ring\ k} 0 = 0$
by (*simp add*: *residue-mult*)

lemma (**in** *padic-integers*) *residue-times-one-r*:
$1 \otimes_{Zp\text{-}res\text{-}ring\ k} y = y \bmod p\hat{}k$
by (*simp add*: *residue-mult*)

lemma (**in** *padic-integers*) *residue-times-one-l*:
$y \otimes_{Zp\text{-}res\text{-}ring\ k} 1 = y \bmod p\hat{}k$
by (*simp add*: *residue-mult-comm residue-times-one-r*)

Similarly to the previous lemmas, many identities about taking residues of p-adic integers hold even for elements which lie outside the carrier of $\mathbb{Z}_p$:

lemma (**in** *padic-integers*) *residue-of-sum*:
$(a \oplus b)\ k = (a\ k) \oplus_{Zp\text{-}res\text{-}ring\ k} (b\ k)$
using *Zp-def residue-ring-def*[*of* $p\hat{}k$] *Zp-defs*(*5*) *padic-add-res*
by *auto*

lemma (**in** *padic-integers*) *residue-of-sum'*:
$(a \oplus b)\ k = ((a\ k) + (b\ k))\ mod\ p\hat{}k$
using *residue-add residue-of-sum* **by** *auto*

lemma (**in** *padic-integers*) *residue-closed*[*simp*]:
assumes $b \in carrier\ Zp$
shows $b\ k \in carrier\ (Zp\text{-}res\text{-}ring\ k)$
using *Zp-def assms padic-integers.Zp-defs*(*3*) *padic-integers-axioms padic-set-res-closed*

by *auto*

lemma (**in** *padic-integers*) *residue-of-diff*:
assumes $b \in carrier\ Zp$
shows $(a \ominus b)\ k = (a\ k) \ominus_{Zp\text{-}res\text{-}ring\ k} (b\ k)$
unfolding *a-minus-def*
using *Zp-def add.inv-closed assms*(*1*) *padic-a-inv prime residue-of-sum* **by** *auto*

lemma (**in** *padic-integers*) *residue-of-prod*:
$(a \otimes b)\ k = (a\ k) \otimes_{Zp\text{-}res\text{-}ring\ k} (b\ k)$
by (*simp add*: *Zp-defs*(*4*) *padic-mult-def*)

lemma (**in** *padic-integers*) *residue-of-prod'*:
$(a \otimes b)\ k = ((a\ k) * (b\ k))\ mod\ (p\hat{}k)$
by (*simp add*: *residue-mult residue-of-prod*)

lemma (**in** *padic-integers*) *residue-of-one*:
assumes $k > 0$
shows $\mathbf{1}\ k = \mathbf{1}_{Zp\text{-}res\text{-}ring\ k}$
$\mathbf{1}\ k = 1$
apply (*simp add*: *Zp-defs*(*1*) *assms padic-one-simp*(*1*))
by (*simp add*: *Zp-def assms padic-int-simps*(*1*) *padic-one-simp*(*1*) *residue-ring-def*)

lemma (**in** *padic-integers*) *residue-of-zero*:
shows $\mathbf{0}\ k = \mathbf{0}_{Zp\text{-}res\text{-}ring\ k}$
$\mathbf{0}\ k = 0$
apply (*simp add*: *Zp-defs*(*2*) *padic-zero-simp*(*1*))
by (*simp add*: *p-residue-alt-def zero-rep*)

lemma(**in** *padic-integers*) *Zp-residue-mult-zero*:
assumes $a\ k = 0$
shows $(a \otimes b)\ k = 0\ (b \otimes a)\ k = 0$
using *assms residue-of-prod'*
by *auto*

lemma(**in** *padic-integers*) *Zp-residue-add-zero*:
assumes $b \in carrier\ Zp$
assumes (a:: *padic-int*) $k = 0$
shows $(a \oplus b)\ k = b\ k\ (b \oplus a)\ k = b\ k$
apply (*metis Zp-def assms*(*1*) *assms*(*2*) *cring.cring-simprules*(*8*) *mod-by-1 padic-int-is-cring power.simps*(*1*)
prime residue-add-closed residue-of-sum residue-of-sum' residues.cring residues.res-zero-eq residues-n)
by (*metis Zp-def assms*(*1*) *assms*(*2*) *cring.cring-simprules*(*16*) *mod-by-1 padic-int-is-cring power.simps*(*1*) *prime residue-add-closed residue-of-sum residue-of-sum' residues.cring*

residues.res-zero-eq residues-n)

P-adic addition and multiplication are globally additive and associative:

lemma *padic-add-assoc0*:
assumes *prime p*
shows *padic-add p* (*padic-add p x y*) *z* = *padic-add p x* (*padic-add p y z*)
using *assms* **unfolding** *padic-add-def*
by (*simp add*: *padic-integers.residue-add-assoc padic-integers-def*)

lemma(**in** *padic-integers*) *add-assoc*:

$a \oplus b \oplus c = a \oplus (b \oplus c)$
using *padic-add-assoc0*[*of p a b c*] *prime Zp-defs* **by** *auto*

lemma *padic-add-comm0*:
assumes *prime p*
shows (*padic-add p x y*)= (*padic-add p y x*)
using *assms* **unfolding** *padic-add-def*
using *padic-integers.residue-add-comm*[*of p*]
by (*simp add*: *padic-integers-def*)

lemma(**in** *padic-integers*) *add-comm*:
$a \oplus b = b \oplus a$
using *padic-add-comm0*[*of p a b*] *prime Zp-defs* **by** *auto*

lemma *padic-mult-assoc0*:
assumes *prime p*
shows *padic-mult p* (*padic-mult p x y*) *z* = *padic-mult p x* (*padic-mult p y z*)
using *assms* **unfolding** *padic-mult-def*
by (*simp add*: *padic-integers.residue-mult-assoc padic-integers-def*)

lemma(**in** *padic-integers*) *mult-assoc*:
$a \otimes b \otimes c = a \otimes (b \otimes c)$
using *padic-mult-assoc0*[*of p a b c*] *prime Zp-defs* **by** *auto*

lemma *padic-mult-comm0*:
assumes *prime p*
shows (*padic-mult p x y*)= (*padic-mult p y x*)
using *assms* **unfolding** *padic-mult-def*
using *padic-integers.residue-mult-comm*[*of p*]
by (*simp add*: *padic-integers-def*)

lemma(**in** *padic-integers*) *mult-comm*:
$a \otimes b = b \otimes a$
using *padic-mult-comm0*[*of p a b*] *prime Zp-defs* **by** *auto*

lemma(**in** *padic-integers*) *mult-zero-l*:
$a \otimes \mathbf{0} = \mathbf{0}$
proof fix *x* **show** ($a \otimes \mathbf{0}$) *x* = **0** *x*
using *Zp-residue-mult-zero*[*of* **0** *x a*]
by (*simp add*: *residue-of-zero*(*2*))
qed

lemma(**in** *padic-integers*) *mult-zero-r*:
$\mathbf{0} \otimes a = \mathbf{0}$
using *mult-zero-l mult-comm* **by** *auto*

lemma (**in** *padic-integers*) *p-residue-ring-car-memI*:
assumes (*m*::*int*) ≥*0*
assumes *m* < *p*^*k*

shows $m \in$ *carrier* (*Zp-res-ring k*)
using *residue-ring-def*[*of p^k*] *assms*(*1*) *assms*(*2*)
by *auto*

lemma (**in** *padic-integers*) *p-residue-ring-car-memE*:
assumes $m \in$ *carrier* (*Zp-res-ring k*)
shows $m < p^k$ $m \geq 0$
using *assms residue-ring-def* **by** *auto*

lemma (**in** *padic-integers*) *residues-closed*:
assumes $a \in$ *carrier Zp*
shows *a k* $\in$ *carrier* (*Zp-res-ring k*)
using *Zp-def assms padic-integers.Zp-defs*(*3*) *padic-integers-axioms padic-set-res-closed*
by *blast*

lemma (**in** *padic-integers*) *mod-in-carrier*:
a mod (p^n) $\in$ *carrier* (*Zp-res-ring n*)
using *p-residue-alt-def p-residue-range'* **by** *auto*

lemma (**in** *padic-integers*) *Zp-residue-a-inv*:
assumes $a \in$ *carrier Zp*
shows ($\ominus$ *a*) *k* = $\ominus_{Zp\text{-}res\text{-}ring\ k}$ (*a k*)
($\ominus$ *a*) *k* = (− (*a k*)) *mod* (p^k)
using *Zp-def assms padic-a-inv prime* **apply** *auto*[*1*]
using *residue-a-inv*
by (*metis Zp-def assms mod-by-1 p-res-ring-zero padic-a-inv padic-integers.prime*

padic-integers-axioms power-0 residue-1-prop residues.res-neg-eq residues-n)

lemma (**in** *padic-integers*) *residue-of-diff'*:
assumes $b \in$ *carrier Zp*
shows (*a* $\ominus$ *b*) *k* = ((*a k*) − (*b k*)) *mod* (p^k)
by (*simp add*: *assms residue-minus-car residue-of-diff residues-closed*)

lemma (**in** *padic-integers*) *residue-UnitsI*:
assumes *n* $\geq$ *1*
assumes (*k*::*int*) > *0*
assumes *k* < p^n
assumes *coprime k p*
shows *k* $\in$ *Units* (*Zp-res-ring n*)
using *residues.res-units-eq assms*
by (*metis* (*mono-tags*, *lifting*) *coprime-power-right-iff mem-Collect-eq not-one-le-zero prime residues-n*)

lemma (**in** *padic-integers*) *residue-UnitsE*:
assumes *n* $\geq$ *1*
assumes *k* $\in$ *Units* (*Zp-res-ring n*)
shows *coprime k p*
using *residues.res-units-eq assms*

by (*simp add*: *p-residues*)

lemma(**in** *padic-integers*) *residue-units-nilpotent*:
assumes $m > 0$
assumes $k = card\ (Units\ (Zp\text{-}res\text{-}ring\ m))$
assumes $x \in (Units\ (Zp\text{-}res\text{-}ring\ m))$
shows $x[\hat{}]_{Zp\text{-}res\text{-}ring\ m}\ k = 1$
proof−
have *0*: *group* (*units-of* (*Zp-res-ring m*))
using *assms*(*1*) *cring-def monoid.units-group padic-integers.R-cring*
padic-integers-axioms ring-def
by *blast*
have *1*: *finite* (*Units* (*Zp-res-ring m*))
using *p-residues assms*(*1*) *residues.finite-Units*
by *auto*
have *2*: $x[\hat{}]_{units\text{-}of\ (Zp\text{-}res\text{-}ring\ m)}\ (order\ (units\text{-}of\ (Zp\text{-}res\text{-}ring\ m))) = \mathbf{1}_{units\text{-}of\ (Zp\text{-}res\text{-}ring\ m)}$
by (*metis 0 assms*(*3*) *group.pow-order-eq-1 units-of-carrier*)
then show *?thesis*
by (*metis 1 assms*(*1*) *assms*(*2*) *assms*(*3*) *cring.units-power-order-eq-one*
padic-integers.R-cring padic-integers.p-residues padic-integers-axioms residues.res-one-eq)
qed

lemma (**in** *padic-integers*) *residue-1-unit*:
assumes $m > 0$
shows $1 \in Units\ (Zp\text{-}res\text{-}ring\ m)$
$\mathbf{1}_{Zp\text{-}res\text{-}ring\ m} \in Units\ (Zp\text{-}res\text{-}ring\ m)$
proof−
have *0*: *group* (*units-of* (*Zp-res-ring m*))
using *assms*(*1*) *cring-def monoid.units-group padic-integers.R-cring*
padic-integers-axioms ring-def
by *blast*
have *1*: $1 = \mathbf{1}_{units\text{-}of\ (Zp\text{-}res\text{-}ring\ m)}$
by (*simp add*: *residue-ring-def units-of-def*)
have $\mathbf{1}_{units\text{-}of\ (Zp\text{-}res\text{-}ring\ m)} \in carrier\ (units\text{-}of\ (Zp\text{-}res\text{-}ring\ m))$
using *0 Group.monoid.intro*[*of units-of* (*Zp-res-ring m*)]
by (*simp add*: *group.is-monoid*)
then show $1 \in Units\ (Zp\text{-}res\text{-}ring\ m)$
using *1* **by** (*simp add*: *units-of-carrier*)
then show $\mathbf{1}_{Zp\text{-}res\text{-}ring\ m} \in Units\ (Zp\text{-}res\text{-}ring\ m)$
by (*simp add*: *1 units-of-one*)
qed

lemma (**in** *padic-integers*) *zero-not-in-residue-units*:
assumes $n \geq 1$
shows $(0::int) \notin Units\ (Zp\text{-}res\text{-}ring\ n)$
using *assms p-residues residues.res-units-eq* **by** *auto*

Cardinality bounds on the units of residue rings:

lemma (**in** *padic-integers*) *residue-units-card-geq-2*:

```
  assumes n ≥2
  shows card (Units (Zp-res-ring n)) ≥ 2
proof(cases p = 2)
  case True
    then have (3::int) ∈ Units (Zp-res-ring n)
    proof−
      have p ≥2
        by (simp add: True)
      then have p^n ≥ 2^n
        using assms
        by (simp add: True)
      then have p^n ≥ 4
        using assms   power-increasing[of 2 n 2]
        by (simp add: True)
      then have (3::int) < p^n
        by linarith
      then have 0: (3::int) ∈ carrier (Zp-res-ring n)
        by (simp add: residue-ring-def)
      have 1: coprime 3 p
        by (simp add: True numeral-3-eq-3)
      show ?thesis using residue-UnitsI[of n 3::int]
        using 1 ‹3 < p ^ n› assms by linarith
    qed
    then have 0: {(1::int), 3} ⊆ Units (Zp-res-ring n)
      using assms padic-integers.residue-1-unit padic-integers-axioms by auto
    have 1: finite (Units (Zp-res-ring n))
     using assms padic-integers.p-residues padic-integers-axioms residues.finite-Units
by auto
    have 2: {(1::int),3}⊆Units (Zp-res-ring n)
      using 0 by auto
    have 3: card {(1::int),3} = 2
      by simp
    show ?thesis
      using 2 1
      by (metis 3 card-mono)
next
    case False
    have 1 ∈ Units (Zp-res-ring n)
      using assms padic-integers.residue-1-unit padic-integers-axioms by auto
    have 2 ∈ Units (Zp-res-ring n)
      apply(rule residue-UnitsI)
      using assms apply linarith
        apply simp
    proof−
      show 2 < p^n
      proof−
        have p^n > p
          by (metis One-nat-def assms le-simps(3) numerals(2) power-one-right
              power-strict-increasing-iff prime prime-gt-1-int)
```

```
      then show ?thesis using False  prime prime-gt-1-int[of p]
        by auto
    qed
    show coprime 2 p
      using False
        by (metis of-nat-numeral prime prime-nat-int-transfer primes-coprime
two-is-prime-nat)
  qed
  then have 0: {(1::int), 2} ⊆ Units (Zp-res-ring n)
    using ‹1 ∈ Units (Zp-res-ring n)› by blast
  have 1: card {(1::int),2} = 2
    by simp
  then show ?thesis
    using residues.finite-Units 0
    by (metis One-nat-def assms card-mono dual-order.trans
      le-simps(3) one-le-numeral padic-integers.p-residues padic-integers-axioms)
qed

lemma (in padic-integers) residue-ring-card:
finite (carrier (Zp-res-ring n)) ∧ card (carrier (Zp-res-ring n)) = nat (p^n)
  using p-res-ring-car[of n]
  by simp

lemma(in comm-monoid) UnitsI:
  assumes a ∈ carrier G
  assumes b ∈ carrier G
  assumes a ⊗ b = 1
  shows a ∈ Units G b ∈ Units G
  unfolding Units-def using comm-monoid-axioms-def assms m-comm[of a b]
  by auto

lemma(in comm-monoid) is-invI:
  assumes a ∈ carrier G
  assumes b ∈ carrier G
  assumes a ⊗ b = 1
  shows inv_G b = a inv_G a = b
  using assms inv-char m-comm
  by auto

lemma (in padic-integers) residue-of-Units:
  assumes k > 0
  assumes a ∈ Units Zp
  shows a k ∈ Units (Zp-res-ring k)
proof−
  have 0: a k ⊗_Zp-res-ring k (inv _Zp a) k = 1
    by (metis R.Units-r-inv assms(1) assms(2) residue-of-one(2) residue-of-prod)
  have 1: a k ∈ carrier (Zp-res-ring k)
    by (simp add: R.Units-closed assms(2) residues-closed)
  have 2: (inv _Zp a) k ∈ carrier (Zp-res-ring k)
```

by (*simp add*: *assms*(*2*) *residues-closed*)
show *?thesis* **using** *0 1 2 comm-monoid.UnitsI*[*of Zp-res-ring k*]
using *assms*(*1*) *p-residues residues.comm-monoid residues.res-one-eq*
by *presburger*
qed

23 *int* and *nat* inclusions in $\mathbb{Z}_p$.

lemma(**in** *ring*) *int-inc-zero*:
$[(0::int)]\cdot \mathbf{1} = \mathbf{0}$
by (*simp add*: *add.int-pow-eq-id*)

lemma(**in** *ring*) *int-inc-zero'*:
assumes $x \in carrier\ R$
shows $[(0::int)] \cdot x = \mathbf{0}$
by (*simp add*: *add.int-pow-eq-id assms*)

lemma(**in** *ring*) *nat-inc-zero*:
$[(0::nat)]\cdot \mathbf{1} = \mathbf{0}$
by *auto*

lemma(**in** *ring*) *nat-mult-zero*:
$[(0::nat)]\cdot x = \mathbf{0}$
by *simp*

lemma(**in** *ring*) *nat-inc-closed*:
fixes $n::nat$
shows $[n] \cdot \mathbf{1} \in carrier\ R$
by *simp*

lemma(**in** *ring*) *nat-mult-closed*:
fixes $n::nat$
assumes $x \in carrier\ R$
shows $[n] \cdot x \in carrier\ R$
by (*simp add*: *assms*)

lemma(**in** *ring*) *int-inc-closed*:
fixes $n::int$
shows $[n] \cdot \mathbf{1} \in carrier\ R$
by *simp*

lemma(**in** *ring*) *int-mult-closed*:
fixes $n::int$
assumes $x \in carrier\ R$
shows $[n] \cdot x \in carrier\ R$
by (*simp add*: *assms*)

lemma(**in** *ring*) *nat-inc-prod*:
fixes $n::nat$

fixes m::*nat*
shows $[m]\cdot([n] \cdot \mathbf{1}) = [(m*n)]\cdot\mathbf{1}$
by (*simp add*: *add.nat-pow-pow mult.commute*)

lemma(**in** *ring*) *nat-inc-prod'*:
fixes n::*nat*
fixes m::*nat*
shows $[(m*n)]\cdot\mathbf{1} = [m]\cdot\ \mathbf{1} \otimes ([n] \cdot \mathbf{1})$
by (*simp add*: *add.nat-pow-pow add-pow-rdistr*)

lemma(**in** *padic-integers*) *Zp-nat-inc-zero*:
shows $[(0::nat)] \cdot x = \mathbf{0}$
by *simp*

lemma(**in** *padic-integers*) *Zp-int-inc-zero'*:
shows $[(0::int)] \cdot x = \mathbf{0}$
using *Zp-nat-inc-zero*[*of x*]
unfolding *add-pow-def int-pow-def* **by** *auto*

lemma(**in** *padic-integers*) *Zp-nat-inc-closed*:
fixes n::*nat*
shows $[n] \cdot \mathbf{1} \in$ *carrier Zp*
by *simp*

lemma(**in** *padic-integers*) *Zp-nat-mult-closed*:
fixes n::*nat*
assumes $x \in$ *carrier Zp*
shows $[n] \cdot x \in$ *carrier Zp*
by (*simp add*: *assms*)

lemma(**in** *padic-integers*) *Zp-int-inc-closed*:
fixes n::*int*
shows $[n] \cdot \mathbf{1} \in$ *carrier Zp*
by *simp*

lemma(**in** *padic-integers*) *Zp-int-mult-closed*:
fixes n::*int*
assumes $x \in$ *carrier Zp*
shows $[n] \cdot x \in$ *carrier Zp*
by (*simp add*: *assms*)

The following lemmas give a concrete description of the inclusion of integers and natural numbers into $\mathbb{Z}_p$:

lemma(**in** *padic-integers*) *Zp-nat-inc-rep*:
fixes n::*nat*
shows $[n] \cdot \mathbf{1} = (\lambda\ m.\ p\text{-}residue\ m\ n)$
apply(*induction n*)
apply (*simp add*: *zero-rep*)
proof−

case (*Suc n*)
fix *n*
assume *A*: [*n*] · **1** = (λ*m*. *p-residue m* (*int n*))
then have *0*: [*Suc n*] · **1** = [*n*]·**1** ⊕ **1** **by** *auto*
show [*Suc n*] · **1** = (λ*m*. *p-residue m* (*Suc n*))
proof fix *m*
show ([*Suc n*] · **1**) *m* = *p-residue m* (*int* (*Suc n*))
proof(*cases m=0*)
case *True*
have *0*: ([*Suc n*] · **1**) *m* ∈ *carrier* (*Zp-res-ring m*)
using *Zp-nat-inc-closed padic-set-res-closed*
by (*simp add*: *residues-closed*)
then have ([*Suc n*] · **1**) *m* = *0*
using *p-res-ring-0 True* **by** *blast*
then show *?thesis*
by (*metis True p-res-ring-0′ p-residue-range′*)
next
case *False*
then have *R*: *residues* (*p*^*m*)
by (*simp add*: *prime residues-n*)
have ([*Suc n*] · **1**) *m* = ([*n*]·**1** ⊕ **1**) *m*
by (*simp add*: *0*)
then have *P0*: ([*Suc n*] · **1**) *m* = *p-residue m* (*int n*) $\oplus_{Zp\text{-}res\text{-}ring\ m}$ **1**$_{Zp\text{-}res\text{-}ring\ m}$
using *A False Zp-def padic-add-res padic-one-def Zp-defs*(*5*) *padic-integers.residue-of-one*(*1*) *padic-integers-axioms* **by** *auto*
then have *P1*:([*Suc n*] · **1**) *m* = *p-residue m* (*int n*) $\oplus_{Zp\text{-}res\text{-}ring\ m}$ *p-residue m* (*1*::*int*)
by (*metis R ext p-residue-alt-def residue-add-assoc residue-add-comm residue-plus-zero-r residue-times-one-r residue-times-zero-l residues.res-one-eq*)
have *P2*: *p-residue m* (*int n*) $\oplus_{Zp\text{-}res\text{-}ring\ m}$ *p-residue m 1* = ((*int n*) *mod* (*p*^*m*)) $\oplus_{Zp\text{-}res\text{-}ring\ m}$ *1*
using *R P0 P1 residue-def residues.res-one-eq*
by (*simp add*: *residues.res-one-eq p-residue-alt-def*)
have *P3*:((*int n*) *mod* (*p*^*m*)) $\oplus_{Zp\text{-}res\text{-}ring\ m}$ *1* = ((*int n*) + *1*) *mod* (*p*^*m*)
using *R residue-ring-def* **by** (*simp add*: *mod-add-left-eq*)
have *p-residue m* (*int n*) $\oplus_{Zp\text{-}res\text{-}ring\ m}$ *p-residue m 1* = (*int* (*Suc n*)) *mod* (*p*^*m*)
by (*metis P2 P3 add.commute of-nat-Suc p-residue-alt-def residue-add*)
then show *?thesis*
using *False R P1 p-residue-def p-residue-alt-def*
by *auto*
qed
qed
qed

lemma(**in** *padic-integers*) *Zp-nat-inc-res*:
fixes *n*::*nat*
shows ([*n*] · **1**) *k* = *n mod* (*p*^*k*)
using *Zp-nat-inc-rep p-residue-def*
by (*simp add*: *p-residue-alt-def*)

lemma(**in** *padic-integers*) *Zp-int-inc-rep*:
fixes *n*::*int*
shows [*n*] · **1** = (λ *m*. *p-residue m n*)
proof(*induction n*)
case (*nonneg n*)
then show *?case* **using** *Zp-nat-inc-rep*
by (*simp add*: *add-pow-int-ge*)
next
case (*neg n*)
show ⋀*n*. [(− *int* (*Suc n*))] · **1** = (λ*m*. *p-residue m* (− *int* (*Suc n*)))
proof
fix *n*
fix *m*
show ([(− *int* (*Suc n*))] · **1**) *m* = *p-residue m* (− *int* (*Suc n*))
proof−
have [(− *int* (*Suc n*))] · **1** = ⊖ ([(*int* (*Suc n*))] · **1**)
using *a-inv-def abelian-group.a-group add-pow-def cring.axioms*(*1*) *domain-def*
negative-zless-0 ring-def R.add.int-pow-neg R.one-closed **by** *blast*
then have ([(− *int* (*Suc n*))] · **1**) *m* = (⊖ ([(*int* (*Suc n*))] · **1**)) *m*
by *simp*
have **1** ∈ *carrier Zp*
using *cring.cring-simprules*(*6*) *domain-def* **by** *blast*
have ([(*int* (*Suc n*))] · **1**) = ([(*Suc n*)] · **1**)
by (*metis add-pow-def int-pow-int*)
then have ([(*int* (*Suc n*))] · **1**) ∈ *carrier Zp* **using** *Zp-nat-inc-closed*
by *simp*
then have *P0*: ([(− *int* (*Suc n*))] · **1**) *m* = ⊖*Zp-res-ring m* (([(*int* (*Suc n*))] · **1**) *m*)
using *Zp-def prime*
using ‹[(− *int* (*Suc n*))] · **1** = ⊖ ([*int* (*Suc n*)] · **1**)› *padic-integers.Zp-residue-a-inv*(*1*)
padic-integers-axioms **by** *auto*
have (([(*int* (*Suc n*))] · **1**) *m*) = (*p-residue m* (*Suc n*))
using *Zp-nat-inc-rep* **by** (*simp add*: *add-pow-int-ge*)
then have *P1*: ([(− *int* (*Suc n*))] · **1**) *m* = ⊖*Zp-res-ring m*(*p-residue m* (*Suc n*))
using *P0* **by** *auto*
have ⊖*Zp-res-ring m*(*p-residue m* (*Suc n*)) = *p-residue m* (− *int* (*Suc n*))
proof(*cases m=0*)
case *True*
then have *0*:⊖*Zp-res-ring m*(*p-residue m* (*Suc n*)) =⊖*Zp-res-ring 0*(*p-residue 0* (*Suc n*))

```
        by blast
      then have 1:⊖Zp-res-ring m(p-residue m (Suc n)) =⊖Zp-res-ring 0 (p-residue
1 (Suc n))
        by (metis p-res-ring-0' residue-a-inv-closed)
       then have 2:⊖Zp-res-ring m(p-residue m (Suc n)) =⊖Zp-res-ring 0 0
        by (metis p-res-ring-0' residue-a-inv-closed)
       then have 3:⊖Zp-res-ring m(p-residue m (Suc n)) =0
        using residue-1-prop p-res-ring-0' residue-a-inv-closed by presburger
       have 4: p-residue m (− int (Suc n)) ∈ carrier (Zp-res-ring 0)
        using p-res-ring-0 True residue-1-zero p-residue-range' by blast
       then show ?thesis
        using 3 True residue-1-zero
        by (simp add: p-res-ring-0')
      next
       case False
       then have R: residues (p^m)
        using padic-integers.p-residues padic-integers-axioms by blast
       have ⊖Zp-res-ring m p-residue m (int (Suc n)) = ⊖Zp-res-ring m (int (Suc
n)) mod (p^m)
        using R residue-def residues.neg-cong residues.res-neg-eq p-residue-alt-def

        by auto
       then have ⊖Zp-res-ring m p-residue m (int (Suc n)) = (−(int (Suc n)))
mod (p^m)
        using R residues.res-neg-eq by auto
       then show ?thesis
        by (simp add: p-residue-alt-def)
      qed
     then show ?thesis
      using P1 by linarith
   qed
  qed
qed

lemma(in padic-integers) Zp-int-inc-res:
  fixes n::int
  shows ([n] · 1) k = n mod (p^k)
  using Zp-int-inc-rep p-residue-def
  by (simp add: p-residue-alt-def)

abbreviation(in padic-integers)(input) p where
p ≡ [p] · 1

lemma(in padic-integers) p-natpow-prod:
p[^](n::nat) ⊗ p[^](m::nat) = p[^](n + m)
  by (simp add: R.nat-pow-mult)

lemma(in padic-integers) p-natintpow-prod:
  assumes (m::int) ≥ 0
```

shows p[^](*n*::*nat*) ⊗ p[^]*m* = p[^](*n* + *m*)
using *p-natpow-prod*[*of n nat m*] *assms int-pow-def*[*of Zp* p *m*] *int-pow-def*[*of Zp* p *n* + *m*]
by (*metis* (*no-types*, *lifting*) *int-nat-eq int-pow-int of-nat-add*)

lemma(**in** *padic-integers*) *p-intnatpow-prod*:
assumes (*n*::*int*) ≥ *0*
shows p[^]*n* ⊗ p[^](*m*::*nat*) = p[^](*m* + *n*)
using *p-natintpow-prod*[*of n m*] *assms mult-comm*[*of* p[^]*n* p[^]*m*]
by *simp*

lemma(**in** *padic-integers*) *p-int-pow-prod*:
assumes (*n*::*int*) ≥ *0*
assumes (*m*::*int*) ≥ *0*
shows p[^]*n* ⊗ p[^]*m* = p[^](*m* + *n*)
proof−
have *nat n* + *nat m* = *nat* (*n* + *m*)
using *assms*
by (*simp add*: *nat-add-distrib*)
then have p [^] (*nat n* + *nat m*) = p[^](*n* + *m*)
using *assms*
by (*simp add*: ‹*nat n* + *nat m* = *nat* (*n* + *m*)›)
then show *?thesis* **using** *assms p-natpow-prod*[*of nat n nat m*]
by (*smt* (*verit*) *pow-nat*)
qed

lemma(**in** *padic-integers*) *p-natpow-prod-Suc*:
p ⊗ p[^](*m*::*nat*) = p[^](*Suc m*)
p[^](*m*::*nat*) ⊗ p = p[^](*Suc m*)
using *R.nat-pow-Suc2* *R.nat-pow-Suc* **by** *auto*

lemma(**in** *padic-integers*) *power-residue*:
assumes *a* ∈ *carrier Zp*
assumes *k* > *0*
shows (*a*[^]$_{Zp}$ (*n*::*nat*)) *k* = (*a k*) ^ *n mod* (*p* ^ *k*)
apply(*induction n*)
using *p-residues assms*(*2*) *residue-of-one*(*1*) *residues.one-cong* **apply** *auto*[*1*]
by (*simp add*: *assms*(*1*) *mod-mult-left-eq power-commutes residue-of-prod'*)

24 The Valuation on $\mathbb{Z}_p$

24.1 The Integer-Valued and Extended Integer-Valued Valuations

fun *fromeint* :: *eint* ⇒ *int* **where**
fromeint (*eint x*) = *x*

The extended-integer-valued *p*-adic valuation on $\mathbb{Z}_p$:

definition(**in** *padic-integers*) *val-Zp* **where**

val-Zp = (λ*x*. (*if* (*x* = **0**) *then* (∞::*eint*) *else* (*eint* (*padic-val p x*))))

We also define an integer-valued valuation on the nonzero elements of $\mathbb{Z}_p$, for simplified reasoning

definition(**in** *padic-integers*) *ord-Zp* **where**
ord-Zp = *padic-val p*

Ord of additive inverse

lemma(**in** *padic-integers*) *ord-Zp-of-a-inv*:
assumes *a* ∈ *nonzero Zp*
shows *ord-Zp a* = *ord-Zp* (⊖*a*)
using *ord-Zp-def Zp-def assms*
padic-val-a-inv prime
by (*simp add*: *domain.nonzero-memE*(*1*) *padic-int-is-domain*)

lemma(**in** *padic-integers*) *val-Zp-of-a-inv*:
assumes *a* ∈ *carrier Zp*
shows *val-Zp a* = *val-Zp* (⊖*a*)
using *R.add.inv-eq-1-iff Zp-def assms padic-val-a-inv prime val-Zp-def* **by** *auto*

Ord-based criterion for being nonzero:

lemma(**in** *padic-integers*) *ord-of-nonzero*:
assumes *x* ∈*carrier Zp*
assumes *ord-Zp x* ≥*0*
shows *x* ≠ **0**
x ∈ *nonzero Zp*
proof−
show *x* ≠ **0**
proof
assume *x* = **0**
then have *ord-Zp x* = −*1*
using *ord-Zp-def padic-val-def Zp-def Zp-defs*(*2*) **by** *auto*
then show *False* **using** *assms*(*2*) **by** *auto*
qed
then show *x* ∈ *nonzero Zp*
using *nonzero-def assms*(*1*)
by (*simp add*: *nonzero-def*)
qed

lemma(**in** *padic-integers*) *not-nonzero-Zp*:
assumes *x* ∈ *carrier Zp*
assumes *x* ∉ *nonzero Zp*
shows *x* = **0**
using *assms*(*1*) *assms*(*2*) *nonzero-def* **by** *fastforce*

lemma(**in** *padic-integers*) *not-nonzero-Qp*:
assumes *x* ∈ *carrier* Q_p
assumes *x* ∉ *nonzero* Q_p
shows *x* = $\mathbf{0}_{Q_p}$

using *assms*(*1*) *assms*(*2*) *nonzero-def* **by** *force*

Relationship between val and ord

lemma(**in** *padic-integers*) *val-ord-Zp*:
assumes $a \neq \mathbf{0}$
shows *val-Zp a* = *eint* (*ord-Zp a*)
by (*simp add*: *assms ord-Zp-def val-Zp-def*)

lemma(**in** *padic-integers*) *ord-pos*:
assumes $x \in$ *carrier Zp*
assumes $x \neq \mathbf{0}$
shows *ord-Zp x* ≥ *0*
proof−
have *x* ≠*padic-zero p*
using *Zp-def assms*(*2*) *Zp-defs*(*2*) **by** *auto*
then have *ord-Zp x* = *int* (*LEAST k. x* (*Suc k*) ≠ $\mathbf{0}_{\textit{residue-ring}\ (p\hat{}\textit{Suc}\ k)}$)
using *ord-Zp-def padic-val-def* **by** *auto*
then show *?thesis*
by *linarith*
qed

lemma(**in** *padic-integers*) *val-pos*:
assumes $x \in$ *carrier Zp*
shows *val-Zp x* ≥ *0*
unfolding *val-Zp-def* **using** *assms*
by (*metis* (*full-types*) *eint-0 eint-ord-simps*(*1*) *eint-ord-simps*(*3*) *ord-Zp-def ord-pos*)

For passing between nat and int castings of ord

lemma(**in** *padic-integers*) *ord-nat*:
assumes $x \in$ *carrier Zp*
assumes $x \neq \mathbf{0}$
shows *int* (*nat* (*ord-Zp x*)) = *ord-Zp x*
using *ord-pos* **by** (*simp add*: *assms*(*1*) *assms*(*2*))

lemma(**in** *padic-integers*) *zero-below-ord*:
assumes $x \in$ *carrier Zp*
assumes *n* ≤ *ord-Zp x*
shows *x n* = *0*
proof−
have *x n* = $\mathbf{0}_{\textit{residue-ring}\ (p\hat{}n)}$
using *ord-Zp-def zero-below-val Zp-def assms*(*1*) *assms*(*2*) *prime padic-int-simps*(*5*)

by *auto*
then show *?thesis* **using** *residue-ring-def*
by *simp*
qed

lemma(**in** *padic-integers*) *zero-below-val-Zp*:
assumes $x \in$ *carrier Zp*

assumes $n \leq$ *val-Zp x*
shows *x n* = *0*
by (*metis assms*(*1*) *assms*(*2*) *eint-ord-simps*(*1*) *ord-Zp-def residue-of-zero*(*2*) *val-Zp-def zero-below-ord*)

lemma(**in** *padic-integers*) *below-ord-zero*:
assumes *x* ∈ *carrier Zp*
assumes *x* (*Suc n*) ≠ *0*
shows *n* ≥ *ord-Zp x*
proof−
have *0*: *x* ∈ *padic-set p*
using *Zp-def assms*(*1*) *Zp-defs*(*3*)
by *auto*
have *1*: *x* (*Suc n*) ≠ $\mathbf{0}_{residue\text{-}ring\ (p\hat{}(Suc\ n))}$
using *residue-ring-def assms*(*2*) **by** *auto*
have *of-nat n* ≥ (*padic-val p x*)
using *0 1 below-val-zero prime* **by** *auto*
then show *?thesis* **using** *ord-Zp-def* **by** *auto*
qed

lemma(**in** *padic-integers*) *below-val-Zp-zero*:
assumes *x* ∈ *carrier Zp*
assumes *x* (*Suc n*) ≠ *0*
shows *n* ≥ *val-Zp x*
by (*metis Zp-def assms*(*1*) *assms*(*2*) *eint-ord-simps*(*1*) *padic-integers.below-ord-zero padic-integers.residue-of-zero*(*2*) *padic-integers.val-ord-Zp padic-integers-axioms*)

lemma(**in** *padic-integers*) *nonzero-imp-ex-nonzero-res*:
assumes *x* ∈ *carrier Zp*
assumes *x* ≠ **0**
shows ∃ *k*. *x* (*Suc k*) ≠ *0*
proof−
have *0*: *x 0* = *0*
using *Zp-def assms*(*1*) *padic-int-simps*(*5*) *padic-set-zero-res prime* **by** *auto*
have ∃ *k*. *k* > *0* ∧ *x k* ≠ *0*
apply(*rule ccontr*) **using** *0 Zp-defs* **unfolding** *padic-zero-def*
by (*metis assms*(*2*) *ext neq0-conv*)
then show *?thesis*
using *not0-implies-Suc* **by** *blast*
qed

lemma(**in** *padic-integers*) *ord-suc-nonzero*:
assumes *x* ∈ *carrier Zp*
assumes *x* ≠**0**
assumes *ord-Zp x* = *n*
shows *x* (*Suc n*) ≠ *0*
proof−
obtain *k* **where** *k-def*: *x* (*Suc k*) ≠ *0*
using *assms*(*1*) *nonzero-imp-ex-nonzero-res assms*(*2*) **by** *blast*

then show *?thesis*
using *assms LeastI nonzero-imp-ex-nonzero-res* **unfolding** *ord-Zp-def padic-val-def*
by (*metis* (*mono-tags*, *lifting*) *Zp-defs*(*2*) *k-def of-nat-eq-iff padic-zero-def padic-zero-simp*(*1*))
qed

lemma(**in** *padic-integers*) *above-ord-nonzero*:
assumes $x \in$ *carrier Zp*
assumes $x \neq \mathbf{0}$
assumes $n >$ *ord-Zp x*
shows $x\ n \neq 0$
proof−
have *P0*: $n \geq$ (*Suc* (*nat* (*ord-Zp x*)))
by (*simp add*: *Suc-le-eq assms*(*1*) *assms*(*2*) *assms*(*3*) *nat-less-iff ord-pos*)
then have *P1*: *p-residue* (*Suc* (*nat* (*ord-Zp x*))) (*x n*) = *x* (*Suc* (*nat* (*ord-Zp x*)))
using *assms*(*1*) *p-residue-padic-int* **by** *blast*
then have *P2*: *p-residue* (*Suc* (*nat* (*ord-Zp x*))) (*x n*) ≠ *0*
using *Zp-def assms*(*1*) *assms*(*2*) *ord-nat padic-integers.ord-suc-nonzero*
padic-integers-axioms **by** *auto*
then show *?thesis*
using *P0 P1 assms*(*1*) *p-residue-padic-int*[*of x* (*Suc* (*nat* (*ord-Zp x*))) *n*]
p-residue-def
by (*metis ord-Zp-def padic-int-simps*(*2*) *padic-integers.zero-rep padic-integers-axioms*
padic-zero-simp(*2*))
qed

lemma(**in** *padic-integers*) *ord-Zp-geq*:
assumes $x \in$ *carrier Zp*
assumes $x\ n = 0$
assumes $x \neq \mathbf{0}$
shows *ord-Zp x* ≥ *n*
proof(*rule ccontr*)
assume ¬ *int n* ≤ *ord-Zp x*
then show *False* **using** *assms*
using *above-ord-nonzero* **by** *auto*
qed

lemma(**in** *padic-integers*) *ord-equals*:
assumes $x \in$ *carrier Zp*
assumes *x* (*Suc n*) ≠ *0*
assumes $x\ n = 0$
shows *ord-Zp x* = *n*
using *assms*(*1*) *assms*(*2*) *assms*(*3*) *below-ord-zero ord-Zp-geq residue-of-zero*(*2*)

by *fastforce*

lemma(**in** *padic-integers*) *ord-Zp-p*:
ord-Zp p = (*1*::*int*)
proof−

```
  have ord-Zp p = int 1
    apply(rule ord-equals[of p])
    using Zp-int-inc-res[of p] prime-gt-1-int prime by auto
  then show ?thesis
    by simp
qed

lemma(in padic-integers) ord-Zp-one:
ord-Zp 1 = 0
proof−
  have ord-Zp ([(1::int)]·1) = int 0
    apply(rule ord-equals)
    using Zp-int-inc-res[of 1] prime-gt-1-int prime by auto
  then show ?thesis
    by simp
qed
```

ord is multiplicative on nonzero elements of Zp

```
lemma(in padic-integers) ord-Zp-mult:
  assumes x ∈ nonzero Zp
  assumes y ∈ nonzero Zp
  shows (ord-Zp (x ⊗Zp y)) = (ord-Zp x) + (ord-Zp y)
  using val-prod[of p x y] prime assms Zp-defs Zp-def nonzero-memE(2) ord-Zp-def

        nonzero-closed nonzero-memE(2)
  by auto

lemma(in padic-integers) ord-Zp-pow:
  assumes x ∈ nonzero Zp
  shows ord-Zp (x[^](n::nat)) = n*(ord-Zp x)
proof(induction n)
  case 0
  have x[^](0::nat) = 1
    using assms(1) nonzero-def by simp
    then show ?case
    by (simp add: ord-Zp-one)
next
  case (Suc n)
  fix n
  assume IH: ord-Zp (x [^] n) = int n * ord-Zp x
  have N: (x [^] n) ∈ nonzero Zp
  proof−
    have ord-Zp x ≥ 0
      using assms
      by (simp add: nonzero-closed nonzero-memE(2) ord-pos)
      then have ord-Zp (x [^] n) ≥ 0
      using IH assms by simp
    then have 0: (x [^] n) ≠ 0
      using ord-of-nonzero(1) by force
```

```
    have 1: (x [^] n) ∈ carrier Zp
     by (simp add: nonzero-closed assms)
      then show ?thesis
      using 0 not-nonzero-Zp by blast
  qed
  have x[^](Suc n) = x ⊗(x[^]n)
    using nonzero-closed assms  R.nat-pow-Suc2 by blast
  then have ord-Zp (x[^](Suc n)) =(ord-Zp x) + ord-Zp (x[^]n)
   using N Zp-def assms padic-integers.ord-Zp-mult padic-integers-axioms by auto
  then have ord-Zp (x[^](Suc n)) =(ord-Zp x) +(int n * ord-Zp x)
    by (simp add: IH)
  then have ord-Zp (x[^](Suc n)) =(1*(ord-Zp x)) +(int n) * (ord-Zp x)
    by simp
  then have ord-Zp (x[^](Suc n)) =(1+ (int n)) * ord-Zp x
    by (simp add: comm-semiring-class.distrib)
  then show ord-Zp (x[^](Suc n)) = int (Suc n)*ord-Zp x
    by simp
qed

lemma(in padic-integers) val-Zp-pow:
  assumes x ∈ nonzero Zp
  shows val-Zp (x[^](n::nat)) = (n*(ord-Zp x))
 using Zp-def domain.nat-pow-nonzero[of Zp] domain-axioms nonzero-memE assms
ord-Zp-def
    padic-integers.ord-Zp-pow padic-integers-axioms val-Zp-def
    nonzero-memE(2)
  by fastforce

lemma(in padic-integers) val-Zp-pow':
  assumes x ∈ nonzero Zp
  shows val-Zp (x[^](n::nat)) = n*(val-Zp x)
 by (metis Zp-def assms not-nonzero-memI padic-integers.val-Zp-pow padic-integers.val-ord-Zp
padic-integers-axioms times-eint-simps(1))

lemma(in padic-integers) ord-Zp-p-pow:
ord-Zp (p[^](n::nat)) = n
 using ord-Zp-pow ord-Zp-p Zp-def Zp-nat-inc-closed ord-of-nonzero(2) padic-integers-axioms
int-inc-closed
       Zp-int-inc-closed by auto

lemma(in padic-integers) ord-Zp-p-int-pow:
  assumes n ≥0
  shows ord-Zp (p[^](n::int)) = n
  by (metis assms int-nat-eq int-pow-int ord-Zp-def ord-Zp-p-pow)

lemma(in padic-integers) val-Zp-p:
(val-Zp p) = 1
 using Zp-def ord-Zp-def padic-val-def val-Zp-def  ord-Zp-p Zp-defs(2) one-eint-def
  by auto
```

lemma(**in** *padic-integers*) *val-Zp-p-pow*:
val-Zp (p[^](*n*::*nat*)) = *eint n*
proof−
have (p[^](*n*::*nat*)) ≠ **0**
by (*metis mult-zero-l n-not-Suc-n of-nat-eq-iff ord-Zp-def ord-Zp-p-pow p-natpow-prod-Suc*(*1*))

then show *?thesis*
using *ord-Zp-p-pow* **by** (*simp add*: *ord-Zp-def val-Zp-def*)
qed

lemma(**in** *padic-integers*) *p-pow-res*:
assumes (*n*::*nat*) ≥ *m*
shows (p[^]*n*) *m* = *0*
by (*simp add*: *assms ord-Zp-p-pow zero-below-ord*)

lemma(**in** *padic-integers*) *p-pow-factor*:
assumes (*n*::*nat*) ≥ *m*
shows (*h* ⊗ (p[^]*n*)) *m* = *0* (*h* ⊗ (p[^]*n*)) *m* = **0**$_{Zp\text{-}res\text{-}ring\ n}$
using *assms p-pow-res p-res-ring-zero*
by(*auto simp*: *residue-of-zero Zp-residue-mult-zero*(*2*))

24.2 The Ultrametric Inequality

Ultrametric inequality for ord

lemma(**in** *padic-integers*) *ord-Zp-ultrametric*:
assumes *x* ∈ *nonzero Zp*
assumes *y* ∈ *nonzero Zp*
assumes *x* ⊕ *y* ∈ *nonzero Zp*
shows *ord-Zp* (*x* ⊕ *y*) ≥ *min* (*ord-Zp x*) (*ord-Zp y*)
unfolding *ord-Zp-def*
using *padic-val-ultrametric*[*of p x y*] *Zp-defs assms nonzero-memE Zp-def prime*
nonzero-closed nonzero-memE(*2*) **by** *auto*

Variants of the ultrametric inequality

lemma (**in** *padic-integers*) *ord-Zp-ultrametric-diff*:
assumes *x* ∈ *nonzero Zp*
assumes *y* ∈ *nonzero Zp*
assumes *x* ≠ *y*
shows *ord-Zp* (*x* ⊖ *y*) ≥ *min* (*ord-Zp x*) (*ord-Zp y*)
using *assms ord-Zp-ultrametric*[*of x* ⊖ *y*]
unfolding *a-minus-def*
by (*metis* (*no-types*, *lifting*) *R.a-transpose-inv R.add.inv-closed R.add.m-closed*
R.l-neg nonzero-closed ord-Zp-of-a-inv ord-of-nonzero(*2*) *ord-pos*)

lemma(**in** *padic-integers*) *ord-Zp-not-equal-imp-notequal*:
assumes *x* ∈ *nonzero Zp*
assumes *y* ∈ *nonzero Zp*
assumes *ord-Zp x* ≠ (*ord-Zp y*)

shows $x \neq y\ x \ominus y \neq \mathbf{0}\ x \oplus y \neq \mathbf{0}$
using *assms*
apply *blast*
using *nonzero-closed assms(1) assms(2) assms(3)* **apply** *auto[1]*
using *nonzero-memE assms*
using *R.minus-equality nonzero-closed*
Zp-def padic-integers.ord-Zp-of-a-inv
padic-integers-axioms **by** *auto*

lemma(**in** *padic-integers*) *ord-Zp-ultrametric-eq*:
assumes $x \in$ *nonzero Zp*
assumes $y \in$ *nonzero Zp*
assumes *ord-Zp* $x >$ (*ord-Zp* y)
shows *ord-Zp* $(x \oplus y) =$ *ord-Zp* y
proof−
have *0*: *ord-Zp* $(x \oplus y) \geq$ *ord-Zp* y
using *assms ord-Zp-not-equal-imp-notequal*[*of x y*]
ord-Zp-ultrametric[*of x y*] *nonzero-memE not-nonzero-Zp*
nonzero-closed **by** *force*
have *1*: *ord-Zp* $y \geq$ *min* (*ord-Zp*$(x \oplus y)$) (*ord-Zp* x)
proof−
have *0*: $x \oplus y \neq x$
using *assms nonzero-memE*
by (*simp add*: *nonzero-closed nonzero-memE(2)*)
have *1*: $x \oplus y \in$ *nonzero Zp*
using *ord-Zp-not-equal-imp-notequal*[*of x y*]
nonzero-closed assms(1) assms(2) assms(3)
not-nonzero-Zp **by** *force*
then show *?thesis*
using *0 assms(1) assms(2) assms(3) ord-Zp-ultrametric-diff*[*of* $x \oplus y\ x$]
by (*simp add*: *R.minus-eq nonzero-closed R.r-neg1 add-comm*)
qed
then show *?thesis*
using *0 assms(3)*
by *linarith*
qed

lemma(**in** *padic-integers*) *ord-Zp-ultrametric-eq′*:
assumes $x \in$ *nonzero Zp*
assumes $y \in$ *nonzero Zp*
assumes *ord-Zp* $x >$ (*ord-Zp* y)
shows *ord-Zp* $(x \ominus y) =$ *ord-Zp* y
using *assms ord-Zp-ultrametric-eq*[*of* $x \ominus y$]
unfolding *a-minus-def*
by (*metis R.add.inv-closed R.add.inv-eq-1-iff nonzero-closed not-nonzero-Zp ord-Zp-of-a-inv*)

lemma(**in** *padic-integers*) *ord-Zp-ultrametric-eq″*:
assumes $x \in$ *nonzero Zp*
assumes $y \in$ *nonzero Zp*

assumes *ord-Zp x* > (*ord-Zp y*)
shows *ord-Zp* (*y* ⊖ *x*) = *ord-Zp y*
by (*metis R.add.inv-closed R.minus-eq*
nonzero-closed Zp-def add-comm
assms(*1*) *assms*(*2*) *assms*(*3*)
ord-Zp-of-a-inv ord-of-nonzero(*2*)
ord-pos padic-integers.ord-Zp-ultrametric-eq padic-integers-axioms)

lemma(**in** *padic-integers*) *ord-Zp-not-equal-ord-plus-minus*:
assumes *x* ∈ *nonzero Zp*
assumes *y* ∈ *nonzero Zp*
assumes *ord-Zp x* ≠ (*ord-Zp y*)
shows *ord-Zp* (*x* ⊖ *y*) = *ord-Zp* (*x* ⊕ *y*)
apply(*cases ord-Zp x* > *ord-Zp y*)
using *assms*
apply (*simp add*: *ord-Zp-ultrametric-eq ord-Zp-ultrametric-eq'*)
using *assms nonzero-memI*
by (*smt* (*verit*) *add-comm ord-Zp-ultrametric-eq ord-Zp-ultrametric-eq''*)

val is multiplicative on nonzero elements

lemma(**in** *padic-integers*) *val-Zp-mult0*:
assumes *x* ∈ *carrier Zp*
assumes *x* ≠**0**
assumes *y* ∈ *carrier Zp*
assumes *y* ≠**0**
shows (*val-Zp* (*x* ⊗$_{Zp}$ *y*)) = (*val-Zp x*) + (*val-Zp y*)
apply(*cases x* ⊗$_{Zp}$ *y* = **0**)
using *assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *integral-iff val-ord-Zp ord-Zp-mult nonzero-memI*
apply (*simp add*: *integral-iff*)
using *assms ord-Zp-mult*[*of x y*] *val-ord-Zp*
by (*simp add*: *nonzero-memI*)

val is multiplicative everywhere

lemma(**in** *padic-integers*) *val-Zp-mult*:
assumes *x* ∈ *carrier Zp*
assumes *y* ∈ *carrier Zp*
shows (*val-Zp* (*x* ⊗$_{Zp}$ *y*)) = (*val-Zp x*) + (*val-Zp y*)
using *assms*(*1*) *assms*(*2*) *integral-iff val-ord-Zp ord-Zp-mult nonzero-memI val-Zp-mult0 val-Zp-def*
by *simp*

lemma(**in** *padic-integers*) *val-Zp-ultrametric0*:
assumes *x* ∈ *carrier Zp*
assumes *x* ≠**0**
assumes *y* ∈ *carrier Zp*
assumes *y* ≠**0**
assumes *x* ⊕ *y* ≠ **0**
shows *min* (*val-Zp x*) (*val-Zp y*) ≤ *val-Zp* (*x* ⊕ *y*)

apply(*cases* $x \oplus y = \mathbf{0}$)
using *assms* **apply** *blast*
using *assms ord-Zp-ultrametric*[*of x y*] *nonzero-memI val-ord-Zp*[*of x*] *val-ord-Zp*[*of y*] *val-ord-Zp*[*of* $x \oplus y$]
by *simp*

Unconstrained ultrametric inequality

lemma(**in** *padic-integers*) *val-Zp-ultrametric*:
assumes $x \in$ *carrier Zp*
assumes $y \in$ *carrier Zp*
shows *min* (*val-Zp x*) (*val-Zp y*) $\leq$ *val-Zp* ($x \oplus y$)
apply(*cases* $x = \mathbf{0}$)
apply (*simp add*: *assms*(*2*))
apply(*cases* $y = \mathbf{0}$)
apply (*simp add*: *assms*(*1*))
apply(*cases* $x \oplus y = \mathbf{0}$)
apply (*simp add*: *val-Zp-def*)
using *assms val-Zp-ultrametric0*[*of x y*]
by *simp*

Variants of the ultrametric inequality

lemma (**in** *padic-integers*) *val-Zp-ultrametric-diff*:
assumes $x \in$ *carrier Zp*
assumes $y \in$ *carrier Zp*
shows *val-Zp* ($x \ominus y$) $\geq$ *min* (*val-Zp x*) (*val-Zp y*)
using *assms val-Zp-ultrametric*[*of x* $\ominus y$] **unfolding** *a-minus-def*
by (*metis R.add.inv-closed R.add.inv-eq-1-iff nonzero-memI ord-Zp-def ord-Zp-of-a-inv val-Zp-def*)

lemma(**in** *padic-integers*) *val-Zp-not-equal-imp-notequal*:
assumes $x \in$ *carrier Zp*
assumes $y \in$ *carrier Zp*
assumes *val-Zp x* $\neq$ *val-Zp y*
shows $x \neq y$ $x \ominus y \neq \mathbf{0}$ $x \oplus y \neq \mathbf{0}$
using *assms*(*3*) **apply** *auto*[*1*]
using *assms*(*1*) *assms*(*2*) *assms*(*3*) *R.r-right-minus-eq* **apply** *blast*
by (*metis R.add.inv-eq-1-iff assms*(*1*) *assms*(*2*) *assms*(*3*) *R.minus-zero R.minus-equality*
not-nonzero-Zp ord-Zp-def ord-Zp-of-a-inv val-ord-Zp)

lemma(**in** *padic-integers*) *val-Zp-ultrametric-eq*:
assumes $x \in$ *carrier Zp*
assumes $y \in$ *carrier Zp*
assumes *val-Zp x* $>$ *val-Zp y*
shows *val-Zp* ($x \oplus y$) = *val-Zp y*
apply(*cases* $x \neq \mathbf{0} \wedge y \neq \mathbf{0} \wedge x \neq y$)
using *assms ord-Zp-ultrametric-eq*[*of x y*] *val-ord-Zp nonzero-memE*
using *not-nonzero-memE val-Zp-not-equal-imp-notequal*(*3*) **apply** *force*
unfolding *val-Zp-def*

using *assms*(*2*) *assms*(*3*) *val-Zp-def* **by** *force*

lemma(**in** *padic-integers*) *val-Zp-ultrametric-eq'*:
assumes $x \in$ *carrier Zp*
assumes $y \in$ *carrier Zp*
assumes *val-Zp* $x >$ (*val-Zp* y)
shows *val-Zp* $(x \ominus y) =$ *val-Zp* y
using *assms val-Zp-ultrametric-eq*[*of* $x \ominus y$]
unfolding *a-minus-def*
by (*metis R.add.inv-closed R.r-neg val-Zp-not-equal-imp-notequal*(*3*))

lemma(**in** *padic-integers*) *val-Zp-ultrametric-eq''*:
assumes $x \in$ *carrier Zp*
assumes $y \in$ *carrier Zp*
assumes *val-Zp* $x >$ (*val-Zp* y)
shows *val-Zp* $(y \ominus x) =$ *val-Zp* y
proof−
have *0*: $y \ominus x = \ominus (x \ominus y)$
using *assms*(*1*,*2*) **unfolding** *a-minus-def*
by (*simp add*: *R.add.m-comm R.minus-add*)
have *1*: *val-Zp* $(x \ominus y) =$ *val-Zp* y
using *assms val-Zp-ultrametric-eq'* **by** *blast*
have *2*: *val-Zp* $(x \ominus y) =$ *val-Zp* $(y \ominus x)$
unfolding *0* **unfolding** *a-minus-def*
by(*rule val-Zp-of-a-inv*, *rule R.ring-simprules*, *rule assms*, *rule R.ring-simprules*, *rule assms*)
show *?thesis* **using** *1* **unfolding** *2* **by** *blast*
qed

lemma(**in** *padic-integers*) *val-Zp-not-equal-ord-plus-minus*:
assumes $x \in$ *carrier Zp*
assumes $y \in$ *carrier Zp*
assumes *val-Zp* $x \neq$ (*val-Zp* y)
shows *val-Zp* $(x \ominus y) =$ *val-Zp* $(x \oplus y)$
by (*metis R.add.inv-closed R.minus-eq R.r-neg R.r-zero add-comm assms*(*1*) *assms*(*2*) *assms*(*3*) *not-nonzero-Zp ord-Zp-def ord-Zp-not-equal-ord-plus-minus val-Zp-def val-Zp-not-equal-imp-notequal*(*3*))

24.3 Units of $\mathbb{Z}_p$

Elements with valuation 0 in Zp are the units

lemma(**in** *padic-integers*) *val-Zp-0-criterion*:
assumes $x \in$ *carrier Zp*
assumes $x\ 1 \neq 0$
shows *val-Zp* $x = 0$
unfolding *val-Zp-def*
using *Zp-def assms*(*1*) *assms*(*2*) *ord-equals padic-set-zero-res prime*
by (*metis One-nat-def Zp-defs*(*3*) *of-nat-0 ord-Zp-def residue-of-zero*(*2*) *zero-eint-def*)

Units in Zp have val 0

lemma(**in** *padic-integers*) *unit-imp-val-Zp0*:
assumes *x* ∈ *Units Zp*
shows *val-Zp x* = *0*
apply(*rule val-Zp-0-criterion*)
apply (*simp add*: *R.Units-closed assms*)
using *assms residue-of-prod*[*of x inv x 1*] *residue-of-one*(*2*)[*of 1*] *R.Units-r-inv*[*of x*]
comm-monoid.UnitsI[*of R 1*] *p-res-ring-1-field*
by (*metis le-eq-less-or-eq residue-of-prod residue-times-zero-r zero-le-one zero-neq-one*)

Elements in Zp with ord 0 are units

lemma(**in** *padic-integers*) *val-Zp0-imp-unit0*:
assumes *val-Zp x* = *0*
assumes *x* ∈ *carrier Zp*
fixes *n*::*nat*
shows (*x* (*Suc n*)) ∈ *Units* (*Zp-res-ring* (*Suc n*))
unfolding *val-Zp-def*
proof−
have *p-res-ring*: *residues* (*p*^(*Suc n*))
using *p-residues* **by** *blast*
have ⋀ *n*. *coprime* (*x* (*Suc n*)) *p*
proof−
fix *n*
show *coprime* (*x* (*Suc n*)) *p*
proof−
have ¬ ¬ *coprime* (*x* (*Suc n*)) *p*
proof
assume ¬ *coprime* (*x* (*Suc n*)) *p*
then have *p dvd* (*x* (*Suc n*)) **using** *prime*
by (*meson coprime-commute prime-imp-coprime prime-nat-int-transfer*)
then obtain *k* **where** (*x* (*Suc n*)) = *k*∗*p*
by *fastforce*
then have *S*:*x* (*Suc n*) *mod p* = *0*
by *simp*
have *x 1* = *0*
proof−
have *Suc n* ≥ *1*
by *simp*
then have *x 1* = *p-residue 1* (*x* (*Suc n*))
using *p-residue-padic-int assms*(*2*) **by** *presburger*
then show *?thesis* **using** *S*
by (*simp add*: *p-residue-alt-def*)
qed
have *x* ≠**0**
proof−
have *ord-Zp x* ≠ *ord-Zp* **0**
using *Zp-def ord-Zp-def padic-val-def assms*(*1*) *ord-of-nonzero*(*1*) *R.zero-closed*

```
            Zp-defs(2) val-Zp-def
        by auto
      then show ?thesis
        by blast
    qed
    then have x 1 ≠ 0
      using assms(1) assms(2)  ord-suc-nonzero
      unfolding val-Zp-def
      by (simp add: ord-Zp-def zero-eint-def)
    then show False
      using ‹x 1 = 0› by blast
  qed
  then show ?thesis
    by auto
 qed
 then have ⋀ n. coprime (x (Suc n)) (p^(Suc n))
  by simp
 then have coprime (x (Suc n)) (p^(Suc n))
  by blast
 then show ?thesis using assms residues.res-units-eq  p-res-ring
  by (metis (no-types, lifting) mod-pos-pos-trivial p-residue-ring-car-memE(1)
        p-residue-ring-car-memE(2)  residues.m-gt-one  residues.mod-in-res-units
residues-closed)
qed

lemma(in padic-integers) val-Zp0-imp-unit0':
 assumes val-Zp x = 0
 assumes x ∈ carrier Zp
 assumes (n::nat) > 0
 shows (x n) ∈ Units (Zp-res-ring n)
 using assms val-Zp0-imp-unit0 gr0-implies-Suc by blast

lemma(in cring) ring-hom-Units-inv:
 assumes a ∈ Units R
 assumes cring S
 assumes h ∈ ring-hom R S
 shows h (inv a) = inv_S h a h a ∈ Units S
proof−
 have 0:h (inv a) ⊗_S h a = 1_S
  using assms Units-closed Units-inv-closed
  by (metis (no-types, lifting) Units-l-inv ring-hom-mult ring-hom-one)
 then show 1: h (inv a) = inv_S h a
  by (metis Units-closed Units-inv-closed assms(1) assms(2) assms(3) comm-monoid.is-invI(1)
cring-def ring-hom-closed)
 show h a ∈ Units S
  apply(rule comm-monoid.UnitsI[of S inv_S h a]) using 0 1 assms
  using cring.axioms(2) apply blast
   apply (metis 1 Units-inv-closed assms(1) assms(3) ring-hom-closed)
```

```
      apply (meson Units-closed assms(1) assms(3) ring-hom-closed)
    using 0 1 by auto
qed

lemma(in padic-integers) val-Zp-0-imp-unit:
  assumes val-Zp x = 0
  assumes x ∈ carrier Zp
  shows x ∈ Units Zp
proof−
  obtain y where y-def:  y= (λn. (if n=0 then 0 else (m-inv (Zp-res-ring n) (x
n))))
    by blast
  have 0:  ⋀m. m > 0 ⟹ y m = inv Zp-res-ring m (x m)
    using y-def by auto
  have 1:  ⋀m. m > 0 ⟹ inv Zp-res-ring m (x m) ∈ carrier (Zp-res-ring m)
  proof− fix m::nat  assume A: m > 0 then show inv Zp-res-ring m (x m) ∈
carrier (Zp-res-ring m)
    using assms val-Zp0-imp-unit0' monoid.Units-inv-closed[of Zp-res-ring m x m]

   by (smt (verit) One-nat-def Zp-def Zp-defs(2) cring.axioms(1) of-nat-0 ord-Zp-def

      padic-integers.R-cring padic-integers.ord-suc-nonzero padic-integers.val-Zp-0-criterion
padic-integers-axioms padic-val-def ring-def)
  qed
  have 2: y ∈ padic-set p
  proof(rule padic-set-memI)
    show 20: ⋀m. y m ∈ carrier (residue-ring (p ^ m))
    proof− fix m show y m ∈ carrier (residue-ring (p ^ m))
      apply(cases m = 0)
      using y-def 0[of m] 1[of m]
      by(auto simp: residue-ring-def y-def)
    qed
    show ⋀m n. m < n ⟹ residue (p ^ m) (y n) = y m
    proof− fix m n::nat assume A: m < n
      show residue (p ^ m) (y n) = y m
      proof(cases m = 0)
        case True
        then show ?thesis
          by (simp add: residue-1-zero y-def)
      next
        case False
        have hom: residue (p ^ m) ∈ ring-hom (Zp-res-ring n) (Zp-res-ring m)
          using A False prime residue-hom-p by auto
        have inv: y n = inv Zp-res-ring n x n using A
          by (simp add: False y-def)
        have unit: x n ∈ Units (Zp-res-ring n)
          using A False Zp-def assms(1) assms(2) val-Zp0-imp-unit0' prime
          by (metis gr0I gr-implies-not0)
        have F0: residue (p ^ m) (x n) = x m
```

using *A Zp-defs(3) assms(2) padic-set-res-coherent prime* **by** *auto*
have *F1*: *residue* $(p \hat{} m)$ $(y\ n) = inv_{Zp\text{-}res\text{-}ring\ m}\ x\ m$
using *F0 R-cring A hom inv unit cring.ring-hom-Units-inv*[*of Zp-res-ring n x n Zp-res-ring m residue* $(p \hat{} m)$]
False
by *auto*
then show *?thesis*
by (*simp add*: *False y-def*)
qed
qed
qed
have *3*: $y \otimes x = \mathbf{1}$
proof
fix *m*
show $(y \otimes x)\ m = \mathbf{1}\ m$
proof(*cases m=0*)
case *True*
then have *L*: $(y \otimes x)\ m\ = 0$
using *Zp-def 1 assms(2) Zp-residue-mult-zero(1) y-def*
by *auto*
have *R*: $\mathbf{1}\ m = 0$
by (*simp add*: *True cring.cring-simprules(6) domain.axioms(1) ord-Zp-one zero-below-ord*)
then show *?thesis* **using** *L R* **by** *auto*
next
case *False*
have *P*: $(y \otimes x)\ m = (y\ m) \otimes_{residue\text{-}ring\ (p\hat{}m)} (x\ m)$
using *Zp-def residue-of-prod* **by** *auto*
have $(y\ m) \otimes_{residue\text{-}ring\ (p\hat{}m)} (x\ m) = 1$
proof−
have $p\hat{}m > 1$
using *False prime prime-gt-1-int* **by** *auto*
then have *residues* $(p\hat{}m)$
using *less-imp-of-nat-less residues.intro* **by** *fastforce*
have *cring* (*residue-ring* $(p\hat{}m)$)
using *residues.cring* ‹*residues* $(p \hat{} m)$›
by *blast*
then have *M*: *monoid* (*residue-ring* $(p\hat{}m)$)
using *cring-def ring-def* **by** *blast*
have *U*: $(x\ m) \in$ *Units* (*residue-ring* $(p\hat{}m)$)
using *False Zp-def assms(1) assms(2) padic-integers.val-Zp0-imp-unit0' padic-integers-axioms* **by** *auto*
have *I*: $y\ m = m\text{-}inv$ (*residue-ring* $(p\hat{}m)$) $(x\ m)$
by (*simp add*: *False y-def*)
have $(y\ m) \otimes_{residue\text{-}ring\ (p\hat{}m)} (x\ m) = \mathbf{1}_{residue\text{-}ring\ (p\hat{}m)}$
using *M U I* **by** (*simp add*: *monoid.Units-l-inv*)
then show *?thesis*
using *residue-ring-def* **by** *simp*
qed

```
      then show ?thesis
        using P Zp-def False residue-of-one(2) by auto
    qed
  qed
  have 4: y ∈ carrier Zp
    using 2 Zp-defs by auto
  show ?thesis
    apply(rule R.UnitsI[of y])
    using assms 4 3 by auto
qed
```

Definition of ord on a fraction is independent of the choice of representatives

```
lemma(in padic-integers) ord-Zp-eq-frac:
  assumes a ∈ nonzero Zp
  assumes b ∈ nonzero Zp
  assumes c ∈ nonzero Zp
  assumes d ∈ nonzero Zp
  assumes a ⊗ d = b ⊗ c
  shows (ord-Zp a) − (ord-Zp b) = (ord-Zp c) − (ord-Zp d)
proof−
  have ord-Zp (a ⊗ d) = ord-Zp (b ⊗ c)
    using assms
    by presburger
  then have (ord-Zp a) + (ord-Zp  d) = (ord-Zp b) + (ord-Zp c)
    using assms(1) assms(2) assms(3) assms(4) ord-Zp-mult by metis
  then show ?thesis
    by linarith
qed

lemma(in padic-integers) val-Zp-eq-frac-0:
  assumes a ∈ nonzero Zp
  assumes b ∈ nonzero Zp
  assumes c ∈ nonzero Zp
  assumes d ∈ nonzero Zp
  assumes a ⊗ d = b ⊗ c
  shows (val-Zp a) − (val-Zp b) = (val-Zp c) − (val-Zp d)
proof−
  have 0:(val-Zp a) − (val-Zp b) = (ord-Zp a) − (ord-Zp b)
    using assms nonzero-memE Zp-defs(2) ord-Zp-def val-Zp-def by auto
  have 1: (val-Zp c) − (val-Zp d) = (ord-Zp c) − (ord-Zp d)
    using assms nonzero-memE   val-ord-Zp[of c] val-ord-Zp[of d]
    by (simp add: nonzero-memE(2))
  then show ?thesis
    using 0 assms(1) assms(2) assms(3) assms(4) assms(5) ord-Zp-eq-frac
    by presburger
qed
```

25 Angular Component Maps on $\mathbb{Z}_p$

The angular component map on $\mathbb{Z}_p$ is just the map which normalizes a point $x \in \mathbb{Z}_p$ by mapping it to a point with valuation 0. It is explicitly defined as the mapping $x \mapsto p^{-ord(p)} * x$ for nonzero x, and $0 \mapsto 0$. By composing these maps with reductions mod p^n we get maps which are equal to the standard residue maps on units of $\mathbb{Z}_p$, but in general unequal elsewhere. Both the angular component map and the angular component map mod p^n are homomorpshims from the multiplicative group of units of $\mathbb{Z}_p$ to the multiplicative group of units of the residue rings, and play a key role in first-order model-theoretic formalizations of the p-adics (see, for example [6], or [2]).

lemma(**in** *cring*) *int-nat-pow-rep*:
[(*k*::*int*)]·**1** [^] (*n*::*nat*) = [(*k*^*n*)]·**1**
apply(*induction n*)
by (*auto simp*: *add.int-pow-pow add-pow-rdistr-int mult.commute*)

lemma(**in** *padic-integers*) *p-pow-rep0*:
fixes *n*::*nat*
shows p[^]*n* = [(*p*^*n*)]·**1**
using *R.int-nat-pow-rep* **by** *auto*

lemma(**in** *padic-integers*) *p-pow-nonzero*:
shows (p[^](*n*::*nat*)) ∈ *carrier Zp*
(p[^](*n*::*nat*)) ≠ **0**
apply *simp*
using *Zp-def nat-pow-nonzero domain-axioms nonzero-memE int-inc-closed ord-Zp-p*

padic-integers.ord-of-nonzero(*2*) *padic-integers-axioms Zp-int-inc-closed*
nonzero-memE(*2*)
by (*metis ord-of-nonzero*(*2*) *zero-le-one*)

lemma(**in** *padic-integers*) *p-pow-nonzero′*:
shows (p[^](*n*::*nat*)) ∈ *nonzero Zp*
using *nonzero-def p-pow-nonzero*
by (*simp add*: *nonzero-def*)

lemma(**in** *padic-integers*) *p-pow-rep*:
fixes *n*::*nat*
shows (p[^]*n*) *k* = (*p*^*n*) *mod* (*p*^*k*)
by (*simp add*: *R.int-nat-pow-rep Zp-int-inc-res*)

lemma(**in** *padic-integers*) *p-pow-car*:
assumes (*n*::*int*)≥ *0*
shows (p[^]*n*) ∈ *carrier Zp*
proof−
have (p[^]*n*) = (p[^](*nat n*))
by (*metis assms int-nat-eq int-pow-int*)

then show *?thesis*
by *simp*
qed

lemma(**in** *padic-integers*) *p-int-pow-nonzero*:
assumes (*n*::*int*) ≥*0*
shows (p[^]*n*) ∈ *nonzero Zp*
by (*metis assms not-nonzero-Zp ord-Zp-p-int-pow ord-of-nonzero*(*1*) *p-pow-car*)

lemma(**in** *padic-integers*) *p-nonzero*:
shows p ∈ *nonzero Zp*
using *p-int-pow-nonzero*[*of 1*]
by (*simp add*: *ord-Zp-p ord-of-nonzero*(*2*))

Every element of Zp is a unit times a power of p.

lemma(**in** *padic-integers*) *residue-factor-unique*:
assumes *k*>*0*
assumes *x* ∈ *carrier Zp*
assumes *u* ∈ *carrier* (*Zp-res-ring k*) ∧ (*u*∗ *p*^*m*) = (*x* (*m*+*k*))
shows *u* = (*THE u*. *u* ∈ *carrier* (*Zp-res-ring k*) ∧ (*u*∗ *p*^*m*) = (*x* (*m*+*k*)))
proof−
obtain *P* **where**
P-def: *P* = (λ *u*. *u* ∈ *carrier* (*Zp-res-ring k*) ∧ (*u*∗ *p*^*m*) = (*x* (*m*+*k*)))
by *simp*
have *0*: *P u*
using *P-def assms*(*3*) **by** *blast*
have *1*: ⋀ *v*. *P v* ⟹ *v* = *u*
by (*metis P-def assms*(*3*) *mult-cancel-right*
not-prime-0 power-not-zero prime)
have *u* = (*THE u*. *P u*)
by (*metis 0 1 the-equality*)
then show *?thesis* **using** *P-def*
by *blast*
qed

lemma(**in** *padic-integers*) *residue-factor-exists*:
assumes *m* = *nat* (*ord-Zp x*)
assumes *k* > *0*
assumes *x* ∈ *carrier Zp*
assumes *x* ≠**0**
obtains *u* **where** *u* ∈ *carrier* (*Zp-res-ring k*) ∧ (*u*∗ *p*^*m*) = (*x* (*m*+*k*))
proof−
have *X0*: (*x* (*m*+*k*)) ∈ *carrier* (*Zp-res-ring* (*m*+*k*))
using *Zp-def assms*(*3*) *padic-set-res-closed residues-closed*
by *blast*
then have *X1*: (*x* (*m*+*k*)) ≥ *0*
using *p-residues assms*(*2*) *residues.res-carrier-eq* **by** *simp*
then have *X2*: (*x* (*m*+*k*)) > *0*
using *assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *above-ord-nonzero*

```
  by (metis add.right-neutral add-cancel-right-right
      add-gr-0 int-nat-eq less-add-same-cancel1
      less-imp-of-nat-less not-gr-zero of-nat-0-less-iff of-nat-add ord-pos)
 have 0: x m = 0
  using  Zp-def assms(1) assms(3) zero-below-val  ord-nat zero-below-ord[of x m]

        assms(4) ord-Zp-def by auto
 then have 1: x (m +k) mod p^m = 0
  using assms(2) assms(3) p-residue-padic-int residue-def
  by (simp add: p-residue-alt-def)
 then have ∃ u.  u*(p^m) = (x (m+k))
  by auto
then obtain u where U0:  u*(p^m) = (x (m+k))
 by blast
 have I: (p^m) > 0
  using prime
  by (simp add: prime-gt-0-int)
 then have U1: (u* p^m) = (x (m+k))
  by (simp add: U0)
 have U2: u ≥ 0
  using I U1 X1
  by (metis U0 less-imp-triv mult.right-neutral mult-less-cancel-left
      of-nat-zero-less-power-iff power.simps(1) times-int-code(1))
 have X3: (x (m+k)) < p^(m+k)
  using assms(3) X0  p-residues assms(2) residues.res-carrier-eq by auto
 have U3: u < p^k
 proof(rule ccontr)
  assume ¬ u < (p ^ k)
  then have (p^k) ≤ u
   by simp
  then have  (p^k * p^m) ≤ u*(p^m)
   using I by simp
  then have p^(m + k) ≤ (x (m+k))
   by (simp add: U0 add.commute semiring-normalization-rules(26))
  then show False
   using X3 by linarith
 qed
 then have u ∈ carrier (Zp-res-ring k)
  using assms(2)  p-residues residues.res-carrier-eq U3 U2 by auto
 then show ?thesis using U1  that by blast
qed

definition(in padic-integers) normalizer where
normalizer m x = (λk. if (k=0) then 0 else (THE u. u ∈ carrier (Zp-res-ring k)
∧ (u* p^m) = (x (m + k)) ) )

definition(in padic-integers) ac-Zp where
ac-Zp x = normalizer (nat (ord-Zp x)) x
```

lemma(**in** *padic-integers*) *ac-Zp-equation*:
assumes $x \in$ *carrier Zp*
assumes $x \neq \mathbf{0}$
assumes $k > 0$
assumes $m =$ *nat* (*ord-Zp x*)
shows (*ac-Zp x k*) $\in$ *carrier* (*Zp-res-ring k*) $\wedge$ (*ac-Zp x k*)$*$($p\hat{}m$) = (*x* (*m*+*k*))
proof−
have *K0*: $k > 0$
using *assms nat-neq-iff* **by** *blast*
have *KM*: $m + k > m$
using *assms*(*3*) *assms*(*4*) **by** *linarith*
obtain *u* **where** *U0*: $u \in$ *carrier* (*Zp-res-ring k*) $\wedge$ ($u*$ $p\hat{}m$) = (*x* (*m*+*k*))
using *assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *residue-factor-exists* **by** *blast*
have *RHS*: *ac-Zp x k* = (*THE u*. *u* $\in$ *carrier* (*Zp-res-ring k*) $\wedge$ $u*$($p\hat{}m$) = (*x* (*m*+*k*)))
proof−
have *K*: $k \neq 0$
by (*simp add*: *K0*)
have *ac-Zp x k* = *normalizer* (*nat* (*ord-Zp x*)) *x k*
using *ac-Zp-def* **by** *presburger*
then have *ac-Zp x k* = *normalizer m x k*
using *assms* **by** *blast*
then show *ac-Zp x k* = (*THE u*. *u* $\in$ *carrier* (*Zp-res-ring k*) $\wedge$ ($u*$ $p\hat{}m$) = (*x* (*m* + *k*)))
using *K* **unfolding** *normalizer-def p-residue-def*
by *simp*
qed
have *LHS*:*u* = (*THE u*. *u* $\in$ *carrier* (*Zp-res-ring k*) $\wedge$ $u*$($p\hat{}m$) = (*x* (*m*+*k*)))
using *assms U0 K0 assms*(*1*) *residue-factor-unique*[*of k x u m*] **by** *metis*
then have *u* = *ac-Zp x k*
by (*simp add*: *RHS*)
then show *?thesis* **using** *U0* **by** *auto*
qed

lemma(**in** *padic-integers*) *ac-Zp-res*:
assumes $m > k$
assumes $x \in$ *carrier Zp*
assumes $x \neq \mathbf{0}$
shows *p-residue k* (*ac-Zp x m*) = (*ac-Zp x k*)
proof(*cases k* = *0*)
case *True*
then show *?thesis*
unfolding *ac-Zp-def normalizer-def*
by (*meson p-res-ring-0′ p-residue-range′*)
next
case *False*
obtain *n* **where** *n-def*: *n* = *nat* (*ord-Zp x*)
by *blast*
have *K0*: $k > 0$ **using** *False* **by** *simp*

```
  obtain uk where Uk0: uk = (ac-Zp x k)
    by simp
  obtain um where Um0: um = (ac-Zp x m)
    by simp
  have Uk1: uk ∈ carrier (Zp-res-ring k) ∧ uk*(p^n) = (x (n+k))
    using K0 Uk0 ac-Zp-equation assms(2) assms(3) n-def by metis
  have Um1: um ∈ carrier (Zp-res-ring m) ∧ um*(p^n) = (x (n+m))
    using Uk1 Um0 ac-Zp-equation assms(1) assms(3) n-def assms(2)
    by (metis neq0-conv not-less0)
  have um mod (p^k) = uk
  proof−
    have (x (n+m)) mod (p^(n + k)) = (x (n+k))
      using assms(1) assms(3) p-residue-padic-int p-residue-def n-def
      by (simp add: assms(2) p-residue-alt-def)
    then have (p^(n + k)) dvd (x (n+m)) − (x (n+k))
      by (metis dvd-minus-mod)
    then obtain d where (x (n+m)) − (x (n+k)) = (p^(n+k))*d
      using dvd-def by blast
    then have ((um*(p^n)) − (uk*(p^n))) =  p^(n+k)*d
      using Uk1 Um1 by auto
    then have ((um − uk)*(p^n)) =  p^(n+k)*d
      by (simp add: left-diff-distrib)
    then have ((um − uk)*(p^n)) =  ((p^k)*d)*(p^n)
      by (simp add: power-add)
    then have (um − uk) =  ((p^k)*d)
      using prime by auto
    then have um mod p^k = uk mod p^k
      by (simp add: mod-eq-dvd-iff)
    then show ?thesis  using Uk1
    by (metis mod-pos-pos-trivial p-residue-ring-car-memE(1) p-residue-ring-car-memE(2))

  qed
  then show ?thesis
    by (simp add: Uk0 Um0 p-residue-alt-def)
qed

lemma(in padic-integers) ac-Zp-in-Zp:
  assumes x ∈ carrier Zp
  assumes x ≠0
  shows ac-Zp x ∈ carrier Zp
proof−
  have ac-Zp x ∈ padic-set p
  proof(rule padic-set-memI)
    show ⋀m. ac-Zp x m ∈ carrier (residue-ring (p ^ m))
    proof−
      fix m
      show ac-Zp x m ∈ carrier (residue-ring (p ^ m))
      proof(cases m = 0)
        case True
```

```
      then have ac-Zp x m = 0
        unfolding ac-Zp-def normalizer-def by auto
      then show ?thesis
        by (simp add: True residue-ring-def)
    next
      case False
      then have m>0
        by blast
      then show ?thesis
        using ac-Zp-equation
        by (metis assms(1) assms(2))
    qed
  qed
  show ⋀m n. m < n ⟹ residue (p ^ m) (ac-Zp x n) = ac-Zp x m
    using ac-Zp-res
    by (simp add: assms(1) assms(2) p-residue-def)
 qed
 then show ?thesis
  by (simp add: Zp-defs(3))
qed

lemma(in padic-integers) ac-Zp-is-Unit:
 assumes x ∈ carrier Zp
 assumes x ≠0
 shows ac-Zp x ∈ Units Zp
proof(rule val-Zp-0-imp-unit)
 show ac-Zp x ∈ carrier Zp
  by (simp add: ac-Zp-in-Zp assms(1) assms(2))
 obtain m where M: m = nat (ord-Zp x)
  by blast
 have AC1: (ac-Zp x 1)*(p^m) = (x (m+1))
  using M ac-Zp-equation assms(1) assms(2)
  by (metis One-nat-def lessI)
 have (x (m+1)) ≠0
  using M assms
  by (metis Suc-eq-plus1 Suc-le-eq nat-int nat-mono nat-neq-iff ord-Zp-geq)
 then have (ac-Zp x 1) ≠ 0
  using AC1 by auto
 then show val-Zp (ac-Zp x) = 0
  using ‹ac-Zp x ∈ carrier Zp› val-Zp-0-criterion
  by blast
qed
```

The typical defining equation for the angular component map.

```
lemma(in padic-integers) ac-Zp-factors-x:
 assumes x ∈ carrier Zp
 assumes x ≠0
 shows x = (p[^](nat (ord-Zp x))) ⊗ (ac-Zp x) x = (p[^](ord-Zp x)) ⊗ (ac-Zp x)
proof−
```

```
  show x = (p[^](nat (ord-Zp x)))⊗ (ac-Zp x)
  proof
    fix k
    show x k = ((p[^](nat (ord-Zp x))) ⊗ (ac-Zp x)) k
    proof(cases k=0)
      case True
      then show ?thesis
        using Zp-def Zp-defs(3) Zp-residue-mult-zero(1) ac-Zp-in-Zp
         assms(1) assms(2) mult-comm padic-set-zero-res prime by auto
    next
      case False
      show ?thesis
      proof(cases k ≤ ord-Zp x)
      case True
      have 0: x k = 0
        using True assms(1) zero-below-ord by blast
      have 1: (p[^](nat (ord-Zp x))) k = 0
          using True assms(1) assms(2) ord-Zp-p-pow ord-nat p-pow-nonzero(1)
zero-below-ord
        by presburger
       have ((p[^](nat (ord-Zp x))) ⊗ (ac-Zp x)) k = (p[^](nat (ord-Zp x))) k *
(ac-Zp x) k mod p^k
        using Zp-def padic-mult-res residue-ring-def
        using residue-of-prod' by blast
      then have ((p[^](nat (ord-Zp x))) ⊗ (ac-Zp x)) k = 0
        by (simp add: 1)
      then show ?thesis using 0
        by metis
      next
      case False
      obtain n where N: n = nat (ord-Zp x)
        by metis
      obtain m where M0: k = n + m
        using False N le-Suc-ex ord-Zp-def by fastforce
      have M1: m >0
        using M0 False N assms(1) assms(2) ord-nat
        by (metis Nat.add-0-right gr0I le-refl less-eq-int-code(1)
            nat-eq-iff2 neq0-conv of-nat-eq-0-iff of-nat-mono)
      have E1: (ac-Zp x m)*(p^n) = (x k)
        using M0 M1 N ac-Zp-equation assms(1) assms(2) by blast
      have E2: (ac-Zp x k)*(p^n) = (x (n + k))
        using M0 M1 N ac-Zp-equation assms(1) assms(2) add-gr-0
        by presburger
       have E3: ((ac-Zp x k) mod (p^k))*((p^n) mod p^k) mod (p^k) = (x (n +
k)) mod (p^k)
        by (metis E2 mod-mult-left-eq mod-mult-right-eq)
      have E4: ((ac-Zp x k) mod (p^k))*(p^n) mod (p^k) = (x k)
       using E2 assms(1) le-add2 mod-mult-left-eq p-residue-padic-int p-residue-def
```

```
    by (metis Zp-int-inc-rep Zp-int-inc-res)

   have E5: (ac-Zp x k)*((p^n) mod p^k) mod (p^k) = (x k)
      using E2 assms(1) p-residue-padic-int p-residue-def by (metis E3 E4
mod-mult-left-eq)
   have E6: (ac-Zp x k) ⊗_(Zp-res-ring k) ((p^n) mod p^k)  = (x k)
    using E5 M0 M1 p-residues  residues.res-mult-eq by auto
   have E7:  ((p^n) mod p^k) ⊗_(Zp-res-ring k)(ac-Zp x k)   = (x k)
    by (simp add: E6 residue-mult-comm)
   have E8: ((p[^](nat (ord-Zp x))) k) ⊗_(Zp-res-ring k) (ac-Zp x k) = (x k)
    using E7 N p-pow-rep
    by metis
   then show ?thesis
    by (simp add: residue-of-prod)
   qed
  qed
  then show x = (p[^](ord-Zp x)) ⊗ (ac-Zp x)
   by (metis assms(1) assms(2) int-pow-int ord-nat)
qed

lemma(in padic-integers) ac-Zp-factors':
  assumes x ∈ nonzero Zp
  shows x = [p] · 1 [^] ord-Zp x ⊗ ac-Zp x
  using assms nonzero-memE
  by (simp add: nonzero-closed nonzero-memE(2) ac-Zp-factors-x(2))

lemma(in padic-integers) ac-Zp-mult:
  assumes x ∈ nonzero Zp
  assumes y ∈ nonzero Zp
  shows ac-Zp (x ⊗ y) = (ac-Zp x) ⊗ (ac-Zp y)
proof−
  have P0: x = (p[^](nat (ord-Zp x))) ⊗ (ac-Zp x)
   using  nonzero-memE ac-Zp-factors-x assms(1)
   by (simp add: nonzero-closed nonzero-memE(2))
  have P1: y = (p[^](nat (ord-Zp y))) ⊗ (ac-Zp y)
   using  nonzero-memE ac-Zp-factors-x assms(2)
   by (simp add: nonzero-closed nonzero-memE(2))

  have x ⊗ y = (p[^](nat (ord-Zp (x ⊗ y)))) ⊗ (ac-Zp (x ⊗ y))
  proof−
   have x ⊗ y ∈ nonzero Zp
    by (simp add:  assms(1) assms(2) nonzero-mult-closed)
     then show ?thesis
    using nonzero-closed nonzero-memE(2) Zp-def
     padic-integers.ac-Zp-factors-x(1) padic-integers-axioms
    by blast
  qed
  then have x ⊗ y =  (p[^](nat ((ord-Zp x) + (ord-Zp y)))) ⊗ (ac-Zp (x ⊗ y))
```

using *assms ord-Zp-mult*[*of x y*]
by (*simp add*: *Zp-def*)
then have $x \otimes y = (\mathrm{p}[\hat{}](nat\ (ord\text{-}Zp\ x)) + nat\ (ord\text{-}Zp\ y))) \otimes (ac\text{-}Zp\ (x \otimes y))$
using *nonzero-closed nonzero-memE*(*2*) *assms*(*1*) *assms*(*2*)
nat-add-distrib ord-pos **by** *auto*
then have $x \otimes y = (\mathrm{p}[\hat{}](nat\ (ord\text{-}Zp\ x))) \otimes (\mathrm{p}[\hat{}](nat(ord\text{-}Zp\ y))) \otimes (ac\text{-}Zp\ (x \otimes y))$
using *p-natpow-prod*
by *metis*
then have *P2*: $(\mathrm{p}[\hat{}](nat\ (ord\text{-}Zp\ x))) \otimes (\mathrm{p}[\hat{}](nat(ord\text{-}Zp\ y))) \otimes (ac\text{-}Zp\ (x \otimes y))$
$= ((\mathrm{p}[\hat{}](nat\ (ord\text{-}Zp\ x))) \otimes (ac\text{-}Zp\ x)) \otimes ((\mathrm{p}[\hat{}](nat\ (ord\text{-}Zp\ y))) \otimes (ac\text{-}Zp\ y))$
using *P0 P1*
by *metis*
have $(\mathrm{p}[\hat{}](nat\ (ord\text{-}Zp\ x))) \otimes (\mathrm{p}[\hat{}](nat(ord\text{-}Zp\ y))) \otimes (ac\text{-}Zp\ (x \otimes y))$
$= ((\mathrm{p}[\hat{}](nat\ (ord\text{-}Zp\ x))) \otimes ((\mathrm{p}[\hat{}](nat\ (ord\text{-}Zp\ y))) \otimes (ac\text{-}Zp\ x)) \otimes (ac\text{-}Zp\ y))$
by (*metis P0 P1 Zp-def* ‹$x \otimes y = [p] \cdot \mathbf{1}\ [\hat{}]\ nat\ (ord\text{-}Zp\ x) \otimes [p] \cdot \mathbf{1}\ [\hat{}]\ nat\ (ord\text{-}Zp\ y) \otimes ac\text{-}Zp\ (x \otimes y)$›
mult-comm padic-integers.mult-assoc padic-integers-axioms)
then have $((\mathrm{p}[\hat{}](nat\ (ord\text{-}Zp\ x))) \otimes (\mathrm{p}[\hat{}](nat(ord\text{-}Zp\ y)))) \otimes (ac\text{-}Zp\ (x \otimes y))$
$=((\mathrm{p}[\hat{}](nat\ (ord\text{-}Zp\ x))) \otimes (\mathrm{p}[\hat{}](nat(ord\text{-}Zp\ y)))) \otimes ((ac\text{-}Zp\ x) \otimes (ac\text{-}Zp\ y))$
using *Zp-def mult-assoc* **by** *auto*
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *R.m-closed*
‹$x \otimes y = [p] \cdot \mathbf{1}\ [\hat{}]\ nat\ (ord\text{-}Zp\ x) \otimes [p] \cdot \mathbf{1}\ [\hat{}]\ nat\ (ord\text{-}Zp\ y) \otimes ac\text{-}Zp\ (x \otimes y)$›
ac-Zp-in-Zp assms(*1*) *assms*(*2*) *integral-iff m-lcancel*
nonzero-closed nonzero-memE(*2*) *p-pow-nonzero*(*1*))
qed

lemma(**in** *padic-integers*) *ac-Zp-one*:
$ac\text{-}Zp\ \mathbf{1} = \mathbf{1}$
by (*metis R.one-closed Zp-def ac-Zp-factors-x*(*2*) *int-pow-0 ord-Zp-one padic-integers.ac-Zp-in-Zp padic-integers-axioms padic-one-id prime zero-not-one*)

lemma(**in** *padic-integers*) *ac-Zp-inv*:
assumes $x \in Units\ Zp$
shows $ac\text{-}Zp\ (inv_{Zp}\ x) = inv_{Zp}\ (ac\text{-}Zp\ x)$
proof−
have $x \otimes (inv_{Zp}\ x) = \mathbf{1}$
using *assms* **by** *simp*
then have $(ac\text{-}Zp\ x) \otimes (ac\text{-}Zp\ (inv_{Zp}\ x)) = ac\text{-}Zp\ \mathbf{1}$
using *ac-Zp-mult*[*of x* (*inv x*)] *R.Units-nonzero*
assms zero-not-one **by** *auto*
then show *?thesis*

using *R.invI(2)[of (ac-Zp x) (ac-Zp (inv$_{Zp}$ x))] assms ac-Zp-in-Zp ac-Zp-one*
by (*metis* (*no-types*, *lifting*) *R.Units-closed R.Units-inv-closed*
R.Units-r-inv integral-iff R.inv-unique ac-Zp-is-Unit)
qed

lemma(**in** *padic-integers*) *ac-Zp-of-Unit*:
assumes *val-Zp x = 0*
assumes *x ∈ carrier Zp*
shows *ac-Zp x = x*
using *assms* **unfolding** *val-Zp-def*
by (*metis R.one-closed Zp-def ac-Zp-factors-x(2) ac-Zp-one eint.inject infinity-ne-i0 mult-assoc*
ord-Zp-def ord-Zp-one padic-integers.ac-Zp-in-Zp padic-integers-axioms padic-one-id prime zero-eint-def zero-not-one)

lemma(**in** *padic-integers*) *ac-Zp-p*:
(*ac-Zp* p) = **1**
proof−
have *0*: p = p [^] *nat* (*ord-Zp* p) ⊗ *ac-Zp* p
using *ac-Zp-factors-x*[*of* p] *ord-Zp-p ord-of-nonzero*(*1*)
by *auto*
have *1*: p [^] *nat* (*ord-Zp* p) = p
by (*metis One-nat-def nat-1 ord-Zp-p p-pow-rep0 power-one-right*)
then have *2*: p = p ⊗ *ac-Zp* p
using *0* **by** *presburger*
have *ac-Zp* p ∈ *carrier Zp*
using *ac-Zp-in-Zp*[*of* p]
by (*simp add*: *ord-Zp-p ord-of-nonzero*(*1*))
then show *?thesis*
by (*metis 1 2 m-lcancel R.one-closed R.r-one*
Zp-int-inc-closed p-pow-nonzero(*2*))
qed

lemma(**in** *padic-integers*) *ac-Zp-p-nat-pow*:
(*ac-Zp* (p [^] (*n*::*nat*))) = **1**
apply(*induction n*)
apply (*simp add*: *ac-Zp-one*)
using *ac-Zp-mult ac-Zp-p int-nat-pow-rep nat-pow-Suc2 R.nat-pow-one*
R.one-closed p-natpow-prod-Suc(*1*) *p-nonzero p-pow-nonzero′ p-pow-rep0*
by *auto*

Facts for reasoning about integer powers in an arbitrary commutative monoid:

lemma(**in** *monoid*) *int-pow-add*:
fixes *n*::*int*
fixes *m*::*int*
assumes *a ∈ Units G*
shows *a* [^] (*n* + *m*) = (*a* [^] *n*) ⊗ (*a* [^] *m*)
proof−
have *0*: *group* (*units-of G*)

by (*simp add*: *units-group*)
have *1*: $a \in$ *carrier* (*units-of G*)
by (*simp add*: *assms units-of-carrier*)
have $\bigwedge n$::*int*. $a\ [\widehat{}]\ n = a\ [\widehat{}]_{\textit{units-of } G}\ n$
proof− **fix** k::*int* **show** $a\ [\widehat{}]\ k = a\ [\widehat{}]_{\textit{units-of } G}\ k$ **using** *1 assms units-of-pow*
by (*metis Units-pow-closed int-pow-def nat-pow-def units-of-inv units-of-pow*)
qed
have *2*: $a\ [\widehat{}]_{\textit{units-of } G}\ (n + m) = (a\ [\widehat{}]_{\textit{units-of } G}\ n) \otimes_{\textit{units-of } G} (a\ [\widehat{}]_{\textit{units-of } G}\ m)$
by (*simp add*: *1 group.int-pow-mult units-group*)
show *?thesis* **using** *0 1 2*
by (*simp add*: ‹$\bigwedge n$. $a\ [\widehat{}]\ n = a\ [\widehat{}]_{\textit{units-of } G}\ n$› *units-of-mult*)
qed

lemma(**in** *monoid*) *int-pow-unit-closed*:
fixes n::*int*
assumes $a \in$ *Units G*
shows $a[\widehat{}]\ n \in$ *Units G*
apply(*cases* $n \geq 0$)
using *units-of-def*[*of G*] *units-group Units-inv-Units*[*of a*]
Units-pow-closed[*of inv a*] *Units-pow-closed*[*of a*]
apply (*metis assms pow-nat*)
using *units-of-def*[*of G*] *units-group Units-inv-Units*[*of a*]
Units-pow-closed[*of inv a*] *Units-pow-closed*[*of a*]
by (*simp add*: *assms int-pow-def nat-pow-def*)

lemma(**in** *monoid*) *nat-pow-of-inv*:
fixes n::*nat*
assumes $a \in$ *Units G*
shows *inv* $a[\widehat{}]\ n =$ *inv* $(a[\widehat{}]\ n)$
by (*metis* (*no-types*, *opaque-lifting*) *Units-inv-Units Units-inv-closed Units-inv-inv Units-pow-closed*
Units-r-inv assms inv-unique′ nat-pow-closed nat-pow-one pow-mult-distrib)

lemma(**in** *monoid*) *int-pow-of-inv*:
fixes n::*int*
assumes $a \in$ *Units G*
shows *inv* $a[\widehat{}]\ n =$ *inv* $(a[\widehat{}]\ n)$
apply(*cases* $n \geq 0$)
apply (*metis assms nat-pow-of-inv pow-nat*)
by (*metis assms int-pow-def2 nat-pow-of-inv*)

lemma(**in** *monoid*) *int-pow-inv*:
fixes n::*int*
assumes $a \in$ *Units G*
shows $a[\widehat{}]\ -n =$ *inv* $a[\widehat{}]\ n$
apply(*cases* $n = 0$)
apply *simp*
apply(*cases* $n > 0$)

using *int-pow-def2*[*of G a* −*n*] *int-pow-of-inv*
apply (*simp add*: *assms*)
using *assms int-pow-def2*[*of G a* −*n*] *int-pow-def2*[*of G a n*] *int-pow-def2*[*of G inv a*]
int-pow-of-inv[*of a n*] *Units-inv-Units*[*of a*] *Units-inv-inv Units-pow-closed*[*of a*]
by (*metis linorder-not-less nat-0-iff nat-eq-iff2 nat-zero-as-int neg-0-less-iff-less*)

lemma(**in** *monoid*) *int-pow-inv'*:
fixes *n*::*int*
assumes *a* ∈ *Units G*
shows *a*[^] −*n* = *inv* (*a*[^] *n*)
by (*simp add*: *assms int-pow-inv int-pow-of-inv*)

lemma(**in** *comm-monoid*) *inv-of-prod*:
assumes *a* ∈ *Units G*
assumes *b* ∈ *Units G*
shows *inv* (*a* ⊗ *b*) = (*inv a*) ⊗ (*inv b*)
by (*metis Units-m-closed assms*(*1*) *assms*(*2*) *comm-monoid.m-comm comm-monoid-axioms group.inv-mult-group monoid.Units-inv-closed monoid-axioms units-group units-of-carrier units-of-inv units-of-mult*)

26 Behaviour of *val_Zp* and *ord_Zp* on Natural Numbers and Integers

If f and g have an equal residue at k, then they differ by a multiple of p^k.

lemma(**in** *padic-integers*) *eq-residue-mod*:
assumes *f* ∈ *carrier Zp*
assumes *g* ∈ *carrier Zp*
assumes *f k* = *g k*
shows ∃ *h*. *h* ∈ *carrier Zp* ∧ *f* = *g* ⊕ (p[^]*k*)⊗*h*
proof(*cases f* = *g*)
case *True*
then show *?thesis*
using *Zp-int-inc-zero' assms*(*1*) **by** *auto*
next
case *False*
have (*f* ⊖ *g*) *k* = *0*
using *assms*
by (*metis R.r-right-minus-eq residue-of-diff residue-of-zero*(*2*))
then have *ord-Zp* (*f* ⊖ *g*) ≥ *k*
using *False assms*
by (*simp add*: *ord-Zp-geq*)
then obtain *m*::*int* **where** *m-def*: *m* ≥ *0* ∧ *ord-Zp* (*f* ⊖ *g*) = *k* + *m*
using *zle-iff-zadd* **by** *auto*
have *f* ⊖ *g* = p[^](*k* + *m*) ⊗ *ac-Zp* (*f* ⊖ *g*)
using *ac-Zp-factors-x*(*2*)[*of f* ⊖ *g*] *False m-def assms*(*1*) *assms*(*2*) **by** *auto*

then have *0*: $f \ominus g = \mathrm{p}[\widehat{}]k \otimes \mathrm{p}\ [\widehat{}]\ m \otimes$ *ac-Zp* $(f \ominus g)$
by (*simp add*: *Zp-def m-def padic-integers.p-natintpow-prod padic-integers-axioms*)
have $\mathrm{p}[\widehat{}]k \otimes \mathrm{p}\ [\widehat{}]\ m \otimes$ *ac-Zp* $(f \ominus g) \in$ *carrier Zp*
using *assms 0* **by** *auto*
then have $f = g \oplus \mathrm{p}[\widehat{}]k \otimes \mathrm{p}\ [\widehat{}]\ m \otimes$ *ac-Zp* $(f \ominus g)$
using *0 assms R.ring-simprules*
by *simp*
then show *?thesis* **using** *mult-assoc*
by (*metis 0 False R.m-closed R.r-right-minus-eq* ‹$[p] \cdot \mathbf{1}\ [\widehat{}]\ k \otimes [p] \cdot \mathbf{1}\ [\widehat{}]\ m \otimes$ *ac-Zp* $(f \ominus g) \in$ *carrier Zp*› *ac-Zp-in-Zp assms*(*1*) *assms*(*2*) *m-def p-pow-car*)
qed

lemma(**in** *padic-integers*) *eq-residue-mod'*:
assumes $f \in$ *carrier Zp*
assumes $g \in$ *carrier Zp*
assumes $f\ k = g\ k$
obtains h **where** $h \in$ *carrier Zp* $\wedge\ f = g \oplus (\mathrm{p}[\widehat{}]k) \otimes h$
using *assms eq-residue-mod* **by** *meson*

Valuations of integers which do not divide p:

lemma(**in** *padic-integers*) *ord-Zp-p-nat-unit*:
assumes (*n*::*nat*) *mod p* $\neq$ *0*
shows *ord-Zp* $([n] \cdot \mathbf{1}) =$ *0*
using *ord-equals*[*of* $[n] \cdot \mathbf{1}$ *0*::*nat*]
by (*simp add*: *Zp-nat-inc-res assms*)

lemma(**in** *padic-integers*) *val-Zp-p-nat-unit*:
assumes (*n*::*nat*) *mod p* $\neq$ *0*
shows *val-Zp* $([n] \cdot \mathbf{1}) =$ *0*
unfolding *val-Zp-def*
using *assms ord-Zp-def ord-Zp-p-nat-unit ord-of-nonzero*(*1*) *zero-eint-def* **by** *auto*

lemma(**in** *padic-integers*) *nat-unit*:
assumes (*n*::*nat*) *mod p* $\neq$ *0*
shows $([n] \cdot \mathbf{1}) \in$ *Units Zp*
using *Zp-nat-mult-closed val-Zp-p-nat-unit*
by (*simp add*: *assms val-Zp-0-imp-unit ord-Zp-p-nat-unit*)

lemma(**in** *padic-integers*) *ord-Zp-p-int-unit*:
assumes (*n*::*int*) *mod p* $\neq$ *0*
shows *ord-Zp* $([n] \cdot \mathbf{1}) =$ *0*
by (*metis One-nat-def Zp-int-inc-closed Zp-int-inc-res assms mod-by-1 of-nat-0 ord-equals power-0 power-one-right*)

lemma(**in** *padic-integers*) *val-Zp-p-int-unit*:
assumes (*n*::*int*) *mod p* $\neq$ *0*
shows *val-Zp* $([n] \cdot \mathbf{1}) =$ *0*
unfolding *val-Zp-def*

using *assms ord-Zp-def ord-Zp-p-int-unit ord-of-nonzero(1) zero-eint-def* **by** *auto*

lemma(**in** *padic-integers*) *int-unit*:
assumes (*n*::*int*) *mod p* ≠ *0*
shows ([*n*]·**1**) ∈ *Units Zp*
by (*simp add*: *assms val-Zp-0-imp-unit val-Zp-p-int-unit*)

lemma(**in** *padic-integers*) *int-decomp-ord*:
assumes *n* = *l*∗(*p*^*k*)
assumes *l mod p* ≠ *0*
shows *ord-Zp* ([*n*]·**1**) = *k*
proof−
have *0*: *n* = *l* ∗ (*p*^*k*)
using *assms*(*1*)
by *simp*
then have (*l* ∗ (*p* ^ *k*) *mod* (*p* ^ (*Suc k*))) ≠ *0*
using *Zp-def Zp-nat-inc-zero assms*(*2*) *p-nonzero nonzero-memE*
padic-integers-axioms R.int-inc-zero nonzero-memE(*2*) **by** *auto*
then have *3*: (*l* ∗ *p* ^ *k*) *mod* (*p* ^ (*Suc k*)) ≠ *0*
by *presburger*
show *?thesis*
using *0 3 Zp-int-inc-res ord-equals* **by** *auto*
qed

lemma(**in** *padic-integers*) *int-decomp-val*:
assumes *n* = *l*∗(*p*^*k*)
assumes *l mod p* ≠ *0*
shows *val-Zp* ([*n*]·**1**) = *k*
using *Zp-def assms*(*1*) *assms*(*2*) *R.int-inc-closed ord-of-nonzero*(*1*) *int-decomp-ord*
padic-integers-axioms val-ord-Zp
by *auto*

$\mathbb{Z}_p$ has characteristic zero:

lemma(**in** *padic-integers*) *Zp-char-0*:
assumes (*n*::*int*) > *0*
shows [*n*]·**1** ≠ **0**
proof−
have *prime* (*nat p*)
using *prime prime-nat-iff-prime*
by *blast*
then obtain *l0 k* **where** *0*: *nat n* = *l0*∗((*nat p*)^*k*) ∧ ¬ (*nat p*) *dvd l0*
using *prime assms prime-power-canonical*[*of nat p nat n*]
by *auto*
obtain *l* **where** *l-def*: *l* = *int l0*
by *blast*
have *1*: *n* = *l*∗(*p*^*k*) ∧ ¬ *p dvd l*
using *0 l-def*
by (*smt* (*verit*) *assms int-dvd-int-iff int-nat-eq of-nat-mult of-nat-power prime*

prime-gt-0-int)
show *?thesis*
apply(*cases l = 1*)
using *1 p-pow-nonzero*(*2*) *p-pow-rep0* **apply** *auto*[*1*]
using *1* **by** (*simp add*: *dvd-eq-mod-eq-0 int-decomp-ord ord-of-nonzero*(*1*))
qed

lemma(**in** *padic-integers*) *Zp-char-0'*:
assumes (*n*::*nat*) > *0*
shows [*n*]·**1** ≠ **0**
proof−
have [*n*]·**1** = [(*int n*)]·**1**
using *assms*
by (*simp add*: *add-pow-def int-pow-int*)
then show *?thesis* **using** *assms Zp-char-0*[*of int n*]
by *simp*
qed

lemma (**in** *domain*) *not-eq-diff-nonzero*:
assumes *a* ≠ *b*
assumes *a* ∈ *carrier R*
assumes *b* ∈*carrier R*
shows *a* ⊖ *b* ∈ *nonzero R*
by (*simp add*: *nonzero-def assms*(*1*) *assms*(*2*) *assms*(*3*))

lemma (**in** *domain*) *minus-a-inv*:
assumes *a* ∈ *carrier R*
assumes *b* ∈ *carrier R*
shows *a* ⊖ *b* = ⊖ (*b* ⊖ *a*)
by (*simp add*: *add.inv-mult-group assms*(*1*) *assms*(*2*) *minus-eq*)

lemma(**in** *ring*) *plus-diff-simp*:
assumes *a* ∈ *carrier R*
assumes *b* ∈ *carrier R*
assumes *c* ∈ *carrier R*
assumes *X* = *a* ⊖ *b*
assumes *Y* = *c* ⊖ *a*
shows *X* ⊕ *Y* = *c* ⊖ *b*
using *assms*
unfolding *a-minus-def*
using *ring-simprules*
by (*simp add*: *r-neg r-neg2*)

lemma (**in** *padic-integers*) *Zp-residue-eq*:
assumes *a* ∈ *carrier Zp*
assumes *b* ∈ *carrier Zp*
assumes *val-Zp* (*a* ⊖ *b*) > *k*
shows (*a k*) = (*b k*)
proof−

have *0*: $(a \ominus b)\ k = a\ k \ominus_{Zp\text{-}res\text{-}ring\ k}\ b\ k$
using *assms*
by (*simp add*: *residue-of-diff*)
have *1*: $(a \ominus b)\ k = 0$
using *assms zero-below-val*
by (*smt* (*verit*) *R.minus-closed Zp-def eint-ord-simps*(*2*) *padic-integers.p-res-ring-zero*

padic-integers.residue-of-zero(*1*) *padic-integers.val-ord-Zp padic-integers.zero-below-ord*
padic-integers-axioms)
show *?thesis*
apply(*cases* $k = 0$)
apply (*metis assms*(*1*) *assms*(*2*) *p-res-ring-0'* *residues-closed*)
using *0 1 assms p-residues R-cring Zp-def assms*(*1*) *assms*(*2*) *cring-def*
padic-set-res-closed
residues.res-zero-eq ring.r-right-minus-eq
by (*metis Zp-defs*(*3*) *linorder-neqE-nat not-less0 p-res-ring-zero*)
qed

lemma (**in** *padic-integers*) *Zp-residue-eq2*:
assumes $a \in carrier\ Zp$
assumes $b \in carrier\ Zp$
assumes $(a\ k) = (b\ k)$
assumes $a \neq b$
shows $val\text{-}Zp\ (a \ominus b) \geq k$
proof−
have $(a \ominus b)\ k = 0$
using *assms residue-of-diff*
by (*simp add*: *Zp-def padic-integers.residue-of-diff'* *padic-integers-axioms*)
then show *?thesis*
using *assms*(*1*) *assms*(*2*) *ord-Zp-def ord-Zp-geq val-Zp-def* **by** *auto*
qed

lemma (**in** *padic-integers*) *equal-val-Zp*:
assumes $a \in carrier\ Zp$
assumes $b \in carrier\ Zp$
assumes $c \in carrier\ Zp$
assumes $val\text{-}Zp\ a = val\text{-}Zp\ b$
assumes $val\text{-}Zp\ (c \ominus a) > val\text{-}Zp\ b$
shows $val\text{-}Zp\ c = val\text{-}Zp\ b$
proof−
have *0*: $val\text{-}Zp\ c = val\text{-}Zp\ (c \ominus a \oplus a)$
using *assms*
by (*simp add*: *R.l-neg R.minus-eq add-assoc*)
have $val\text{-}Zp\ c \geq min\ (val\text{-}Zp\ (c \ominus a))\ (val\text{-}Zp\ a)$
using *val-Zp-ultrametric*[*of* $(c \ominus a)\ a$] *assms*(*1*)
assms(*3*) *ord-Zp-ultrametric-eq''*
by (*simp add*: *0*)
then have *1*: $val\text{-}Zp\ c \geq (val\text{-}Zp\ a)$
by (*metis assms*(*4*) *assms*(*5*) *dual-order.order-iff-strict less-le-trans min-le-iff-disj*)

```
  have  val-Zp c = (val-Zp a)
  proof(rule ccontr)
    assume A: val-Zp c ≠ val-Zp a
    then have 0: val-Zp c > val-Zp a
      using 1 A by auto
    then have val-Zp (c ⊕ (a ⊖ c)) ≥ min (val-Zp c) (val-Zp (a ⊖ c))
      by (simp add: assms(1) assms(3) val-Zp-ultrametric)
    then have 1: val-Zp a ≥ min (val-Zp c) (val-Zp (a ⊖ c))
     using assms(1) assms(3) assms(4) assms(5) val-Zp-ultrametric-eq' 0 by auto

    have 2: val-Zp (a ⊖ c) > val-Zp a
      using 0 assms(1) assms(3) assms(4) assms(5)
        val-Zp-ultrametric-eq' by auto
    then have val-Zp a > val-Zp a
      using 0 1 2 val-Zp-of-a-inv
      by (metis assms(1) assms(3) assms(4) assms(5) val-Zp-ultrametric-eq')
    then show False
      by blast
  qed
  then show ?thesis
    using assms(4)
    by simp
qed

lemma (in padic-integers) equal-val-Zp':
  assumes a ∈ carrier Zp
  assumes b ∈ carrier Zp
  assumes c ∈ carrier Zp
  assumes val-Zp a = val-Zp b
  assumes val-Zp c > val-Zp b
  shows val-Zp (a ⊕ c) = val-Zp b
proof−
  have 0: val-Zp b < val-Zp (a ⊕ c ⊖ a)
   by (simp add: R.minus-eq nonzero-closed R.r-neg1 add-comm assms(1) assms(3)
assms(5))
  have 1: val-Zp a ≠ val-Zp (⊖ c)
    using assms(3) assms(4) assms(5)
    by (metis eq-iff not-less val-Zp-of-a-inv)
  then show ?thesis
   by (meson 0 R.semiring-axioms assms(1) assms(2) assms(3) assms(4) equal-val-Zp
semiring.semiring-simprules(1))
qed

lemma (in padic-integers) val-Zp-of-minus:
  assumes a ∈ carrier Zp
  shows val-Zp a = val-Zp (⊖ a)
  using assms not-nonzero-Zp ord-Zp-def ord-Zp-of-a-inv val-Zp-def
  by auto
```

end
theory *Padic-Int-Topology*
imports *Padic-Integers Function-Ring*
begin

type-synonym *padic-int-seq* = *nat* ⇒ *padic-int*

type-synonym *padic-int-fun* = *padic-int* ⇒ *padic-int*

sublocale *padic-integers* < *FunZp?*: *U-function-ring Zp*
  unfolding *U-function-ring-def*
  by (*simp add*: *R.ring-axioms*)

context *padic-integers*
begin

27 Sequences over Zp

The p-adic valuation can be thought of as equivalent to the p-adic absolute value, but with the notion of size inverted so that small numbers have large valuation, and zero has maximally large valuation. The p-adic distance between two points is just the valuation of the difference of those points, and is thus equivalent to the metric induced by the p-adic absolute value. For background on valuations and absolute values for p-adic rings see [4]. In what follows, we develop the topology of the p-adic from a valuative perspective rather than a metric perspective. Though equivalent to the metric approach in the p-adic case, this approach is more general in that there exist valued rings whose valuations take values in non-Archimedean ordered ablelian groups which do not embed into the real numbers.

27.1 The Valuative Distance Function on $\mathbb{Z}_p$

The following lemmas establish that the p-adic distance function satifies the standard properties of an ultrametric. It is symmetric, obeys the ultrametric inequality, and only identical elements are infinitely close.

definition *val-Zp-dist* :: *padic-int* ⇒ *padic-int* ⇒ *eint* **where**
val-Zp-dist a b ≡ *val-Zp* (*a* ⊖ *b*)

lemma *val-Zp-dist-sym*:
  assumes *a* ∈ *carrier Zp*
  assumes *b* ∈ *carrier Zp*
  shows *val-Zp-dist a b* = *val-Zp-dist b a*
proof−
  have *1*: *a* ⊖ *b* = ⊖ (*b* ⊖ *a*) **using** *assms*(*1*) *assms*(*2*)
    using *minus-a-inv* **by** *blast*

then show *?thesis*
using *R.minus-closed Zp-def assms(1) assms(2) padic-integers.val-Zp-of-minus*

padic-integers-axioms val-Zp-dist-def **by** *auto*
qed

lemma *val-Zp-dist-ultrametric*:
assumes $a \in$ *carrier Zp*
assumes $b \in$ *carrier Zp*
assumes $c \in$ *carrier Zp*
shows *val-Zp-dist b c* $\geq$ *min* (*val-Zp-dist a c*) (*val-Zp-dist a b*)
proof−
let *?X* = $b \ominus a$
let *?Y* = $a \ominus c$
let *?Z* = $b \ominus c$
have *0*: *?Z* = *?X* $\oplus$ *?Y*
using *R.add.m-comm assms(1) assms(2) assms(3) R.plus-diff-simp* **by** *auto*
have *4*: *val-Zp ?Z* $\geq$ *min* (*val-Zp ?X*) (*val-Zp ?Y*)
using *0 assms(1) assms(2) assms(3) val-Zp-ultrametric* **by** *auto*
then show *?thesis*
using *assms val-Zp-dist-sym*
unfolding *val-Zp-dist-def*
by (*simp add*: *min.commute*)
qed

lemma *val-Zp-dist-infty*:
assumes $a \in$ *carrier Zp*
assumes $b \in$ *carrier Zp*
assumes *val-Zp-dist a b* $= \infty$
shows $a = b$
using *assms* **unfolding** *val-Zp-dist-def*
by (*metis R.r-right-minus-eq not-eint-eq val-ord-Zp*)

lemma *val-Zp-dist-infty'*:
assumes $a \in$ *carrier Zp*
assumes $b \in$ *carrier Zp*
assumes $a = b$
shows *val-Zp-dist a b* $= \infty$
using *assms* **unfolding** *val-Zp-dist-def*
by (*simp add*: *val-Zp-def*)

The following property will be useful in the proof of Hensel's Lemma: two p-adic integers are close together if and only if their residues are equal at high orders.

lemma *val-Zp-dist-res-eq*:
assumes $a \in$ *carrier Zp*
assumes $b \in$ *carrier Zp*
assumes *val-Zp-dist a b* $> k$
shows (*a k*) = (*b k*)

using *assms(1) assms(2) assms(3) val-Zp-dist-def*
by (*simp add*: *Zp-residue-eq*)

lemma *val-Zp-dist-res-eq2*:
assumes $a \in$ *carrier Zp*
assumes $b \in$ *carrier Zp*
assumes $(a\ k) = (b\ k)$
shows *val-Zp-dist a b* $\geq k$
using *assms(1) assms(2) assms(3) Zp-residue-eq2*
unfolding *val-Zp-dist-def*
by (*simp add*: *val-Zp-def*)

lemma *val-Zp-dist-triangle-eqs*:
assumes $a \in$ *carrier Zp*
assumes $b \in$ *carrier Zp*
assumes $c \in$ *carrier Zp*
assumes *val-Zp-dist a b* $> n$
assumes *val-Zp-dist a c* $> n$
assumes $(k::nat) < n$
shows $a\ k = b\ k$
$a\ k = c\ k$
$b\ k = c\ k$
unfolding *val-Zp-dist-def*
proof−
show *0*: $a\ k = b\ k$
using *assms(1) assms(2) assms(4) assms(6) val-Zp-dist-res-eq*
by (*metis less-imp-le-nat p-residue-padic-int*)
show *1*: $a\ k = c\ k$
using *assms(1) assms(3) assms(5) assms(6) val-Zp-dist-res-eq*
by (*meson eint-ord-simps(1) le-less-trans less-imp-triv not-less of-nat-le-iff*)
show $b\ k = c\ k$
using *0 1* **by** *auto*
qed

27.2 Cauchy Sequences

The definition of Cauchy sequence here is equivalent to standard the metric notion, and is identical to the one found on page 50 of [4].

lemma *closed-seqs-diff-closed*:
assumes $s \in$ *closed-seqs Zp*
assumes $a \in$ *carrier Zp*
shows $s\ m \ominus a \in$ *carrier Zp*
using *assms*
by (*simp add*: *closed-seqs-memE*)

definition *is-Zp-cauchy* :: *padic-int-seq* ⇒ *bool* **where**
is-Zp-cauchy s = (($s \in$ *closed-seqs Zp*) ∧ (∀ (*n*::*int*). ∃ (*N*::*nat*). ∀ *m k*::*nat*.

$$(m{>}N \wedge k{>}N \longrightarrow (\textit{val-Zp-dist}\ (s\ m)\ (s\ k)) > \textit{eint}\ n)))$$

Relation for a sequence which converges to a point:

definition *Zp-converges-to* :: *padic-int-seq* ⇒ *padic-int* ⇒ *bool* **where**
Zp-converges-to s a = ((*a* ∈ *carrier Zp* ∧ *s* ∈ *closed-seqs Zp*)
∧ (∀ (*n*::*int*). (∃ (*k*:: *nat*). (∀ (*m*::*nat*).
(*m* > *k* ⟶ (*val-Zp* ((*s m*) ⊖ *a*)) > *eint n*)))))

lemma *is-Zp-cauchy-imp-closed*:
assumes *is-Zp-cauchy s*
shows *s* ∈ *closed-seqs Zp*
using *assms* **unfolding** *is-Zp-cauchy-def* **by** *blast*

Analogous to the lemmas about residues and p-adic distances, we can characterize Cauchy sequences without reference to a distance function: a sequence is Cauchy if and only if for every natural number k, the k^{th} residues of the elements in the sequence are eventually all equal.

lemma *is-Zp-cauchy-imp-res-eventually-const-0*:
assumes *is-Zp-cauchy s*
fixes *n*::*nat*
obtains *N* **where** ⋀ *n0 n1*. *n0* > *N* ∧ *n1* > *N* ⟹ (*s n0*) *n* = (*s n1*) *n*
proof−
have ∃ (*N*::*nat*). ∀ *m k*::*nat*. (*m*>*N* ∧ *k*>*N* ⟶ (*val-Zp-dist* (*s m*) (*s k*)) > (*int n*))
using *assms is-Zp-cauchy-def* **by** *blast*
then obtain *N* **where** *P0*: ∀ *m k*::*nat*. (*m*>*N* ∧ *k*>*N* ⟶ (*val-Zp-dist* (*s m*) (*s k*)) > (*int n*))
by *blast*
have *P1*: ⋀ *n0 n1*. *n0* > *N* ∧ *n1* > *N* ⟹ (*s n0*) *n* = (*s n1*) *n*
proof−
fix *n0 n1*
assume *A*: *n0* > *N* ∧ *n1* > *N*
have (*n0*>*N* ∧ *n1*>*N* ⟶ (*val-Zp-dist* (*s n0*) (*s n1*)) > (*int n*))
using *P0* **by** *blast*
then have *C0*: (*val-Zp-dist* (*s n0*) (*s n1*)) > (*int n*)
using *A* **by** *blast*
show (*s n0*) *n* = (*s n1*) *n*
proof−
have *A0*: (*val-Zp-dist* (*s n0*) (*s n1*)) > (*int n*)
using *C0* **by** *blast*
have *A1*: *s n0* ∈ *carrier Zp*
using *is-Zp-cauchy-imp-closed*[*of s*] *assms*
by (*simp add*: *closed-seqs-memE*)
have *A2*: *s n1* ∈ *carrier Zp*
using *is-Zp-cauchy-def assms closed-seqs-memE*[*of - Zp*]
by *blast*
show *?thesis*
using *A0 val-Zp-dist-res-eq A1 A2* **by** *metis*
qed
qed
then show *?thesis*

```
    using that by blast
qed

lemma is-Zp-cauchyI:
  assumes s ∈ closed-seqs Zp
  assumes ⋀ n. (∃ N. (∀ n0 n1. n0 > N ∧ n1 > N ⟶ (s n0) n = (s n1) n))
  shows is-Zp-cauchy s
proof−
  have (∀ (n::int). ∃ (N::nat). ∀ m k::nat. (m>N ∧ k>N ⟶ (val-Zp-dist (s m)
(s k)) > n))
  proof
    fix n
    show ∃ (N::nat). ∀ m k::nat. (m>N ∧ k>N ⟶ (val-Zp-dist (s m) (s k)) >
eint n)
    proof−
      have (∃ N. (∀ n0 n1. n0 > N ∧ n1 > N ⟶ (s n0) (Suc (nat n)) = (s n1)
(Suc (nat n))))
        using assms(2) by blast
      then obtain N where N-def: (∀ n0 n1. n0 > N ∧ n1 > N ⟶ (s n0) (Suc
(nat n)) = (s n1) (Suc (nat n)))
        by blast
      have 0: n ≤ eint (int (nat n))
        by simp
      have ∀ m k. N < m ∧ N < k ⟶ (nat n) < val-Zp-dist (s m) (s k)
      proof
      fix m
      show ∀ k. N < m ∧ N < k ⟶ int (nat n) < val-Zp-dist (s m) (s k)
      proof
        fix k
        show N < m ∧ N < k ⟶ int(nat n) < val-Zp-dist (s m) (s k)
        proof
          assume A: N < m ∧ N < k
          then have E: (s m) (Suc(nat n)) = (s k) (Suc(nat n))
            using N-def by blast
          then show  int (nat n) < val-Zp-dist (s m) (s k)
          proof−
            have 0: (s m) ∈ carrier Zp
              by (simp add: assms(1) closed-seqs-memE)
            have 1: (s k) ∈ carrier Zp
             using Zp-def assms(1) closed-seqs-memE[of - Zp] padic-integers-axioms
by blast
            have int (Suc (nat n)) ≤ val-Zp-dist (s m) (s k)
                  using    E 0 1 val-Zp-dist-res-eq[of (s m) (s k) Suc (nat n)]
val-Zp-dist-res-eq2
              by blast
            then have int (nat n) < val-Zp-dist (s m) (s k)
              by (metis Suc-ile-eq add.commute of-nat-Suc)
            then show ?thesis
              by blast
```

```
        qed
      qed
    qed
    qed
    hence ∀ m k. N < m ∧ N < k ⟶ n < val-Zp-dist (s m) (s k)
    using 0
    by (simp add: order-le-less-subst2)
    thus ?thesis by blast
  qed
 qed
 then show ?thesis
  using is-Zp-cauchy-def assms  by blast
qed

lemma is-Zp-cauchy-imp-res-eventually-const:
 assumes is-Zp-cauchy s
 fixes n::nat
 obtains N r where r ∈ carrier (Zp-res-ring n) and ⋀ m. m > N ⟹ (s m) n
= r
proof−
 obtain N where A0: ⋀ n0 n1. n0 > N ∧ n1 > N ⟹ (s n0) n = (s n1) n
  using assms is-Zp-cauchy-imp-res-eventually-const-0 padic-integers-axioms by
blast
 obtain r where A1: s (Suc N) n = r
  by simp
 have 0: r ∈ carrier (Zp-res-ring n)
  using  Zp-def ‹s (Suc N) n = r› assms closed-seqs-memE[of - Zp]
   is-Zp-cauchy-def padic-integers-axioms residues-closed
  by blast
 have 1: ⋀ m. m > N ⟹ (s m) n = r
 proof−
  fix m
  assume  m > N
  then show (s m) n = r
   using A0 A1  by blast
 qed
 then show ?thesis
  using 0 1  that by blast
qed
```

This function identifies the eventual residues of the elements of a cauchy sequence. Since a p-adic integer is defined to be the map which identifies its residues, this map will turn out to be precisely the limit of a cauchy sequence.

```
definition res-lim :: padic-int-seq ⇒ padic-int where
res-lim s = (λ k. (THE r. (∃ N.  (∀  m. m > N ⟶ (s m) k = r))))

lemma res-lim-Zp-cauchy-0:
 assumes is-Zp-cauchy s
```

assumes $f = (\mathit{res\text{-}lim}\ s)\ k$
shows $(\exists N.\ \ (\forall\ m.\ (m > N \longrightarrow (s\ m)\ k = f)))$
$f \in \mathit{carrier}\ (\mathit{Zp\text{-}res\text{-}ring}\ k)$
proof−
obtain $N\ r$ **where** $P0$: $r \in \mathit{carrier}\ (\mathit{Zp\text{-}res\text{-}ring}\ k)$ **and** $P1$: $\bigwedge\ m.\ m > N \Longrightarrow (s\ m)\ k = r$
by (*meson assms*(*1*) *is-Zp-cauchy-imp-res-eventually-const*)
obtain P **where** $P\text{-}def$: $P = (\lambda\ x.\ (\exists N.\ \ (\forall\ m.\ m > N \longrightarrow (s\ m)\ k = x)))$
by *simp*
have *0*: $P\ r$
using *P1 P-def* **by** *auto*
have *1*: $(\bigwedge x.\ P\ x \Longrightarrow x = r)$
proof−
fix x
assume $A\text{-}x$: $P\ x$
obtain *N0* **where** $(\forall\ m.\ m > N0 \longrightarrow (s\ m)\ k = x)$
using $A\text{-}x$ *P-def* **by** *blast*
let *?M* = *max N0 N*
have *C0*: $s\ (\mathit{Suc}\ ?M)\ k = x$
by (*simp add*: ‹$\forall\, m{>}N0.\ s\ m\ k = x$›)
have *C1*: $s\ (\mathit{Suc}\ ?M)\ k = r$
by (*simp add*: *P1*)
show $x = r$
using *C0 C1* **by** *auto*
qed
have *R*: $r = (\mathit{THE}\ x.\ P\ x)$
using *the-equality 0 1* **by** *metis*
have $(\mathit{res\text{-}lim}\ s)\ k = (\mathit{THE}\ r.\ \exists\, N.\ \forall\, m{>}N.\ s\ m\ k = r)$
using *res-lim-def* **by** *simp*
then have $f = (\mathit{THE}\ r.\ \exists\, N.\ \forall\, m{>}N.\ s\ m\ k = r)$
using *assms* **by** *auto*
then have $f = (\mathit{THE}\ r.\ P\ r)$
using *P-def* **by** *auto*
then have $r = f$
using *R* **by** *auto*
then show $(\exists N.\ \ (\forall\ m.\ (m > N \longrightarrow (s\ m)\ k = f)))$ **using** *0 P-def*
by *blast*
show $f \in \mathit{carrier}\ (\mathit{Zp\text{-}res\text{-}ring}\ k)$
using *P0* ‹$r = f$› **by** *auto*
qed

lemma *res-lim-Zp-cauchy*:
assumes *is-Zp-cauchy s*
obtains N **where** $\bigwedge\ m.\ \ (m > N \longrightarrow (s\ m)\ k = (\mathit{res\text{-}lim}\ s)\ k)$
using *res-lim-Zp-cauchy-0 assms* **by** *presburger*

lemma *res-lim-in-Zp*:
assumes *is-Zp-cauchy s*
shows *res-lim s* ∈ *carrier Zp*

```
proof−
  have res-lim s ∈ padic-set p
  proof(rule padic-set-memI)
    show ⋀m. res-lim s m ∈ carrier (residue-ring (p^ m))
      using res-lim-Zp-cauchy-0      by (simp add:  assms)
    show ⋀m n. m < n ⟹ residue (p^ m) (res-lim s n) = res-lim s m
    proof−
      fix m n
      obtain N where  N0: ⋀ l.  (l > N ⟶ (s l) m = (res-lim s) m)
        using assms res-lim-Zp-cauchy by blast
      obtain M where M0: ⋀ l.  (l > M ⟶ (s l) n = (res-lim s) n)
        using assms prod.simps(2) res-lim-Zp-cauchy by auto
      obtain K where K-def: K = max M N
        by simp
      have Km: ⋀ l.  (l > K ⟶ (s l) m = (res-lim s) m)
        using K-def N0  by simp
      have Kn: ⋀ l.  (l > K ⟶ (s l) n = (res-lim s) n)
        using K-def M0  by simp
      assume m < n
      show residue (p^ m) (res-lim s n) = res-lim s m
      proof−
        obtain l where l-def: l = Suc K
          by simp
        have ln: (res-lim s n) = (s l) n
          by (simp add: Kn l-def)
        have lm: (res-lim s m) = (s l) m
          by (simp add: Km l-def)
        have s-car: s l ∈ carrier Zp
          using assms is-Zp-cauchy-def closed-seqs-memE[of - Zp] by blast
        then show ?thesis
          using s-car lm ln ‹m < n› p-residue-def p-residue-padic-int by auto
      qed
    qed
  qed
  then show ?thesis
    by (simp add: Zp-def padic-int-simps(5))
qed
```

27.3 Completeness of $\mathbb{Z}_p$

We can use the developments above to show that a sequence of p-adic integers is convergent if and only if it is cauchy, and that limits of convergent sequences are always unique.

```
lemma is-Zp-cauchy-imp-has-limit:
  assumes is-Zp-cauchy s
  assumes a = res-lim s
  shows Zp-converges-to s a
proof−
  have 0: (a ∈ carrier Zp ∧ s ∈ closed-seqs Zp)
```

```
    using assms(1) assms(2) is-Zp-cauchy-def res-lim-in-Zp by blast
  have 1: (∀(n::int). (∃(k:: nat). (∀( m::nat).
          (m > k ⟶ (val-Zp ((s m) ⊖ a)) ≥ n))))
  proof
    fix n
    show ∃k. ∀m>k.  eint n ≤ val-Zp (s m ⊖ a)
    proof−
      obtain K where K-def: ⋀m. (m > K ⟶ (s m) (nat n) = (res-lim s) (nat
n))
        using assms(1) res-lim-Zp-cauchy
        by blast
      have ∀m>K.  n ≤ val-Zp-dist (s m) a
      proof
        fix m
        show K < m ⟶  n ≤ val-Zp-dist (s m) a
        proof
          assume A: K < m
          show  n ≤ val-Zp-dist (s m) a
          proof−
            have X: (s m) ∈ carrier Zp
              using 0 closed-seqs-memE[of - Zp]
              by blast
            have (s m) (nat n) = (res-lim s) (nat n)
              using A K-def  by blast
            then have (s m) (nat n) = a (nat n)
              using assms(2) by blast
            then have int (nat n) ≤ val-Zp-dist (s m) a
              using  X val-Zp-dist-res-eq2 0 by blast
            then show ?thesis
              by (metis eint-ord-simps(1) int-ops(1) less-not-sym nat-eq-iff2 not-le
order-trans zero-eint-def)
          qed
        qed
      qed
      then show ?thesis
        using val-Zp-dist-def by auto
    qed
  qed
  then show ?thesis using
    0 Zp-converges-to-def
    by (metis Suc-ile-eq val-Zp-dist-def)
qed

lemma convergent-imp-Zp-cauchy:
  assumes s ∈ closed-seqs Zp
  assumes a ∈ carrier Zp
  assumes Zp-converges-to s a
  shows is-Zp-cauchy s
  apply(rule is-Zp-cauchyI)
```

```
  using assms apply simp
proof−
  fix n
  show ∃ N. ∀ n0 n1. N < n0 ∧ N < n1 ⟶ s n0 n = s n1 n
  proof−
    obtain k where k-def:∀ m>k.  n < val-Zp-dist (s m) a
      using assms val-Zp-dist-def
      unfolding Zp-converges-to-def
      by presburger
    have A0:  ∀ n0 n1. k < n0 ∧ k < n1 ⟶ s n0 n = s n1 n
    proof− have ⋀n0 n1. k < n0 ∧ k < n1 ⟶ s n0 n = s n1 n
      proof
      fix n0 n1
      assume A:  k < n0 ∧ k < n1
      show  s n0 n = s n1 n
      proof−
        have 0: n < val-Zp-dist (s n0) a
          using k-def using  A
          by blast
        have 1: n < val-Zp-dist (s n1) a
          using k-def using  A
          by blast
        hence 2: (s n0) n = a n
          using 0 assms val-Zp-dist-res-eq[of s n0 a n] closed-seqs-memE
          by blast
        hence 3: (s n1) n = a n
          using 1 assms val-Zp-dist-res-eq[of s n1 a n] closed-seqs-memE
          by blast
        show ?thesis
          using 2 3
          by auto
      qed
       qed
       thus ?thesis by blast
    qed
    show ?thesis
      using A0
      by blast
  qed
qed

lemma unique-limit:
  assumes s ∈ closed-seqs Zp
  assumes a ∈ carrier Zp
  assumes b ∈ carrier Zp
  assumes Zp-converges-to s a
  assumes Zp-converges-to s b
  shows a = b
proof−
```

```
  have ⋀k. a k = b k
  proof−
    fix k::nat
    obtain k0 where k0-def:∀ m>k0.  k < val-Zp-dist (s m) a
      using assms  unfolding val-Zp-dist-def Zp-converges-to-def
      by blast
    obtain k1 where k1-def:∀ m>k1. k <  val-Zp-dist (s m) b
      using assms unfolding val-Zp-dist-def Zp-converges-to-def
      by blast
    have k0-prop: ⋀m. m> k0 ⟹ (s m) k = a k proof− fix m assume A: m >
k0
      then show (s m) k = a k
        using k0-def assms closed-seqs-memE[of s Zp] val-Zp-dist-res-eq[of - a k]
of-nat-Suc
       by blast
    qed
    have k1-prop: ⋀m. m> k1 ⟹ (s m) k = b k
      using k1-def assms closed-seqs-memE[of s Zp]
      by (simp add: val-Zp-dist-res-eq)
    have ⋀ m. m > (max k0 k1) ⟹ a k = b k
      using k0-prop k1-prop
      by force
    then show a k = b k
      by blast
  qed
  then show ?thesis
    by blast
qed

lemma unique-limit':
  assumes s ∈ closed-seqs Zp
  assumes a ∈ carrier Zp
  assumes Zp-converges-to s a
  shows a = res-lim s
  using unique-limit[of s a res-lim s] assms
        convergent-imp-Zp-cauchy is-Zp-cauchy-imp-has-limit res-lim-in-Zp
  by blast

lemma Zp-converges-toE:
  assumes s ∈ closed-seqs Zp
  assumes a ∈ carrier Zp
  assumes Zp-converges-to s a
  shows ∃ N. ∀ k > N. s k n = a n
  by (metis assms(1) assms(2) assms(3)
      convergent-imp-Zp-cauchy
      res-lim-Zp-cauchy unique-limit')

lemma Zp-converges-toI:
  assumes s ∈ closed-seqs Zp
```

```
  assumes a ∈ carrier Zp
  assumes ⋀n. ∃ N. ∀ k > N. s k n = a n
  shows Zp-converges-to s a
proof−
  have 0: (a ∈ carrier Zp ∧ s ∈ closed-seqs Zp)
    using assms
    by auto
  have 1: (∀ n::int. ∃ k. ∀ m>k.  n < val-Zp-dist (s m) a)
  proof
    fix n::int
    show ∃ k. ∀ m>k.  n < val-Zp-dist (s m) a
    proof(cases n < 0)
      case True
      have ∀ m>0.  n < val-Zp-dist (s m) a
      proof
        fix m ::nat
        show 0 < m ⟶  n < val-Zp-dist (s m) a
        proof
          have 0: eint n < 0
            by (simp add: True zero-eint-def)
          have 1:  s m ⊖ a ∈ carrier Zp
            using assms
            by (simp add: closed-seqs-diff-closed)
          thus  n < val-Zp-dist (s m) a
            using 0  True val-pos[of s m ⊖ a]
            unfolding val-Zp-dist-def
            by auto
        qed
      qed
      then show ?thesis
        by blast
    next
      case False
      then have P0: n ≥ 0
        by auto
      obtain N where N-def: ∀ k > N. s k (Suc (nat n)) = a (Suc (nat n))
        using assms by blast
      have ∀ m>N.  n < val-Zp-dist (s m) a
      proof
        fix m
        show  N < m ⟶  n < val-Zp-dist (s m) a
        proof
          assume A: N < m
          then have A0: s m (Suc (nat n)) = a (Suc (nat n))
            using N-def by blast
          have (Suc (nat n)) ≤ val-Zp-dist (s m) a
          using assms A0 val-Zp-dist-res-eq2[of s m a Suc (nat n)] closed-seqs-memE

            by blast
```

```
        thus n < val-Zp-dist (s m) a
          using False
            by (meson P0 eint-ord-simps(2) less-Suc-eq less-le-trans nat-less-iff)

      qed
    qed
    then show ?thesis
      by blast
  qed
 qed
 show ?thesis
  unfolding Zp-converges-to-def
  using 0 1
  by (simp add: val-Zp-dist-def)
qed
```

Sums and products of cauchy sequences are cauchy:

```
lemma sum-of-Zp-cauchy-is-Zp-cauchy:
 assumes is-Zp-cauchy s
 assumes is-Zp-cauchy t
 shows is-Zp-cauchy (s ⊕_{Zp^ω} t)
proof(rule is-Zp-cauchyI)
 show (s ⊕_{Zp^ω} t) ∈ closed-seqs Zp
  using assms seq-plus-closed is-Zp-cauchy-def by blast
 show ⋀n. ∃ N. ∀ n0 n1. N < n0 ∧ N < n1 ⟶ (s ⊕_{Zp^ω} t) n0 n = (s ⊕_{Zp^ω} t)
n1 n
 proof−
  fix n
  show ∃ N. ∀ n0 n1. N < n0 ∧ N < n1 ⟶ (s ⊕_{Zp^ω} t) n0 n = (s ⊕_{Zp^ω} t) n1
n
  proof−
   obtain N1 where N1-def: ∀ n0 n1. N1 < n0 ∧ N1 < n1 ⟶ s n0 n = s n1
n
   using assms(1) is-Zp-cauchy-imp-res-eventually-const-0 padic-integers-axioms
by blast
   obtain N2 where N2-def: ∀ n0 n1. N2 < n0 ∧ N2 < n1 ⟶ t n0 n = t n1
n
   using assms(2) is-Zp-cauchy-imp-res-eventually-const-0 padic-integers-axioms
by blast
   obtain M where M-def: M = max N1 N2
    by simp
   have ∀ n0 n1. M < n0 ∧ M < n1 ⟶ (s ⊕_{Zp^ω} t) n0 n = (s ⊕_{Zp^ω} t) n1 n
   proof
    fix n0
    show ∀ n1. M < n0 ∧ M < n1 ⟶ (s ⊕_{Zp^ω} t) n0 n = (s ⊕_{Zp^ω} t) n1 n
    proof
     fix n1
     show  M < n0 ∧ M < n1 ⟶ (s ⊕_{Zp^ω} t) n0 n = (s ⊕_{Zp^ω} t) n1 n
```

```
        proof
          assume A: M < n0 ∧ M < n1
          have Fs:  s n0 n = s n1 n using A M-def N1-def by auto
          have Ft:  t n0 n = t n1 n using A M-def N2-def by auto
          then show (s ⊕_{Zp^ω} t) n0 n = (s ⊕_{Zp^ω} t) n1 n
            using seq-plus-simp[of s t] assms
        by (simp add: Fs is-Zp-cauchy-imp-closed residue-of-sum)
        qed
      qed
    qed
    then show ?thesis
      by blast
  qed
 qed
qed

lemma prod-of-Zp-cauchy-is-Zp-cauchy:
  assumes is-Zp-cauchy s
  assumes is-Zp-cauchy t
  shows is-Zp-cauchy (s ⊗_{Zp^ω} t)
proof(rule is-Zp-cauchyI)
  show (s ⊗_{Zp^ω} t) ∈ closed-seqs Zp
    using assms(1) assms(2) is-Zp-cauchy-def seq-mult-closed by auto
  show ⋀n. ∃ N. ∀ n0 n1. N < n0 ∧ N < n1 ⟶ (s ⊗_{Zp^ω} t) n0 n = (s ⊗_{Zp^ω} t)
n1 n
  proof−
    fix n
    show ∃ N. ∀ n0 n1. N < n0 ∧ N < n1 ⟶ (s ⊗_{Zp^ω} t) n0 n = (s ⊗_{Zp^ω} t) n1
n
    proof−
      obtain N1 where N1-def: ∀ n0 n1. N1 < n0 ∧ N1 < n1 ⟶ s n0 n = s n1
n
      using assms(1) is-Zp-cauchy-imp-res-eventually-const-0 padic-integers-axioms
by blast
      obtain N2 where N2-def: ∀ n0 n1. N2 < n0 ∧ N2 < n1 ⟶ t n0 n = t n1
n
      using assms(2) is-Zp-cauchy-imp-res-eventually-const-0 padic-integers-axioms
by blast
      obtain M where M-def: M = max N1 N2
        by simp
      have ∀ n0 n1. M < n0 ∧ M < n1 ⟶ (s ⊗_{Zp^ω} t) n0 n = (s ⊗_{Zp^ω} t) n1 n
      proof
        fix n0
        show ∀ n1. M < n0 ∧ M < n1 ⟶ (s ⊗_{Zp^ω} t) n0 n = (s ⊗_{Zp^ω} t) n1 n
        proof
          fix n1
          show  M < n0 ∧ M < n1 ⟶ (s ⊗_{Zp^ω} t) n0 n = (s ⊗_{Zp^ω} t) n1 n
          proof
```

```
      assume A: M < n0 ∧ M < n1
      have Fs:  s n0 n = s n1 n using A M-def N1-def by auto
      have Ft:  t n0 n = t n1 n using A M-def N2-def by auto
      then show (s ⊗_Zpω t) n0 n = (s ⊗_Zpω t) n1 n
        using seq-mult-simp[of s t] is-Zp-cauchy-imp-closed assms
        by (simp add: Fs residue-of-prod)
    qed
   qed
  qed
  then show ?thesis
   by blast
 qed
qed
```

Constant sequences are cauchy:

```
lemma constant-is-Zp-cauchy:
 assumes is-constant-seq Zp s
 shows is-Zp-cauchy s
proof(rule is-Zp-cauchyI)
 show s ∈ closed-seqs Zp
 proof(rule closed-seqs-memI)
  fix k
  show s k ∈ carrier Zp
   using assms is-constant-seq-imp-closed
   by (simp add: is-constant-seq-imp-closed closed-seqs-memE)
 qed
 obtain x where ⋀ k. s k = x
  using assms
  by (meson is-constant-seqE)
 then show ⋀n. ∃ N. ∀ n0 n1. N < n0 ∧ N < n1 ⟶ s n0 n = s n1 n
  by simp
qed
```

Scalar multiplies of cauchy sequences are cauchy:

```
lemma smult-is-Zp-cauchy:
 assumes is-Zp-cauchy s
 assumes a ∈ carrier Zp
 shows is-Zp-cauchy (a ⊙_Zpω s)
 apply(rule is-Zp-cauchyI)
 apply (meson U-function-ring.ring-seq-smult-closed U-function-ring-axioms assms(1)
assms(2) is-Zp-cauchy-def)
 using assms ring-seq-smult-eval[of s a] is-Zp-cauchy-imp-closed[of s]
 by (metis res-lim-Zp-cauchy residue-of-prod)

lemma Zp-cauchy-imp-approaches-res-lim:
 assumes is-Zp-cauchy s
 assumes a = res-lim s
 obtains N where ⋀n. n> N ⟹ val-Zp (a ⊖ (s n)) > eint k
```

proof−
have $(\forall\, n{::}int.\ \exists\, k.\ \forall\, m{>}k.\ \ n < \textit{val-Zp-dist}\ (s\ m)\ a)$
using *Zp-converges-to-def*[*of s a*] *assms is-Zp-cauchy-imp-has-limit*[*of s a*] *val-Zp-dist-def*
by *simp*
then have $\exists\, N.\ \forall\, m{>}N.\ \ k < \textit{val-Zp-dist}\ (s\ m)\ a$
by *blast*
then obtain N **where** *N-def*: $\forall\, m{>}N.\ \ k < \textit{val-Zp-dist}\ (s\ m)\ a$
by *blast*
have $\bigwedge n.\ n{>}\ N \Longrightarrow \textit{val-Zp}\ (a \ominus (s\ n)) > k$
proof−
fix m
assume $m > N$
then have *0*: $k < \textit{val-Zp-dist}\ (s\ m)\ a$
using *N-def*
by (*simp add*: *val-Zp-def*)
show $k < \textit{val-Zp}\ (a \ominus s\ m)$
using *0 assms*(*1*) *assms*(*2*) *is-Zp-cauchy-def closed-seqs-memE*[*of - Zp*] *val-Zp-dist-def val-Zp-dist-sym res-lim-in-Zp* **by** *auto*
qed
then show *?thesis*
using *that*
by *blast*
qed

28 Continuous Functions

For convenience, we will use a slightly unorthodox definition of continuity here. Since $\mathbb{Z}_p$ is complete, a function is continuous if and only if its compositions with cauchy sequences are cauchy sequences. Thus we can define a continuous function on $\mathbb{Z}_p$ as a function which carries cauchy sequences to cauchy sequences under composition. Note that this does not generalize to a definition of continuity for functions defined on incomplete subsets of $\mathbb{Z}_p$. For example, the function $1/x$ defined on $\mathbb{Z}_p - \{0\}$ clearly does not have this property but is continuous. However, towards a proof of Hensel's Lemma we will only need to consider polynomial functions and so this definition suffices for our purposes.

28.1 Defining Continuous Functions and Basic Examples

abbreviation *Zp-constant-function* (‹$\mathfrak{c}_{Zp}$›) **where**
$\mathfrak{c}_{Zp}\ a \equiv \textit{constant-function}\ (\textit{carrier Zp})\ a$

definition *is-Zp-continuous* ::*padic-int-fun* ⇒ *bool* **where**
is-Zp-continuous f = $(f \in \textit{carrier}\ (\textit{Fun Zp}) \wedge (\forall\ s.\ \textit{is-Zp-cauchy}\ s \longrightarrow \textit{is-Zp-cauchy}(f \circ s)))$

lemma *Zp-continuous-is-Zp-closed*:
assumes *is-Zp-continuous f*
shows *f* ∈ *carrier* (*Fun Zp*)
using *assms is-Zp-continuous-def* **by** *blast*

lemma *is-Zp-continuousI*:
assumes *f* ∈ *carrier* (*Fun Zp*)
assumes ⋀*s*. *is-Zp-cauchy s* ⟹ *is-Zp-cauchy* (*f* ∘ *s*)
shows *is-Zp-continuous f*
proof−
have (∀ *s*. *is-Zp-cauchy s* ⟶ *is-Zp-cauchy*(*f* ∘ *s*))
proof
fix *s*
show *is-Zp-cauchy s* ⟶ *is-Zp-cauchy* (*f* ∘ *s*)
by (*simp add*: *assms*(*2*))
qed
then show *?thesis*
using *assms*(*1*) *is-Zp-continuous-def* **by** *blast*
qed

lemma *Zp-continuous-is-closed*:
assumes *is-Zp-continuous f*
shows *f* ∈ *carrier* (*Fun Zp*)
using *assms* **unfolding** *is-Zp-continuous-def* **by** *blast*

lemma *constant-is-Zp-continuous*:
assumes *a* ∈ *carrier Zp*
shows *is-Zp-continuous* (*const a*)
proof(*rule is-Zp-continuousI*)
show $\mathfrak{c}_{Zp}$ *a* ∈ *carrier* (*function-ring* (*carrier Zp*) *Zp*)
by (*simp add*: *assms constant-function-closed*)
show ⋀*s*. *is-Zp-cauchy s* ⟹ *is-Zp-cauchy* ($\mathfrak{c}_{Zp}$ *a* ∘ *s*)
proof−
fix *s*
assume *A*: *is-Zp-cauchy s*
have *is-constant-seq Zp* ($\mathfrak{c}_{Zp}$ *a* ∘ *s*)
using *constant-function-comp-is-constant-seq*[*of a s*] *A assms*
is-Zp-cauchy-imp-closed **by** *blast*
then show *is-Zp-cauchy* ($\mathfrak{c}_{Zp}$ *a* ∘ *s*)
using *A assms constant-is-Zp-cauchy*
by *blast*
qed
qed

lemma *sum-of-cont-is-cont*:
assumes *is-Zp-continuous f*
assumes *is-Zp-continuous g*
shows *is-Zp-continuous* (*f* ⊕$_{Fun\ Zp}$ *g*)
apply(*rule is-Zp-continuousI*)

using *assms Zp-continuous-is-closed assms function-sum-comp-is-seq-sum*[*of - f g*]
apply (*simp add*: *fun-add-closed*)
using *assms*(*1*) *assms*(*2*) *function-sum-comp-is-seq-sum is-Zp-cauchy-def is-Zp-continuous-def sum-of-Zp-cauchy-is-Zp-cauchy* **by** *auto*

lemma *prod-of-cont-is-cont*:
assumes *is-Zp-continuous f*
assumes *is-Zp-continuous g*
shows *is-Zp-continuous* ($f \otimes_{Fun\ Zp} g$)
apply(*rule is-Zp-continuousI*)
using *assms Zp-continuous-is-closed assms*
apply (*simp add*: *fun-mult-closed*)
using *function-mult-comp-is-seq-mult*[*of - f g*] *assms*(*1*) *assms*(*2*) *is-Zp-cauchy-imp-closed*

is-Zp-continuous-def prod-of-Zp-cauchy-is-Zp-cauchy **by** *auto*

lemma *smult-is-continuous*:
assumes *is-Zp-continuous f*
assumes $a \in$ *carrier Zp*
shows *is-Zp-continuous* ($a \odot_{Fun\ Zp} f$)
apply(*rule is-Zp-continuousI*)
using *assms* **apply** (*simp add*: *assms function-smult-closed is-Zp-continuous-def*)
using *ring-seq-smult-comp-assoc assms*
by (*simp add*: *is-Zp-cauchy-imp-closed is-Zp-continuous-def smult-is-Zp-cauchy*)

lemma *id-Zp-is-Zp-continuous*:
is-Zp-continuous ring-id
apply(*rule is-Zp-continuousI*)
by (*auto simp add*: *is-Zp-cauchy-imp-closed ring-id-seq-comp*)

lemma *nat-pow-is-Zp-continuous*:
assumes *is-Zp-continuous f*
shows *is-Zp-continuous* ($f[\hat{}]_{Fun\ Zp}(n::nat)$)
apply(*induction n*)
using *constant-is-Zp-continuous function-one-is-constant* **apply** *force*
by (*simp add*: *assms prod-of-cont-is-cont*)

lemma *ring-id-pow-closed*:
(*ring-id*)$[\hat{}]_{Fun\ Zp}$ (*n*::*nat*) $\in$ *carrier* (*Fun Zp*)
by (*simp add*: *function-ring-is-monoid monoid.nat-pow-closed*)

lemma *monomial-equation*:
assumes $c \in$ *carrier Zp*
shows *monomial c n* = $c \odot_{Fun\ Zp}$ (*ring-id*)$[\hat{}]_{Fun\ Zp} n$
apply(*rule function-ring-car-eqI*)
apply (*simp add*: *assms monomial-functions*)
using *assms function-smult-closed*[*of c ring-id* $[\hat{}]_{Fun\ Zp}$ *n*] *ring-id-pow-closed*
apply *blast*

unfolding *monomial-function-def*
using *assms function-smult-eval*[*of c* (*ring-id*)[^]$_{Fun\ Zp}$ (*n*::*nat*)]
function-nat-pow-eval[*of ring-id - n*]
by (*simp add*: *ring-id-eval ring-id-pow-closed*)

lemma *monomial-is-Zp-continuous*:
assumes *c* ∈ *carrier Zp*
shows *is-Zp-continuous* (*monomial c n*)
using *monomial-equation*[*of c n*] *nat-pow-is-Zp-continuous*
by (*simp add*: *assms id-Zp-is-Zp-continuous smult-is-continuous*)

28.2 Composition by a Continuous Function Commutes with Taking Limits of Sequences

Due to our choice of definition for continuity, a little bit of care is required to show that taking the limit of a cauchy sequence commutes with composition by a continuous function. For a sequence $(s_n)_{n\in\mathbb{N}}$ converging to a point t, we can consider the alternating sequence $(s_0, t, s_1, t, s_2, t, \dots)$ which is also cauchy. Clearly composition with f yields $(f(s_0), f(t), f(s_1), f(t), f(s_2), f(t), \dots)$ from which we can see that the limit must be $f(t)$.

definition *alt-seq* **where**
alt-seq s = (λ*k*. (*if* (*even k*) *then* (*s k*) *else* (*res-lim s*)))

lemma *alt-seq-Zp-cauchy*:
assumes *is-Zp-cauchy s*
shows *is-Zp-cauchy* (*alt-seq s*)
proof(*rule is-Zp-cauchyI*)
show (*alt-seq s*) ∈ *closed-seqs Zp*
unfolding *alt-seq-def* **using** *assms is-Zp-cauchy-imp-closed*
by (*simp add*: *closed-seqs-memE closed-seqs-memI res-lim-in-Zp*)
fix *n*
show ∃ *N*. ∀ *n0 n1*. *N* < *n0* ∧ *N* < *n1* ⟶ *alt-seq s n0 n* = *alt-seq s n1 n*
proof−
obtain *N* **where** *N*: ∀ *n0 n1*. *N* < *n0* ∧ *N* < *n1* ⟶ *s n0 n* = *s n1 n*
using *assms is-Zp-cauchy-imp-res-eventually-const-0 padic-integers-axioms*
by *blast*
have *alt-seq s n0 n* = *alt-seq s n1 n*
if *N* < *n0 N* < *n1* **for** *n0 n1*
using *N* **apply** (*auto simp*: *alt-seq-def*)
using *that* **apply** *blast*
apply (*metis that*(*1*) *assms gt-ex linorder-not-less order-less-le-trans order-less-trans res-lim-Zp-cauchy*)
apply (*metis that*(*2*) *assms gt-ex linorder-neqE-nat order-less-trans res-lim-Zp-cauchy*)
done
then show *?thesis*
by *blast*
qed
qed

lemma *alt-seq-limit*:
assumes *is-Zp-cauchy s*
shows *res-lim*(*alt-seq s*) = *res-lim s*
proof−
have ⋀*k*. *res-lim*(*alt-seq s*) *k* = *res-lim s k*
proof−
fix *k*
obtain *N* **where** *N-def*: ∀ *m*. *m*> *N* ⟶ *s m k* = *res-lim s k*
using *assms res-lim-Zp-cauchy*
by *blast*
obtain *N*′ **where** *N*′*-def*: ∀ *m*. *m*> *N*′ ⟶ (*alt-seq s*) *m k* = *res-lim* (*alt-seq s*) *k*
using *assms res-lim-Zp-cauchy*
alt-seq-Zp-cauchy
by *blast*
have ⋀*m*. *m* > (*max N N*′) ⟹ *s m k* = *res-lim* (*alt-seq s*) *k*
proof−
fix *m*
assume *A0*: *m* > (*max N N*′)
have *A1*: *s m k* = *res-lim s k*
using *A0 N-def*
by *simp*
have *A2*: (*alt-seq s*) *m k* = *res-lim* (*alt-seq s*) *k*
using *A0 N*′*-def*
by *simp*
have *A3*: (*alt-seq s*) *m k* = *res-lim s k*
using *alt-seq-def A1 A2*
by *presburger*
show *s m k* = *res-lim* (*alt-seq s*) *k*
using *A1 A2 A3*
by *auto*
qed
then have *P*:⋀*m*. *m* > (*max N N*′) ⟹ (*res-lim s k*) = *res-lim* (*alt-seq s*) *k*
using *N-def*
by *auto*
show *res-lim*(*alt-seq s*) *k* = *res-lim s k*
using *P*[*of Suc* (*max N N*′)]
by *auto*
qed
then show *?thesis*
by (*simp add*: *ext*)
qed

lemma *res-lim-pushforward*:
assumes *is-Zp-continuous f*
assumes *is-Zp-cauchy s*
assumes *t* = *alt-seq s*
shows *res-lim* (*f* ∘ *t*) = *f* (*res-lim t*)

```
proof−
  have 0: Zp-converges-to (f ∘ t) (res-lim (f ∘ t))
    using assms alt-seq-Zp-cauchy is-Zp-cauchy-imp-has-limit
          is-Zp-continuous-def
    by blast
  have ⋀k. res-lim (f ∘ t) k = f (res-lim t) k
  proof−
    fix k
    show res-lim (f ∘ t) k = f (res-lim t) k
    proof−
      obtain N where N-def: ⋀m. m> N ⟹ (f ∘ t) m k = (res-lim (f ∘ t)) k
        using 0
       by (meson convergent-imp-Zp-cauchy Zp-converges-to-def res-lim-Zp-cauchy)
      obtain M where M-def: M = 2∗(Suc N) + 1
        by simp
      have 0: t M = res-lim s
        using assms
        unfolding alt-seq-def
        by (simp add: M-def)
      have 1: (f ∘ t) M k = (res-lim (f ∘ t)) k
        using N-def M-def
        by auto
      have 2: (f ∘ t) M k = f (t M) k
        by simp
      have 3: (f ∘ t) M k = f (res-lim s) k
        using 0 2 by simp
      have 4: (f ∘ t) M k = f (res-lim t) k
        using 3 assms alt-seq-limit[of s]
        by auto
      show ?thesis
        using 4 1 by auto
    qed
  qed
  then show ?thesis by(simp add: ext)
qed

lemma res-lim-pushforward':
  assumes is-Zp-continuous f
  assumes is-Zp-cauchy s
  assumes t = alt-seq s
  shows res-lim (f ∘ s) = res-lim (f ∘ t)
proof−
  obtain a where a-def: a = res-lim (f ∘ s)
    by simp
  obtain b where b-def: b = res-lim (f ∘ t)
    by simp
  have ⋀k. a k = b k
  proof−
    fix k
```

obtain *Na* **where** *Na-def*: $\bigwedge m.\ m > Na \Longrightarrow (f \circ s)\ m\ k = a\ k$
using *a-def assms is-Zp-continuous-def*
padic-integers-axioms res-lim-Zp-cauchy
by *blast*
obtain *Nb* **where** *Nb-def*: $\bigwedge m.\ m > Nb \Longrightarrow (f \circ t)\ m\ k = b\ k$
using *b-def assms is-Zp-continuous-def*
padic-integers-axioms res-lim-Zp-cauchy
alt-seq-Zp-cauchy
by *blast*
obtain *M* **where** *M-def*: $M = 2*(max\ Na\ Nb) + 1$
by *simp*
have *M0*: *odd M*
by (*simp add*: *M-def*)
have *M1*: $M > Na$
using *M-def*
by *auto*
have *M2*: $M > Nb$
using *M-def*
by *auto*
have *M3*: $t\ M = res\text{-}lim\ s$
using *assms alt-seq-def M0*
by *auto*
have *M4*: $((f \circ t)\ M) = f\ (res\text{-}lim\ s)$
using *M3*
by *auto*
have *M5*: $((f \circ t)\ M)\ k = b\ k$
using *M2 Nb-def* **by** *auto*
have *M6*: $f\ (res\text{-}lim\ s) = f\ (res\text{-}lim\ t)$
using *assms alt-seq-limit*[*of s*]
by *auto*
have *M7*: $f\ (res\text{-}lim\ t)\ k = b\ k$
using *M4 M5 M6* **by** *auto*
have *M8*: $(f \circ s)\ M\ k = (f \circ s)\ (Suc\ M)\ k$
using *M1 Na-def* **by** *auto*
have *M9*: $(f \circ s)\ (Suc\ M) = (f \circ t)\ (Suc\ M)$
using *assms* **unfolding** *alt-seq-def*
using *M-def*
by *auto*
have *M10*: $(f \circ t)\ M\ k = (f \circ t)\ (Suc\ M)\ k$
using *M2 Nb-def* **by** *auto*
have *M11*: $(f \circ t)\ M\ k = (f \circ s)\ M\ k$
using *M10 M8 M9* **by** *auto*
show $a\ k = b\ k$
using *M1 M11 M5 Na-def* **by** *auto*
qed
then show *?thesis* **using** *a-def b-def ext*[*of a b*] **by** *auto*
qed

lemma *continuous-limit*:

assumes *is-Zp-continuous f*
assumes *is-Zp-cauchy s*
shows *Zp-converges-to* (*f* ∘ *s*) (*f* (*res-lim s*))
proof−
obtain *t* **where** *t-def*: *t* = *alt-seq s*
by *simp*
have *0*: *Zp-converges-to* (*f* ∘ *s*) (*res-lim* (*f* ∘ *s*))
using *assms*(*1*) *assms*(*2*) *is-Zp-continuous-def*
is-Zp-cauchy-imp-has-limit padic-integers-axioms **by** *blast*
have *1*: *Zp-converges-to* (*f* ∘ *s*) (*res-lim* (*f* ∘ *t*))
using *0 assms*(*1*) *assms*(*2*) *res-lim-pushforward*′ *t-def* **by** *auto*
have *2*: *Zp-converges-to* (*f* ∘ *s*) (*f* (*res-lim t*))
using *1 assms*(*1*) *assms*(*2*) *res-lim-pushforward padic-integers-axioms t-def* **by** *auto*
then show *Zp-converges-to* (*f* ∘ *s*) (*f* (*res-lim s*))
by (*simp add*: *alt-seq-limit assms*(*2*) *t-def*)
qed

end
end
theory *Padic-Int-Polynomials*
imports *Padic-Int-Topology Cring-Poly*
begin

context *padic-integers*
begin

This theory states and proves basic lemmas connecting the topology on $\mathbb{Z}_p$ with the functions induced by polynomial evaluation over $\mathbb{Z}_p$. This will imply that polynomial evaluation applied to a Cauchy Sequence will always produce a cauchy sequence, which is a key fact in the proof of Hensel's Lemma.

type-synonym *padic-int-poly* = *nat* ⇒ *padic-int*

lemma *monom-term-car*:
assumes *c* ∈ *carrier Zp*
assumes *x* ∈ *carrier Zp*
shows *c* ⊗ *x*[^](*n*::*nat*) ∈ *carrier Zp*
using *assms R.nat-pow-closed*
by (*simp add*: *monoid.nat-pow-closed cring.cring-simprules*(*5*) *cring-def ring-def*)

Univariate polynomial ring over Zp

abbreviation(*input*) *Zp-x* **where**
Zp-x ≡ *UP Zp*

lemma *Zp-x-is-UP-cring*:
UP-cring Zp
using *UP-cring.intro domain-axioms domain-def* **by** *auto*

lemma *Zp-x-is-UP-domain*:
UP-domain Zp
by (*simp add*: *UP-domain-def domain-axioms*)

lemma *Zp-x-domain*:
domain Zp-x
by (*simp add*: *UP-domain.UP-domain Zp-x-is-UP-domain*)

lemma *pow-closed*:
assumes $a \in$ *carrier Zp*
shows $a[\hat{}](n::nat) \in$ *carrier Zp*
by (*meson domain-axioms domain-def cring-def assms monoid.nat-pow-closed ring-def*)

lemma(**in** *ring*) *pow-zero*:
assumes $(n::nat)>0$
shows $\mathbf{0}[\hat{}]\ n = \mathbf{0}$
by (*simp add*: *assms nat-pow-zero*)

lemma *sum-closed*:
assumes $f \in$ *carrier Zp*
assumes $g \in$ *carrier Zp*
shows $f \oplus g \in$ *carrier Zp*
by (*simp add*: *assms*(*1*) *assms*(*2*) *cring.cring-simprules*(*1*))

lemma *mult-zero*:
assumes $f \in$ *carrier Zp*
shows $f \otimes \mathbf{0} = \mathbf{0}$
$\mathbf{0} \otimes f = \mathbf{0}$
apply (*simp add*: *assms cring.cring-simprules*(*27*))
by (*simp add*: *assms cring.cring-simprules*(*26*))

lemma *monom-poly-is-Zp-continuous*:
assumes $c \in$ *carrier Zp*
assumes $f =$ *monom Zp-x c n*
shows *is-Zp-continuous* (*to-fun f*)
using *monomial-is-Zp-continuous assms monom-to-monomial* **by** *auto*

lemma *degree-0-is-Zp-continuous*:
assumes $f \in$ *carrier Zp-x*
assumes *degree f* $= 0$
shows *is-Zp-continuous* (*to-fun f*)
using *const-to-constant*[*of lcf f*] *assms constant-is-Zp-continuous ltrm-deg-0*
by (*simp add*: *cfs-closed*)

lemma *UP-sum-is-Zp-continuous*:
assumes $a \in$ *carrier Zp-x*
assumes $b \in$ *carrier Zp-x*

assumes *is-Zp-continuous* (*to-fun a*)
assumes *is-Zp-continuous* (*to-fun b*)
shows *is-Zp-continuous* (*to-fun* ($a \oplus_{Zp\text{-}x} b$))
using *sum-of-cont-is-cont assms*
by (*simp add*: *to-fun-Fun-add*)

lemma *polynomial-is-Zp-continuous*:
assumes $f \in$ *carrier Zp-x*
shows *is-Zp-continuous* (*to-fun f*)
apply(*rule poly-induct3*)
apply (*simp add*: *assms*)
using *UP-sum-is-Zp-continuous* **apply** *blast*
using *monom-poly-is-Zp-continuous* **by** *blast*
end

Notation for polynomial function application

context *padic-integers*
begin
notation *to-fun* (**infixl**‹•› *70*)

Evaluating polynomials in the residue rings

lemma *res-to-fun-monic-monom*:
assumes $a \in$ *carrier Zp*
assumes $b \in$ *carrier Zp*
assumes *a k* = *b k*
shows (*monom Zp-x* **1** *n* · *a*) *k* = (*monom Zp-x* **1** *n* · *b*) *k*
proof(*induction n*)
case *0*
then show *?case*
using *UP-cring.to-fun-X-pow Zp-x-is-UP-domain assms*(*1*) *assms*(*2*) *nat-pow-0 to-fun-one monom-one*
by *presburger*
next
case (*Suc n*)
fix *n*::*nat*
assume *IH*: *to-fun* (*monom Zp-x* **1** *n*) *a k* = *to-fun* (*monom Zp-x* **1** *n*) *b k*
show *to-fun* (*monom Zp-x* **1** (*Suc n*)) *a k* = *to-fun* (*monom Zp-x* **1** (*Suc n*)) *b k*
proof−
have *LHS0*: (*monom Zp-x* **1** (*Suc n*) · *a*) *k* = ((*monom Zp-x* **1** *n* · *a*) ⊗ (*X* · *a*)) *k*
by (*simp add*: *UP-cring.to-fun-monic-monom Zp-x-is-UP-cring assms*(*1*))
then show *?thesis*
using *assms IH Zp-x-is-UP-domain*
Zp-continuous-is-Zp-closed
by (*metis* (*mono-tags*, *lifting*) *R.one-closed X-poly-def monom-closed monom-one-Suc*
residue-of-prod to-fun-X to-fun-mult)
qed

qed

lemma *res-to-fun-monom*:
assumes $a \in$ *carrier Zp*
assumes $b \in$ *carrier Zp*
assumes $c \in$ *carrier Zp*
assumes $a\ k = b\ k$
shows $(monom\ Zp\text{-}x\ c\ n \cdot a)\ k = (monom\ Zp\text{-}x\ c\ n \cdot b)\ k$
using *res-to-fun-monic-monom assms*
by (*metis* (*mono-tags*, *opaque-lifting*) *to-fun-monic-monom to-fun-monom residue-of-prod*)

lemma *to-fun-res-ltrm*:
assumes $a \in$ *carrier Zp*
assumes $b \in$ *carrier Zp*
assumes $f \in$ *carrier Zp-x*
assumes $a\ k = b\ k$
shows $((ltrm\ f) \cdot a)\ k = ((ltrm\ f) \cdot b)\ k$
by (*simp add*: *lcf-closed assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *res-to-fun-monom*)

Polynomial application commutes with taking residues

lemma *to-fun-res*:
assumes $f \in$ *carrier Zp-x*
assumes $a \in$ *carrier Zp*
assumes $b \in$ *carrier Zp*
assumes $a\ k = b\ k$
shows $(f \cdot a)\ k = (f \cdot b)\ k$
apply(*rule poly-induct3*[*of f*])
apply (*simp add*: *assms*(*1*))
using *assms*(*2*) *assms*(*3*) *to-fun-plus residue-of-sum* **apply** *presburger*
using *assms*(*2*) *assms*(*3*) *assms*(*4*) *res-to-fun-monom* **by** *blast*

If a and b have equal kth residues, then so do f'(a) and f'(b)

lemma *deriv-res*:
assumes $f \in$ *carrier Zp-x*
assumes $a \in$ *carrier Zp*
assumes $b \in$ *carrier Zp*
assumes $a\ k = b\ k$
shows $(deriv\ f\ a)\ k = (deriv\ f\ b)\ k$
using *assms to-fun-res*[*of pderiv f a b k*]
by (*simp add*: *pderiv-closed pderiv-eval-deriv*)

Propositions about evaluation:

lemma *to-fun-monom-plus*:
assumes $a \in$ *carrier Zp*
assumes $g \in$ *carrier Zp-x*
assumes $c \in$ *carrier Zp*
shows $(monom\ Zp\text{-}x\ a\ n \oplus_{Zp\text{-}x} g) \cdot c = a \otimes c[\frown]n \oplus (g \cdot c)$
by (*simp add*: *assms*(*1*) *assms*(*2*) *assms*(*3*) *to-fun-monom to-fun-plus*)

lemma *to-fun-monom-minus*:
assumes $a \in$ *carrier Zp*
assumes $g \in$ *carrier Zp-x*
assumes $c \in$ *carrier Zp*
shows $(monom\ Zp\text{-}x\ a\ n \ominus_{Zp\text{-}x} g) \cdot c = a \otimes c[\hat{}]n \ominus (g \cdot c)$
by (*simp add*: *UP-cring.to-fun-diff Zp-x-is-UP-cring assms*(*1*) *assms*(*2*) *assms*(*3*) *to-fun-monom*)

end

end
theory *Hensels-Lemma*
imports *Padic-Int-Polynomials*
begin

The following proof of Hensel's Lemma is directly adapted from Keith Conrad's proof which is given in an online note [1]. The same note was used as the basis for a formalization of Hensel's Lemma by Robert Lewis in the Lean proof assistant [5].

29 Auxiliary Lemmas for Hensel's Lemma

lemma(**in** *ring*) *minus-sum*:
assumes $a \in$ *carrier R*
assumes $b \in$ *carrier R*
shows $\ominus (a \oplus b) = \ominus a \oplus \ominus b$
by (*simp add*: *assms*(*1*) *assms*(*2*) *local.minus-add*)

context *padic-integers*
begin

lemma *poly-diff-val*:
assumes $f \in$ *carrier Zp-x*
assumes $a \in$ *carrier Zp*
assumes $b \in$ *carrier Zp*
shows $val\text{-}Zp\ (f \cdot a \ominus f \cdot b) \geq val\text{-}Zp\ (a \ominus b)$
proof−
obtain *c* **where** *c-def*: $c \in carrier\ Zp \wedge (f \cdot a \ominus f \cdot b) = (a \ominus b) \otimes c$
using *assms*
by (*meson to-fun-diff-factor*)
have *1*: $val\text{-}Zp\ c \geq 0$
using *c-def val-pos* **by** *blast*
have *2*: $val\text{-}Zp\ (f \cdot a \ominus f \cdot b) = val\text{-}Zp\ (a \ominus b) + (val\text{-}Zp\ c)$
using *c-def val-Zp-mult*
by (*simp add*: *assms*(*2*) *assms*(*3*))
then show *?thesis*
using *1* **by** *auto*

qed

Restricted p-adic division

definition *divide* **where**
divide x y = (*if x* = **0** *then* **0** *else*
(p[^](*nat* (*ord-Zp x* − *ord-Zp y*)) ⊗ *ac-Zp x* ⊗ (*inv ac-Zp y*)))

lemma *divide-closed*:
assumes *x* ∈ *carrier Zp*
assumes *y* ∈ *carrier Zp*
assumes *y* ≠ **0**
shows *divide x y* ∈ *carrier Zp*
unfolding *divide-def*
apply(*cases x* = **0**)
apply *auto*[*1*]
using *assms ac-Zp-is-Unit*
by (*simp add*: *ac-Zp-in-Zp*)

lemma *divide-formula*:
assumes *x* ∈ *carrier Zp*
assumes *y* ∈ *carrier Zp*
assumes *y* ≠ **0**
assumes *val-Zp x* ≥ *val-Zp y*
shows *y* ⊗ *divide x y* = *x*
apply(*cases x* = **0**)
apply (*simp add*: *divide-def mult-zero-l*)
proof− **assume** *A*: *x* ≠ **0**
have *0*: *y* ⊗ *divide x y* = p[^] *nat* (*ord-Zp y*) ⊗ *ac-Zp y* ⊗ (p[^](*nat* (*ord-Zp x* − *ord-Zp y*)) ⊗ *ac-Zp x* ⊗ (*inv ac-Zp y*))
using *assms ac-Zp-factors-x*[*of x*] *ac-Zp-factors-x*[*of y*] *A divide-def*
by *auto*
hence *1*: *y* ⊗ *divide x y* = p[^] *nat* (*ord-Zp y*) ⊗ (p[^](*nat* (*ord-Zp x* − *ord-Zp y*)) ⊗ *ac-Zp x* ⊗ *ac-Zp y* ⊗ (*inv ac-Zp y*))
using *mult-assoc mult-comm* **by** *auto*
have *2*: (*nat* (*ord-Zp y*) + *nat* (*ord-Zp x* − *ord-Zp y*)) = *nat* (*ord-Zp x*)
using *assms ord-pos*[*of x*] *ord-pos*[*of y*] *A val-ord-Zp* **by** *auto*
have *y* ⊗ *divide x y* = p[^] *nat* (*ord-Zp y*) ⊗ p[^](*nat* (*ord-Zp x* − *ord-Zp y*)) ⊗ *ac-Zp x*
using *1 A assms*
by (*simp add*: *ac-Zp-in-Zp ac-Zp-is-Unit mult-assoc*)
thus *y* ⊗ *divide x y* = *x*
using *2 A ac-Zp-factors-x*(*1*) *assms*(*1*) *p-natpow-prod* **by** *auto*
qed

lemma *divide-nonzero*:
assumes *x* ∈ *nonzero Zp*
assumes *y* ∈ *nonzero Zp*
assumes *val-Zp x* ≥ *val-Zp y*
shows *divide x y* ∈ *nonzero Zp*

by (*metis assms*(1) *assms*(2) *assms*(3) *divide-closed divide-formula mult-zero-l nonzero-closed nonzero-memE*(2) *nonzero-memI*)

lemma *val-of-divide*:
assumes $x \in$ *carrier Zp*
assumes $y \in$ *nonzero Zp*
assumes *val-Zp* $x \geq$ *val-Zp* y
shows *val-Zp* (*divide* x y) = *val-Zp* x − *val-Zp* y
proof−
have *0*: $y \otimes$ *divide* x $y = x$
by (*simp add*: *assms*(1) *assms*(2) *assms*(3) *divide-formula nonzero-closed nonzero-memE*(2))
hence *val-Zp* y + *val-Zp* (*divide* x y) = *val-Zp* x
using *assms*(1) *assms*(2) *divide-closed nonzero-closed not-nonzero-memI val-Zp-mult*
by *fastforce*
thus *?thesis*
by (*metis Extended-Int.eSuc-minus-1 add-0 assms*(2) *eint-minus-comm p-nonzero val-Zp-eq-frac-0 val-Zp-p*)
qed

lemma *val-of-divide'*:
assumes $x \in$ *carrier Zp*
assumes $y \in$ *carrier Zp*
assumes $y \neq \mathbf{0}$
assumes *val-Zp* $x \geq$ *val-Zp* y
shows *val-Zp* (*divide* x y) = *val-Zp* x − *val-Zp* y
using *Zp-def assms*(1) *assms*(2) *assms*(3) *assms*(4) *padic-integers.not-nonzero-Zp*

padic-integers.val-of-divide padic-integers-axioms **by** *blast*
end

lemma(**in** *UP-cring*) *taylor-deg-1-eval'''*:
assumes $f \in$ *carrier P*
assumes $a \in$ *carrier R*
assumes $b \in$ *carrier R*
assumes c = *to-fun* (*shift* (*2*::*nat*) (T_a f)) ($\ominus b$)
assumes $b \otimes$ (*deriv* f a) = (*to-fun* f a)
shows *to-fun* f ($a \ominus b$) = ($c \otimes b[\widehat{}](2::nat)$)
proof−
have *0*: *to-fun* f ($a \ominus b$) = (*to-fun* f a) $\ominus$ (*deriv* f $a \otimes b$) $\oplus$ ($c \otimes b[\widehat{}](2::nat)$)
using *assms taylor-deg-1-eval''*
by *blast*
have *1*: (*to-fun* f a) $\ominus$ (*deriv* f $a \otimes b$) = $\mathbf{0}$
using *assms*
proof −
have $\forall f\ a.\ f \notin$ *carrier P* $\vee$ $a \notin$ *carrier R* $\vee$ *to-fun* f $a \in$ *carrier R*
using *to-fun-closed* **by** *presburger*
then show *?thesis*
using *R.m-comm R.r-right-minus-eq assms*(1) *assms*(2) *assms*(3) *assms*(5)

```
    by (simp add: deriv-closed)
  qed
  have 2: to-fun f (a ⊖ b) = 𝟬 ⊕ (c ⊗ b[^](2::nat))
    using 0 1
    by simp
  then show ?thesis using assms
    by (simp add: taylor-closed to-fun-closed shift-closed)
qed

lemma(in padic-integers) res-diff-zero-fact:
  assumes a ∈ carrier Zp
  assumes b ∈ carrier Zp
  assumes (a ⊖ b) k = 0
  shows a k = b k a k ⊖Zp-res-ring k b k = 0
   apply(cases k = 0)
 apply (metis assms(1) assms(2) p-res-ring-0 p-res-ring-0' p-res-ring-car p-residue-padic-int
p-residue-range' zero-le)
    apply (metis R.add.inv-closed R.add.m-lcomm R.minus-eq R.r-neg R.r-zero
Zp-residue-add-zero(2) assms(1) assms(2) assms(3))
    using assms(2) assms(3) residue-of-diff by auto

lemma(in padic-integers) res-diff-zero-fact':
  assumes a ∈ carrier Zp
  assumes b ∈ carrier Zp
  assumes a k = b k
  shows a k ⊖Zp-res-ring k b k = 0
  by (simp add: assms(3) residue-minus)

lemma(in padic-integers) res-diff-zero-fact'':
  assumes a ∈ carrier Zp
  assumes b ∈ carrier Zp
  assumes a k = b k
  shows (a ⊖ b) k = 0
  by (simp add: assms(2) assms(3) res-diff-zero-fact' residue-of-diff)

lemma(in padic-integers) is-Zp-cauchyI':
assumes s ∈ closed-seqs Zp
assumes ∀ n::nat. ∃ k::int.∀ m.  m ≥  k ⟶ val-Zp (s (Suc m) ⊖ s m) ≥ n
shows is-Zp-cauchy s
proof(rule is-Zp-cauchyI)
  show A0: s ∈ closed-seqs Zp
    by (simp add: assms(1))
  show ⋀n. ∃ N. ∀ n0 n1. N < n0 ∧ N < n1 ⟶ s n0 n = s n1 n
  proof−
    fix n
    show ∃ N. ∀ n0 n1. N < n0 ∧ N < n1 ⟶ s n0 n = s n1 n
    proof(induction n)
      case 0
      then show ?case
```

```
    proof−
      have ∀ n0 n1. 0 < n0 ∧ 0 < n1 ⟶ s n0 0 = s n1 0
        apply auto
      proof−
        fix n0 n1::nat
        assume A: n0 > 0 n1 > 0
        have 0: s n0 ∈ carrier Zp
          using A0
          by (simp add: closed-seqs-memE)
        have 1: s n1 ∈ carrier Zp
          using A0
          by (simp add: closed-seqs-memE)
        show  s n0 (0::nat) = s n1 (0::nat)
          using A0 Zp-def 0 1 residues-closed
          by (metis p-res-ring-0')
      qed
      then show ?thesis
        by blast
    qed
  next
    case (Suc n)
    fix n
    assume IH: ∃ N. ∀ n0 n1. N < n0 ∧ N < n1 ⟶ s n0 n = s n1 n
    show  ∃ N. ∀ n0 n1. N < n0 ∧ N < n1 ⟶ s n0 (Suc n) = s n1 (Suc n)
    proof−
      obtain N where N-def: ∀ n0 n1. N < n0 ∧ N < n1 ⟶ s n0 n = s n1 n
        using IH
        by blast
      obtain k where k-def: ∀ m.  (Suc m) ≥ k ⟶ val-Zp (s (Suc (Suc m)) ⊖
s (Suc m)) ≥ Suc (Suc n)
        using assms  Suc-n-not-le-n
        by (meson nat-le-iff)
      have ∀ n0 n1.  Suc (max N (max n k)) < n0 ∧  Suc (max N (max n k))<
n1 ⟶ s n0 (Suc n) = s n1 (Suc n)
        apply auto
      proof−
        fix n0 n1
        assume A: Suc (max N (max n k)) < n0  Suc (max N (max n k)) < n1
        show s n0 (Suc n) = s n1 (Suc n)
        proof−
          obtain K where K-def: K = Suc (max N (max n k))
            by simp
          have P0: ⋀m. s ((Suc m)+ K) (Suc n) = s (Suc K) (Suc n)
            apply auto
          proof−
            fix m
            show s (Suc (m + K)) (Suc n) = s (Suc K) (Suc n)
            apply(induction m)
              apply auto
```

```
          proof−
            fix m
            assume A0:  s (Suc (m + K)) (Suc n) = s (Suc K) (Suc n)
            show  s (Suc (Suc (m + K))) (Suc n) = s (Suc K) (Suc n)
            proof−
              have I: k < m + K
                using K-def
                by linarith
               have val-Zp (s (Suc (Suc (m + K))) ⊖ s (Suc (m + K))) ≥  Suc
(Suc n)
              proof−
                have (Suc (m + K)) > k
                  by (simp add: I less-Suc-eq)
                then show ?thesis
                  using k-def less-imp-le-nat
                  by blast
              qed
               hence D: val-Zp (s (Suc (Suc (m + K))) ⊖ s (Suc (m + K))) >
(Suc n)
                using Suc-ile-eq by fastforce
              have s (Suc (Suc (m + K))) (Suc n) =  s (Suc (m + K)) (Suc n)
              proof−
                have (s (Suc (Suc (m + K))) ⊖ s (Suc (m + K)))  (Suc n) = 0
                  using D assms(1) res-diff-zero-fact''[of s (Suc (Suc (m + K)))
s (Suc (m + K))  Suc n]
                   val-Zp-dist-res-eq[of s (Suc (Suc (m + K))) s (Suc (m + K))
Suc n] unfolding val-Zp-dist-def
              by (simp add: closed-seqs-memE)
                 hence 0: (s (Suc (Suc (m + K)))  (Suc n) ⊖Zp-res-ring (Suc n)
(s (Suc (m + K)))  (Suc n)) = 0
                  using res-diff-zero-fact(2)[of s (Suc (Suc (m + K))) s (Suc (m
+ K)) Suc n ]
                       assms(1)
                  by (simp add: closed-seqs-memE)

                show ?thesis
                proof−
                  have 00: cring (Zp-res-ring (Suc n))
                    using R-cring by blast
                 have 01:  s (Suc (Suc (m + K))) (Suc n) ∈ carrier (Zp-res-ring
(Suc n))
                    using assms(1) closed-seqs-memE residues-closed by blast
                   have 02: (⊖Zp-res-ring (Suc n) (s (Suc (m + K)) (Suc n))) ∈
carrier (Zp-res-ring (Suc n))
                by (meson 00 assms(1) cring.cring-simprules(3) closed-seqs-memE
residues-closed)
                  show ?thesis
                    unfolding a-minus-def
                    using  00 01 02
```

```
                                cring.sum-zero-eq-neg[of Zp-res-ring (Suc n) s (Suc (Suc
(m + K))) (Suc n)
                                                    ⊖Zp-res-ring (Suc n) s (Suc (m + K)) (Suc n)]
                          by (metis 0  a-minus-def assms(1) cring.cring-simprules(21)
closed-seqs-memE
                              p-res-ring-zero residues-closed)
                      qed
                    qed
                    then show ?thesis using A0 assms(1)
                      by simp
                  qed
                qed
              qed
              have ∃ m0. n0 = (Suc m0) + K
              proof−
                have n0 > K
                  by (simp add: A(1) K-def)
                then have n0 = (Suc (n0 − K − 1)) + K
                  by auto
                then show ?thesis by blast
              qed
              then obtain m0 where m0-def: n0 = (Suc m0) + K
                by blast
              have ∃ m0. n1 = (Suc m0) + K
              proof−
                have n1 > K
                  by (simp add: A(2) K-def)
                then have n1 = (Suc (n1 − K − 1)) + K
                  by auto
                then show ?thesis by blast
              qed
              then obtain m1 where m1-def: n1 = (Suc m1) + K
                by blast
              have 0: s n0 (Suc n) = s (Suc K) (Suc n)
                using m0-def P0[of m0] by auto
              have 1: s n1 (Suc n) = s (Suc K) (Suc n)
                using m1-def P0[of m1] by auto
              show ?thesis using 0 1
                by auto
            qed
          qed
          then show ?thesis
            by blast
        qed
      qed
    qed
  qed
```

30 The Proof of Hensel's Lemma

30.1 Building a Locale for the Proof of Hensel's Lemma

locale *hensel* = *padic-integers*+
  fixes *f*::*padic-int-poly*
  fixes *a*::*padic-int*
  assumes *f-closed*[*simp*]: *f* ∈ *carrier Zp-x*
  assumes *a-closed*[*simp*]: *a* ∈ *carrier Zp*
  assumes *fa-nonzero*[*simp*]: *f*•*a* ≠**0**
  assumes *hensel-hypothesis*[*simp*]: (*val-Zp* (*f*•*a*) > *2*∗ *val-Zp* ((*pderiv f*)•*a*))

sublocale *hensel* < *cring Zp*
  by (*simp add*: *R.is-cring*)

context *hensel*
begin

abbreviation *f* ′ **where**
f ′ ≡ *pderiv f*

lemma *f*′*-closed*:
f ′ ∈ *carrier Zp-x*
  using *f-closed pderiv-closed* **by** *blast*

lemma *f*′*-vals-closed*:
  assumes *a* ∈ *carrier Zp*
  shows *f* ′•*a* ∈ *carrier Zp*
  by (*simp add*: *UP-cring.to-fun-closed Zp-x-is-UP-cring f*′*-closed*)

lemma *fa-closed*:
(*f*•*a*) ∈ *carrier Zp*
  by (*simp add*: *UP-cring.to-fun-closed Zp-x-is-UP-cring*)

lemma *f*′*a-closed*:
(*f* ′•*a*) ∈ *carrier Zp*
proof−
  have *f* ′ ∈ *carrier Zp-x*
    by (*simp add*: *f*′*-closed*)
  then show *?thesis*
    by (*simp add*: *f*′*-vals-closed*)
qed

lemma *fa-nonzero*′:
(*f*•*a*) ∈ *nonzero Zp*
  using *fa-closed fa-nonzero not-nonzero-Zp* **by** *blast*

lemma *f*′*a-nonzero*[*simp*]:
(*f* ′•*a*) ≠ **0**
proof(*rule ccontr*)

assume $\neg\ (f'\bullet a) \neq \mathbf{0}$
then have $(f'\bullet a) = \mathbf{0}$
by *blast*
then have $\infty <$ *val-Zp* $(f\bullet a)$ **using** *hensel-hypothesis*
by (*simp add*: *val-Zp-def*)
thus *False*
using *eint-ord-simps*(*6*) **by** *blast*
qed

lemma *f'a-nonzero'*:
$(f'\bullet a) \in$ *nonzero Zp*
using *f'a-closed f'a-nonzero not-nonzero-Zp* **by** *blast*

lemma *f'a-not-infinite*[*simp*]:
val-Zp $(f'\bullet a) \neq \infty$
by (*metis eint-ord-code*(*3*) *hensel-hypothesis linorder-not-less times-eint-simps*(*4*))

lemma *f'a-nonneg-val*[*simp*]:
val-Zp $((f'\bullet a)) \geq$ *0*
using *f'a-closed val-pos* **by** *blast*

lemma *hensel-hypothesis-weakened*:
val-Zp $(f\bullet a) >$ *val-Zp* $(f'\bullet a)$
proof−
have *0*: *0* $\leq$ *val-Zp* $(f'\bullet a) \wedge$ *val-Zp* $(f'\bullet a) \neq \infty$
using *f'a-closed val-ord-Zp val-pos* **by** *force*
have *1*: *1* $<$ *eint 2*
by (*simp add*: *one-eint-def*)
thus *?thesis* **using** *0 eint-mult-mono'*[*of val-Zp* $(f'\bullet a)$ *1 2*] *hensel-hypothesis*
by (*metis linorder-not-less mult-one-left order-trans*)
qed

30.2 Constructing the Newton Sequence

definition *newton-step* :: *padic-int* $\Rightarrow$ *padic-int* **where**
newton-step x $=$ *x* $\ominus$ (*divide* $(f\bullet x)$ $(f'\bullet x)$)

lemma *newton-step-closed*:
newton-step a $\in$ *carrier Zp*
using *divide-closed* **unfolding** *newton-step-def*
using *f'a-closed f'a-nonzero fa-closed local.a-closed* **by** *blast*

fun *newton-seq* :: *padic-int-seq* (‹*ns*›) **where**
newton-seq 0 $=$ *a*|
newton-seq (*Suc n*) $=$ *newton-step* (*newton-seq n*)

30.3 Key Properties of the Newton Sequence

lemma *hensel-factor-id*:
(*divide* $(f\bullet a)$ $(f'\bullet a)$) $\otimes$ $((f'\bullet a))$ $= (f\bullet a)$

using *hensel-hypothesis hensel-axioms divide-formula f'a-closed fa-closed hensel-hypothesis-weakened mult-comm*
by *auto*

definition *hensel-factor* (‹*t*›) **where**
hensel-factor = *val-Zp* (*f*•*a*) − *2*∗(*val-Zp* (*f'*•*a*))

lemma *t-pos*[*simp*]:
t > *0*
using *hensel-factor-def hensel-hypothesis*
by (*simp add*: *eint-minus-le*)

lemma *t-neq-infty*[*simp*]:
t ≠ ∞
by (*simp add*: *hensel-factor-def val-Zp-def*)

lemma *t-times-pow-pos*[*simp*]:
(*2*^(*n*::*nat*))∗*t* > *0*
apply(*cases n* = *0*)
using *one-eint-def* **apply** *auto*[*1*]
using *eint-mult-mono'*[*of t 1 2*^*n*] *t-pos*
by (*smt* (*verit*) *eint-ord-simps*(*2*) *linorder-not-less mult-one-left neq0-conv one-eint-def order-less-le order-trans self-le-power t-neq-infty*)

lemma *newton-seq-props-induct*:
shows ⋀*k*. *k* ≤ *n* ⟹ (*ns k*) ∈ *carrier Zp*
∧ *val-Zp* (*f'*•(*ns k*)) = *val-Zp* ((*f'*•*a*))
∧ *val-Zp* (*f*•(*ns k*)) ≥ *2*∗(*val-Zp* (*f'*•*a*)) + (*2*^*k*)∗*t*
proof(*induction n*)
case *0*
then have *kz*: *k* = *0*
by *simp*
have *B0*: (*ns k*) ∈ *carrier Zp*
using *kz*
by *simp*
have *B1*: *val-Zp* (*f'* • *ns k*) = (*val-Zp* (*f'*•*a*))
using *kz newton-seq.simps*(*1*)
by *presburger*
have *B2*: *val-Zp* (*f* • (*ns k*)) ≥ (*2* ∗ (*val-Zp* (*f'*•*a*))) + *2* ^ *k* ∗ *t*
proof−
have *B20*: (*2* ∗ (*val-Zp* (*f'*•*a*))) + *2* ^ *k* ∗ *t* = (*2* ∗ (*val-Zp* (*f'*•*a*))) + *t*
proof−
have (*2* ∗ (*val-Zp* (*f'*•*a*))) + *2* ^ *k* ∗ *t* = (*2* ∗ (*val-Zp* (*f'*•*a*))) + *t*
using *kz one-eint-def* **by** *auto*
then show *?thesis*
by *blast*
qed
then have (*2* ∗ (*val-Zp* (*f'*•*a*))) + *2* ^ *k* ∗ *t* = (*2* ∗ (*val-Zp* (*f'*•*a*))) + *val-Zp* (*f*•*a*) − *2*∗(*val-Zp* (*f'*•*a*))

```
    unfolding hensel-factor-def
    by (simp add: val-Zp-def)
  then have (2 * (val-Zp (f'•a))) + 2 ^ k * t =  val-Zp (f•a)
    by (metis add-diff-cancel-eint eint-ord-simps(6) hensel-hypothesis)
  thus ?thesis       by (simp add: kz)
 qed
 thus ?case
  using B0 B1 by blast
next
 case (Suc n)
 show ?case
 proof(cases k ≤ n)
  case True
  then show ?thesis using  Suc.IH
   by blast
  next
   case False
   have F1: (ns n) ∈ carrier Zp
    using  Suc.IH   by blast
   have F2: val-Zp (f'•(ns n)) = val-Zp ((f'•a))
    using  Suc.IH  by blast
   have F3: val-Zp (f•(ns n)) ≥ 2*(val-Zp (f'•a)) + (2^n)*t
    using  Suc.IH  by blast
   have kval: k = Suc n
    using False Suc.prems le-Suc-eq by blast
   have F6: val-Zp (f•(ns n)) ≥ val-Zp (f'•(ns n))
   proof−
    have 2*(val-Zp (f'•a))  ≥ val-Zp (f'•a)
     using f'a-closed val-pos eint-mult-mono'[of val-Zp (f'•a) 1 2]
        by (metis Groups.add-ac(2) add.right-neutral eSuc-eint eint-0-iff(2)
eint-add-left-cancel-le
     eint-ord-simps(2) f'a-nonneg-val f'a-not-infinite infinity-ne-i1 linorder-not-less

          mult-one-left not-one-less-zero one-add-one one-eint-def order-less-le
order-trans zero-one-eint-neq(1))
    hence  2*(val-Zp (f'•a)) + (2^n)*t  ≥ val-Zp (f'•a)
     using t-times-pow-pos[of n]
      by (metis (no-types, lifting) add.right-neutral eint-add-left-cancel-le or-
der-less-le order-trans)
    then show ?thesis
     using F2 F3 by auto
   qed
   have F5:  divide (f•(ns n))(f'•(ns n)) ∈ carrier Zp
   proof−
    have 00: f • ns n ∈ carrier Zp
     by (simp add: F1 to-fun-closed)
    have val-Zp ((f'•a)) ≠ val-Zp 0
     by (simp add:  val-Zp-def)
    then have 01: f' • ns n ∈ nonzero Zp
```

```
      using F2 F1 Zp-x-is-UP-cring f'-closed nonzero-def
    proof −
      have f' • ns n ∈ carrier Zp
     using F1 Zp-continuous-is-Zp-closed f'-closed  polynomial-is-Zp-continuous
        by (simp add: to-fun-closed)
      then show ?thesis
        using F2 ‹val-Zp (f'•a) ≠ val-Zp 0› not-nonzero-Zp by fastforce
    qed
    then show ?thesis
      using F6
      by (metis 00 F2 ‹val-Zp (f'•a) ≠ val-Zp 0› divide-closed nonzero-closed)

  qed
  have F4:  (ns k) ⊖ (ns n) = (⊖ divide (f•(ns n))(f'•(ns n)))
    using F1 F5 newton-seq.simps(2)[of n] kval
    unfolding newton-step-def
     by (metis R.l-neg R.minus-closed R.minus-zero R.plus-diff-simp R.r-neg2
R.r-right-minus-eq
       a-minus-def local.a-closed minus-a-inv)
   have F7: val-Zp (divide (f•(ns n))(f'•(ns n))) = val-Zp (f•(ns n)) − val-Zp
(f'•(ns n))
    apply(rule val-of-divide)
      apply (simp add: F1 to-fun-closed)
        using F1 f'-closed to-fun-closed F2 not-nonzero-Zp val-Zp-def apply
fastforce
        by (simp add: F6)
  show ?thesis
  proof
    show P0:ns k ∈ carrier Zp
    proof−
      have A0: ns k = ns n ⊖ (divide (f• (ns n)) (f'•(ns n)))
        by (simp add: kval newton-step-def)
      have A1: val-Zp (f'•(ns n)) = val-Zp (f'•a)
        using  Suc.IH
        by blast
      have A2: val-Zp (f•(ns n)) ≥val-Zp (f'•a)
      proof−
         have A20: (2 * val-Zp (f'•a)) + 2 ^ n * (val-Zp (f•a) − 2 * val-Zp
(f'•a)) ≥val-Zp (f'•a)
        proof−
          have val-Zp (f•a) − 2 * val-Zp (f'•a) > 0
            using hensel-hypothesis eint-minus-le by blast
          then have   (2 ^ n) * (val-Zp (f•a) − 2 * val-Zp (f'•a))
                  ≥ (val-Zp (f•a) − 2 * val-Zp (f'•a))
            using eint-pos-int-times-ge by auto
         then have    ((2 * val-Zp (f'•a)) + 2 ^ n * (val-Zp (f•a) − 2 * val-Zp
(f'•a)))
                  ≥ (2 * val-Zp (f'•a)) + (val-Zp (f•a) − 2 * val-Zp (f'•a))
          by (simp add: val-Zp-def)
```

```
          then have    ((2 * val-Zp (f'•a)) + 2 ^ n * (val-Zp (f•a) - 2 * val-Zp
(f'•a)))
                    ≥ (val-Zp (f•a) )
            by simp
          then show    ((2 * val-Zp (f'•a)) + 2 ^ n * (val-Zp (f•a) - 2 * val-Zp
(f'•a)))
                    ≥ (val-Zp (f'•a) )
            using hensel-hypothesis-weakened by auto
        qed
       have A21:val-Zp (f•(ns n)) ≥ (2 * val-Zp (f'•a)) + 2 ^ n * (val-Zp (f•a)
- 2 * val-Zp (f'•a))
          using  Suc.IH unfolding hensel-factor-def
          by blast
        show ?thesis using A21 A20
          by auto
      qed
      have A3: ns n ∈ carrier Zp
        using  Suc.IH by blast
      have A4: val-Zp (f•(ns n)) ≥val-Zp (f'•(ns n))
        using A1 A2
        by presburger
      have A5: f•(ns n) ∈ carrier Zp
      by (simp add: F1 UP-cring.to-fun-closed Zp-x-is-UP-cring)
      have A6: (f'•(ns n)) ∈ nonzero Zp
      proof−
        have (f'•(ns n)) ∈ carrier  Zp
          by (simp add: F1 UP-cring.to-fun-closed Zp-x-is-UP-cring f'-closed)

        have val-Zp (f'•(ns n)) ≠ ∞
          using A1
          by (simp add:  val-Zp-def)
        then show ?thesis
          using ‹f' • ns n ∈ carrier Zp› not-nonzero-Zp val-Zp-def
          by meson
      qed
      have A7:  (divide (f• (ns n)) (f'•(ns n))) ∈ carrier Zp
        using A5 A6 A4 A3 F5 by linarith
      then show ?thesis
        using A0 A3 cring.cring-simprules(4)
        by (simp add: F1 F5 cring.cring-simprules(4))
    qed
    have P1: val-Zp (f' • ns k) = val-Zp (f'•a)
    proof(cases (f' • ns k) = (f' • ns n))
      case True
      then show ?thesis using  Suc.IH
        by (metis order-refl)
    next
      case False
      have val-Zp ((f' • ns k) ⊖ (f' • ns n)) ≥ val-Zp ((ns k) ⊖ (ns n))
```

using *False P0 f′-closed poly-diff-val Suc.IH*
by *blast*
then have *val-Zp ((f′ · ns k) ⊖ (f′ · ns n)) ≥ val-Zp (⊖ divide (f·(ns n))(f′·(ns n)))*
using *F4* **by** *argo*
then have *val-Zp ((f′ · ns k) ⊖ (f′ · ns n)) ≥ val-Zp (divide (f·(ns n))(f′·(ns n)))*
using *F5 val-Zp-of-minus*
by *presburger*
then have *P10*: *val-Zp ((f′ · ns k) ⊖ (f′ · ns n)) ≥ val-Zp (f·(ns n)) − val-Zp (f′·(ns n))*
using *F7* **by** *metis*
have *P11*: *val-Zp (f′·(ns n)) ≠ ∞*
by (*simp add*: *F2*)
then have *val-Zp ((f′ · ns k) ⊖ (f′ · ns n)) ≥ (2 ∗ val-Zp (f′·a)) + 2 ^ n ∗ t − val-Zp (f′·(ns n))*
using *F3 P10*
by (*smt* (*verit*) *eint-add-cancel-fact eint-add-left-cancel-le order-trans*)
then have *P12*: *val-Zp ((f′ · ns k) ⊖ (f′ · ns n)) ≥ (2 ∗(val-Zp (f′·a))) + 2 ^ n ∗ t − (val-Zp (f′·a))*
by (*simp add*: *F2*)
have *P13*:*val-Zp ((f′ · ns k) ⊖ (f′ · ns n)) ≥ (val-Zp (f′·a)) + 2 ^ n ∗ t*
proof−
have *(2 ∗(val-Zp (f′·a))) + (2 ^ n ∗ t) − (val-Zp (f′·a)) = (2 ∗(val-Zp (f′·a))) − (val-Zp (f′·a)) + (2 ^ n ∗ t)*
using *eint-minus-comm* **by** *blast*
then show *?thesis* **using** *P12*
using *f′a-not-infinite* **by** *force*
qed
then have *P14*: *val-Zp ((f′ · ns k) ⊖ (f′ · ns n)) > (val-Zp (f′·a))*
using *f′a-not-infinite ge-plus-pos-imp-gt t-times-pow-pos* **by** *blast*
show *?thesis*
by (*meson F1 F2 P0 P14 equal-val-Zp f′-closed f′a-closed to-fun-closed*)
qed
have *P2*: *val-Zp (f·(ns k)) ≥ 2∗(val-Zp (f′·a)) + (2^k)∗t*
proof−
have *P23*: *2 ∗ (val-Zp (f′·a)) + ((2 ^ k) ∗ t) ≤ val-Zp (f · ns k)*
proof−
have *0*: *ns n ∈ carrier Zp*
by (*simp add*: *F1*)
have *1*: *local.divide (f · ns n) (f′ · ns n) ∈ carrier Zp*
using *F5* **by** *blast*
have *2*: *(poly-shift-iter 2 (taylor (ns n) f)) · ⊖ local.divide (f · ns n) (f′ · ns n) ∈ carrier Zp*
using *F1 F5 shift-closed 1*
by (*simp add*: *taylor-closed to-fun-closed*)
have *3*: *divide (f · ns n) (f′ · ns n) ⊗ deriv f (ns n) = f · ns n*
by (*metis F1 F2 F6 divide-formula f′-closed f′a-not-infinite f-closed*

mult-comm pderiv-eval-deriv to-fun-closed val-Zp-def)
have *4*: $f \in$ *carrier Zp-x*
by *simp*
obtain *c* **where** *c-def*: $c =$ *poly-shift-iter* (*2*::*nat*) (*taylor* (*ns n*) *f*) $\cdot \ominus$ *local.divide* ($f \cdot$ *ns n*) ($f' \cdot$ *ns n*)
by *blast*
then have *c-def'*: $c \in$ *carrier Zp* $\wedge$ $f \cdot$ (*ns n* $\ominus$ *local.divide* ($f \cdot$ *ns n*) ($f' \cdot$ *ns n*)) = $c \otimes$ *local.divide* ($f \cdot$ *ns n*) ($f' \cdot$ *ns n*) [^] (*2*::*nat*)
using *0 1 2 3 4 UP-cring.taylor-deg-1-eval'''*[*of Zp f ns n* (*divide* (*f*•(*ns n*)) (*f'*•(*ns n*))) *c*]
Zp-x-is-UP-cring
by *blast*
have *P230*: *f*•(*ns k*) = (*c* $\otimes$ (*divide* (*f*•(*ns n*)) (*f'*•(*ns n*)))[^](*2*::*nat*))
using *c-def'*
by (*simp add*: *kval newton-step-def*)
have *P231*: *val-Zp* (*f*•(*ns k*)) = *val-Zp c* + *2*∗(*val-Zp* (*f*•(*ns n*)) − *val-Zp*(*f'*•(*ns n*)))
proof−
have *P2310*: *val-Zp* (*f*•(*ns k*)) = *val-Zp c* + *val-Zp* ((*divide* (*f*•(*ns n*)) (*f'*•(*ns n*)))[^](*2*::*nat*))
by (*simp add*: *F5 P230 c-def' val-Zp-mult*)
have *P2311*: *val-Zp* ((*divide* (*f*•(*ns n*)) (*f'*•(*ns n*)))[^](*2*::*nat*)) = *2*∗(*val-Zp* (*f*•(*ns n*)) − *val-Zp*(*f'*•(*ns n*)))
by (*metis F5 F7 R.pow-zero mult.commute not-nonzero-Zp of-nat-numeral times-eint-simps*(*3*) *val-Zp-def val-Zp-pow' zero-less-numeral*)
thus *?thesis*
by (*simp add*: *P2310*)
qed
have *P232*: *val-Zp* (*f*•(*ns k*)) ≥ *2*∗(*val-Zp* (*f*•(*ns n*)) − *val-Zp*(*f'*•(*ns n*)))
by (*simp add*: *P231 c-def' val-pos*)
have *P236*: *val-Zp* (*f*•(*ns k*)) ≥ *2*∗(*2* ∗*val-Zp* (*f'*•*a*) + *2* ^ *n* ∗ *t*) − *2*∗ *val-Zp*(*f'*•(*ns n*))
using *P232 F3 eint-minus-ineq''*[*of val-Zp*(*f'*•(*ns n*)) (*2* ∗*val-Zp* (*f'*•*a*)) + *2* ^ *n* ∗ *t val-Zp* (*f*•(*ns n*)) *2*]
F2 eint-pow-int-is-pos **by** *auto*
hence *P237*: *val-Zp* (*f*•(*ns k*)) ≥(*4*∗*val-Zp* (*f'*•*a*)) + (*2*∗((*2* ^ *n*)∗ *t*)) − *2*∗ *val-Zp*(*f'*•(*ns n*))
proof−
have *2*∗(*2*∗*val-Zp* (*f'*•*a*) + *2* ^ *n* ∗ *t*) = (*4*∗*val-Zp* (*f'*•*a*)) + *2*∗(*2* ^ *n*)∗ *t*
using *distrib-left*[*of 2 2*∗*val-Zp* (*f'*•*a*) *2* ^ *n* ∗ *t*] *mult.assoc mult-one-right one-eint-def plus-eint-simps*(*1*)
hensel-factor-def val-Zp-def **by** *auto*
then show *?thesis*
using *P236*
by (*metis mult.assoc*)
qed

```
            hence P237:  val-Zp (f•(ns k)) ≥ 4*val-Zp (f'•a) + 2*(2 ^ n)* t −
2* val-Zp((f'•a))
              by (metis F2 mult.assoc)
            hence P238: val-Zp (f•(ns k)) ≥ 2*val-Zp (f'•a) + 2*(2 ^ n)* t
                  using eint-minus-comm[of 4*val-Zp (f'•a)  2*(2 ^ n)* t 2*
val-Zp((f'•a))]
              by (simp add: eint-int-minus-distr)
            thus ?thesis
              by (simp add: kval)
          qed
          thus ?thesis
            by blast
        qed
        show val-Zp (to-fun f' (ns k)) = val-Zp (f'•a) ∧
              2 * val-Zp (f'•a) + eint (2 ^ k) * t ≤ val-Zp (to-fun f (ns k))
          using P1 P2 by blast
      qed
    qed
qed

lemma newton-seq-closed:
shows ns m ∈ carrier Zp
  using newton-seq-props-induct
  by blast

lemma f-of-newton-seq-closed:
shows f • ns m ∈ carrier Zp
  by (simp add: to-fun-closed newton-seq-closed)

lemma newton-seq-fact1[simp]:
 val-Zp (f'•(ns k)) = val-Zp ((f'•a))
using newton-seq-props-induct by blast

lemma newton-seq-fact2:
⋀k.  val-Zp (f•(ns k)) ≥ 2*(val-Zp (f'•a)) + (2^k)*t
  by (meson le-iff-add newton-seq-props-induct)

lemma newton-seq-fact3:
val-Zp (f•(ns l)) ≥ val-Zp (f'•(ns l))
proof−
  have 2*(val-Zp (f'•a)) + (2^l)*t ≥ (val-Zp (f'•a))
    using f'a-closed ord-pos t-pos
   by (smt (verit) eint-pos-int-times-ge f'a-nonneg-val f'a-not-infinite ge-plus-pos-imp-gt
linorder-not-less nat-mult-not-infty order-less-le t-times-pow-pos)
  then show val-Zp (f • ns l) ≥ val-Zp (f' • ns l)
   using f'a-closed f'a-nonzero newton-seq-fact1[of l] newton-seq-fact2[of l]  val-Zp-def

    proof −
    show ?thesis
```

using ‹*eint 2 * val-Zp (f'•a) + eint (2 ^ l) * t ≤ val-Zp (to-fun f (ns l))*›
‹*val-Zp (f'•a) ≤ eint 2 * val-Zp (f'•a) + eint (2 ^ l) * t*› **by** *force*
qed
qed

lemma *newton-seq-fact4*[*simp*]:
assumes *f•(ns l) ≠***0**
shows *val-Zp (f•(ns l)) ≥ val-Zp (f'•(ns l))*
using *newton-seq-fact3* **by** *blast*

lemma *newton-seq-fact5*:
divide (f • ns l) (f' • ns l) ∈ carrier Zp
apply(*rule divide-closed*)
apply (*simp add*: *to-fun-closed newton-seq-closed*)
apply (*simp add*: *f'-closed to-fun-closed newton-seq-closed*)
by (*metis f'a-not-infinite newton-seq-fact1 val-Zp-def*)

lemma *newton-seq-fact6*:
(f'•(ns l)) ∈ nonzero Zp
apply(*rule ccontr*)
using *nonzero-memI nonzero-memE*
f'a-nonzero newton-seq-fact1 val-Zp-def
by (*metis* (*no-types*, *lifting*) *divide-closed f'-closed f'a-closed fa-closed hensel-factor-id*

hensel-hypothesis-weakened mult-zero-l newton-seq-closed order-less-le to-fun-closed val-Zp-mult)

lemma *newton-seq-fact7*:
(ns (Suc n)) ⊖ (ns n) = ⊖divide (f•(ns n)) (f'•(ns n))
using *newton-seq.simps*(*2*)[*of n*] *newton-seq-fact5*[*of n*]
newton-seq-closed[*of Suc n*] *newton-seq-closed*[*of n*]
R.ring-simprules
unfolding *newton-step-def a-minus-def*
by (*smt* (*verit*))

lemma *newton-seq-fact8*:
assumes *f•(ns l) ≠***0**
shows *divide (f • ns l) (f' • ns l) ∈ nonzero Zp*
using *assms divide-nonzero*[*of f • ns l f' • ns l*]
nonzero-memI
using *f-of-newton-seq-closed newton-seq-fact3 newton-seq-fact6* **by** *blast*

lemma *newton-seq-fact9*:
assumes *f•(ns n) ≠***0**
shows *val-Zp((ns (Suc n)) ⊖ (ns n)) = val-Zp (f•(ns n)) − val-Zp (f'•(ns n))*
using *newton-seq-fact7 val-of-divide newton-seq-fact6 assms nonzero-memI*
f-of-newton-seq-closed newton-seq-fact4 newton-seq-fact5
by (*metis val-Zp-of-minus*)

Assuming no element of the Newton sequence is a root of f, the Newton

sequence is Cauchy.

lemma *newton-seq-is-Zp-cauchy-0*:
assumes $\bigwedge k.\ f\cdot(ns\ k) \neq \mathbf{0}$
shows *is-Zp-cauchy ns*
proof(*rule is-Zp-cauchyI'*)
show *P0*: $ns \in$ *closed-seqs Zp*
proof(*rule closed-seqs-memI*)
show $\bigwedge k.$ *ns k* $\in$ *carrier Zp*
by (*simp add*: *newton-seq-closed*)
qed
show $\forall n.\ \exists k.\ \forall m.\ k \leq int\ m \longrightarrow int\ n \leq$ *val-Zp* (*ns* (*Suc m*) $\ominus$ *ns m*)
proof
fix *n*
show $\exists k.\ \forall m.\ k \leq int\ m \longrightarrow int\ n \leq$ *val-Zp* (*ns* (*Suc m*) $\ominus$ *ns m*)
proof(*induction n*)
case *0*
have *B0*: $\forall$ *n0 n1*. *0* $<$ *n0* $\wedge$ *0* $<$ *n1* $\longrightarrow$ *ns n0 0* $=$ *ns n1 0*
apply *auto*
proof−
fix *n0 n1*::*nat*
assume *A*: *0* $<$ *n0 0* $<$ *n1*
show *ns n0 0* $=$ *ns n1 0*
proof−
have *0*: *ns n0* $\in$ *carrier Zp*
using *P0*
by (*simp add*: *newton-seq-closed*)
have *1*: *ns n1* $\in$ *carrier Zp*
using *P0*
by (*simp add*: *newton-seq-closed*)
show *?thesis*
using *0 1 Zp-defs*(*3*) *prime*
by (*metis p-res-ring-0' residue-closed*)
qed
qed
have $\forall m.$ *1* $\leq$ *int m* $\longrightarrow$ *int 0* $\leq$ *val-Zp-dist* (*newton-step* (*ns m*)) (*ns m*)
proof
fix *m*
show *1* $\leq$ *int m* $\longrightarrow$ *int 0* $\leq$ *val-Zp-dist* (*newton-step* (*ns m*)) (*ns m*)
proof
assume *1* $\leq$ *int m*
then have *C0*:*ns* (*Suc m*) *0* $=$ *ns m 0*
using *B0*
by (*metis int-one-le-iff-zero-less int-ops*(*1*) *less-Suc-eq-0-disj of-nat-less-iff*)
then show *int 0* $\leq$ *val-Zp-dist* (*newton-step* (*ns m*)) (*ns m*)
proof−
have (*newton-step* (*ns m*)) $\neq$(*ns m*)
proof−
have *A0*: *divide* ($f\cdot$(*ns m*)) ($f'\cdot$(*ns m*)) $\neq \mathbf{0}$
proof−

```
          have 0: (f•(ns m)) ≠ 0
            using assms by auto
          have 1:  (f'•(ns m)) ∈ carrier Zp
               by (simp add: UP-cring.to-fun-closed Zp-x-is-UP-cring f'-closed
newton-seq-closed)
          have 2:  (f'•(ns m)) ≠ 0
         using newton-seq-fact6 not-nonzero-memI by blast
          show ?thesis using 0 1 2
         by (metis R.r-null divide-formula f-closed to-fun-closed newton-seq-closed
newton-seq-fact4)
        qed
        have A2: local.divide (f • ns m) (f' • ns m) ∈ carrier Zp
          using newton-seq-fact5 by blast
        have A3: ns m ∈ carrier Zp
          by (simp add: newton-seq-closed)
        have A4: newton-step (ns m) ∈ carrier Zp
          by (metis newton-seq.simps(2) newton-seq-closed)
        show ?thesis
          apply(rule ccontr)
          using A4 A3 A2 A0 newton-step-def[of (ns m)]
          by (simp add: a-minus-def)
      qed
      then show ?thesis using C0
        by (metis newton-seq.simps(2) newton-seq-closed val-Zp-dist-res-eq2)
    qed
   qed
   qed
   then show ?case
     using val-Zp-def val-Zp-dist-def
     by (metis int-ops(1) newton-seq.simps(2) zero-eint-def)
 next
   case (Suc n)
   show ∃ k. ∀ m. k ≤ int m ⟶ int (Suc n) ≤ val-Zp (ns (Suc m) ⊖ ns m)
   proof−
     obtain k0 where k0-def: k0 ≥0 ∧ (∀ m. k0 ≤ int m ⟶ int n ≤ val-Zp
(ns (Suc m) ⊖ ns m))
       using Suc.IH
       by (metis int-nat-eq le0 nat-le-iff of-nat-0-eq-iff )
     have I0: ⋀l. val-Zp (ns (Suc l) ⊖ ns l) = val-Zp (f• (ns l)) − val-Zp (f'•(ns
l))
      proof−
       fix l
       have I00:(ns (Suc l) ⊖ ns l) = (⊖ divide (f•(ns l)) (f'•(ns l)))
       proof−
         have local.divide (f • ns l) (f' • ns l) ∈ carrier Zp
           by (simp add: newton-seq-fact5)
         then show ?thesis
           using newton-seq.simps(2)[of l] newton-seq-closed R.ring-simprules
           unfolding newton-step-def a-minus-def
```

by (*metis add-comm*)
qed
have *I01*: *val-Zp* (*ns* (*Suc l*) ⊖ *ns l*) = *val-Zp* (*divide* (*f*•(*ns l*)) (*f′*•(*ns l*)))
proof−
have *I010*: (*divide* (*f*•(*ns l*)) (*f′*•(*ns l*))) ∈*carrier Zp*
by (*simp add*: *newton-seq-fact5*)
have *I011*: (*divide* (*f*•(*ns l*)) (*f′*•(*ns l*))) ≠ **0**
proof−
have *A*: (*f*•(*ns l*)) ≠**0**
by (*simp add*: *assms*)
have *B*: (*f′*•(*ns l*)) ∈*carrier Zp*
using *nonzero-memE newton-seq-fact6* **by** *auto*
then have *C*: (*f′*•(*ns l*)) ∈*nonzero Zp*
using *f′a-closed fa-closed fa-nonzero hensel-factor-id hensel-hypothesis-weakened newton-seq-fact1*[*of l*] *not-nonzero-Zp val-Zp-def*
by *fastforce*
then show *?thesis* **using** *I010 A*
by (*metis B R.r-null divide-formula f-closed to-fun-closed newton-seq-closed newton-seq-fact4 nonzero-memE*(*2*))
qed
then have *val-Zp* (*divide* (*f*•(*ns l*)) (*f′*•(*ns l*)))
= *val-Zp* (⊖ *divide* (*f*•(*ns l*)) (*f′*•(*ns l*)))
using *I010 not-nonzero-Zp val-Zp-of-minus* **by** *blast*
then show *?thesis* **using** *I00* **by** *metis*
qed
have *I02*: *val-Zp* (*f*•(*ns l*)) ≥ *val-Zp* (*f′*•(*ns l*))
using *assms newton-seq-fact4*
by *blast*
have *I03*: (*f*•(*ns l*)) ∈ *nonzero Zp*
by (*meson UP-cring.to-fun-closed Zp-x-is-UP-cring assms f-closed newton-seq-closed not-nonzero-Zp*)
have *I04*: *f′*•(*ns l*) ∈ *nonzero Zp*
by (*simp add*: *newton-seq-fact6*)
have *I05* : *val-Zp* (*divide* (*f*•(*ns l*)) (*f′*•(*ns l*))) = *val-Zp* (*f*• (*ns l*)) − *val-Zp* (*f′*•(*ns l*))
using *I02 I03 I04 I01 assms newton-seq-fact9* **by** *auto*
then show *val-Zp* (*ns* (*Suc l*) ⊖ *ns l*) = *val-Zp* (*f*• (*ns l*)) − *val-Zp* (*f′*•(*ns l*))
using *I01* **by** *simp*
qed
have ∀ *m*. *int*(*Suc n*) + *k0* + *1* ≤ *int m* ⟶ *int* (*Suc n*) ≤ *val-Zp-dist* (*newton-step* (*ns m*)) (*ns m*)
proof
fix *m*
show *int* (*Suc n*) + *k0* + *1* ≤ *int m* ⟶ *int* (*Suc n*) ≤ *val-Zp-dist* (*newton-step* (*ns m*)) (*ns m*)
proof
assume *A*: *int* (*Suc n*) + *k0* + *1* ≤ *int m*

show *int (Suc n) ≤ val-Zp-dist (newton-step (ns m)) (ns m)*
proof−
have *0*: *val-Zp-dist (newton-step (ns m)) (ns m) = val-Zp (f• (ns m)) − val-Zp (f′•(ns m))*
using *I0 val-Zp-dist-def* **by** *auto*
have *1*: *val-Zp (f• (ns m)) − val-Zp (f′•(ns m)) > int n*
proof−
have *val-Zp (f• (ns m)) ≥ 2∗(val-Zp (f′•a)) + (2^m)∗t*
by (*simp add*: *newton-seq-fact2*)
then have *10*:*val-Zp (f• (ns m)) − val-Zp (f′•(ns m)) ≥ 2∗(val-Zp (f′•a)) + (2^m)∗t − val-Zp (f′•(ns m))*
by (*simp add*: *eint-minus-ineq*)
have *2^m ∗ t > m*
apply(*induction m*)
using *one-eint-def zero-eint-def* **apply** *auto*[*1*]
proof− **fix** *m*
assume *IH* : *int m < 2 ^ m ∗ t*
then have *((2 ^ (Suc m)) ∗ t) = 2∗ ((2 ^ m) ∗ t)*
by (*metis mult.assoc power-Suc times-eint-simps*(*1*))
then show *int (Suc m) < 2 ^ Suc m ∗ t*
using *IH t-neq-infty* **by** *force*
qed
then have *100*: *2^m ∗ t > int m*
by *blast*
have *int m ≥2 + (int n + k0)*
using *A* **by** *simp*
hence *1000*: *2^m ∗ t > 2 + (int n + k0)*
using *100*
by (*meson eint-ord-simps*(*2*) *less-le-trans linorder-not-less*)
have *2 + (int n + k0) > 1 + int n*
using *k0-def* **by** *linarith*
then have *2^m ∗ t > 1 + int n*
using *1000 eint-ord-simps*(*2*) *k0-def less-le-trans linorder-not-less*
proof −
have *eint (2 + (int n + k0)) < t ∗ eint (int (2 ^ m))*
by (*metis 1000 mult.commute numeral-power-eq-of-nat-cancel-iff*)
then have *eint (int (Suc n)) < t ∗ eint (int (2 ^ m))*
by (*metis ‹1 + int n < 2 + (int n + k0)› eint-ord-simps*(*2*) *less-trans of-nat-Suc*)
then show *?thesis*
by (*simp add*: *mult.commute*)
qed
hence *2∗val-Zp (f′•a) + eint (2 ^ m) ∗ t ≥ 2∗(val-Zp (f′•a)) + 1 + int n*
by (*smt* (*verit*) *eSuc-eint eint-add-left-cancel-le iadd-Suc iadd-Suc-right order-less-le*)
then have *11*: *val-Zp (f• (ns m)) − val-Zp (f′•(ns m)) ≥ 2∗(val-Zp (f′•a)) + 1 + int n − val-Zp (f′•(ns m))*
using *10*

```
            by (smt (verit) ‹eint 2 * val-Zp (f'•a) + eint (2 ^ m) * t ≤ val-Zp
(to-fun f (ns m))›
                f'a-not-infinite eint-minus-ineq hensel-axioms newton-seq-fact1
order-trans)
          have 12: val-Zp (f'•(ns m)) = val-Zp (f'•a)
            using nonzero-memE newton-seq-fact1 newton-seq-fact6 val-Zp-def
val-Zp-def
           by auto
          then have 13: val-Zp (f• (ns m)) − val-Zp (f'•(ns m))
                      ≥ 2*(val-Zp (f'•a)) + (1 + int n) − val-Zp ((f'•a))
           using 11
           by (metis eint-1-iff(1) group-cancel.add1 plus-eint-simps(1))
          then have 14:val-Zp (f• (ns m)) − val-Zp (f'•(ns m))
                      ≥ 1 + int n + val-Zp ((f'•a))
           using eint-minus-comm[of 2*(val-Zp (f'•a)) 1 + int n val-Zp ((f'•a))]

           by (simp add: Groups.add-ac(2))
          then show ?thesis
        by (smt (verit) Suc-ile-eq add.right-neutral eint.distinct(2) f'a-nonneg-val
ge-plus-pos-imp-gt order-less-le)
          qed
          then show ?thesis
           by (smt (verit) 0 Suc-ile-eq of-nat-Suc)
         qed
        qed
       qed
       then show ?thesis
        using val-Zp-def val-Zp-dist-def
        by (metis newton-seq.simps(2))
      qed
    qed
  qed
qed

lemma eventually-zero:
f • ns (k + m) = 0 ⟹ f • ns (k + Suc m) = 0
proof−
  assume A: f • ns (k + m) = 0
  have 0: ns (k + Suc m) = ns (k + m) ⊖ (divide (f • ns (k + m)) (f' • ns (k +
m)))
    by (simp add: newton-step-def)
  have 1: (divide (f • ns (k + m)) (f' • ns (k + m))) = 0
    by (simp add: A divide-def)
  show f • ns (k + Suc m) = 0
    using A 0 1
    by (simp add: a-minus-def newton-seq-closed)
qed
```

The Newton Sequence is Cauchy:

```
lemma newton-seq-is-Zp-cauchy:
is-Zp-cauchy ns
proof(cases ∀ k. f•(ns k) ≠0)
  case True
  then show ?thesis using newton-seq-is-Zp-cauchy-0
    by blast
next
  case False
  obtain k where k-def:f•(ns k) = 0
    using False by blast
  have 0: ⋀m. (ns (m + k)) = (ns k)
  proof−
    fix m
    show (ns (m + k)) = (ns k)
    proof(induction m)
      case 0
      then show ?case
        by simp
    next
      case (Suc m)
      show (ns (Suc m + k)) = (ns k)
      proof−
        have f • ns (m + k) = 0
          by (simp add: Suc.IH k-def)
        then have divide ( f • ns (m + k)) (f' • ns (m + k)) = 0
          by (simp add: divide-def)
        then show ?thesis using newton-step-def
          by (simp add: Suc.IH a-minus-def newton-seq-closed)
      qed
    qed
  qed
  show is-Zp-cauchy ns
    apply(rule is-Zp-cauchyI)
    apply (simp add: closed-seqs-memI newton-seq-closed)
  proof−
    show ⋀n.⋀n. ∃ N. ∀ n0 n1. N < n0 ∧ N < n1 ⟶ ns n0 n = ns n1 n
    proof−
      fix n
      show ∃ N. ∀ n0 n1. N < n0 ∧ N < n1 ⟶ ns n0 n = ns n1 n
      proof−
        have ∀ n0 n1. k < n0 ∧ k < n1 ⟶ ns n0 n = ns n1 n
          apply auto
        proof−
          fix n0 n1
          assume A0: k < n0
          assume A1: k < n1
          obtain m0 where m0-def: n0 = k + m0
            using A0 less-imp-add-positive by blast
          obtain m1 where m1-def: n1 = k + m1
```

using *A1 less-imp-add-positive* **by** *auto*
show *ns n0 n = ns n1 n*
using *0 m0-def m1-def*
by (*metis add.commute*)
qed
then show *?thesis* **by** *blast*
qed
qed
qed
qed

30.4 The Proof of Hensel's Lemma

lemma *pre-hensel*:
val-Zp (*a* ⊖ (*ns n*)) > *val-Zp* (*f'*•*a*)
∃ *N*. ∀ *n*. *n*> *N* ⟶ (*val-Zp* (*a* ⊖ (*ns n*)) = *val-Zp* (*divide* (*f*•*a*) (*f'*•*a*)))
val-Zp (*f'*•(*ns n*)) = *val-Zp* (*f'*•*a*)
proof−
show *val-Zp* (*a* ⊖ (*ns n*)) > *val-Zp* (*f'*•*a*)
proof(*induction n*)
case *0*
then show *?case*
by (*simp add*: *val-Zp-def*)
next
case (*Suc n*)
show *val-Zp* (*a* ⊖ (*ns* (*Suc n*))) > *val-Zp* (*f'*•*a*)
proof−
have *I0*: *val-Zp* ((*ns* (*Suc n*)) ⊖ (*ns n*)) > *val-Zp* (*f'*•*a*)
proof(*cases* (*ns* (*Suc n*)) = (*ns n*))
case *True*
then show *?thesis*
by (*simp add*: *newton-seq-closed val-Zp-def*)
next
case *False*
have *00*:(*ns* (*Suc n*)) ⊖ (*ns n*) = ⊖*divide* (*f*•(*ns n*)) (*f'*•(*ns n*))
using *newton-seq-fact7* **by** *blast*
then have *0*: *val-Zp*((*ns* (*Suc n*)) ⊖ (*ns n*)) = *val-Zp* (*divide* (*f*•(*ns n*)) (*f'*•(*ns n*)))
using *newton-seq-fact5 val-Zp-of-minus* **by** *presburger*
have *1*: (*f*•(*ns n*)) ∈ *nonzero Zp*
by (*metis False R.minus-zero R.r-right-minus-eq 00 divide-def f-closed to-fun-closed*
newton-seq-closed not-nonzero-Zp)
have *2*: *f'*•(*ns n*) ∈ *nonzero Zp*
by (*simp add*: *newton-seq-fact6*)
have *val-Zp* (*f*•(*ns n*)) ≥ *val-Zp* (*f'*•(*ns n*))
using *nonzero-memE* ‹*f* • *ns n* ∈ *nonzero Zp*› *newton-seq-fact4* **by** *blast*
then have *3*:*val-Zp*((*ns* (*Suc n*)) ⊖ (*ns n*)) = *val-Zp* (*f*•(*ns n*)) − *val-Zp* (*f'*•(*ns n*))

using *0 1 2 newton-seq-fact9 nonzero-memE(2)* **by** *blast*
have *4*: *val-Zp* $(f \cdot ns\ n) \geq (2 * val\text{-}Zp\ (f'\cdot a)) + 2 \hat{}\ n * t$
using *newton-seq-fact2*[*of n*] **by** *metis*
then have *5*: *val-Zp*$((ns\ (Suc\ n)) \ominus (ns\ n)) \geq ((2 * val\text{-}Zp\ (f'\cdot a)) + 2 \hat{}\ n * t) - val\text{-}Zp\ (f'\cdot(ns\ n))$
using *3 eint-minus-ineq f'a-not-infinite newton-seq-fact1* **by** *presburger*
have *6*: $((ns\ (Suc\ n)) \ominus (ns\ n)) \in$ *nonzero Zp*
using *False not-eq-diff-nonzero newton-seq-closed* **by** *blast*
then have *val-Zp*$((ns\ (Suc\ n)) \ominus (ns\ n)) \geq (2 * val\text{-}Zp\ (f'\cdot a)) + 2 \hat{}\ n * t - val\text{-}Zp\ ((f'\cdot a))$
using *5* **by** *auto*
then have *7*: *val-Zp*$((ns\ (Suc\ n)) \ominus (ns\ n)) \geq (val\text{-}Zp\ (f'\cdot a)) + 2 \hat{}\ n * t$
by (*simp add*: *eint-minus-comm*)
then show *val-Zp*$((ns\ (Suc\ n)) \ominus (ns\ n)) > (val\text{-}Zp\ (f'\cdot a))$
using *f'a-not-infinite ge-plus-pos-imp-gt t-times-pow-pos* **by** *blast*
qed
have *val-Zp* $((ns\ (Suc\ n)) \ominus (ns\ n)) = val\text{-}Zp\ ((ns\ n) \ominus (ns\ (Suc\ n)))$
using *newton-seq-closed*[*of n*] *newton-seq-closed*[*of Suc n*]
val-Zp-def val-Zp-dist-def val-Zp-dist-sym val-Zp-def
by *auto*
then have *I1*: *val-Zp* $((ns\ n) \ominus (ns\ (Suc\ n))) > val\text{-}Zp\ (f'\cdot a)$
using *I0*
by *presburger*
have *I2*: $(a \ominus (ns\ n)) \oplus ((ns\ n) \ominus (ns\ (Suc\ n))) = (a \ominus (ns\ (Suc\ n)))$
by (*metis R.plus-diff-simp add-comm local.a-closed newton-seq-closed*)

then have *val-Zp* $(a \ominus (ns\ (Suc\ n))) \geq min\ (val\text{-}Zp\ (a \ominus ns\ n))\ (val\text{-}Zp\ (ns\ n \ominus ns\ (Suc\ n)))$
by (*metis R.minus-closed local.a-closed newton-seq-closed val-Zp-ultrametric*)

thus *?thesis*
using *I1 Suc.IH eint-min-ineq* **by** *blast*
qed
qed
show *val-Zp* $(f'\cdot(ns\ n)) = val\text{-}Zp\ (f'\cdot a)$
using *newton-seq-fact1* **by** *blast*
show $\exists N. \forall n.\ n > N \longrightarrow (val\text{-}Zp\ (a \ominus (ns\ n)) = val\text{-}Zp\ (divide\ (f\cdot a)\ (f'\cdot a)))$
proof−
have *P*: $\bigwedge m.\ m > 1 \Longrightarrow (val\text{-}Zp\ (a \ominus (ns\ m)) = val\text{-}Zp\ (divide\ (f\cdot a)\ (f'\cdot a)))$
proof−
fix *n*::*nat*
assume *AA*: *n >1*
show $(val\text{-}Zp\ (a \ominus (ns\ n)) = val\text{-}Zp\ (divide\ (f\cdot a)\ (f'\cdot a)))$
proof(*cases* $(ns\ 1) = a$)
case *True*
have *T0*: $\bigwedge k.\ \forall n.\ n \leq k \longrightarrow\ ns\ n = a$
proof−
fix *k*
show $\forall n.\ n \leq k \longrightarrow\ ns\ n = a$

```
      proof(induction k)
        case 0
        then show ?case
          by simp
      next
        case (Suc k)
        show ∀ n≤Suc k. ns n = a apply auto
        proof−
          fix n
          assume A: n ≤Suc k
          show ns n = a
          proof(cases n < Suc k)
            case True
            then show ?thesis using Suc.IH by auto
          next
            case False thus ?thesis
              using A Suc.IH True by auto
          qed
        qed
      qed
      show val-Zp (a ⊖ ns n) = val-Zp (local.divide (f•a) (f'•a))
        by (metis T0  Zp-def Zp-defs(3) f'a-closed f'a-nonzero fa-nonzero
          hensel.fa-closed hensel-axioms hensel-hypothesis-weakened le-eq-less-or-eq

          newton-seq-fact9 not-nonzero-Qp order-less-le val-of-divide)
    next
      case False
      have F0: (1::nat) ≤ n
        using AA by simp
      have (f•a) ≠ 0
        by simp
      have ⋀k. val-Zp (a ⊖ ns (Suc k)) = val-Zp (local.divide (f•a) (f'•a))
      proof−
        fix k
        show  val-Zp (a ⊖ ns (Suc k)) = val-Zp (local.divide (f•a) (f'•a))
        proof(induction k)
          case 0
          have (a ⊖ ns (Suc 0)) = (local.divide (f•a) (f'•a))
           by (metis R.minus-minus Zp-def hensel.newton-seq-fact7 hensel-axioms

              local.a-closed minus-a-inv newton-seq.simps(1) newton-seq.simps(2)
newton-seq-fact5 newton-step-closed)
          then show ?case by simp
        next
          case (Suc k)
         have I0: ns (Suc (Suc k)) = ns (Suc k) ⊖ (divide (f•(ns (Suc k))) (f'•(ns
(Suc k))))
            by (simp add: newton-step-def)
```

have *I1*: *val-Zp* (*f*•(*ns* (*Suc k*))) ≥ *val-Zp*(*f′*•(*ns* (*Suc k*)))
using *newton-seq-fact3* **by** *blast*
have *I2*: (*divide* (*f*•(*ns* (*Suc k*))) (*f′*•(*ns* (*Suc k*)))) ∈ *carrier Zp*
using *newton-seq-fact5* **by** *blast*
have *I3*: *ns* (*Suc* (*Suc k*)) ⊖ *ns* (*Suc k*) = ⊖(*divide* (*f*•(*ns* (*Suc k*))) (*f′*•(*ns* (*Suc k*))))
using *I0 I2 newton-seq-fact7* **by** *blast*
then have *val-Zp* (*ns* (*Suc* (*Suc k*)) ⊖ *ns* (*Suc k*)) = *val-Zp* (*divide* (*f*•(*ns* (*Suc k*))) (*f′*•(*ns* (*Suc k*))))
using *I2 val-Zp-of-minus*
by *presburger*
then have *val-Zp* (*ns* (*Suc* (*Suc k*)) ⊖ *ns* (*Suc k*)) = *val-Zp* (*f*•(*ns* (*Suc k*))) − *val-Zp* (*f′*•(*ns* (*Suc k*)))
by (*metis I1 R.zero-closed Zp-def newton-seq-fact6 newton-seq-fact9 padic-integers.val-of-divide padic-integers-axioms*)
then have *I4*: *val-Zp* (*ns* (*Suc* (*Suc k*)) ⊖ *ns* (*Suc k*)) = *val-Zp* (*f*•(*ns* (*Suc k*))) − *val-Zp* ((*f′*•*a*))
using *newton-seq-fact1* **by** *presburger*
have *F3*: *val-Zp* (*a* ⊖ *ns* (*Suc k*)) = *val-Zp* (*local.divide* (*f*•*a*) (*f′*•*a*))
using *Suc.IH* **by** *blast*
have *F4*: *a* ⊖ *ns* (*Suc* (*Suc k*)) = (*a* ⊖ (*ns* (*Suc k*))) ⊕ (*ns* (*Suc k*)) ⊖ *ns* (*Suc* (*Suc k*))
by (*metis R.ring-simprules*(17) *a-minus-def add-comm local.a-closed newton-seq-closed*)
have *F5*: *val-Zp* ((*ns* (*Suc k*)) ⊖ *ns* (*Suc* (*Suc k*))) > *val-Zp* (*a* ⊖ (*ns* (*Suc k*)))
proof−
have *F50*: *val-Zp* ((*ns* (*Suc k*)) ⊖ *ns* (*Suc* (*Suc k*))) = *val-Zp* (*f*•(*ns* (*Suc k*))) − *val-Zp* ((*f′*•*a*))
by (*metis I4 R.minus-closed minus-a-inv newton-seq-closed val-Zp-of-minus*)

have *F51*: *val-Zp* (*f*•(*ns* (*Suc k*))) > *val-Zp* ((*f*•*a*))
proof−
have *F510*: *val-Zp* (*f*•(*ns* (*Suc k*))) ≥ 2∗*val-Zp* (*f′*•*a*) + 2^(*Suc k*)∗*t*
using *newton-seq-fact2* **by** *blast*
hence *F511*: *val-Zp* (*f*•(*ns* (*Suc k*))) ≥ 2∗*val-Zp* (*f′*•*a*) + 2∗*t*
using *eint-plus-times*[*of t* 2∗*val-Zp* (*f′*•*a*) 2^(*Suc k*) *val-Zp* (*f*•(*ns* (*Suc k*))) 2] *t-pos*
by (*simp add*: *order-less-le*)
have *F512*: 2∗*val-Zp* (*f′*•*a*) + 2∗*t* = 2 ∗*val-Zp* (*f*•*a*) − 2∗ *val-Zp* (*f′*•*a*)
unfolding *hensel-factor-def*
using *eint-minus-distr*[*of val-Zp* (*f*•*a*) 2 ∗ *val-Zp* (*f′*•*a*) 2]
eint-minus-comm[*of* - - *eint* 2 ∗ (*eint* 2 ∗ *val-Zp* (*f′*•*a*))]
by (*smt* (*verit*) *eint-2-minus-1-mult eint-add-cancel-fact eint-minus-comm f′a-not-infinite hensel-hypothesis nat-mult-not-infty order-less-le*)
hence 2∗*val-Zp* (*f′*•*a*) + 2∗*t* > *val-Zp* (*f*•*a*)
using *hensel-hypothesis*
by (*smt* (*verit*) *add-diff-cancel-eint eint-add-cancel-fact eint-add-left-cancel-le*

```
        eint-pos-int-times-gt f'a-not-infinite hensel-factor-def nat-mult-not-infty
order-less-le t-neq-infty t-pos)
        thus ?thesis using F512
          using F511 less-le-trans by blast
      qed
      thus ?thesis
    by (metis F3 F50 Zp-def divide-closed eint-add-cancel-fact eint-minus-ineq

                  f'a-closed f'a-nonzero f'a-not-infinite fa-closed fa-nonzero
hensel.newton-seq-fact7
             hensel-axioms newton-seq.simps(1) newton-seq-fact9 order-less-le
val-Zp-of-minus)
     qed
     have a ⊖ ns (Suc k) ⊕ (ns (Suc k) ⊖ ns (Suc (Suc k))) = a ⊖ ns (Suc
(Suc k))
       by (metis F4 a-minus-def add-assoc)
      then show F6: val-Zp (a ⊖ ns (Suc (Suc k))) = val-Zp (local.divide
(f•a) (f'•a))
       using F5 F4 F3
       by (metis R.minus-closed local.a-closed newton-seq-closed order-less-le
val-Zp-not-equal-ord-plus-minus val-Zp-ultrametric-eq'')
    qed
   qed
   thus ?thesis
    by (metis AA less-imp-add-positive plus-1-eq-Suc)
  qed
 qed
qed

lemma hensel-seq-comp-f:
 res-lim ((to-fun f) ∘ ns) = 0
proof−
  have A: is-Zp-cauchy ((to-fun f) ∘ ns)
   using f-closed is-Zp-continuous-def newton-seq-is-Zp-cauchy polynomial-is-Zp-continuous

    by blast
  have Zp-converges-to ((to-fun f) ∘ ns) 0
   apply(rule Zp-converges-toI)
   using A is-Zp-cauchy-def apply blast
    apply simp
  proof−
   fix n
   show  ∃ N. ∀ k>N. (((to-fun f) ∘ ns) k) n = 0 n
   proof−
    have 0: ⋀k. (k::nat)>3 ⟶  val-Zp (f•(ns k)) > k
```

```
    proof
      fix k::nat
      assume A: k >3
      show val-Zp (f•(ns k)) > k
      proof−
        have 0:  val-Zp (f•(ns k)) ≥  2∗(val-Zp (f'•a)) + (2^k)∗t
          using newton-seq-fact2 by blast
        have 1: 2∗(val-Zp (f'•a)) + (2^k)∗t > k
        proof−
          have (2^k)∗t ≥ (2^k)
            apply(cases t = ∞)
             apply simp
          using t-pos eint-mult-mono'
          proof −
            obtain ii :: eint ⇒ int where
              f1: ∀ e. (∞ ≠ e ∨ (∀ i. eint i ≠ e)) ∧ (eint (ii e) = e ∨ ∞ = e)
              by (metis not-infinity-eq)
            then have 0 < ii t
             by (metis (no-types) eint-ord-simps(2) t-neq-infty t-pos zero-eint-def)
            then show ?thesis
          using f1 by (metis eint-pos-int-times-ge eint-mult-mono linorder-not-less

                        mult.commute order-less-le t-neq-infty t-pos t-times-pow-pos)
          qed
          hence  2∗(val-Zp (f'•a)) + (2^k)∗t ≥ (2^k)
           by (smt (verit) Groups.add-ac(2) add.right-neutral eint-2-minus-1-mult
eint-pos-times-is-pos
                  eint-pow-int-is-pos f'a-nonneg-val ge-plus-pos-imp-gt idiff-0-right
linorder-not-less
                nat-mult-not-infty order-less-le t-neq-infty)
          then have   2∗(val-Zp (f'•a)) + (2^k)∗t > k
            using A  of-nat-1 of-nat-add of-nat-less-two-power
          by (smt (verit) eint-ord-simps(1) linorder-not-less order-trans)
          then show ?thesis
            by metis
        qed
        thus ?thesis
          using 0 less-le-trans by blast
      qed
    qed
    have 1: ⋀k. (k::nat)>3 ⟶  (f•(ns k)) k = 0
    proof
      fix k::nat
      assume B: 3<k
      show  (f•(ns k)) k = 0
      proof−
        have B0:  val-Zp (f•(ns k)) > k
          using 0 B
          by blast
```

```
        then show ?thesis
          by (simp add: f-of-newton-seq-closed zero-below-val-Zp)
      qed
    qed
    have ∀ k>(max 3 n). (((to-fun f) ∘ ns) k) n = 0 n
      apply auto
    proof−
      fix k::nat
      assume A: 3< k
      assume A': n < k
      have A0: (f•(ns k)) k = 0
        using 1[of k] A by auto
      then have (f•(ns k)) n = 0
        using A A'
        using above-ord-nonzero[of (f•(ns k))]
     by (smt (verit) UP-cring.to-fun-closed Zp-x-is-UP-cring f-closed le-eq-less-or-eq

          newton-seq-closed of-nat-mono residue-of-zero(2) zero-below-ord)
      then show A1:  to-fun f (ns k) n = 0 n
        by (simp add: residue-of-zero(2))
    qed
    then show ?thesis by blast
  qed
 qed
 then show ?thesis
   by (metis Zp-converges-to-def unique-limit')
qed

lemma full-hensels-lemma:
 obtains α where
      f•α = 0 and α ∈ carrier Zp
      val-Zp (a ⊖ α) > val-Zp (f'•a)
      (val-Zp (a ⊖ α) = val-Zp (divide (f•a) (f'•a)))
      val-Zp (f'•α) = val-Zp (f'•a)
proof(cases ∃ k. f•(ns k) =0)
 case True
 obtain k where k-def: f•(ns k) =0
   using True by blast
 obtain N where N-def: ∀ n. n> N ⟶ (val-Zp (a ⊖ (ns n)) = val-Zp (divide
(f•a) (f'•a)))
   using pre-hensel(2) by blast
 have Z: ⋀n. n ≥k ⟹ f•(ns n) =0
 proof−
   fix n
   assume A: n ≥k
   obtain l where l-def:n = k + l
     using A le-Suc-ex
     by blast
   have ⋀m. f•(ns (k+m)) =0
```

```
    proof−
      fix m
      show f•(ns (k+m)) =0
        apply(induction m)
         apply (simp add: k-def)
        using  eventually-zero
        by simp
    qed
    then show f•(ns n) =0
      by (simp add: l-def)
  qed
  obtain M where M-def: M = N + k
    by simp
  then have M-root: f•(ns M) =0
    by (simp add: Z)
  obtain α where alpha-def: α= ns M
    by simp
  have T0: f•α = 0
    using alpha-def M-root
    by auto
  have T1:    val-Zp (a ⊖ α) > val-Zp (f'•a)
    using alpha-def pre-hensel(1) by blast
  have T2: (val-Zp (a ⊖ α) = val-Zp (divide (f•a) (f'•a)))
    by (metis M-def N-def alpha-def fa-nonzero k-def
       less-add-same-cancel1 newton-seq.elims zero-less-Suc)
  have T3:  val-Zp (f'•α) = val-Zp (f'•a)
    using alpha-def newton-seq-fact1 by blast
  show ?thesis using T0 T1 T2 T3
    using that alpha-def newton-seq-closed
    by blast
next
  case False
  then have Nz: ⋀k. f•(ns k) ≠0
    by blast
  have ns-cauchy: is-Zp-cauchy ns
    by (simp add: newton-seq-is-Zp-cauchy)
  have fns-cauchy: is-Zp-cauchy ((to-fun f) ∘ ns)
    using f-closed is-Zp-continuous-def ns-cauchy polynomial-is-Zp-continuous by
blast
  have F0: res-lim ((to-fun f) ∘ ns) = 0
  proof−
    show ?thesis
      using hensel-seq-comp-f by auto
  qed
  obtain α where alpha-def: α = res-lim ns
    by simp
  have F1: (f•α)= 0
    using F0 alpha-def alt-seq-limit
      ns-cauchy polynomial-is-Zp-continuous res-lim-pushforward
```

res-lim-pushforward' **by** *auto*
have *F2*: *val-Zp* (*a* ⊖ *α*) > *val-Zp* (*f'*•*a*) ∧ *val-Zp* (*a* ⊖ *α*) = *val-Zp* (*local.divide* (*f*•*a*) (*f'*•*a*))
proof−
have *0*: *Zp-converges-to ns α*
by (*simp add*: *alpha-def is-Zp-cauchy-imp-has-limit ns-cauchy*)
have *val-Zp* (*a* ⊖ *α*) < ∞
using *0 F1 R.r-right-minus-eq Zp-converges-to-def Zp-def hensel.fa-nonzero hensel-axioms local.a-closed val-Zp-def*
by *auto*
hence *1* + *max* (*eint 2* + *val-Zp* (*f'*•*a*)) (*val-Zp* (*α* ⊖ *a*)) < ∞
by (*metis 0 R.minus-closed Zp-converges-to-def eint.distinct*(*2*) *eint-ord-simps*(*4*)
f'a-not-infinite infinity-ne-i1 local.a-closed max-def minus-a-inv sum-infinity-imp-summand-infinity val-Zp-of-minus)
then obtain *l* **where** *l-def*: *eint l* = *1* + *max* (*eint 2* + *val-Zp* (*f'*•*a*)) (*val-Zp* (*α* ⊖ *a*))
by *auto*
then obtain *N* **where** *N-def*: (∀ *m*>*N*. *1* + *max* (*2* + *val-Zp* (*f'*•*a*)) (*val-Zp* (*α* ⊖ *a*)) < *val-Zp-dist* (*ns m*) *α*)
using *0 l-def Zp-converges-to-def*[*of ns α*] **unfolding** *val-Zp-dist-def*
by *metis*
obtain *N'* **where** *N'-def*: ∀ *n*>*N'*. *val-Zp* (*a* ⊖ *ns n*) = *val-Zp* (*local.divide* (*f*•*a*) (*f'*•*a*))
using *pre-hensel*(*2*) **by** *blast*
obtain *K* **where** *K-def*: *K* = *Suc* (*max N N'*)
by *simp*
then have *F21*: (*1*+ (*max* (*2* + *val-Zp* (*f'*•*a*)) (*val-Zp* (*α* ⊖ *a*)))) < *val-Zp-dist* (*ns K*) *α*
by (*metis N-def lessI linorder-not-less max-def order-trans*)
have *F22*: *a* ≠ *ns K*
by (*smt* (*verit*, *del-insts*) *F21 Nz alpha-def eint-1-iff*(*1*) *eint-pow-int-is-pos less-le*
local.a-closed max-less-iff-conj newton-seq-is-Zp-cauchy-0 not-less pos-add-strict res-lim-in-Zp val-Zp-dist-def val-Zp-dist-sym)
show *?thesis*
proof(*cases ns K* = *α*)
case *True*
then show *?thesis*
using *pre-hensel F1 False* **by** *blast*
next
case *False*
assume *ns K* ≠ *α*
show *?thesis*
proof−
have *P0*: (*a* ⊖ *α*) ∈ *nonzero Zp*
by (*metis* (*mono-tags*, *opaque-lifting*) *F1 not-eq-diff-nonzero* ‹*Zp-converges-to ns α*› *a-closed Zp-converges-to-def fa-nonzero*)
have *P1*: (*α* ⊖ (*ns K*)) ∈ *nonzero Zp*

using *False not-eq-diff-nonzero* ‹*Zp-converges-to ns α*›
Zp-converges-to-def newton-seq-closed
by (*metis* (*mono-tags*, *opaque-lifting*))
have *P2*: *a* ⊖ (*ns K*) ∈ *nonzero Zp*
using *F22 not-eq-diff-nonzero*
a-closed newton-seq-closed
by *blast*
have *P3*: (*a* ⊖ *α*) = *a* ⊖ (*ns K*) ⊕ ((*ns K*) ⊖ *α*)
by (*metis R.plus-diff-simp* ‹*Zp-converges-to ns α*› *add-comm Zp-converges-to-def local.a-closed newton-seq-closed*)
have *P4*: *val-Zp* (*a* ⊖ *α*) ≥ *min* (*val-Zp* (*a* ⊖ (*ns K*))) (*val-Zp* ((*ns K*) ⊖ *α*))
using *0 P3 Zp-converges-to-def newton-seq-closed val-Zp-ultrametric*
by *auto*
have *P5*: *val-Zp* (*a* ⊖ (*ns K*)) > *val-Zp* (*f′*•*a*)
using *pre-hensel*(*1*)[*of K*]
by *metis*
have *1* + *max* (*eint 2* + *val-Zp* (*f′*•*a*)) (*val-Zp* (*α* ⊖ *a*)) > *val-Zp* (*f′*•*a*)
proof−
have *1* + *max* (*eint 2* + *val-Zp* (*f′*•*a*)) (*val-Zp* (*α* ⊖ *a*)) > (*eint 2* + *val-Zp* (*f′*•*a*))
proof −
obtain *ii* :: *int* **where**
f1: *eint ii* = *1* + *max* (*eint 2* + *val-Zp* (*f′*•*a*)) (*val-Zp* (*α* ⊖ *a*))
by (*meson l-def*)
then have *1* + (*eint 2* + *val-Zp* (*f′*•*a*)) ≤ *eint ii*
by *simp*
then show *?thesis*
using *f1* **by** (*metis Groups.add-ac*(*2*) *iless-Suc-eq linorder-not-less*)
qed
thus *?thesis*
by (*smt* (*verit*) *Groups.add-ac*(*2*) *eint-pow-int-is-pos f′a-not-infinite ge-plus-pos-imp-gt order-less-le*)
qed
hence *P6*: *val-Zp* ((*ns K*) ⊖ *α*) > *val-Zp* (*f′*•*a*)
using *F21* **unfolding** *val-Zp-dist-def*
by *auto*
have *P7*: *val-Zp* (*a* ⊖ *α*) > *val-Zp* (*f′*•*a*)
using *P4 P5 P6 eint-min-ineq* **by** *blast*
have *P8*: *val-Zp* (*a* ⊖ *α*) = *val-Zp* (*local.divide* (*f*•*a*) (*f′*•*a*))
proof−
have *1* + *max* (*2* + *val-Zp* (*f′*•*a*)) (*val-Zp-dist α a*) ≤ *val-Zp-dist* (*ns K*) *α*
using *False F21*
by (*simp add*: *val-Zp-dist-def*)
then have *val-Zp*(*α* ⊖ (*ns K*)) > *max* (*2* + *val-Zp* (*f′*•*a*)) (*val-Zp-dist α a*)
by (*metis 0 Groups.add-ac*(*2*) *P1 Zp-converges-to-def eSuc-mono iless-Suc-eq l-def*

minus-a-inv newton-seq-closed nonzero-closed val-Zp-dist-def val-Zp-of-minus)

then have *val-Zp*($\alpha \ominus$ (*ns K*)) > *val-Zp* (*a* $\ominus$ α)
using ‹*Zp-converges-to ns* α› *Zp-converges-to-def val-Zp-dist-def val-Zp-dist-sym*

by *auto*
then have *P80*: *val-Zp* (*a* $\ominus$ α) = *val-Zp* (*a* $\ominus$ (*ns K*))
using *P0 P1 Zp-def val-Zp-ultrametric-eq*[*of* α $\ominus$ *ns K a* $\ominus$ α] *0 R.plus-diff-simp*
Zp-converges-to-def local.a-closed newton-seq-closed nonzero-closed **by** *auto*
have *P81*: *val-Zp* (*a* $\ominus$ *ns K*) = *val-Zp* (*local.divide* (*f*•*a*) (*f'*•*a*))
using *K-def N'-def*
by (*metis* (*no-types*, *lifting*) *lessI linorder-not-less max-def order-less-le order-trans*)
then show *?thesis*
by (*simp add*: *P80*)
qed
thus *?thesis*
using *P7* **by** *blast*
qed
qed
qed
have *F3*: *val-Zp* (*f'* • α) = *val-Zp* (*f'*•*a*)
proof−
have *F31*: (*f'* • α) = *res-lim* ((*to-fun f'*) ∘ *ns*)
using *alpha-def alt-seq-limit ns-cauchy polynomial-is-Zp-continuous res-lim-pushforward res-lim-pushforward' f'-closed*
by *auto*
obtain *N* **where** *N-def*: *val-Zp* (*f'*•α $\ominus$ *f'*•(*ns N*)) > *val-Zp* ((*f'*•*a*))
by (*smt* (*verit*) *F2 False R.minus-closed Suc-ile-eq Zp-def alpha-def f'-closed f'a-nonzero*
local.a-closed minus-a-inv newton-seq.simps(*1*) *newton-seq-is-Zp-cauchy-0 order-trans*
padic-integers.poly-diff-val padic-integers-axioms res-lim-in-Zp val-Zp-def val-Zp-of-minus)
show *?thesis*
by (*metis False N-def alpha-def equal-val-Zp f'-closed newton-seq-closed newton-seq-is-Zp-cauchy-0 newton-seq-fact1 res-lim-in-Zp to-fun-closed*)
qed
show *?thesis*
using *F1 F2 F3 that alpha-def ns-cauchy res-lim-in-Zp*
by *blast*
qed

end

31 Removing Hensel's Lemma from the Hensel Locale

context *padic-integers*
begin

lemma *hensels-lemma*:
assumes $f \in$ *carrier Zp-x*
assumes $a \in$ *carrier Zp*
assumes $(pderiv\ f)\cdot a \neq \mathbf{0}$
assumes $f\cdot a \neq \mathbf{0}$
assumes *val-Zp* $(f\cdot a) > 2 *$ *val-Zp* $((pderiv\ f)\cdot a)$
obtains α **where**
$f\cdot\alpha = \mathbf{0}$ **and** $\alpha \in$ *carrier Zp*
val-Zp $(a \ominus \alpha) >$ *val-Zp* $((pderiv\ f)\cdot a)$
val-Zp $(a \ominus \alpha) =$ *val-Zp* $(divide\ (f\cdot a)\ ((pderiv\ f)\cdot a))$
val-Zp $((pderiv\ f)\cdot\alpha) =$ *val-Zp* $((pderiv\ f)\cdot a)$
proof−
have *hensel p f a*
using *assms*
by (*simp add*: *Zp-def hensel.intro hensel-axioms.intro padic-integers-axioms*)
then show *?thesis*
using *hensel.full-hensels-lemma Zp-def that*
by *blast*
qed

Uniqueness of the root found in Hensel's lemma

lemma *hensels-lemma-unique-root*:
assumes $f \in$ *carrier Zp-x*
assumes $a \in$ *carrier Zp*
assumes $(pderiv\ f)\cdot a \neq \mathbf{0}$
assumes $f\cdot a \neq \mathbf{0}$
assumes (*val-Zp* $(f\cdot a) > 2 *$ *val-Zp* $((pderiv\ f)\cdot a)$)
assumes $f\cdot\alpha = \mathbf{0}$
assumes $\alpha \in$ *carrier Zp*
assumes *val-Zp* $(a \ominus \alpha) >$ *val-Zp* $((pderiv\ f)\cdot a)$
assumes $f\cdot\beta = \mathbf{0}$
assumes $\beta \in$ *carrier Zp*
assumes *val-Zp* $(a \ominus \beta) >$ *val-Zp* $((pderiv\ f)\cdot a)$
assumes *val-Zp* $((pderiv\ f)\cdot\alpha) =$ *val-Zp* $((pderiv\ f)\cdot a)$
shows $\alpha = \beta$
proof−
have $\alpha \neq a$
using *assms*(*4*) *assms*(*6*) **by** *auto*
have $\beta \neq a$
using *assms*(*4*) *assms*(*9*) **by** *auto*
have *0*: *val-Zp* $(\beta \ominus \alpha) >$ *val-Zp* $((pderiv\ f)\cdot a)$
proof−

have $\beta \ominus \alpha = \ominus ((a \ominus \beta) \ominus (a \ominus \alpha))$
by (*metis R.minus-eq R.plus-diff-simp assms*(10) *assms*(2) *assms*(7) *minus-a-inv*)
hence *val-Zp* $(\beta \ominus \alpha) =$ *val-Zp* $((a \ominus \beta) \ominus (a \ominus \alpha))$
using *R.minus-closed assms*(10) *assms*(2) *assms*(7) *val-Zp-of-minus* **by** *presburger*
thus *?thesis* **using** *val-Zp-ultrametric-diff*[*of* $a \ominus \beta$ $a \ominus \alpha$]
by (*smt* (*verit*) *R.minus-closed assms*(10) *assms*(11) *assms*(2) *assms*(7) *assms*(8) *min.absorb2 min-less-iff-conj*)
qed
obtain h **where** *h-def*: $h = \beta \ominus \alpha$
by *blast*
then have *h-fact*: $h \in$ *carrier Zp* $\wedge\ \beta = \alpha \oplus h$
by (*metis R.l-neg R.minus-closed R.minus-eq R.r-zero add-assoc add-comm assms*(10) *assms*(7))
then have *1*: $f \cdot (\alpha \oplus h) = \mathbf{0}$
using *assms*
by *blast*
obtain c **where** *c-def*: $c \in$ *carrier Zp* $\wedge\ f \cdot (\alpha \oplus h) = (f \cdot \alpha) \oplus (deriv\ f\ \alpha) \otimes h \oplus c \otimes (h[\hat{}](2::nat))$
using *taylor-deg-1-eval'*[*of* $f\ \alpha\ h$ - $f \cdot \alpha$ *deriv f* α]
by (*meson taylor-closed assms*(1) *assms*(7) *to-fun-closed h-fact shift-closed*)
then have $(f \cdot \alpha) \oplus (deriv\ f\ \alpha) \otimes h \oplus c \otimes (h[\hat{}](2::nat)) = \mathbf{0}$
by (*simp add*: *1*)
then have *2*: $(deriv\ f\ \alpha) \otimes h \oplus c \otimes (h[\hat{}](2::nat)) = \mathbf{0}$
by (*simp add*: *assms*(1) *assms*(6) *assms*(7) *deriv-closed h-fact*)
have *3*: $((deriv\ f\ \alpha) \oplus c \otimes h) \otimes h = \mathbf{0}$
proof−
have $((deriv\ f\ \alpha) \oplus c \otimes h) \otimes h = ((deriv\ f\ \alpha) \otimes h \oplus (c \otimes h) \otimes h)$
by (*simp add*: *R.r-distr UP-cring.deriv-closed Zp-x-is-UP-cring assms*(1) *assms*(7) *c-def h-fact mult-comm*)
then have $((deriv\ f\ \alpha) \oplus c \otimes h) \otimes h = (deriv\ f\ \alpha) \otimes h \oplus (c \otimes (h \otimes h))$
by (*simp add*: *mult-assoc*)
then have $((deriv\ f\ \alpha) \oplus c \otimes h) \otimes h = (deriv\ f\ \alpha) \otimes h \oplus (c \otimes (h[\hat{}](2::nat)))$
using *nat-pow-def*[*of Zp h 2*]
by (*simp add*: *h-fact*)
then show *?thesis*
using *2*
by *simp*
qed
have $h = \mathbf{0}$
proof(*rule ccontr*)
assume $h \neq \mathbf{0}$
then have $(deriv\ f\ \alpha) \oplus c \otimes h = \mathbf{0}$
using *2 3*
by (*meson R.m-closed assms*(1) *assms*(7) *c-def deriv-closed h-fact local.integral sum-closed*)
then have $(deriv\ f\ \alpha) = \ominus\ c \otimes h$
by (*simp add*: *R.l-minus R.sum-zero-eq-neg UP-cring.deriv-closed Zp-x-is-UP-cring*

assms(*1*) *assms*(*7*) *c-def h-fact*)
then have *val-Zp* (*deriv f* α) = *val-Zp* ($c \otimes h$)
by (*meson R.m-closed* ‹*deriv f* $\alpha \oplus c \otimes h = \mathbf{0}$› *assms*(*1*) *assms*(*7*) *c-def deriv-closed h-fact val-Zp-not-equal-imp-notequal*(*3*))
then have *P*: *val-Zp* (*deriv f* α) = *val-Zp h* + *val-Zp c*
using *val-Zp-mult c-def h-fact* **by** *force*
hence *val-Zp* (*deriv f* α) $\geq$ *val-Zp h*
using *val-pos*[*of c*]
by (*simp add*: *c-def*)
then have *val-Zp* (*deriv f* α) $\geq$ *val-Zp* ($\beta \ominus \alpha$)
using *h-def* **by** *blast*
then have *val-Zp* (*deriv f* α) > *val-Zp* ((*pderiv f*)$\cdot a$)
using *0* **by** *auto*
then show *False* **using** *pderiv-eval-deriv*[*of f* α]
using *assms*(*1*) *assms*(*12*) *assms*(*7*) **by** *auto*
qed
then show $\alpha = \beta$
using *assms*(*10*) *assms*(*7*) *h-def*
by *auto*
qed

lemma *hensels-lemma′*:
assumes $f \in$ *carrier Zp-x*
assumes $a \in$ *carrier Zp*
assumes *val-Zp* ($f \cdot a$) > *2*∗*val-Zp* ((*pderiv f*)$\cdot a$)
shows $\exists!\alpha \in$ *carrier Zp*. $f \cdot \alpha = \mathbf{0} \wedge$ *val-Zp* ($a \ominus \alpha$) > *val-Zp* ((*pderiv f*)$\cdot a$)
proof(*cases* $f \cdot a = \mathbf{0}$)
case *True*
have *T0*: *pderiv f* $\cdot$ $a \neq \mathbf{0}$
apply(*rule ccontr*) **using** *assms*(*3*)
unfolding *val-Zp-def* **by** *simp*
then have *T1*: $a \in$ *carrier Zp* $\wedge$ $f \cdot a = \mathbf{0} \wedge$ *val-Zp* ($a \ominus a$) > *val-Zp* ((*pderiv f*)$\cdot a$)
using *assms True*
by(*simp add*: *val-Zp-def*)
have *T2*: $\bigwedge b.\ b \in$ *carrier Zp* $\wedge$ $f \cdot b = \mathbf{0} \wedge$ *val-Zp* ($a \ominus b$) > *val-Zp* ((*pderiv f*)$\cdot a$) $\Longrightarrow a = b$
proof− **fix** *b* **assume** *A*: $b \in$ *carrier Zp* $\wedge$ $f \cdot b = \mathbf{0} \wedge$ *val-Zp* ($a \ominus b$) > *val-Zp* ((*pderiv f*)$\cdot a$)
obtain *h* **where** *h-def*: $h = b \ominus a$
by *blast*
then have *h-fact*: $h \in$ *carrier Zp* $\wedge$ $b = a \oplus h$
by (*metis A R.l-neg R.minus-closed R.minus-eq R.r-zero add-assoc add-comm assms*(*2*))
then have *1*: $f \cdot (a \oplus h) = \mathbf{0}$
using *assms A* **by** *blast*
obtain *c* **where** *c-def*: $c \in$ *carrier Zp* $\wedge$ $f \cdot (a \oplus h) = (f \cdot a) \oplus$ (*deriv f a*)$\otimes h \oplus c \otimes (h[\widehat{}](2::nat))$
using *taylor-deg-1-eval′*[*of f a h - f* $\cdot$ *a deriv f a*]

by (*meson taylor-closed assms*(*1*) *assms*(*2*) *to-fun-closed h-fact shift-closed*)

then have $(f \cdot a) \oplus (deriv\ f\ a) \otimes h \oplus c \otimes (h[\hat{}](2::nat)) = \mathbf{0}$
by (*simp add*: *1*)
then have *2*: $(deriv\ f\ a) \otimes h \oplus c \otimes (h[\hat{}](2::nat)) = \mathbf{0}$
by (*simp add*: *True assms*(*1*) *assms*(*2*) *deriv-closed h-fact*)
hence *3*: $((deriv\ f\ a) \oplus c \otimes h) \otimes h = \mathbf{0}$
proof −
have $((deriv\ f\ a) \oplus c \otimes h) \otimes h = ((deriv\ f\ a) \otimes h \oplus (c \otimes h) \otimes h)$
by (*simp add*: *R.l-distr assms*(*1*) *assms*(*2*) *c-def deriv-closed h-fact*)
then have $((deriv\ f\ a) \oplus c \otimes h) \otimes h = (deriv\ f\ a) \otimes h \oplus (c \otimes (h \otimes h))$
by (*simp add*: *mult-assoc*)
then have $((deriv\ f\ a) \oplus c \otimes h) \otimes h = (deriv\ f\ a) \otimes h \oplus (c \otimes (h[\hat{}](2::nat)))$
using *nat-pow-def*[*of Zp h 2*]
by (*simp add*: *h-fact*)
then show *?thesis*
using *2*
by *simp*
qed
have $h = \mathbf{0}$
proof(*rule ccontr*)
assume $h \neq \mathbf{0}$
then have $(deriv\ f\ a) \oplus c \otimes h = \mathbf{0}$
using *2 3*
by (*meson R.m-closed UP-cring.deriv-closed Zp-x-is-UP-cring assms*(*1*) *assms*(*2*) *c-def h-fact local.integral sum-closed*)
then have $(deriv\ f\ a) = \ominus\ c \otimes h$
using *R.l-minus R.minus-equality assms*(*1*) *assms*(*2*) *c-def deriv-closed h-fact* **by** *auto*
then have $val\text{-}Zp\ (deriv\ f\ a) = val\text{-}Zp\ (c \otimes h)$
by (*meson R.m-closed* ‹$deriv\ f\ a \oplus c \otimes h = \mathbf{0}$› *assms*(*1*) *assms*(*2*) *c-def deriv-closed h-fact val-Zp-not-equal-imp-notequal*(*3*))
then have *P*: $val\text{-}Zp\ (deriv\ f\ a) = val\text{-}Zp\ h + val\text{-}Zp\ c$
by (*simp add*: *c-def h-fact val-Zp-mult*)
have $val\text{-}Zp\ (deriv\ f\ a) \geq val\text{-}Zp\ h$
using *P val-pos*[*of c*] *c-def*
by *simp*
then have $val\text{-}Zp\ (deriv\ f\ a) \geq val\text{-}Zp\ (b \ominus a)$
using *h-def* **by** *blast*
then have $val\text{-}Zp\ (deriv\ f\ a) > val\text{-}Zp\ ((pderiv\ f) \cdot a)$
by (*metis* (*no-types*, *lifting*) *A assms*(*2*) *h-def h-fact minus-a-inv not-less order-trans val-Zp-of-minus*)
then have *P0*:$val\text{-}Zp\ (deriv\ f\ a) > val\text{-}Zp\ (deriv\ f\ a)$
by (*metis UP-cring.pderiv-eval-deriv Zp-x-is-UP-cring assms*(*1*) *assms*(*2*))

thus *False* **by** *auto*
qed
then show $a = b$
by (*simp add*: *assms*(*2*) *h-fact*)

qed
show *?thesis*
using *T1 T2*
by *blast*
next
case *False*
have *F0*: *pderiv f* ⋅ *a* ≠ **0**
apply(*rule ccontr*) **using** *assms*(*3*)
unfolding *val-Zp-def* **by** *simp*
obtain α **where** *alpha-def*:
f⋅α = **0** α ∈ *carrier Zp*
val-Zp (*a* ⊖ α) > *val-Zp* ((*pderiv f*)⋅*a*)
(*val-Zp* (*a* ⊖ α) = *val-Zp* (*divide* (*f*⋅*a*) ((*pderiv f*)⋅*a*)))
val-Zp ((*pderiv f*)⋅α) = *val-Zp* ((*pderiv f*)⋅*a*)
using *assms hensels-lemma F0 False* **by** *blast*
have *0*: ⋀*x*. *x* ∈ *carrier Zp* ∧ *f* ⋅ *x* = **0** ∧ *val-Zp* (*a* ⊖ *x*) > *val-Zp* (*pderiv f* ⋅ *a*) ∧ *val-Zp* (*pderiv f* ⋅ *a*) ≠ *val-Zp* (*a* ⊖ *x*) ⟹ *x*= α
using *alpha-def assms hensels-lemma-unique-root*[*of f a* α] *F0 False* **by** *blast*

have *1*: α ∈ *carrier Zp* ∧ *f* ⋅ α = **0** ∧ *val-Zp* (*a* ⊖ α) > *val-Zp* (*pderiv f* ⋅ *a*) ∧ *val-Zp* (*pderiv f* ⋅ *a*) ≠ *val-Zp* (*a* ⊖ α)
using *alpha-def order-less-le* **by** *blast*
thus *?thesis*
using *0*
by (*metis* (*no-types*, *opaque-lifting*) *R.minus-closed alpha-def*(*1*−*3*) *assms*(*2*) *equal-val-Zp val-Zp-ultrametric-eq'*)
qed

32 Some Applications of Hensel's Lemma to Root Finding for Polynomials over $\mathbb{Z}_p$

lemma *Zp-square-root-criterion*:
assumes *p* ≠ *2*
assumes *a* ∈ *carrier Zp*
assumes *b* ∈ *carrier Zp*
assumes *val-Zp b* ≥ *val-Zp a*
assumes *a* ≠ **0**
assumes *b* ≠ **0**
shows ∃ *y* ∈ *carrier Zp*. *a*[^](*2*::*nat*) ⊕ p⊗*b*[^](*2*::*nat*) = (*y* [^]*Zp* (*2*::*nat*))
proof−
have *bounds*: *val-Zp a* < ∞ *val-Zp a* ≥ *0 val-Zp b* < ∞ *val-Zp b* ≥ *0*
using *assms*(*2*) *assms*(*3*) *assms*(*6*) *assms*(*5*) *val-Zp-def val-pos*[*of b*] *val-pos*[*of a*]
by *auto*
obtain *f* **where** *f-def*: *f* = *monom Zp-x* **1** *2* ⊕*Zp-x* *to-polynomial Zp* (⊖ (*a*[^](*2*::*nat*)⊕ p⊗*b*[^](*2*::*nat*)))
by *simp*
have ∃ α. *f*⋅α = **0** ∧ α ∈ *carrier Zp*

```
proof−
  have 0: f ∈ carrier Zp-x
    using f-def
    by (simp add: X-closed assms(2) assms(3) to-poly-closed)
  have 1: (pderiv f)•a = [(2::nat)] · 1 ⊗ a
  proof−
    have pderiv f = pderiv (monom Zp-x 1 2)
    using assms f-def pderiv-add[of monom Zp-x 1 2] to-poly-closed R.nat-pow-closed

          pderiv-deg-0
      unfolding to-polynomial-def
      using P.nat-pow-closed P.r-zero R.add.inv-closed X-closed Zp-int-inc-closed
deg-const monom-term-car pderiv-closed sum-closed
    by (metis (no-types, lifting) R.one-closed monom-closed)
    then have 20: pderiv f = monom (Zp-x) ([(2::nat) ] · 1) (1::nat)
      using pderiv-monom[of 1 2]
      by simp
    have 21: [(2::nat)] · 1 ≠ 0
      using Zp-char-0'[of 2] by simp
    have 22: (pderiv f)•a = [(2::nat)] · 1 ⊗ (a[^](1::nat))
      using 20
      by (simp add: Zp-nat-inc-closed assms(2) to-fun-monom)
    then show ?thesis
      using assms(2)
      by (simp add: cring.cring-simprules(12))
  qed
  have 2: (pderiv f)•a ≠ 0
    using 1 assms
    by (metis Zp-char-0' Zp-nat-inc-closed local.integral zero-less-numeral)
  have 3: f•a = ⊖ (p⊗b[^](2::nat))
  proof−
    have 3: f•a =
  monom (UP Zp) 1 2 • a ⊕
  to-polynomial Zp (⊖ (a [^] (2::nat) ⊕ [p] · 1 ⊗ b [^] (2::nat)))•a
      unfolding f-def apply(rule to-fun-plus)
       apply (simp add: assms(2) assms(3) to-poly-closed)
      apply simp
      by (simp add: assms(2))
    have 30: f•a = a[^](2::nat)  ⊖ (a[^](2::nat) ⊕ p⊗b[^](2::nat))
    unfolding 3 by (simp add: R.minus-eq assms(2) assms(3) to-fun-monic-monom
to-fun-to-poly)
    have 31: f•a = a[^](2::nat)  ⊖ a[^](2::nat) ⊖ (p⊗b[^](2::nat))
    proof−
      have 310: a[^](2::nat) ∈ carrier Zp
        using assms(2) pow-closed
        by blast
      have 311: p⊗(b[^](2::nat)) ∈ carrier Zp
        by (simp add: assms(3) monom-term-car)
      have  ⊖ (a [^] (2::nat)⊕(p ⊗ b [^] (2::nat))) = ⊖ (a [^] (2::nat)) ⊕ ⊖ (p
```

```
⊗ (b [^] (2::nat)))
        using 310 311 R.minus-add by blast
      then show ?thesis
       by (simp add: 30 R.minus-eq add-assoc)
    qed
    have 32: f•a = (a[^](2::nat)  ⊖ a[^](2::nat)) ⊖ (p⊗b[^](2::nat))
      using 31 unfolding a-minus-def
      by blast
    have 33: p⊗b[^](2::nat) ∈ carrier Zp
      by (simp add: Zp-nat-inc-closed assms(3) monom-term-car)
    have 34: a[^](2::nat) ∈ carrier Zp
      using assms(2) pow-closed by blast
    then have 34: (a[^](2::nat)  ⊖ a[^](2::nat)) = 0
      by simp
    have 35: f•a = 0 ⊖ (p⊗b[^](2::nat))
      by (simp add: 32 34)
    then show ?thesis
      using 33 unfolding a-minus-def
      by (simp add: cring.cring-simprules(3))
  qed
  have 4: f•a ≠0
    using 3 assms
  by (metis R.add.inv-eq-1-iff R.m-closed R.nat-pow-closed Zp.integral Zp-int-inc-closed
      mult-zero-r nonzero-pow-nonzero p-natpow-prod-Suc(1) p-pow-nonzero(2))

  have 5: val-Zp (f•a) = 1 + 2*val-Zp b
  proof−
    have val-Zp (f•a) = val-Zp (p⊗b[^](2::nat))
      using 3 Zp-int-inc-closed assms(3) monom-term-car val-Zp-of-minus by
presburger
    then have val-Zp (p⊗b[^](2::nat)) = 1 + val-Zp (b[^](2::nat))
      by (simp add: assms(3) val-Zp-mult val-Zp-p)
    then show ?thesis
      using assms(3) assms(6)
     using Zp-def ‹val-Zp (to-fun f a) = val-Zp ([p] · 1 ⊗ b [^] 2)› not-nonzero-Zp
       padic-integers-axioms val-Zp-pow' by fastforce
  qed
  have 6: val-Zp ((pderiv f)•a) = val-Zp a
  proof−
    have 60: val-Zp ([(2::nat)] · 1 ⊗ a) = val-Zp ([(2::nat)] · 1) + val-Zp a
     by (simp add: Zp-char-0' assms(2) assms(5) val-Zp-mult ord-of-nonzero(2)
ord-pos)
    have val-Zp ([(2::nat)] · 1) = 0
    proof−
      have (2::nat) < p
        using prime assms prime-ge-2-int by auto
      then have (2::nat) mod p = (2::nat)
        by simp
      then show ?thesis
```

```
        by (simp add: val-Zp-p-nat-unit)
      qed
      then show ?thesis
        by (simp add: 1 60)
    qed
    then have 7: val-Zp (f•a) > 2* val-Zp ((pderiv f)•a)
      using bounds 5 assms(4)
      by (simp add: assms(5) assms(6) one-eint-def val-Zp-def)
    obtain α where
      A0: f•α = 0  α ∈ carrier Zp
      using hensels-lemma[of f a] 0 2 4 7 assms(2)
      by blast
    show ?thesis
      using A0  by blast
  qed
  then obtain α where α-def: f•α = 0 ∧ α ∈ carrier Zp
    by blast
  have f•α = α [^](2::nat)  ⊖ (a[^](2::nat)⊕ p⊗b[^](2::nat))
  proof−
    have 0: f•α =
    monom (UP Zp) 1 2 · α ⊕
    to-polynomial Zp (⊖ (a [^] (2::nat) ⊕ [p] · 1 ⊗ b [^] (2::nat)))•α
      unfolding f-def apply(rule to-fun-plus)
       apply (simp add: assms(2) assms(3) to-poly-closed)
      apply simp
      by (simp add: α-def)
    thus ?thesis
      by (simp add: R.minus-eq α-def assms(2) assms(3) to-fun-monic-monom
to-fun-to-poly)
  qed
  then show ?thesis
   by (metis R.r-right-minus-eq Zp-int-inc-closed α-def assms(2) assms(3) monom-term-car
pow-closed sum-closed)
qed

lemma Zp-semialg-eq:
  assumes a ∈ nonzero Zp
  shows ∃ y ∈ carrier Zp. 1 ⊕ (p [^] (3::nat))⊗ (a [^] (4::nat)) = (y [^] (2::nat))
proof−
   obtain f where f-def: f = monom Zp-x 1 2 ⊕_Zp-x to-poly (⊖ (1 ⊕ (p [^]
(3::nat))⊗ (a [^] (4::nat))))
    by simp
  have a-car: a ∈ carrier Zp
    by (simp add: nonzero-memE assms)
  have f ∈ carrier Zp-x
    using f-def
    by (simp add: a-car to-poly-closed)
  hence 0:f•1 = 1 ⊖ (1 ⊕ (p [^] (3::nat))⊗ (a [^] (4::nat)))
    using f-def
```

by (*simp add*: *R.minus-eq assms nat-pow-nonzero nonzero-mult-in-car p-pow-nonzero′ to-fun-monom-plus to-fun-to-poly to-poly-closed*)
then have *1*: $f \cdot \mathbf{1} = \ominus$ (p [^] (*3*::*nat*))$\otimes$ (*a* [^] (*4*::*nat*))
unfolding *a-minus-def*
by (*smt* (*verit*) *R.add.inv-closed R.l-minus R.minus-add R.minus-minus R.nat-pow-closed R.one-closed R.r-neg1 a-car monom-term-car p-pow-nonzero*(*1*))
then have *val-Zp* ($f \cdot \mathbf{1}$) = *3* + *val-Zp* (*a* [^] (*4*::*nat*))
using *assms val-Zp-mult*[*of* p [^] (*3*::*nat*) (*a* [^] (*4*::*nat*))]
val-Zp-p-pow p-pow-nonzero[*of 3*::*nat*] *val-Zp-of-minus*
by (*metis R.l-minus R.nat-pow-closed a-car monom-term-car of-nat-numeral*)
then have *2*: *val-Zp* ($f \cdot \mathbf{1}$) = *3* + *4*∗ *val-Zp a*
using *assms val-Zp-pow′* **by** *auto*
have *pderiv f* = *pderiv* (*monom Zp-x* **1** *2*)
using *assms f-def pderiv-add*[*of monom Zp-x* **1** *2*] *to-poly-closed R.nat-pow-closed pderiv-deg-0*
unfolding *to-polynomial-def*
by (*metis* (*no-types*, *lifting*) *P.r-zero R.add.inv-closed R.add.m-closed R.one-closed*

UP-zero-closed a-car deg-const deg-nzero-nzero monom-closed monom-term-car p-pow-nonzero(*1*))
then have *3*: *pderiv f* = [(*2*::*nat*)] · **1** $\odot_{Zp\text{-}x}$ *X*
by (*metis P.nat-pow-eone R.one-closed Suc-1 X-closed diff-Suc-1 monom-rep-X-pow pderiv-monom′*)
hence *4*: *val-Zp* ((*pderiv f*)·**1**) = *val-Zp* ([(*2*::*nat*)] · **1**)
by (*metis R.add.nat-pow-eone R.nat-inc-prod R.nat-inc-prod′ R.nat-pow-one R.one-closed*
Zp-nat-inc-closed ‹*pderiv f* = *pderiv* (*monom Zp-x* **1** *2*)› *pderiv-monom to-fun-monom*)
have (*2*::*int*) = (*int* (*2*::*nat*))
by *simp*
then have *5*: [(*2*::*nat*)] · **1** = ([(*int* (*2*::*nat*))] · **1**)
using *add-pow-def int-pow-int*
by *metis*
have *6*: *val-Zp* ((*pderiv f*)·**1**) ≤ *1*
apply(*cases p* = *2*)
using *4 5 val-Zp-p* **apply** *auto*[*1*]
proof−
assume *p* ≠ *2*
then have *60*: *coprime 2 p*
using *prime prime-int-numeral-eq primes-coprime two-is-prime-nat* **by** *blast*

have *61*: *2* < *p*
using *60 prime*
by (*smt* (*verit*) ‹*p* ≠ *2*› *prime-gt-1-int*)
then show *?thesis*
by (*smt* (*verit*) *4 5* ‹*2* = *int 2*› *mod-pos-pos-trivial nonzero-closed p-nonzero val-Zp-p val-Zp-p-int-unit val-pos*)
qed
have *7*: *val-Zp* ($f \cdot \mathbf{1}$) ≥ *3*

proof−
have *eint 4 ∗ val-Zp a ≥ 0*
using *2 val-pos*[*of a*]
by (*metis R.nat-pow-closed a-car assms of-nat-numeral val-Zp-pow′ val-pos*)
thus *?thesis*
using *2* **by** *auto*
qed
have *2∗val-Zp* ((*pderiv f*)•**1**) ≤ *2∗1*
using *6 one-eint-def eint-mult-mono′*
by (*smt* (*verit*) ‹*2 = int 2*› *eint.distinct*(*2*) *eint-ile eint-ord-simps*(*1*) *eint-ord-simps*(*2*) *mult.commute*
ord-Zp-p ord-Zp-p-pow ord-Zp-pow p-nonzero p-pow-nonzero(*1*) *times-eint-simps*(*1*) *val-Zp-p val-Zp-pow′ val-pos*)
hence *8*: *2 ∗ val-Zp* ((*pderiv f*)• **1**) < *val-Zp* (*f*•**1**)
using *7 le-less-trans*[*of 2 ∗ val-Zp* ((*pderiv f*)• **1**) *2*::*eint 3*]
less-le-trans[*of 2 ∗ val-Zp* ((*pderiv f*)• **1**) *3 val-Zp* (*f*•**1**)] *one-eint-def*
by *auto*
obtain *α* **where** *α-def*: *f*•*α* = **0** **and** *α-def′* :*α* ∈ *carrier Zp*
using *2 6 7 hensels-lemma′ 8* ‹*f* ∈ *carrier Zp-x*› **by** *blast*
have *0*: (*monom Zp-x* **1** *2*) • *α* = *α* [^] (*2*::*nat*)
by (*simp add*: *α-def′ to-fun-monic-monom*)
have *1*: *to-poly* (⊖ (**1** ⊕ (p [^] (*3*::*nat*))⊗ (*a* [^] (*4*::*nat*)))) • *α* =⊖(**1** ⊕ (p [^] (*3*::*nat*))⊗ (*a* [^] (*4*::*nat*)))
by (*simp add*: *α-def′ a-car to-fun-to-poly*)
then have *α* [^] (*2*::*nat*) ⊖ (**1** ⊕ (p [^] (*3*::*nat*))⊗ (*a* [^] (*4*::*nat*))) = **0**
using *α-def α-def′*
by (*simp add*: *R.minus-eq a-car f-def to-fun-monom-plus to-poly-closed*)
then show *?thesis*
by (*metis R.add.m-closed R.nat-pow-closed R.one-closed R.r-right-minus-eq α-def′ a-car monom-term-car p-pow-nonzero*(*1*))
qed

lemma *Zp-nth-root-lemma*:
assumes *a* ∈ *carrier Zp*
assumes *a* ≠ **1**
assumes *n* > *1*
assumes *val-Zp* (**1** ⊖ *a*) > *2∗val-Zp* ([(*n*::*nat*)]• **1**)
shows ∃ *b* ∈ *carrier Zp*. *b*[^]*n* = *a*
proof−
obtain *f* **where** *f-def*: *f* = *monom Zp-x* **1** *n* ⊕$_{Zp\text{-}x}$ *monom Zp-x* (⊖*a*) *0*
by *simp*
have *f* ∈ *carrier Zp-x*
using *f-def monom-closed assms*
by *simp*
have *0*: *pderiv f* = *monom Zp-x* ([*n*]• **1**) (*n*−*1*)
by (*simp add*: *assms*(*1*) *f-def pderiv-add pderiv-monom*)
have *1*: *f* • **1** = **1** ⊖ *a*
using *f-def*
by (*metis R.add.inv-closed R.minus-eq R.nat-pow-one R.one-closed assms*(*1*)

to-fun-const to-fun-monom to-fun-monom-plus monom-closed)
have *2*: (*pderiv f*) ⋅ **1** = ([*n*]⋅ **1**)
using *0 to-fun-monom assms*
by *simp*
have *3*: *val-Zp* (*f* ⋅ **1**) > *2*∗ *val-Zp* ((*pderiv f*) ⋅ **1**)
using *1 2 assms*
by (*simp add*: *val-Zp-def*)
have *4*: *f* ⋅ **1** ≠ **0**
using *1 assms*(*1*) *assms*(*2*) **by** *auto*
have *5*: (*pderiv f*) ⋅ **1** ≠ **0**
using *2 Zp-char-0′ assms*(*3*) **by** *auto*
obtain *β* **where** *beta-def*: *β* ∈ *carrier Zp* ∧ *f* ⋅ *β* = **0**
using *hensels-lemma*[*of f* **1**]
by (*metis 3 5 R.one-closed* ‹*f* ∈ *carrier Zp-x*›)
then have (*β* [^] *n*) ⊖ *a* = **0**
using *f-def R.add.inv-closed assms*(*1*) *to-fun-const*[*of* ⊖ *a*] *to-fun-monic-monom*[*of* *β n*] *to-fun-plus monom-closed*
unfolding *a-minus-def*
by (*simp add*: *beta-def*)
then have *β* ∈ *carrier Zp* ∧ *β* [^] *n* = *a*
using *beta-def nonzero-memE not-eq-diff-nonzero assms*(*1*) *pow-closed*
by *blast*
then show *?thesis* **by** *blast*
qed

end
end
theory *Zp-Compact*
imports *Padic-Int-Topology*
begin

context *padic-integers*
begin

lemma *res-ring-car*:
carrier (*Zp-res-ring k*) = {*0..p* ^ *k* − *1*}
unfolding *residue-ring-def* **by** *simp*

The refinement of a sequence by a function *nat* ⇒ *nat*

definition *take-subseq* :: (*nat* ⇒ *'a*) ⇒ (*nat* ⇒ *nat*) ⇒ (*nat* ⇒ *'a*) **where**
take-subseq s f = (*λk*. *s* (*f k*))

Predicate for increasing function on the natural numbers

definition *is-increasing* :: (*nat* ⇒ *nat*) ⇒ *bool* **where**
is-increasing f = (∀ *n m*::*nat*. *n*>*m* ⟶ (*f n*) > (*f m*))

Elimination and introduction lemma for increasing functions

lemma *is-increasingI*:
assumes ⋀ *n m*::*nat*. *n*>*m* ⟹ (*f n*) > (*f m*)

shows *is-increasing f*
unfolding *is-increasing-def*
using *assms*
by *blast*

lemma *is-increasingE*:
assumes *is-increasing f*
assumes $n > m$
shows $f\ n > f\ m$
using *assms*
unfolding *is-increasing-def*
by *blast*

The subsequence predicate

definition *is-subseq-of* :: $(nat \Rightarrow 'a) \Rightarrow (nat \Rightarrow 'a) \Rightarrow bool$ **where**
is-subseq-of s s' = $(\exists (f{::}nat \Rightarrow nat).$ *is-increasing f* $\wedge$ *s'* = *take-subseq s f*)

Subsequence introduction lemma

lemma *is-subseqI*:
assumes *is-increasing f*
assumes *s'* = *take-subseq s f*
shows *is-subseq-of s s'*
using *assms*
unfolding *is-subseq-of-def*
by *auto*

lemma *is-subseq-ind*:
assumes *is-subseq-of s s'*
shows $\exists\ l.\ s'\ k = s\ l$
using *assms*
unfolding *is-subseq-of-def take-subseq-def* **by** *blast*

lemma *is-subseq-closed*:
assumes *s* $\in$ *closed-seqs Zp*
assumes *is-subseq-of s s'*
shows *s'* $\in$ *closed-seqs Zp*
apply(*rule closed-seqs-memI*)
using *is-subseq-ind assms closed-seqs-memE*
by *metis*

Given a sequence and a predicate, returns the function from nat to nat which represents the increasing sequences of indices n on which P (s n) holds.

primrec *seq-filter* :: $(nat \Rightarrow 'a) \Rightarrow ('a \Rightarrow bool) \Rightarrow nat \Rightarrow nat$ **where**
seq-filter s P $(0{::}nat)$ = $(LEAST\ k{::}nat.\ P\ (s\ k))$|
seq-filter s P $(Suc\ n)$ = $(LEAST\ k{::}\ nat.\ (P\ (s\ k)) \wedge k > (seq\text{-}filter\ s\ P\ n))$

lemma *seq-filter-pre-increasing*:
assumes $\forall\, n{::}nat.\ \exists\, m.\ m > n \wedge P\ (s\ m)$
shows *seq-filter s P n* < *seq-filter s P* $(Suc\ n)$

apply(*auto*)
proof(*induction n*)
case *0*
have ∃ *k*. *P* (*s k*) **using** *assms*(*1*) **by** *blast*
then have ∃ *k*::*nat*. (*LEAST k*::*nat*. (*P* (*s k*))) ≥ *0*
by *blast*
obtain *k* **where** (*LEAST k*::*nat*. (*P* (*s k*))) = *k* **by** *simp*
have ∃ *l*. *l* = (*LEAST l*::*nat*. (*P* (*s l*) ∧ *l* > *k*))
by *simp*
thus *?case*
by (*metis* (*no-types*, *lifting*) *LeastI assms*)
next
case (*Suc n*)
then show *?case*
by (*metis* (*no-types*, *lifting*) *LeastI assms*)
qed

lemma *seq-filter-increasing*:
assumes ∀ *n*::*nat*. ∃ *m*. *m* > *n* ∧ *P* (*s m*)
shows *is-increasing* (*seq-filter s P*)
by (*metis assms seq-filter-pre-increasing is-increasingI lift-Suc-mono-less*)

definition *filtered-seq* :: (*nat* ⇒ *'a*) ⇒ (*'a* ⇒ *bool*) ⇒ (*nat* ⇒ *'a*) **where**
filtered-seq s P = *take-subseq s* (*seq-filter s P*)

lemma *filter-exist*:
assumes *s* ∈ *closed-seqs Zp*
assumes ∀ *n*::*nat*. ∃ *m*. *m* > *n* ∧ *P* (*s m*)
shows ⋀*m*. *n*≤*m* ⟹ *P* (*s* (*seq-filter s P n*))
proof(*induct n*)
case *0*
then show *?case*
using *LeastI assms*(*2*) **by** *force*
next
case (*Suc n*)
then show *?case*
by (*smt* (*verit*) *LeastI assms*(*2*) *seq-filter.simps*(*2*))
qed

In a filtered sequence, every element satisfies the filtering predicate

lemma *fil-seq-pred*:
assumes *s* ∈ *closed-seqs Zp*
assumes *s'* = *filtered-seq s P*
assumes ∀ *n*::*nat*. ∃ *m*. *m* > *n* ∧ *P* (*s m*)
shows ⋀*m*::*nat*. *P* (*s'* *m*)
proof−
have ∃ *k*. *P* (*s k*) **using** *assms*(*3*)
by *blast*
fix *m*

obtain k **where** *kdef*: $k = seq\text{-}filter\ s\ P\ m$ **by** *auto*
have $\exists\, k.\ P\ (s\ k)$
using *assms*(*3*) **by** *auto*
then have $P\ (s\ k)$
by (*metis* (*full-types*) *assms*(*1*) *assms*(*3*) *kdef le-refl less-imp-triv not-less-eq filter-exist*)
then have $s'\ m = s\ k$
by (*simp add*: *assms*(*2*) *filtered-seq-def kdef take-subseq-def*)
hence $P\ (s'\ m)$
by (*simp add*: ‹$P\ (s\ k)$›)
thus $\bigwedge m.\ P\ (s'\ m)$ **using** *assms*(*2*) *assms*(*3*) *dual-order.strict-trans filter-exist filtered-seq-def*
lessI less-Suc-eq-le take-subseq-def
by (*metis* (*mono-tags*, *opaque-lifting*) *assms*(*1*))
qed

definition *kth-res-equals* :: $nat \Rightarrow int \Rightarrow (padic\text{-}int \Rightarrow bool)$ **where**
kth-res-equals k n a = $(a\ k = n)$

definition *indicator*:: $(nat \Rightarrow 'a) \Rightarrow ('a \Rightarrow bool)$ **where**
indicator s a = $(\exists\, n{::}nat.\ s\ n = a)$

Choice function for a subsequence with constant kth residue. Could be made constructive by choosing the LEAST n if we wanted.

definition *const-res-subseq* :: $nat \Rightarrow padic\text{-}int\text{-}seq \Rightarrow padic\text{-}int\text{-}seq$ **where**
const-res-subseq k s = $(SOME\ s'{::}(padic\text{-}int\text{-}seq).\ (\exists\ n.\ is\text{-}subseq\text{-}of\ s\ s' \wedge s' = (filtered\text{-}seq\ s\ (kth\text{-}res\text{-}equals\ k\ n)) \wedge (\forall\, m.\ s'\ m\ k = n)))$

The constant kth residue value for the sequence obtained by the previous function

definition *const-res* :: $nat \Rightarrow padic\text{-}int\text{-}seq \Rightarrow int$ **where**
const-res k s = $(THE\ n.\ (\forall\ m.\ (const\text{-}res\text{-}subseq\ k\ s)\ m\ k = n))$

definition *maps-to-n*:: $int \Rightarrow (nat \Rightarrow int) \Rightarrow bool$ **where**
maps-to-n n f = $(\forall\, (k{::}nat).\ f\ k \in \{0..n\})$

definition *drop-res* :: $int \Rightarrow (nat \Rightarrow int) \Rightarrow (nat \Rightarrow int)$ **where**
drop-res k f n = $(if\ (f\ n) = k\ then\ 0\ else\ f\ n)$

lemma *maps-to-nE*:
assumes *maps-to-n n f*
shows $(f\ k) \in \{0..n\}$
using *assms*
unfolding *maps-to-n-def*
by *blast*

lemma *maps-to-nI*:
assumes $\bigwedge n.\ f\ n \in \{0\ ..\ k\}$

shows *maps-to-n k f*
using *assms maps-to-n-def* **by** *auto*

lemma *maps-to-n-drop-res*:
assumes *maps-to-n* (*Suc n*) *f*
shows *maps-to-n n* (*drop-res* (*Suc n*) *f*)
proof−
fix *k*
have *drop-res* (*Suc n*) *f k* ∈ {*0*..*n*}
proof(*cases f k* = *Suc n*)
case *True*
then have *drop-res* (*Suc n*) *f k* = *0*
unfolding *drop-res-def* **by** *auto*
then show *?thesis*
using *assms local.drop-res-def maps-to-n-def* **by** *auto*
next
case *False*
then show *?thesis*
using *assms atLeast0-atMost-Suc maps-to-n-def drop-res-def*
by *auto*
qed
then have ⋀*k*. *drop-res* (*Suc n*) *f k* ∈ {*0*..*n*}
using *assms local.drop-res-def maps-to-n-def* **by** *auto*
then show *maps-to-n n* (*drop-res* (*Suc n*) *f*) **using** *maps-to-nI*
using *maps-to-n-def* **by** *blast*
qed

lemma *drop-res-eq-f*:
assumes *maps-to-n* (*Suc n*) *f*
assumes ¬ (∀ *m*. ∃ *n*. *n*>*m* ∧ (*f n* = (*Suc k*)))
shows ∃ *N*. ∀ *n*. *n*>*N* ⟶ *f n* = *drop-res* (*Suc k*) *f n*
proof−
have ∃ *m*. ∀ *n*. *n* ≤ *m* ∨ (*f n*) ≠ (*Suc k*)
using *assms*
by (*meson Suc-le-eq nat-le-linear*)
then have ∃ *m*. ∀ *n*. *n* ≤ *m* ∨ (*f n*) = *drop-res* (*Suc k*) *f n*
using *drop-res-def* **by** *auto*
then show *?thesis*
by (*meson less-Suc-eq-le order.asym*)
qed

lemma *maps-to-n-infinite-seq*:
shows ⋀*f*. *maps-to-n* (*k*::*nat*) *f* ⟹ ∃ *l*::*int*. ∀ *m*. ∃ *n*. *n*>*m* ∧ (*f n* = *l*)
proof(*induction k*)
case *0*
then have ⋀*n*. *f n* ∈ {*0*}
using *maps-to-nE*[*of 0 f*] **by** *auto*
then show ∃ *l*. ∀ *m*. ∃ *n*. *m* < *n* ∧ *f n* = *l*

```
    by blast
next
  case (Suc k)
  assume IH: ⋀f. maps-to-n k f ⟹ ∃ l. ∀ m. ∃ n. m < n ∧ f n = l
  fix f
  assume A: maps-to-n (Suc k) f
  show ∃ l. ∀ m. ∃ n. n>m ∧ (f n = l)
  proof(cases  ∀ m. ∃ n. n>m ∧ (f n = (Suc k)))
    case True
    then show ?thesis by blast
  next
    case False
    then obtain N where N-def: ∀ n. n>N ⟶ f n = drop-res (Suc k) f n
      using drop-res-eq-f drop-res-def
      by fastforce
    have  maps-to-n k (drop-res (Suc k) f)
      using A maps-to-n-drop-res by blast
    then have  ∃ l. ∀ m. ∃ n. m < n ∧ (drop-res (Suc k) f) n = l
      using IH by blast
    then obtain l where l-def: ∀ m. ∃ n. m < n ∧ (drop-res (Suc k) f) n = l
      by blast
    have ∀ m. ∃ n. n>m ∧ (f n = l)
      apply auto
    proof−
      fix m
      show ∃ n>m. f n = l
      proof−
        obtain n where N'-def: (max m N) < n ∧ (drop-res (Suc k) f) n = l
          using l-def by blast
        have f n =  (drop-res (Suc k) f) n
          using N'-def N-def
          by simp
        then show ?thesis
          using N'-def by auto
      qed
    qed
    then show ?thesis
      by blast
  qed
qed

lemma int-nat-p-pow-minus:
int (nat (p ^ k − 1)) = p ^ k − 1
  by (simp add: prime prime-gt-0-int)

lemma maps-to-n-infinite-seq-res-ring:
⋀f. f ∈ (UNIV::nat set) → carrier (Zp-res-ring k) ⟹ ∃ l. ∀ m. ∃ n. n>m ∧ (f n
= l)
apply(rule maps-to-n-infinite-seq[of nat (p^k − 1)])
```

unfolding *maps-to-n-def res-ring-car int-nat-p-pow-minus* **by** *blast*

definition *index-to-residue* :: *padic-int-seq* ⇒ *nat* ⇒ *nat* ⇒ *int* **where**
index-to-residue s k m = ((*s m*) *k*)

lemma *seq-maps-to-n*:
assumes *s* ∈ *closed-seqs Zp*
shows (*index-to-residue s k*) ∈ *UNIV* → *carrier* (*Zp-res-ring k*)
proof−
have *A1*: ⋀*m*. (*s m*) ∈ *carrier Zp*
using *assms closed-seqs-memE* **by** *auto*
have *A2*: ⋀*m*. (*s m k*) ∈ *carrier* (*Zp-res-ring k*)
using *assms* **by** (*simp add*: *A1*)
have ⋀*m*. *index-to-residue s k m* = *s m k*
using *index-to-residue-def*
by *auto*
thus *index-to-residue s k* ∈ *UNIV* → *carrier* (*residue-ring* (*p* ^ *k*))
using *A2* **by** *simp*
qed

lemma *seq-pr-inc*:
assumes *s* ∈ *closed-seqs Zp*
shows ∃ *l*. ∀ *m*. ∃ *n* > *m*. (*kth-res-equals k l*) (*s n*)
proof−
fix *k l m*
have *0*: (*kth-res-equals k l*) (*s m*) ⟹ (*s m*) *k* = *l*
by (*simp add*: *kth-res-equals-def*)
have *1*: ⋀*k m*. *s m k* = *index-to-residue s k m*
by (*simp add*: *index-to-residue-def*)
have *2*: (*index-to-residue s k*) ∈ *UNIV* → *carrier* (*Zp-res-ring k*)
using *seq-maps-to-n assms* **by** *blast*
have *3*: ⋀*m*. *s m k* ∈ *carrier* (*Zp-res-ring k*)
proof−
fix *m* **have** *30*: *s m k* = *index-to-residue s k m*
using *1* **by** *blast*
show *s m k* ∈ *carrier* (*Zp-res-ring k*)
unfolding *30* **using** *2* **by** *blast*
qed
obtain *j* **where** *j-def*: *j* = *nat* (*p*^*k* − *1*)
by *blast*
have *j-to-int*: *int j* = *p*^*k* − *1*
using *j-def*
by (*simp add*: *prime prime-gt-0-int*)
have ∃ *l*. ∀ *m*. ∃ *n*. *n* > *m* ∧ (*index-to-residue s k n* = *l*)
by(*rule maps-to-n-infinite-seq-res-ring*[*of* - *k*], *rule seq-maps-to-n*, *rule assms*)
hence ∃ *l*. ∀ *m*. ∃ *n*. *n* > *m* ∧ (*s n k* = *l*)
by (*simp add*: *index-to-residue-def*)
thus ∃ *l*. ∀ *m*. ∃ *n* > *m*. (*kth-res-equals k l*) (*s n*)
using *kth-res-equals-def* **by** *auto*

qed

lemma *kth-res-equals-subseq*:
assumes $s \in$ *closed-seqs Zp*
shows $\exists\, n.$ *is-subseq-of s* (*filtered-seq s* (*kth-res-equals k n*)) $\wedge$ ($\forall\, m.$ (*filtered-seq s* (*kth-res-equals k n*)) *m k* = *n*)
proof−
obtain *l* **where** *l-def*: $\forall\ m.\ \exists\, n > m.$ (*kth-res-equals k l*) (*s n*)
using *assms seq-pr-inc* **by** *blast*
have *0*: *is-subseq-of s* (*filtered-seq s* (*kth-res-equals k l*))
unfolding *filtered-seq-def*
apply(*rule is-subseqI*[*of seq-filter s* (*kth-res-equals k l*)])
apply(*rule seq-filter-increasing*, *rule l-def*)
by *blast*
have *1*: ($\forall\, m.$ (*filtered-seq s* (*kth-res-equals k l*)) *m k* = *l*)
using *l-def*
by (*meson assms kth-res-equals-def fil-seq-pred padic-integers-axioms*)
show *?thesis* **using** *0 1* **by** *blast*
qed

lemma *const-res-subseq-prop-0*:
assumes $s \in$ *closed-seqs Zp*
shows $\exists\, l.$ ((((*const-res-subseq k s*) = *filtered-seq s* (*kth-res-equals k l*)) $\wedge$ (*is-subseq-of s* (*const-res-subseq k s*)) $\wedge$ ($\forall\, m.$(*const-res-subseq k s*) *m k* = *l*))
proof−
have $\exists\, n.$ (*is-subseq-of s* (*filtered-seq s* (*kth-res-equals k n*)) $\wedge$ ($\forall\, m.$ (*filtered-seq s* (*kth-res-equals k n*)) *m k* = *n*))
by (*simp add*: *kth-res-equals-subseq assms*)
then have $\exists\, s'.$ ($\exists\, n.$ (*is-subseq-of s s'*) $\wedge$ (*s'* = *filtered-seq s* (*kth-res-equals k n*)) $\wedge$ ($\forall\, m.$ *s' m k* = *n*))
by *blast*
then show *?thesis*
using *const-res-subseq-def*[*of k s*] *const-res-subseq-def someI-ex*
by (*smt* (*verit*) *const-res-subseq-def someI-ex*)
qed

lemma *const-res-subseq-prop-1*:
assumes $s \in$ *closed-seqs Zp*
shows ($\forall\, m.$(*const-res-subseq k s*) *m k* = (*const-res k s*))
using *const-res-subseq-prop-0*[*of s*] *const-res-def*[*of k s*]
by (*smt* (*verit*) *assms const-res-subseq-def const-res-def the-equality*)

lemma *const-res-subseq*:
assumes $s \in$ *closed-seqs Zp*
shows *is-subseq-of s* (*const-res-subseq k s*)
using *assms const-res-subseq-prop-0*[*of s k*] **by** *blast*

lemma *const-res-range*:
assumes $s \in$ *closed-seqs Zp*

assumes $k > 0$
shows *const-res k s* $\in$ *carrier* (*Zp-res-ring k*)
proof−
have *0*: (*const-res-subseq k s*) *0* $\in$ *carrier Zp*
using *const-res-subseq*[*of s k*] *is-subseq-closed*[*of s const-res-subseq k s*]
assms(*1*) *closed-seqs-memE* **by** *blast*
have *1*: (*const-res-subseq k s*) *0 k* $\in$ *carrier* (*Zp-res-ring k*)
using *0* **by** *simp*
then show *?thesis*
using *assms const-res-subseq-prop-1*[*of s k*]
by (*simp add*: ‹*s* $\in$ *closed-seqs Zp*›)
qed

fun *res-seq* ::*padic-int-seq* $\Rightarrow$ *nat* $\Rightarrow$ *padic-int-seq* **where**
res-seq s 0 = *s*|
res-seq s (*Suc k*) = *const-res-subseq* (*Suc k*) (*res-seq s k*)

lemma *res-seq-res*:
assumes *s* $\in$ *closed-seqs Zp*
shows (*res-seq s k*) $\in$ *closed-seqs Zp*
apply(*induction k*)
apply (*simp add*: *assms*)
by (*simp add*: *const-res-subseq is-subseq-closed*)

lemma *res-seq-res′*:
assumes *s* $\in$ *closed-seqs Zp*
shows $\bigwedge$*n*. *res-seq s* (*Suc k*) *n* (*Suc k*) = *const-res* (*Suc k*) (*res-seq s k*)
using *assms res-seq-res*[*of s k*] *const-res-subseq-prop-1*[*of* (*res-seq s k*) *Suc k*]
by *simp*

lemma *res-seq-subseq*:
assumes *s* $\in$ *closed-seqs Zp*
shows *is-subseq-of* (*res-seq s k*) (*res-seq s* (*Suc k*))
by (*metis assms const-res-subseq-prop-0 res-seq-res*
res-seq.simps(*2*))

lemma *is-increasing-id*:
is-increasing (λ *n*. *n*)
by (*simp add*: *is-increasingI*)

lemma *is-increasing-comp*:
assumes *is-increasing f*
assumes *is-increasing g*
shows *is-increasing* (*f* $\circ$ *g*)
using *assms*(*1*) *assms*(*2*) *is-increasing-def*
by *auto*

lemma *is-increasing-imp-geq-id*[*simp*]:
assumes *is-increasing f*

shows $f\ n \geq n$
apply(*induction n*)
apply *simp*
by (*metis* (*mono-tags*, *lifting*) *assms is-increasing-def*
leD lessI not-less-eq-eq order-less-le-subst2)

lemma *is-subseq-ofE*:
assumes $s \in$ *closed-seqs Zp*
assumes *is-subseq-of s s'*
shows $\exists k.\ k \geq n \wedge s'\ n = s\ k$
proof−
obtain *f* **where** *is-increasing f* $\wedge$ $s' =$ *take-subseq s f*
using *assms*(*2*) *is-subseq-of-def* **by** *blast*
then have $f\ n \geq n \wedge s'\ n = s\ (f\ n)$
unfolding *take-subseq-def*
by *simp*
then show *?thesis* **by** *blast*
qed

lemma *is-subseq-of-id*:
assumes $s \in$ *closed-seqs Zp*
shows *is-subseq-of s s*
proof−
have $s =$ *take-subseq s* $(\lambda n.\ n)$
unfolding *take-subseq-def*
by *auto*
then show *?thesis* **using** *is-increasing-id*
using *is-subseqI*
by *blast*
qed

lemma *is-subseq-of-trans*:
assumes $s \in$ *closed-seqs Zp*
assumes *is-subseq-of s s'*
assumes *is-subseq-of s' s''*
shows *is-subseq-of s s''*
proof−
obtain *f* **where** *f-def*: *is-increasing f* $\wedge$ $s' =$ *take-subseq s f*
using *assms*(*2*) *is-subseq-of-def*
by *blast*
obtain *g* **where** *g-def*: *is-increasing g* $\wedge$ $s'' =$ *take-subseq s' g*
using *assms*(*3*) *is-subseq-of-def*
by *blast*
have $s'' =$ *take-subseq s* $(f \circ g)$
proof
fix *x*
show $s''\ x =$ *take-subseq s* $(f \circ g)\ x$
using *f-def g-def* **unfolding** *take-subseq-def*

```
    by auto
  qed
  then show ?thesis
    using f-def g-def is-increasing-comp is-subseq-of-def
    by blast
qed

lemma res-seq-subseq':
  assumes s ∈ closed-seqs Zp
  shows is-subseq-of s (res-seq s k)
proof(induction k)
  case 0
  then show ?case using is-subseq-of-id
    by (simp add: assms)
next
  case (Suc k)
  fix k
  assume is-subseq-of s (res-seq s k)
  then show is-subseq-of s (res-seq s (Suc k))
    using assms is-subseq-of-trans res-seq-subseq
    by blast
qed

lemma res-seq-subseq'':
  assumes s ∈ closed-seqs Zp
  shows is-subseq-of (res-seq s n) (res-seq s (n + k))
  apply(induction k)
  apply (simp add: assms is-subseq-of-id res-seq-res)
  using add-Suc-right assms is-subseq-of-trans res-seq-res res-seq-subseq by pres-
burger


definition acc-point :: padic-int-seq ⇒ padic-int where
acc-point s k = (if (k = 0) then (0::int) else ((res-seq s k) 0 k))

lemma res-seq-res-1:
  assumes s ∈ closed-seqs Zp
  shows res-seq s (Suc k) 0 k = res-seq s k 0 k
proof−
  obtain n where n-def: res-seq s (Suc k) 0 = res-seq s k n
    by (metis assms is-subseq-of-def res-seq-subseq take-subseq-def)
  have res-seq s (Suc k) 0 k = res-seq s k n k
    using n-def by auto
  thus ?thesis
    using assms padic-integers.p-res-ring-0'
      padic-integers-axioms res-seq.elims residues-closed
  proof −
    have ∀ n. s n ∈ carrier Zp
      by (simp add: assms closed-seqs-memE)
```

then show *?thesis*
by (*metis ‹res-seq s (Suc k) 0 k = res-seq s k n k› assms padic-integers.p-res-ring-0' padic-integers-axioms res-seq.elims res-seq-res' residues-closed*)
qed
qed

lemma *acc-point-cres*:
assumes *s ∈ closed-seqs Zp*
shows (*acc-point s (Suc k)*) = (*const-res (Suc k) (res-seq s k)*)
proof−
have *Suc k > 0* **by** *simp*
have (*res-seq s (Suc k)*) = *const-res-subseq (Suc k) (res-seq s k)*
by *simp*
then have (*const-res-subseq (Suc k) (res-seq s k)*) *0* (*Suc k*) = *const-res (Suc k) (res-seq s k)*
using *assms res-seq-res' padic-integers-axioms* **by** *auto*
have *acc-point s (Suc k)* = *res-seq s (Suc k) 0 (Suc k)* **using** *acc-point-def* **by** *simp*
then have *acc-point s (Suc k)* = (*const-res-subseq (Suc k) (res-seq s k)*) *0* (*Suc k*)
by *simp*
thus *?thesis*
by (*simp add*: ‹(*const-res-subseq (Suc k) (res-seq s k)*) *0* (*Suc k*) = *const-res (Suc k) (res-seq s k)*›)
qed

lemma *acc-point-res*:
assumes *s ∈ closed-seqs Zp*
shows *residue* (*p* ^ *k*) (*acc-point s (Suc k)*) = *acc-point s k*
proof(*cases k = 0*)
case *True*
then show *?thesis*
by (*simp add*: *acc-point-def residue-1-zero*)
next
case *False*
assume *k ≠ 0* **show** *residue* (*p* ^ *k*) (*acc-point s (Suc k)*) = *acc-point s k*
using *False acc-point-def assms lessI less-imp-le nat.distinct*(*1*) *res-seq-res-1 res-seq-res*
Zp-defs(*3*) *closed-seqs-memE prime* **by** (*metis padic-set-res-coherent*)
qed

lemma *acc-point-closed*:
assumes *s ∈ closed-seqs Zp*
shows *acc-point s ∈ carrier Zp*
proof−
have *acc-point s ∈ padic-set p*
proof(*rule padic-set-memI*)
show ⋀*m*. *acc-point s m ∈ carrier* (*residue-ring* (*p* ^ *m*))
proof−

```
    fix m
    show acc-point s m ∈ carrier (residue-ring (p ^ m))
    proof(cases m = 0)
      case True
      then show ?thesis
        by (simp add: acc-point-def residue-ring-def)
    next
      case False
      assume m ≠ 0
      then have acc-point s m = res-seq s m 0 m
        by (simp add: acc-point-def)
     then show ?thesis  using const-res-range[of (const-res-subseq (m−1) s) m]
acc-point-def[of s m]
        by (metis False Suc-pred acc-point-cres assms const-res-range neq0-conv
res-seq-res)
    qed
  qed
  show ⋀m n. m < n ⟹ residue (p ^ m) (acc-point s n) = acc-point s m
  proof−
    fix m n::nat
    assume A: m < n
    show residue (p ^ m) (acc-point s n) = acc-point s m
    proof−
      obtain l where l-def: l = n − m − 1
        by simp
      have residue (p ^ m) (acc-point s (Suc (m + l))) = acc-point s m
      proof(induction l)
        case 0
        then show ?case
          by (simp add: acc-point-res assms)
      next
        case (Suc l)
        then show ?case
       using Zp-defs(3) acc-point-def add-Suc-right assms  le-add1 closed-seqs-memE
nat.distinct(1)
              padic-integers.prime padic-integers-axioms res-seq-res res-seq-res-1
          by (metis padic-set-res-coherent)
      qed
      then show ?thesis
        by (metis A Suc-diff-Suc Suc-eq-plus1 add-Suc-right add-diff-inverse-nat
diff-diff-left
            l-def le-less-trans less-not-refl order-less-imp-le)
    qed
  qed
 qed
 then show ?thesis
   by (simp add: Zp-defs(3))
qed
```

Choice function for a subsequence of s which converges to a, if it exists

fun *convergent-subseq-fun* :: *padic-int-seq* ⇒ *padic-int* ⇒ (*nat* ⇒ *nat*) **where**
convergent-subseq-fun s a 0 = 0|
convergent-subseq-fun s a (*Suc n*) = (*SOME k*. *k* > (*convergent-subseq-fun s a n*)
∧ (*s k* (*Suc n*)) = *a* (*Suc n*))

definition *convergent-subseq* :: *padic-int-seq* ⇒ *padic-int-seq* **where**
convergent-subseq s = *take-subseq s* (*convergent-subseq-fun s* (*acc-point s*))

lemma *increasing-conv-induction-0-pre*:
assumes *s* ∈ *closed-seqs Zp*
assumes *a* = *acc-point s*
shows ∃ *k* > *convergent-subseq-fun s a n*. (*s k* (*Suc n*)) = *a* (*Suc n*)
proof−
obtain *l*::*nat* **where** *l* > *0* **by** *blast*
have *is-subseq-of s* (*res-seq s* (*Suc n*))
using *assms*(*1*) *res-seq-subseq*′ **by** *blast*
then obtain *m* **where** *s m* = *res-seq s* (*Suc n*) *l* ∧ *m* ≥ *l*
by (*metis is-increasing-imp-geq-id is-subseq-of-def take-subseq-def*)
have *a* (*Suc n*) = *res-seq s* (*Suc n*) *0* (*Suc n*)
by (*simp add*: *acc-point-def assms*(*2*))
have *s m* (*Suc n*) = *a* (*Suc n*)
by (*metis* ‹*a* (*Suc n*) = *res-seq s* (*Suc n*) *0* (*Suc n*)› ‹*s m* = *res-seq s* (*Suc n*) *l* ∧ *l* ≤ *m*› *assms*(*1*) *res-seq-res*′)
thus *?thesis*
using ‹*0* < *l*› ‹*s m* = *res-seq s* (*Suc n*) *l* ∧ *l* ≤ *m*› *less-le-trans* ‹*s m* (*Suc n*) = *a* (*Suc n*)›
by (*metis* ‹*a* (*Suc n*) = *res-seq s* (*Suc n*) *0* (*Suc n*)› ‹*is-subseq-of s* (*res-seq s* (*Suc n*))›
assms(*1*) *lessI is-subseq-ofE res-seq-res*′)
qed

lemma *increasing-conv-subseq-fun-0*:
assumes *s* ∈ *closed-seqs Zp*
assumes ∃ *s*′. *s*′ = *convergent-subseq s*
assumes *a* = *acc-point s*
shows *convergent-subseq-fun s a* (*Suc n*) > *convergent-subseq-fun s a n*
apply(*auto*)
proof(*induction n*)
case *0*
then show *?case*
by (*metis* (*mono-tags*, *lifting*) *assms*(*1*) *assms*(*3*) *increasing-conv-induction-0-pre someI-ex*)
next
case (*Suc k*)
then show *?case*
by (*metis* (*mono-tags*, *lifting*) *assms*(*1*) *assms*(*3*) *increasing-conv-induction-0-pre someI-ex*)
qed

lemma *increasing-conv-subseq-fun*:
assumes $s \in$ *closed-seqs Zp*
assumes $a =$ *acc-point s*
assumes $\exists\, s'.\ s' =$ *convergent-subseq s*
shows *is-increasing* (*convergent-subseq-fun s a*)
by (*metis assms*(*1*) *assms*(*2*) *increasing-conv-subseq-fun-0 is-increasingI lift-Suc-mono-less*)

lemma *convergent-subseq-is-subseq*:
assumes $s \in$ *closed-seqs Zp*
shows *is-subseq-of s* (*convergent-subseq s*)
using *assms convergent-subseq-def increasing-conv-subseq-fun is-subseqI* **by** *blast*

lemma *is-closed-seq-conv-subseq*:
assumes $s \in$ *closed-seqs Zp*
shows (*convergent-subseq s*) $\in$ *closed-seqs Zp*
by (*simp add*: *assms convergent-subseq-def closed-seqs-memI closed-seqs-memE take-subseq-def*)

lemma *convergent-subseq-res*:
assumes $s \in$ *closed-seqs Zp*
assumes $a =$ *acc-point s*
shows *convergent-subseq s l l* = *residue* (*p* ^ *l*) (*acc-point s l*)
proof−
have $\exists\, k.$ *convergent-subseq s l* = *s k* $\wedge$ *s k l* = *a l*
proof−
have *convergent-subseq s l* = *s* (*convergent-subseq-fun s a l*)
by (*simp add*: *assms*(*2*) *convergent-subseq-def take-subseq-def*)
obtain *k* **where** *kdef*: (*convergent-subseq-fun s a l*) = *k*
by *simp*
have *convergent-subseq s l* = *s k*
by (*simp add*: ‹*convergent-subseq s l* = *s* (*convergent-subseq-fun s a l*)› *kdef*)
have *s k l* = *a l*
proof(*cases l* = *0*)
case *True*
then show *?thesis*
using *acc-point-def assms*(*1*) *assms*(*2*)
by (*metis closed-seqs-memE p-res-ring-0*′ *residues-closed*)
next
case *False*
have *0* < *l*
using *False* **by** *blast*
then have *k* > *convergent-subseq-fun s a* (*l*−*1*)
by (*metis One-nat-def Suc-pred assms*(*1*) *assms*(*2*) *increasing-conv-subseq-fun-0 kdef*)
then have *s k l* = *a l* **using** *kdef*
assms(*1*) *assms*(*2*) *convergent-subseq-fun.simps*(*2*) *increasing-conv-induction-0-pre*

padic-integers-axioms someI-ex One-nat-def ‹*0* < *l*› *increasing-conv-induction-0-pre*

```
      by (smt (verit) Suc-pred)
    then show ?thesis
      by simp
  qed
  then have convergent-subseq s l =  s k ∧ s k l = a l
    using ‹convergent-subseq s l = s k› by blast
  thus ?thesis
    by blast
 qed
 thus ?thesis
  using acc-point-closed assms(1) assms(2) Zp-defs(3) prime padic-set-res-coherent
by force
qed

lemma convergent-subseq-res':
 assumes s ∈ closed-seqs Zp
 assumes n > l
 shows convergent-subseq s n l = convergent-subseq s l l
proof−
 have 0: convergent-subseq s l l = residue (p ^ l) (acc-point s l)
   using assms(1) convergent-subseq-res by auto
 have 1: convergent-subseq s n n = residue (p ^ n) (acc-point s n)
   by (simp add: assms(1) convergent-subseq-res)
 have 2: convergent-subseq s n l = residue (p ^ l) (convergent-subseq s l l)
   using 0 assms 1 Zp-defs(3) acc-point-closed is-closed-seq-conv-subseq
     closed-seqs-memE le-refl less-imp-le-nat prime
   by (metis padic-set-res-coherent)
 show ?thesis using 0 1 2 Zp-defs(3) assms(1) is-closed-seq-conv-subseq closed-seqs-memE
le-refl prime
   by (metis padic-set-res-coherent)
qed

lemma convergent-subsequence-is-convergent:
 assumes s ∈ closed-seqs Zp
 assumes a = acc-point s
 shows Zp-converges-to (convergent-subseq s) (acc-point s)
proof(rule Zp-converges-toI)
 show acc-point s ∈ carrier Zp
   using acc-point-closed assms  by blast
 show convergent-subseq s ∈ carrier (Zp^ω)
   using is-closed-seq-conv-subseq assms by simp
 show ⋀n. ∃ N. ∀ k>N. convergent-subseq s k n = acc-point s n
 proof−
   fix n
   show ∃ N. ∀ k>N. convergent-subseq s k n = acc-point s n
   proof(induction n)
     case 0
     then show ?case
       using acc-point-closed[of s] assms convergent-subseq-def closed-seqs-memE
```

```
of-nat-0
            ord-pos take-subseq-def zero-below-ord is-closed-seq-conv-subseq[of s]
      by (metis residue-of-zero(2))
   next
     case (Suc n)
     have acc-point s (Suc n) = res-seq s (Suc n) 0 (Suc n)
       by (simp add: acc-point-def)
     obtain k where kdef: convergent-subseq-fun s a (Suc n) = k by simp
     have Suc n > 0 by simp
     then have k > (convergent-subseq-fun s a n)
       using assms(1) assms(2) increasing-conv-subseq-fun-0 kdef by blast
      then have  k > (convergent-subseq-fun s a n) ∧ (s k (Suc n)) = a (Suc n)
using kdef
     by (metis (mono-tags, lifting) assms(1) assms(2) convergent-subseq-fun.simps(2)
increasing-conv-induction-0-pre someI-ex)
     have s k (Suc n) = a (Suc n)
      using ‹convergent-subseq-fun s a n < k ∧ s k (Suc n) = a (Suc n)› by blast
     then have convergent-subseq s (Suc n) (Suc n) = a (Suc n)
       by (metis assms(2) convergent-subseq-def kdef take-subseq-def)
     then have ∀ l > n.  convergent-subseq s l (Suc n) = a (Suc n)
       using convergent-subseq-res'
       by (metis Suc-lessI assms(1))
     then show ?case
       using assms(2) by blast
   qed
 qed
qed

theorem Zp-is-compact:
 assumes s ∈ closed-seqs Zp
 shows ∃ s'. is-subseq-of s s' ∧ (Zp-converges-to s' (acc-point s))
 using assms convergent-subseq-is-subseq convergent-subsequence-is-convergent
 by blast

end
end
```

References

[1] K. Conrad. Hensel's lemma.

[2] J. Denef. p-adic semi-algebraic sets and cell decomposition. *Journal für die reine und angewandte Mathematik*, 369:154–166, 1986.

[3] D. Dummit. *Abstract algebra*. John Wiley & Sons, Inc, Hoboken, NJ, 2004.

[4] A. Engler. *Valued fields*. Springer, Berlin New York, 2005.

[5] R. Y. Lewis. A formal proof of hensel's lemma over the p-adic integers. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, pages 15–26, New York, NY, USA, 2019. Association for Computing Machinery.

[6] J. Pas. On the angular component map modulo p. *The Journal of Symbolic Logic*, 55(3):1125–1129, 1990.