example-Bicomplex

By jmaransay

April 17, 2009

Contents

1 Definition of a ring of completion homomorphisms

theory *HomGroupCompletion*
imports
~~/src/HOL/Algebra/Ring
begin

2 Definition of completion functions and some related lemmas

constdefs
completion :: [('a, 'c) *monoid-scheme*, ('b, 'd) *monoid-scheme*, ('a => 'b)] => ('a => 'b)
completion G G' f == (%x. *if* x ∈ *carrier G then f x else one G'*)

lemma *completion-in-funcset*: (!!x. x ∈ *carrier G* ==> *f x* ∈ *carrier G'*) ==> (*completion G G' f*) ∈ *carrier G* –> *carrier G'*
⟨*proof*⟩

lemma *completion-in-hom*: **includes** *group-hom* G G' h **shows** *completion* G G' $h \in$ *hom* G G'
⟨*proof*⟩

lemma *completion-apply-carrier* [*simp*]: $x \in$ *carrier* G ==> *completion* G G' h x = h x
⟨*proof*⟩

lemma *completion-apply-not-carrier* [*simp*]: $x \notin$ *carrier* G ==> *completion* G G' h x = *one* G'
⟨*proof*⟩

lemma *completion-ext*: (!!x. $x \in$ *carrier* G ==> h x = g x) ==> (*completion* G G' h) = (*completion* G G' g)
⟨*proof*⟩

lemma *inj-on-completion-eq*: *inj-on* (*completion* G G' h) (*carrier* G) = *inj-on* h (*carrier* G)
⟨*proof*⟩

constdefs
completion-fun :: [('a, 'c) *monoid-scheme*, ('b, 'd) *monoid-scheme*] => ('a => 'b) *set*
completion-fun G G' == {f. f = (%x. *if* $x \in$ *carrier* G *then* f x *else one* G')}

constdefs
completion-fun2 :: [('a, 'c) *monoid-scheme*, ('b, 'd) *monoid-scheme*] => ('a => 'b) *set*
completion-fun2 G G' == {f. $\exists$ g. f = *completion* G G' g}

lemma *f-in-completion-fun2-f-completion*: $f \in$ *completion-fun2* G G' ==> f = *completion* G G' f
⟨*proof*⟩

lemma *completion-in-completion-fun*: *completion* G G' $h \in$ *completion-fun* G G'
⟨*proof*⟩

lemma *completion-in-completion-fun2*: **shows** *completion* G G' $h \in$ *completion-fun2* G G'
⟨*proof*⟩

lemma *completion-fun-completion-fun2*: *completion-fun* G G' = *completion-fun2* G G'
⟨*proof*⟩

lemma *completion-id-in-completion-fun*: **shows** *completion* G G' *id* $\in$ *completion-fun* G G'
⟨*proof*⟩

lemma *completion-closed2*: **assumes** h: $h \in$ *completion-fun2* G G' **and** x: $x \notin$ *carrier* G **shows** $h\ x = $ *one* G'
⟨*proof*⟩

2.1 Homomorphisms defined as completions

constdefs
hom-completion :: [('*a*, '*c*) *monoid-scheme*, ('*b*, '*d*) *monoid-scheme*] => ('*a* => '*b*)*set*
hom-completion G G' == {h. $h \in$ *completion-fun2* G G' & $h \in$ *hom* G G'}

lemma *hom-completionI*: **assumes** $h \in$ *completion-fun2* G G' **and** $h \in$ *hom* G G' **shows** $h \in$ *hom-completion* G G'
⟨*proof*⟩

lemma *hom-completion-is-hom*: **assumes** f: $f \in$ *hom-completion* G G' **shows** $f \in$ *hom* G G'
⟨*proof*⟩

lemma *hom-completion-mult*: **assumes** $h \in$ *hom-completion* G G' **and** $x \in$ *carrier* G **and** $y \in$ *carrier* G
shows h (*mult* G x y) = *mult* G' (h x) (h y)
⟨*proof*⟩

lemma *hom-completion-closed*: **assumes** h: $h \in$ *hom-completion* G G' **and** x: $x \in$ *carrier* G **shows** $h\ x \in$ *carrier* G'
⟨*proof*⟩

lemma *hom-completion-one*[*simp*]: **includes** *group* G + *group* G'
assumes h: $h \in$ *hom-completion* G G' **shows** h (*one* G) = *one* G'
⟨*proof*⟩

lemma *comp-sum*: **includes** *group* G
assumes h: $h \in$ *hom* G G **and** h': $h' \in$ *hom* G G **and** x: $x \in$ *carrier* G **and** y: $y \in$ *carrier* G
shows h' (h (*mult* G x y)) = *mult* G (h' (h x)) (h' (h y))
⟨*proof*⟩

lemma *comp-is-hom*: **includes** *group* G
assumes h: $h \in$ *hom* G G **and** h': $h' \in$ *hom* G G
shows $h' \circ h \in$ *hom* G G
⟨*proof*⟩

Usual composition *op* $\circ$ of completion homomorphisms is closed

lemma *hom-completion-comp*: **includes** *group* G
assumes $f \in$ *hom-completion* G G **and** $g \in$ *hom-completion* G G
shows $f \circ g \in$ *hom-completion* G G
⟨*proof*⟩

2.2 Completion homomorphisms with usual composition form a monoid

The underlying algebraic structures in our development, except otherwise stated, will be commutative groups or differential groups

lemma (**in** *comm-group*) *hom-completion-monoid*:
shows *monoid* (| *carrier* = *hom-completion G G*, *mult* = *op o*, *one* = (λx. *if x* $\in$ *carrier G then id x else* **1**) |)
(**is** *monoid ?H-CO*)
⟨*proof*⟩

Homomorphisms, without the completion condition, are also a monoid with usual composition and the identity

lemma (**in** *group*) *hom-group-monoid*:
shows *monoid* (| *carrier* = *hom G G*, *mult* = *op o*, *one* = *id* |)
(**is** *monoid ?HOM*)
⟨*proof*⟩

2.3 Preliminary facts about addition of homomorphisms

lemma *homI*:
assumes *closed*: $\bigwedge x.\ x \in carrier\ G \Longrightarrow f\ x \in carrier\ H$
and *mult*: $\bigwedge x\ y.\ [\![x \in carrier\ G;\ y \in carrier\ G]\!] \Longrightarrow f\ (x \otimes_G y) = f\ x \otimes_H f\ y$
shows $f \in hom\ G\ H$ ⟨*proof*⟩

The operation we are going to use as addition for homomorphisms is based on the multiplicative operation of the underlying algebraic structures

The three following lemmas show how we can define the addition of homomorphisms in different ways with satisfactory result

lemma (**in** *comm-group*) *hom-mult-is-hom*: **assumes** *F*: $f \in hom\ G\ G$ **and** *G*: $g \in hom\ G\ G$ **shows** $(\lambda x.\ f\ x \otimes g\ x) \in hom\ G\ G$
⟨*proof*⟩

lemma (**in** *comm-group*) *hom-mult-is-hom-rest*:
assumes *f*: $f \in hom\ G\ G$ **and** *g*: $g \in hom\ G\ G$
shows $(\lambda x{\in}carrier\ G.\ f\ x \otimes g\ x) \in hom\ G\ G$ (**is** *?fg* $\in$ -)
⟨*proof*⟩

lemma (**in** *comm-group*) *hom-mult-is-hom-completion*:
assumes *f*: $f \in hom\ G\ G$ **and** *g*: $g \in hom\ G\ G$
shows (λx. *if* $x \in carrier\ G$ *then* $f\ x \otimes g\ x$ *else* **1**) $\in hom\ G\ G$
(**is** *?fg* $\in$ -)
⟨*proof*⟩

The inverse for the addition of homomorphisms will be given by the $\lambda x.\ inv$ $f\ x$ operation

lemma (**in** *comm-group*) *hom-inv-is-hom*: **assumes** f: $f \in$ *hom* G G **shows** (λx. *inv* f x) $\in$ *hom* G G
⟨*proof*⟩

Lemma *?f* $\in$ *hom* G $G \Longrightarrow$ (λx. *inv ?f* x) $\in$ *hom* G G proves that the multiplicative inverse of the underlying structure preserves the homomorphism definition

locale *group-end* = *group-hom* G G h

Due to the partial definitions of domains, it would not be possible to prove that $h \circ$ (λx. *inv* h x) = (λx. $\mathbf{1}$); the closer fact that can be proven is $h \circ$ (λx. *inv* h x) = ($\lambda x \in$ *carrier* G. $\mathbf{1}$);

lemma (**in** *comm-group*) *hom-completion-inv-is-hom-completion*:
  assumes $f \in$ *hom-completion* G G
  shows (λx. *if* $x \in$ *carrier* G *then inv* f x *else* $\mathbf{1}$) $\in$ *hom-completion* G G
⟨*proof*⟩

lemma (**in** *comm-group*) *hom-completion-mult-inv-is-hom-completion*:
  assumes $f \in$ *hom-completion* G G
  shows $\exists$ $g \in$*hom-completion* G G. (λx. *if* $x \in$ *carrier* G *then* g $x \otimes f$ x *else* $\mathbf{1}$) = (λx. $\mathbf{1}$)
⟨*proof*⟩

2.4 Completion homomorphisms are a commutative group with the underlying operation

lemma (**in** *comm-group*) *hom-completion-mult-comm-group*:
  shows *comm-group* (|*carrier* = *hom-completion* G G, *mult* = λf. λg. (λx. *if* $x \in$ *carrier* G *then* f $x \otimes g$ x *else* $\mathbf{1}$),
  one = (λx. *if* $x \in$ *carrier* G *then* $\mathbf{1}$ *else* $\mathbf{1}$)|)
  (**is** *comm-group ?H-CO*)
⟨*proof*⟩

lemma (**in** *comm-group*) *hom-completion-mult-comm-group2*:
  shows *comm-group* (| *carrier* = *hom-completion* G G, *mult* = λf. λg. (λx. *if* $x \in$ *carrier* G *then* f $x \otimes g$ x *else* $\mathbf{1}$), *one* = (λx. $\mathbf{1}$)|)
⟨*proof*⟩

lemma (**in** *comm-group*) *hom-completion-mult-comm-monoid*:
  includes *comm-group* G
  shows *comm-monoid* (| *carrier* = *hom-completion* G G, *mult* = λf. λg. (λx. *if* $x \in$ *carrier* G *then* f $x \otimes g$ x *else* $\mathbf{1}$), *one* = (λx. $\mathbf{1}$)|)
⟨*proof*⟩

2.5 Endomorphisms with suitable operations form a ring

The distributive law is proved first

lemma (**in** *comm-group*) *r-mult-dist-add*: **assumes**$f \in$ *hom-completion* G G **and** $g \in$ *hom-completion* G G **and** $h \in$ *hom-completion* G G
shows $(\lambda x.$ *if* $x \in$ *carrier* G *then* $f\ x \otimes g\ x$ *else* $\mathbf{1})$ o $h = (\lambda x.$ *if* $x \in$ *carrier* G *then* $(f\ o\ h)\ x \otimes (g\ o\ h)\ x$ *else* $\mathbf{1})$
⟨*proof*⟩

lemma (**in** *comm-group*) *l-mult-dist-add*: **assumes**$f \in$ *hom-completion* G G **and** $g \in$ *hom-completion* G G **and** $h \in$ *hom-completion* G G
shows $h \circ (\lambda x.$ *if* $x \in$ *carrier* G *then* $f\ x \otimes g\ x$ *else* $\mathbf{1}) = (\lambda x.$ *if* $x \in$ *carrier* G *then* $(h\ o\ f)\ x \otimes (h\ o\ g)\ x$ *else* $\mathbf{1})$
⟨*proof*⟩

Endomorphisms with the previous operations form a ring

lemma (**in** *comm-group*) *hom-completion-ring*:
shows *ring* (| *carrier* = *hom-completion* G G, *mult* = *op* o, *one* = $(\lambda x.$ *if* $x \in$ *carrier* G *then id* x *else* $\mathbf{1})$,
zero = $(\lambda x.$ *if* $x \in$ *carrier* G *then* $\mathbf{1}$ *else* $\mathbf{1})$, *add* = $\lambda f.\ \lambda g.\ (\lambda x.$ *if* $x \in$ *carrier* G *then* $f\ x \otimes g\ x$ *else* $\mathbf{1})$|)
⟨*proof*⟩

locale *hom-completion-ring* = *comm-group* G + *ring* R +
assumes R = (| *carrier* = *hom-completion* G G, *mult* = *op* o,
one = $(\lambda x.$ *if* $x \in$ *carrier* G *then id* x *else* $\mathbf{1})$,
zero = $(\lambda x.$ *if* $x \in$ *carrier* G *then one* G *else* $\mathbf{1})$,
add = $\lambda f.\ \lambda g.\ (\lambda x.$ *if* $x \in$ *carrier* G *then* $f\ x \otimes g\ x$ *else* $\mathbf{1})$|)

Some examples where it is shown the usefulness of the previous proofs

lemma (**in** *hom-completion-ring*) *r-dist-minus*:
[|$f \in$ *carrier* R; $g \in$ *carrier* R; $h \in$ *carrier* R|]
==> $(f \ominus_2 g) \otimes_2 h = (f \otimes_2 h) \ominus_2 (g \otimes_2 h)$ ⟨*proof*⟩

lemma (**in** *hom-completion-ring*) *sublemma*:
[| $f \in$ *carrier* R; $h \in$ *carrier* R; $f \otimes_2 h = h$ |] ==> $(\mathbf{1}_2 \ominus_2 f) \otimes_2 h = \mathbf{0}_2$ ⟨*proof*⟩

2.6 Definition of differential group

According to Section 2.3 in Aransay's memoir, in the following we will be dealing with ungraded algebraic structures.

The Basic Perturbation Lemma is usually stated in terms of differential structures; these include diferential groups as well as chain complexes.

Moreover, chain complexes can be defined in terms of differential groups (more concretely, as indexed collections of differential groups).

The proof of the Basic Perturbation Lemma does not include any reference to graded structures or proof obligations derived from the degree information.

Thus, we preferred to state and prove the Basic Perturbation Lemma in terms of ungrades structures (differential and abelian groups), for the sake of simplicity, and avoid implementing and dealing with graded structures (chain complexes and graded groups).

record $'a$ *diff-group* $=$ $'a$ *monoid* $+$
  diff $::$ $'a \Rightarrow 'a$ (*differı 81*)

locale *diff-group* $=$ *comm-group* D $+$
  assumes *diff-hom* : *differ* $\in$ *hom-completion* D D
  and *diff-nilpot* : *differ* $\circ$ *differ* $= (\lambda x.\ \mathbf{1})$

lemma *diff-groupI*:
  includes *struct* D
  assumes *m-closed*:
  $!!x\ y.\ [|\ x \in carrier\ D;\ y \in carrier\ D\ |] ==> x \otimes y \in carrier\ D$
  and *one-closed*: $\mathbf{1} \in carrier\ D$
  and *m-assoc*:
  $!!x\ y\ z.\ [|\ x \in carrier\ D;\ y \in carrier\ D;\ z \in carrier\ D\ |] ==> (x \otimes y) \otimes z = x \otimes (y \otimes z)$
  and *m-comm*:
  $!!x\ y.\ [|\ x \in carrier\ D;\ y \in carrier\ D\ |] ==> x \otimes y = y \otimes x$
  and *l-one*: $!!x.\ x \in carrier\ D ==> \mathbf{1} \otimes x = x$
  and *l-inv-ex*: $!!x.\ x \in carrier\ D ==> \exists\, y \in carrier\ D.\ y \otimes x = \mathbf{1}$
  and *diff-hom*: *differ* $\in$ *hom-completion* D D
  and *diff-nilpot*: $!!x.\ (differ)\ ((differ)\ x) = \mathbf{1}$
  shows *diff-group* D
  ⟨*proof*⟩

2.7 Definition of homomorphisms between differential groups

locale *hom-completion-diff* $=$ *diff-group* C $+$ *diff-group* D $+$ *var* f $+$
  assumes *f-hom-completion*: $f \in$ *hom-completion* C D
  and *f-coherent*: $f \circ differ_1 = differ_2 \circ f$

constdefs (**structure** C **and** D)
  hom-diff $::$ - $=>$ - $=>$ $('a => 'b)$ *set*
  hom-diff C D $== \{f.\ f \in$ *hom-completion* C D & $(f \circ (differ_C) = (differ_D) \circ f)\}$

lemma *hom-diff-is-hom-completion*: **assumes** h: $h \in$ *hom-diff* C D
  shows $h \in$ *hom-completion* C D
  ⟨*proof*⟩

lemma *hom-diff-closed*: **assumes** h: $h \in$ *hom-diff* C D **and** x: $x \in carrier\ C$ **shows** $h\ x \in carrier\ D$
  ⟨*proof*⟩

lemma *hom-diff-mult*: **assumes** h: $h \in$ *hom-diff* C D **and** x: $x \in carrier\ C$ **and** y: $y \in carrier\ C$ **shows** $h\ (x \otimes_C y) = h\ (x) \otimes_D h\ (y)$
  ⟨*proof*⟩

lemma *hom-diff-coherent*: **assumes** *h*: $h \in$ *hom-diff C D* **shows** $h \circ differ_C = differ_D \circ h$
⟨*proof*⟩

lemma (**in** *diff-group*) *hom-diff-comp-closed*: **assumes** $f \in$ *hom-diff D D* **and** $g \in$ *hom-diff D D* **shows** $g \circ f \in$ *hom-diff D D*
⟨*proof*⟩

lemma (**in** *diff-group*) *hom-diff-monoid*:
shows *monoid* (|*carrier* = *hom-diff D D*, *mult* = *op o*, *one* = (λx. *if* $x \in$ *carrier D then id x else* **1**)|)
(**is** *monoid ?DIFF*)
⟨*proof*⟩

2.8 Completion homomorphisms between differential structures form a commutative group with the underlying operation

lemma (**in** *diff-group*) *hom-diff-mult-closed*: **assumes** $f \in$ *hom-diff D D* **and** $g \in$ *hom-diff D D*
shows (λx. *if* $x \in$ *carrier D then* $f\ x \otimes g\ x$ *else* **1**) $\in$ *hom-diff D D*
⟨*proof*⟩

lemma (**in** *diff-group*) *hom-diff-inv-def*: **assumes** $f \in$ *hom-diff D D*
shows (λx. *if* $x \in$ *carrier D then inv f x else* **1**) $\in$ *hom-diff D D*
⟨*proof*⟩ **thm** *inv-one*
⟨*proof*⟩

lemma (**in** *diff-group*) *hom-diff-inv*: **assumes** $f \in$ *hom-diff D D*
shows $\exists\, g \in$ *hom-diff D D*. (λx. *if* $x \in$ *carrier D then* $g\ x \otimes f\ x$ *else* **1**) = (λx. **1**)
⟨*proof*⟩

2.9 Differential homomorphisms form a commutative group with the underlying operation

lemma (**in** *diff-group*) *hom-diff-mult-comm-group*:
shows *comm-group* (|*carrier* = *hom-diff D D*, *mult* = λf. λg. (λx. *if* $x \in$ *carrier D then* $f\ x \otimes g\ x$ *else* **1**),
one = (λx. *if* $x \in$ *carrier D then* **1** *else* **1**)|)
(**is** *comm-group ?DIFF*)
⟨*proof*⟩

The completion homomorphisms between differential groups are a ring with suitable operations

lemma (**in** *diff-group*) *hom-diff-ring*:
shows *ring* (| *carrier* = *hom-diff D D*, *mult* = *op o*, *one* = (λx. *if* $x \in$ *carrier D then id x else* **1**),
zero = (λx. *if* $x \in$ *carrier D then* **1** *else* **1**), *add* = λf. λg. (λx. *if* $x \in$ *carrier D then* $f\ x \otimes g\ x$ *else* **1**)|)

(**is** *ring ?DIFF*)
$\langle proof \rangle$

end

theory *HomGroupsCompletion*
imports
HomGroupCompletion
begin

2.10 Homomorphisms seen as algebraic structures

Homomorphisms with the underlying operation are closed

lemma *hom-mult-completion-is-hom*:
includes *comm-group* G + *comm-group* G'
shows [|f : *hom* G G'; g : *hom* G G' |] ==> (%x. *if* $x \in$ *carrier* G *then* $f\ x \otimes_2$ $g\ x$ *else* $\mathbf{1}_2$) : *hom* G G'
$\langle proof \rangle$

lemma *hom-completion-mult-is-hom-completion*:
includes *comm-group* G + *comm-group* G'
assumes $f \in$ *hom-completion* G G' **and** $g \in$ *hom-completion* G G'
shows (λx. *if* $x \in$ *carrier* G *then* $f\ x \otimes_{G'}$ $g\ x$ *else* $\mathbf{1}_{G'}$) $\in$ *hom-completion* G G'
$\langle proof \rangle$

Proof of the existence of an inverse homomorphism

lemma *hom-completions-mult-inv-is-hom-completion*:
includes *comm-group* G + *comm-group* G'
assumes $f \in$ *hom-completion* G G'
shows $\exists\, g \in$*hom-completion* G G'. (λx. *if* $x \in$ *carrier* G *then* $g\ x \otimes_{G'} f\ x$ *else* $\mathbf{1}_{G'}$) = (λx. $\mathbf{1}_{G'}$)
$\langle proof \rangle$

2.11 Completion homomorphisms between two algebraic structures form a commutative group

lemma *hom-completion-groups-mult-comm-group*:
includes *comm-group* G + *comm-group* G'
shows *comm-group* (| *carrier* = *hom-completion* G G', *mult* = λf. λg. (λx. *if* $x \in$ *carrier* G *then* $f\ x \otimes_2$ $g\ x$ *else* $\mathbf{1}_2$),
one = (λx. *if* $x \in$ *carrier* G *then* $\mathbf{1}_2$ *else* $\mathbf{1}_2$)|)
(**is** *comm-group ?H-CO*)
$\langle proof \rangle$

2.12 Previous facts about homomorphisms of differential structures

lemma *hom-diff-mult-is-hom-diff*:

includes *diff-group* D + *diff-group* D'
assumes $f \in$ *hom-diff* D D' **and** $g \in$ *hom-diff* D D'
shows $(\lambda x.$ *if* $x \in$ *carrier* D *then* $f\ x \otimes_{D'}\ g\ x$ *else* $\mathbf{1}_{D'}) \in$ *hom-diff* D D'
⟨*proof*⟩

lemma *hom-diff-mult-inv-is-hom-diff*:
includes *diff-group* D + *diff-group* D'
assumes $f \in$ *hom-diff* D D'
shows $\exists\, g \in$ *hom-diff* D D'. $(\lambda x.$ *if* $x \in$ *carrier* D *then* $g\ x \otimes_{D'} f\ x$ *else* $\mathbf{1}_{D'}) =$ $(\lambda x.\ \mathbf{1}_{D'})$
⟨*proof*⟩

The set of completion differential homomorphisms between two differential groups are a commutative group

lemma *hom-diff-groups-mult-comm-group*:
includes *diff-group* D + *diff-group* D'
shows *comm-group* (| *carrier* = *hom-diff* D D', *mult* = $\lambda f.\ \lambda g.\ (\lambda x.$ *if* $x \in$ *carrier* D *then* $f\ x \otimes_2\ g\ x$ *else* $\mathbf{1}_2)$,
one = $(\lambda x.$ *if* $x \in$ *carrier* D *then* $\mathbf{1}_2$ *else* $\mathbf{1}_2)$|)
(**is** *comm-group ?H-DI*)
⟨*proof*⟩

The following result has been already proved in *comm-group* (|*carrier* = *hom-diff* D D, *mult* = $\lambda f\ g\ x.$ *if* $x \in$ *carrier* D *then* $f\ x \otimes g\ x$ *else* $\mathbf{1}$, *one* = $\lambda x.$ *if* $x \in$ *carrier* D *then* $\mathbf{1}$ *else* $\mathbf{1}$|); now that we have provided a proof of a similar result but for two different differential groups, D and D', it can be trivially proved for the case $D = D'$

lemma (**in** *diff-group*) *hom-diff-group-mult-comm-group-inst*:
shows *comm-group* (| *carrier* = *hom-diff* D D, *mult* = $\lambda f.\ \lambda g.\ (\lambda x.$ *if* $x \in$ *carrier* D *then* $f\ x \otimes\ g\ x$ *else* $\mathbf{1})$,
one = $(\lambda x.$ *if* $x \in$ *carrier* D *then* $\mathbf{1}$ *else* $\mathbf{1})$|)
⟨*proof*⟩

end

3 Previous definitions and Propositions 2.2.9, 2.2.10 and Lemma 2.2.11 in Aransay's memoir

theory *lemma-2-2-11*
imports
~~/src/HOL/Algebra/Coset
HomGroupsCompletion
begin

Definitions and results leading to prove that the *ker* and *image* sets of a given homomorphism are subgroups and give place to suitable algebraic structures

locale *comm-group-hom* = *group-hom* +
assumes *comm-group-G*: *comm-group G*
and *comm-group-H*: *comm-group H*
and *hom-completion-h*: $h \in$ *completion-fun2 G H*

lemma *comm-group-hom* [*intro*]: **assumes** *G*: *comm-group G* **and** *H*: *comm-group H* **and** *h*: $h \in$ *hom-completion G H*
shows *comm-group-hom G H h*
⟨*proof*⟩

lemma (**in** *comm-group-hom*) *subgroup-kernel*: *subgroup* (*kernel G H h*) *G*
⟨*proof*⟩

lemma (**in** *comm-group-hom*) *kernel-comm-group*: *comm-group* (| *carrier* = (*kernel G H h*), *mult* = *mult G*, *one* = *one G*|)
⟨*proof*⟩

locale *diff-group-hom-diff* = *comm-group-hom D C h* +
assumes *diff-group-axioms-D*: *diff-group-axioms D*
and *diff-group-axioms-C*: *diff-group-axioms C*
and *diff-hom-h*: $h \circ \mathit{differ}_D = \mathit{differ}_C \circ h$

lemma *diff-group-hom-diffI*: **assumes** *d-g-D*: *diff-group D* **and** *d-g-C*: *diff-group C* **and** *h-hom*: $h \in$ *hom-diff D C*
shows *diff-group-hom-diff D C h*
⟨*proof*⟩

lemma (**in** *diff-group-hom-diff*) *diff-group-D*: **shows** *diff-group D*
⟨*proof*⟩

lemma (**in** *diff-group-hom-diff*) *diff-group-C*: **shows** *diff-group C*
⟨*proof*⟩

lemma (**in** *diff-group-hom-diff*) *hom-diff-h*: **shows** $h \in$ *hom-diff D C*
⟨*proof*⟩

lemma (**in** *diff-group-hom-diff*) *group-hom-D-D-differ*: **shows** *group-hom D D* (differ_D)
⟨*proof*⟩

lemma (**in** *diff-group-hom-diff*) *group-hom-C-C-differ*: **shows** *group-hom C C* (differ_C)
⟨*proof*⟩

lemma (**in** *diff-group-hom-diff*) *subgroup-kernel*: *subgroup* (*kernel D C h*) *D*
⟨*proof*⟩

The following lemma corresponds to Proposition 2.2.9 in Aransay's thesis

Due to the use of completion functions for the differential, we need to define the *diff* function, which originally was a completion from D into D, as a completion from the kernel into the original differential group D

lemma (**in** *diff-group-hom-diff*) *kernel-diff-group*:
  diff-group ⦇ *carrier* = (*kernel* D C h), *mult* = *mult* D, *one* = *one* D,
  diff = *completion* ⦇*carrier* = (*kernel* D C h), *mult* = *mult* D, *one* = *one* D, *diff* = *diff* D ⦈ D (*diff* D)⦈
  (**is** *diff-group* *?KER*)
⟨*proof*⟩

The following lemma corresponds to Proposition 2.2.10 in Aransay's thesis; here it is proved for a generic homomorphism h

lemma (**in** *diff-group-hom-diff*) *image-diff-group*:
  diff-group ⦇ *carrier* = *image* h (*carrier* D), *mult* = *mult* C, *one* = *one* C,
  diff = *completion* ⦇ *carrier* = *image* h (*carrier* D), *mult* = *mult* C, *one* = *one* C, *diff* = *diff* C⦈ C (*diff* C)⦈
  (**is** *diff-group* ⦇ *carrier* = *?img-set*, *mult* = *mult* C, *one* = *one* C, *diff* = *?compl* ⦈ **is** *diff-group* *?IMG*)
⟨*proof*⟩

Before proving Lemma 2.2.11, we first must introduce the definition of *reduction*

locale *reduction* = *diff-group* D + *diff-group* C + *var* f + *var* g + *var* h +
  assumes *f-hom-diff*: $f \in$ *hom-diff* D C
  and *g-hom-diff*: $g \in$ *hom-diff* C D
  and *h-hom-compl*: $h \in$ *hom-completion* D D
  and *fg*: $f \circ g = (\lambda x.\ \textit{if } x \in \textit{carrier } C \textit{ then id } x \textit{ else } \mathbf{1}_C)$
  and *gf-dh-hd*: $(\lambda x.\ \textit{if } x \in \textit{carrier } D \textit{ then } (g \circ f)\ x \otimes (\textit{if } x \in \textit{carrier } D \textit{ then } ((\textit{differ}) \circ h)\ x \otimes (h \circ (\textit{differ}))\ x \textit{ else } \mathbf{1}_D) \textit{ else } \mathbf{1}_D) =$
  $(\lambda x.\ \textit{if } x \in \textit{carrier } D \textit{ then id } x \textit{ else } \mathbf{1}_D)$
  and *fh*: $f \circ h = (\lambda x.\ \textit{if } x \in \textit{carrier } D \textit{ then } \mathbf{1}_C \textit{ else } \mathbf{1}_C)$
  and *hg*: $h \circ g = (\lambda x.\ \textit{if } x \in \textit{carrier } C \textit{ then } \mathbf{1}_D \textit{ else } \mathbf{1}_D)$
  and *hh*: $h \circ h = (\lambda x.\ \textit{if } x \in \textit{carrier } D \textit{ then } \mathbf{1}_D \textit{ else } \mathbf{1}_D)$

Due to the nature of the formula $(\lambda x.\ \textit{if } x \in \textit{carrier } D \textit{ then } (g \circ f)\ x \otimes (\textit{if } x \in \textit{carrier } D \textit{ then } (\textit{differ} \circ h)\ x \otimes (h \circ \textit{differ})\ x \textit{ else } \mathbf{1}) \textit{ else } \mathbf{1}) = (\lambda x.\ \textit{if } x \in \textit{carrier } D \textit{ then id } x \textit{ else } \mathbf{1})$, we associate first the addition of $d \circ h$ and $h \circ d$, and then $g \circ f$

lemma *reductionI*:
  includes *struct* D + *struct* C
  assumes *src-diff-group*: *diff-group* D
  and *trg-diff-group*: *diff-group* C
  assumes $f \in$ *hom-diff* D C
  and $g \in$ *hom-diff* C D
  and $h \in$ *hom-completion* D D
  and $f \circ g = (\lambda x.\ \textit{if } x \in \textit{carrier } C \textit{ then id } x \textit{ else } \mathbf{1}_C)$
  and $(\lambda x.\ \textit{if } x \in \textit{carrier } D \textit{ then } (g \circ f)\ x \otimes (\textit{if } x \in \textit{carrier } D \textit{ then } ((\textit{differ}) \circ h)\ x \otimes (h \circ (\textit{differ}))\ x \textit{ else } \mathbf{1}_D) \textit{ else } \mathbf{1}_D) =$

$(\lambda x.\ \textit{if } x \in \textit{carrier } D \textit{ then id } x \textit{ else } \mathbf{1}_D)$
and $f \circ h = (\lambda x.\ \textit{if } x \in \textit{carrier } D \textit{ then } \mathbf{1}_C \textit{ else } \mathbf{1}_C)$
and $h \circ g = (\lambda x.\ \textit{if } x \in \textit{carrier } C \textit{ then } \mathbf{1}_D \textit{ else } \mathbf{1}_D)$
and $h \circ h = (\lambda x.\ \textit{if } x \in \textit{carrier } D \textit{ then } \mathbf{1}_D \textit{ else } \mathbf{1}_D)$
shows *reduction D C f g h*
⟨*proof*⟩

lemma (**in** *reduction*) *C-diff-group*: **shows** *diff-group C* ⟨*proof*⟩

lemma (**in** *reduction*) *D-diff-group*: **shows** *diff-group D* ⟨*proof*⟩

lemma (**in** *reduction*) *D-C-f-diff-group-hom-diff*: **shows** *diff-group-hom-diff D C f* ⟨*proof*⟩

lemma (**in** *reduction*) *D-C-f-group-hom*: **shows** *group-hom D C f* ⟨*proof*⟩

lemma (**in** *reduction*) *C-D-g-diff-group-hom-diff*: **shows** *diff-group-hom-diff C D g* ⟨*proof*⟩

lemma (**in** *reduction*) *C-D-g-group-hom*: **shows** *group-hom C D g* ⟨*proof*⟩

3.1 Definition of isomorphic differential groups

Lemma 2.2.11, which corresponds to the first lemma in the BPL proof, has been already proved in our first approach.

It requires introducing first the notion of isomorphic differential groups; the definition is based on the one of isomorphic monoids presented in Group.thy for homomorphisms, by extending it to be coherent with the differentials.

constdefs
iso-diff :: $('a, 'c)$ *diff-group-scheme* => $('b, 'd)$ *diff-group-scheme* => $('a => 'b)$ *set* (**infixr** $\cong_{diff}$ *60*)
$D \cong_{diff} C == \{h.\ h \in \textit{hom-diff } D\ C\ \&\ \textit{bij-betw } h\ (\textit{carrier } D)\ (\textit{carrier } C)\}$

lemma *iso-diffI*: **assumes** *closed*: $\bigwedge x.\ x \in \textit{carrier } D \Longrightarrow h\ x \in \textit{carrier } C$
and *mult*: $\bigwedge x\ y.\ [\![x \in \textit{carrier } D;\ y \in \textit{carrier } D]\!] \Longrightarrow h\ (x \otimes_D y) = h\ x \otimes_C h\ y$
and *complect*: $\exists\, g.\ h = (\lambda x.\ \textit{if } x \in \textit{carrier } D \textit{ then } g\ x \textit{ else } \mathbf{1}_C)$
and *coherent*: $\bigwedge x.\ h\ ((\textit{differ}_D)\ x) = (\textit{differ}_C)\ (h\ x)$
and *inj-on*: $\bigwedge x\ y.\ [\![\ x \in \textit{carrier } D;\ \ y \in \textit{carrier } D;\ \ h\ (x) = h\ (y)\ ]\!] \Longrightarrow x{=}y$
and *image*: $\bigwedge y.\ y{\in}\textit{carrier } C \Longrightarrow \exists\, x \in \textit{carrier } D.\ y = h\ (x)$
shows $h \in D \cong_{diff} C$
⟨*proof*⟩

definition
iso-inv-diff :: $('a, 'c)$ *diff-group-scheme* => $('b, 'd)$ *diff-group-scheme* => $(('a => 'b) \times ('b => 'a))$ *set* (**infixr** $\cong_{invdiff}$ *60*)

where $D \cong_{invdiff} C == \{(f,\ g).\ f \in (D \cong_{diff} C)\ \&\ g \in (C \cong_{diff} D)\ \&\ (f \circ g = completion\ C\ C\ id)\ \&\ (g \circ f = completion\ D\ D\ id)\}$

lemma *iso-inv-diffI*: **assumes** *f*: $f \in (D \cong_{diff} C)$ **and** *g*: $g \in (C \cong_{diff} D)$ **and** *fg-id*: $(f \circ g = completion\ C\ C\ id)$
and *gf-id*: $(g \circ f = completion\ D\ D\ id)$ **shows** $(f,\ g) \in (D \cong_{invdiff} C)$
⟨*proof*⟩

lemma *iso-inv-diff-iso-diff*: **assumes** *f-f'*: $(f,\ f') \in (D \cong_{invdiff} C)$ **shows** $f \in (D \cong_{diff} C)$
⟨*proof*⟩

lemma *iso-inv-diff-iso-diff2*: **assumes** *f-f'*: $(f,\ f') \in (D \cong_{invdiff} C)$ **shows** $f' \in (C \cong_{diff} D)$
⟨*proof*⟩

lemma *iso-inv-diff-id*: **assumes** *f-f'*: $(f,\ f') \in (D \cong_{invdiff} C)$ **shows** $f' \circ f = completion\ D\ D\ id$
⟨*proof*⟩

lemma *iso-inv-diff-id2*: **assumes** *f-f'*: $(f,\ f') \in (D \cong_{invdiff} C)$ **shows** $f \circ f' = completion\ C\ C\ id$
⟨*proof*⟩

lemma *iso-inv-diff-rev*: **assumes** *f-f'*: $(f,\ f') \in (D \cong_{invdiff} C)$ **shows** $(f',\ f) \in (C \cong_{invdiff} D)$
⟨*proof*⟩

lemma *iso-diff-hom-diff*: **assumes** *h*: $h \in D \cong_{diff} C$ **shows** $h \in hom\text{-}diff\ D\ C$
⟨*proof*⟩

3.2 Previous facts for Lemma 2.2.11

lemma (**in** *reduction*) *g-f-hom-diff*: **shows** $g \circ f \in hom\text{-}diff\ D\ D$
⟨*proof*⟩

lemma (**in** *reduction*) *D-D-g-f-diff-group-hom-diff*: **shows** $diff\text{-}group\text{-}hom\text{-}diff\ D\ D\ (g \circ f)$ ⟨*proof*⟩

lemma (**in** *reduction*) *D-D-g-f-group-hom*: **shows** $group\text{-}hom\ D\ D\ (g \circ f)$ ⟨*proof*⟩

The following lemma proves that, in a general reduction, *f*, *g*, *h*, the set image of $g \circ f$ with the operations inherited from D is a differential group.

lemma (**in** *reduction*) *image-g-f-diff-group*: **shows** $diff\text{-}group\ (\!|carrier = image\ (g \circ f)\ (carrier\ D),\ mult = mult\ D,\ one = one\ D,$
$diff = completion\ (\!|carrier = image\ (g \circ f)\ (carrier\ D),\ mult = mult\ D,\ one = one\ D,\ diff = diff\ D\ |\!)\ D\ (diff\ D)\ |\!)$
⟨*proof*⟩

3.3 Lemma 2.2.11

The following lemmas correpond to Lemma 2.2.11 in Aransay's thesis

In the version in the thesis, two differential groups are defined to be isomorphic whenever there exists two homomorphisms f and g such that their composition is the identity in both directions

The Isabelle definition is slightly different, and it requires proving that there exists *one homomorphism*, which is, additionally, injective and surjective

This is the reason why the lemma is proved in Isabelle in four different lemmas; the first two, prove that the isomorphism exists, and then we prove that they are mutually inverse

We first introduce a locale which only contains some abbreviations, the main reason is to shorten proofs and statements

We will avoid the use of record update operations

locale *lemma-2-2-11* = *reduction D C f g h*

FIXME:Probably the following *definitions* would be more suitably stored as *abbreviations* or *notations*

context *lemma-2-2-11*
begin

definition *im-gf* **where** *im-gf* == *image* $(g \circ f)$ (*carrier D*)

definition *diff-group-im-gf* **where** *diff-group-im-gf* == (|*carrier* = *image* $(g \circ f)$ (*carrier D*), *mult* = *mult D*, *one* = *one D*,
diff =*completion* (|*carrier* = *image* $(g \circ f)$ (*carrier D*), *mult* = *mult D*, *one* = *one D*, *diff* = *diff D*|) *D* (*diff D*)|)

definition *diff-im-gf* **where** *diff-im-gf* == *completion diff-group-im-gf D* (*diff D*)

end

lemma (**in** *lemma-2-2-11*) *lemma-2-2-11-first-part*: **shows** $g \in (C \cong_{diff}$ *diff-group-im-gf*)
⟨*proof*⟩

The inverse of g is the restriction of f to the image set of $g \circ f$

lemma (**in** *lemma-2-2-11*) *lemma-2-2-11-second-part*: **shows** *completion diff-group-im-gf C f* $\in$ (*diff-group-im-gf* $\cong_{diff}$ *C*)
(**is** *?compl-f* $\in$ (*?IM* $\cong_{diff}$ *C*))
⟨*proof*⟩

We now prove that g and the restriction of f are inverse of each other.

lemma (**in** *lemma-2-2-11*) *lemma-2-2-11-third-part*: **shows** *completion diff-group-im-gf C f* $\circ\ g = (\lambda x.$ *if* $x \in$ *carrier C then id x else* $\mathbf{1}_C)$

(**is** *?compl-f* $\circ$ *g* = *?id-C*)
⟨*proof*⟩

lemma (**in** *lemma-2-2-11*) *lemma-2-2-11-fourth-part*:
shows *g* $\circ$ *completion diff-group-im-gf C f* = (λx. *if* $x \in$ *carrier diff-group-im-gf then id x else* $\mathbf{1}_{\textit{diff-group-im-gf}}$)
(**is** *g* $\circ$ *?compl-f* = *?id-IM*)
⟨*proof*⟩

The following is just the recollection of the four parts in which we have divided the proof of Lemma 2.2.11

The following statement should be compared to Lemma 2.2.11 in Aransay memoir

lemma (**in** *lemma-2-2-11*) *lemma-2-2-11*: **shows** (*g*, *completion diff-group-im-gf C f*) $\in$ (*C* $\cong_{\textit{invdiff}}$ *diff-group-im-gf*)
⟨*proof*⟩

end

4 Propositions 2.2.12, 2.2.13 and Lemma 2.2.14 in Aransay's memoir

theory *lemma-2-2-14*
imports
lemma-2-2-11
begin

4.1 Previous definitions for Lemma 2.2.14

In the following we introduce some locale specifications and definitions that will ease our proofs

For instance, we introduce the locale *ring-endomorphisms* which will allow us to apply equational reasoning with endomorphisms

In the *ring-endomorphisms* specification we introduce as an assumption the fact *ring-R*, stating that completion endomorphisms are a ring; we have proved this fact in the library *HomGroupCompletion.thy* and here it should be introduced by means of an *interpretation*, but some technical limitations in the interpretation mechanism led us to introduce this fact as an assumption

locale *ring-endomorphisms* = *diff-group D* + *ring R* +
assumes *ring-R*: *R* = (| *carrier* = *hom-completion D D*, *mult* = *op o*,
one = (λx. *if* $x \in$ *carrier D then id x else* $\mathbf{1}$),
zero = (λx. *if* $x \in$ *carrier D then* $\mathbf{1}$ *else* $\mathbf{1}$),
add = λf. λg. (λx. *if* $x \in$ *carrier D then f x* $\otimes$ *g x else* $\mathbf{1}$)|)

locale *lemma-2-2-14* = *ring-endomorphisms D R* + *var h* +
  assumes *h-hom*: $h \in$ *hom-completion D D*
  and *h-nil*: $h \otimes_R h = \mathbf{0}_R$
  and *hdh-h*: $h \otimes_R$ *differ* $\otimes_R h = h$

context *lemma-2-2-14*
begin

definition *p* **where** *p* == (*differ*) $\otimes_R$ *h* $\oplus_R$ *h* $\otimes_R$ (*differ*)

definition *ker-p* **where** *ker-p* == *kernel D D p*

definition *diff-group-ker-p* **where** *diff-group-ker-p* == (| *carrier* = *kernel D D p*, *mult* = *mult D*, *one* = *one D*,
  diff = *completion* (|*carrier* = *kernel D D p*, *mult* = *mult D*, *one* = *one D*, *diff* = *diff D*|) *D* (*diff D*)|)
definition *inc-ker-p* **where** *inc-ker-p* == (λx. *if* $x \in$ (*kernel D D p*) *then x else* $\mathbf{1}_D$)

end

lemma (**in** *ring-endomorphisms*) *D-diff-group*: **shows** *diff-group D* ⟨*proof*⟩

lemma (**in** *ring-endomorphisms*) *diff-in-R* [*simp*]: **shows** *differ* $\in$ *carrier R* ⟨*proof*⟩

lemma (**in** *lemma-2-2-14*) *h-in-R* [*simp*]: **shows** $h \in$ *carrier R* ⟨*proof*⟩

lemma (**in** *lemma-2-2-14*) *p-in-R* [*simp*]: **shows** $p \in$ *carrier R* ⟨*proof*⟩

lemma (**in** *ring-endomorphisms*) *diff-nilpot*[*simp*]: **shows** *differ* $\otimes_R$ *differ* = $\mathbf{0}_R$ ⟨*proof*⟩

4.2 Proposition 2.2.12

The following two lemmas correspond to Proposition 2.2.12 in Aransay's memoir

lemma (**in** *lemma-2-2-14*) *p-in-hom-diff*: **shows** $p \in$ *hom-diff D D*
⟨*proof*⟩

lemma (**in** *lemma-2-2-14*) *ker-p-diff-group*: *diff-group diff-group-ker-p*
  ⟨*proof*⟩

4.3 Proposition 2.2.13

The following lemma corresponds to Proposition 2.2.13 in Aransay's Ph.D.

lemma (**in** *ring-endomorphisms*) *image-subset*: **assumes** *p-in-R*: $p \in$ *carrier R* **and** *p-idemp*: $p \otimes_R p = p$

shows *image* $(\mathbf{1}_R \ominus_R p)$ $(carrier\ D) \subseteq kernel\ D\ D\ p$
$\langle proof \rangle$

lemma (**in** *group-hom*) *ker-m-closed*: **assumes** *x-in-ker*: $x \in kernel\ G\ H\ h$ **and** *y-in-ker*: $y \in kernel\ G\ H\ h$
shows $x \otimes y \in kernel\ G\ H\ h$
$\langle proof \rangle$

4.4 Lemma 2.2.14

The following lemma, proved in a generic ring, will help us to prove that $p = d \otimes_R h \oplus_R h \otimes_R d$ is a projector

lemma (**in** *ring*) *idemp-prod*: **assumes** *a*: $a \in carrier\ R$ **and** *b*: $b \in carrier\ R$ **and** *a-idemp*: $a \otimes a = a$ **and** *b-idemp*: $b \otimes b = b$
and *a-b*: $a \otimes b = \mathbf{0}$ **and** *b-a*: $b \otimes a = \mathbf{0}$ **shows** $(a \oplus b) \otimes (a \oplus b) = (a \oplus b)$
$\langle proof \rangle$

The following lemma corresponds to the first part of Lemma 2.2.14 as stated in Aransay's memoir

lemma (**in** *lemma-2-2-14*) *p-projector*: **shows** $p \otimes_R p = p$
$\langle proof \rangle$

lemma (**in** *abelian-group*) *minus-equality*:
$[|\ x \in carrier\ G;\ y \in carrier\ G;\ y \oplus x = \mathbf{0}\ |] ==> \ominus\ x = y$
$\langle proof \rangle$

lemma (**in** *abelian-monoid*) *minus-unique*:
$[|\ x \in carrier\ G;\ y \in carrier\ G;\ y' \in carrier\ G;\ y \oplus x = \mathbf{0};\ x \oplus y' = \mathbf{0}\ |] ==> y = y'$
$\langle proof \rangle$

When proving that R is a ring, you have to give an element such that it satisfies the condition of the additive inverse; nevertheless, when you really want to know the explicit expression of this inverse, there is no direct way to recover it. This makes a difference with the rest of constants and operations in a ring, such as the addition, the product, or the units.

This is the reason why we had to introduce the following lemma, giving us the expression of the additive inverse of any element a in R

lemma (**in** *ring-endomorphisms*) *minus-interpret*: **assumes** *a*: $a \in carrier\ R$
shows $(\ominus_R\ a) = (\lambda x.\ if\ x \in carrier\ D\ then\ inv_D\ (a\ x)\ else\ \mathbf{1}_D)$
$\langle proof \rangle$

The following proof is a nice example of how we can take advantage of reasoning with endomorphisms as elements of a ring, making use of the automatic tactics for this structure (*by algebra*, ...)

lemma (**in** *lemma-2-2-14*) *one-minus-p-hom-diff*: **shows** $\mathbf{1}_R \ominus_R p \in$ *hom-diff D D*
⟨*proof*⟩

The following lemma allows us to change the codomain of a homomorphism, whenever its image set is a subset of the new codomain

lemma (**in** *diff-group-hom-diff*) *h-image-hom-diff*: **assumes** *image-subset*: *image h (carrier D)* $\subseteq C'$
shows $h \in$ *hom-diff D* (| *carrier* = C', *mult* = *mult C*, *one* = *one C*,
diff = *completion* (|*carrier* = C', *mult* = *mult C*, *one* = *one C*, *diff* = *diff C*|) *C* (*diff C*)|)
⟨*proof*⟩

We denote as inclusion, *inc*, a homomorphism from a subgroup into a group such that it maps every element to the same element

lemma *inc-ker-hom-diff*: **includes** *diff-group D*
assumes *h-hom-diff*: $h \in$ *hom-diff D D*
shows (λx. *if* $x \in$ *kernel D D h then x else* $\mathbf{1}_D$) $\in$
hom-diff (|*carrier* = *kernel D D h*, *mult* = *mult D*, *one* = *one D*,
diff = *completion* (|*carrier* = *kernel D D h*, *mult* = *mult D*, *one* = *one D*, *diff* = *diff D*|) *D* (*diff D*)|) *D*
(**is** *?inc-KER* $\in$ *hom-diff ?KER -*)
⟨*proof*⟩

The following lemma corresponds to the second part of Lemma 2.2.14 in Aransay's memoir; we prove that a given triple of homomorphisms is a reduction

lemma (**in** *lemma-2-2-14*) *lemma-2-2-14*: **shows** *reduction D diff-group-ker-p* ($\mathbf{1}_R \ominus_R p$) *inc-ker-p h*
(**is** *reduction D ?KER* ($\mathbf{1}_R \ominus_R p$) *?inc-KER h*)
⟨*proof*⟩

end

5 Infinite Sets and Related Concepts

theory *Infinite-Set*
imports *ATP-Linkup*
begin

5.1 Infinite Sets

Some elementary facts about infinite sets, mostly by Stefan Merz. Beware! Because "infinite" merely abbreviates a negation, these lemmas may not work well with *blast*.

abbreviation
infinite :: *'a set* $\Rightarrow$ *bool* **where**

infinite S == $\neg$ *finite* S

Infinite sets are non-empty, and if we remove some elements from an infinite set, the result is still infinite.

lemma *infinite-imp-nonempty*: *infinite* S ==> $S \neq \{\}$
⟨*proof*⟩

lemma *infinite-remove*:
infinite $S \Longrightarrow$ *infinite* $(S - \{a\})$
⟨*proof*⟩

lemma *Diff-infinite-finite*:
assumes T: *finite* T **and** S: *infinite* S
shows *infinite* $(S - T)$
⟨*proof*⟩

lemma *Un-infinite*: *infinite* $S \Longrightarrow$ *infinite* $(S \cup T)$
⟨*proof*⟩

lemma *infinite-super*:
assumes T: $S \subseteq T$ **and** S: *infinite* S
shows *infinite* T
⟨*proof*⟩

As a concrete example, we prove that the set of natural numbers is infinite.

lemma *finite-nat-bounded*:
assumes S: *finite* $(S{::}nat\ set)$
shows $\exists\, k.\ S \subseteq \{..{<}k\}$ (**is** $\exists\, k.$ *?bounded* $S\ k$)
⟨*proof*⟩

lemma *finite-nat-iff-bounded*:
finite $(S{::}nat\ set) = (\exists\, k.\ S \subseteq \{..{<}k\})$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *finite-nat-iff-bounded-le*:
finite $(S{::}nat\ set) = (\exists\, k.\ S \subseteq \{..k\})$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *infinite-nat-iff-unbounded*:
infinite $(S{::}nat\ set) = (\forall\, m.\ \exists\, n.\ m{<}n \wedge n{\in}S)$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *infinite-nat-iff-unbounded-le*:
infinite $(S{::}nat\ set) = (\forall\, m.\ \exists\, n.\ m{\leq}n \wedge n{\in}S)$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

For a set of natural numbers to be infinite, it is enough to know that for any number larger than some k, there is some larger number that is an element of the set.

lemma *unbounded-k-infinite*:
  assumes k: $\forall m.\ k<m \longrightarrow (\exists n.\ m<n \wedge n\in S)$
  shows *infinite* $(S::nat\ set)$
⟨*proof*⟩

lemma *nat-infinite* [*simp*]: *infinite* $(UNIV :: nat\ set)$
  ⟨*proof*⟩

lemma *nat-not-finite* [*elim*]: *finite* $(UNIV::nat\ set) \Longrightarrow R$
  ⟨*proof*⟩

Every infinite set contains a countable subset. More precisely we show that a set S is infinite if and only if there exists an injective function from the naturals into S.

lemma *range-inj-infinite*:
  inj $(f::nat \Rightarrow 'a) \Longrightarrow$ *infinite* $(range\ f)$
⟨*proof*⟩

lemma *int-infinite* [*simp*]:
  shows *infinite* $(UNIV::int\ set)$
⟨*proof*⟩

The "only if" direction is harder because it requires the construction of a sequence of pairwise different elements of an infinite set S. The idea is to construct a sequence of non-empty and infinite subsets of S obtained by successively removing elements of S.

lemma *linorder-injI*:
  assumes *hyp*: $!!x\ y.\ x < (y::'a::linorder) ==> f\ x \neq f\ y$
  shows *inj f*
⟨*proof*⟩

lemma *infinite-countable-subset*:
  assumes *inf*: *infinite* $(S::'a\ set)$
  shows $\exists f.\ inj\ (f::nat \Rightarrow 'a) \wedge range\ f \subseteq S$
⟨*proof*⟩

lemma *infinite-iff-countable-subset*:
    infinite $S = (\exists f.\ inj\ (f::nat \Rightarrow 'a) \wedge range\ f \subseteq S)$
  ⟨*proof*⟩

For any function with infinite domain and finite range there is some element that is the image of infinitely many domain elements. In particular, any infinite sequence of elements from a finite set contains some element that occurs infinitely often.

lemma *inf-img-fin-dom*:

assumes *img*: *finite* $(f`A)$ **and** *dom*: *infinite* A
shows $\exists\, y \in f`A.$ *infinite* $(f\ -`\ \{y\})$
$\langle proof\rangle$

lemma *inf-img-fin-domE*:
assumes *finite* $(f`A)$ **and** *infinite* A
obtains y **where** $y \in f`A$ **and** *infinite* $(f\ -`\ \{y\})$
$\langle proof\rangle$

5.2 Infinitely Many and Almost All

We often need to reason about the existence of infinitely many (resp., all but finitely many) objects satisfying some predicate, so we introduce corresponding binders and their proof rules.

definition
Inf-many :: $('a \Rightarrow bool) \Rightarrow bool$ (**binder** *INFM* *10*) **where**
Inf-many P = *infinite* $\{x.\ P\ x\}$

definition
Alm-all :: $('a \Rightarrow bool) \Rightarrow bool$ (**binder** *MOST* *10*) **where**
Alm-all $P = (\neg\ (INFM\ x.\ \neg\ P\ x))$

notation (*xsymbols*)
Inf-many (**binder** $\exists_\infty$ *10*) **and**
Alm-all (**binder** $\forall_\infty$ *10*)

notation (*HTML* **output**)
Inf-many (**binder** $\exists_\infty$ *10*) **and**
Alm-all (**binder** $\forall_\infty$ *10*)

lemma *INF-EX*:
$(\exists_\infty x.\ P\ x) \Longrightarrow (\exists\, x.\ P\ x)$
$\langle proof\rangle$

lemma *MOST-iff-finiteNeg*: $(\forall_\infty x.\ P\ x) =$ *finite* $\{x.\ \neg\ P\ x\}$
$\langle proof\rangle$

lemma *ALL-MOST*: $\forall\, x.\ P\ x \Longrightarrow \forall_\infty x.\ P\ x$
$\langle proof\rangle$

lemma *INF-mono*:
assumes *inf*: $\exists_\infty x.\ P\ x$ **and** *q*: $\bigwedge x.\ P\ x \Longrightarrow Q\ x$
shows $\exists_\infty x.\ Q\ x$
$\langle proof\rangle$

lemma *MOST-mono*: $\forall_\infty x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow \forall_\infty x.\ Q\ x$
$\langle proof\rangle$

lemma *INF-nat*: $(\exists_\infty n.\ P\ (n{::}nat)) = (\forall\, m.\ \exists\, n.\ m{<}n \wedge P\ n)$

$\langle proof \rangle$

lemma *INF-nat-le*: $(\exists_\infty n.\ P\ (n::nat)) = (\forall m.\ \exists n.\ m \le n \wedge P\ n)$
$\langle proof \rangle$

lemma *MOST-nat*: $(\forall_\infty n.\ P\ (n::nat)) = (\exists m.\ \forall n.\ m < n \longrightarrow P\ n)$
$\langle proof \rangle$

lemma *MOST-nat-le*: $(\forall_\infty n.\ P\ (n::nat)) = (\exists m.\ \forall n.\ m \le n \longrightarrow P\ n)$
$\langle proof \rangle$

5.3 Enumeration of an Infinite Set

The set's element type must be wellordered (e.g. the natural numbers).

consts
enumerate $::\ 'a::wellorder\ set => (nat => 'a::wellorder)$
primrec
enumerate-0: $enumerate\ S\ 0 = (LEAST\ n.\ n \in S)$
enumerate-Suc: $enumerate\ S\ (Suc\ n) = enumerate\ (S - \{LEAST\ n.\ n \in S\})\ n$

lemma *enumerate-Suc'*:
$enumerate\ S\ (Suc\ n) = enumerate\ (S - \{enumerate\ S\ 0\})\ n$
$\langle proof \rangle$

lemma *enumerate-in-set*: $infinite\ S \Longrightarrow enumerate\ S\ n : S$
$\langle proof \rangle$

declare *enumerate-0* [*simp del*] *enumerate-Suc* [*simp del*]

lemma *enumerate-step*: $infinite\ S \Longrightarrow enumerate\ S\ n < enumerate\ S\ (Suc\ n)$
$\langle proof \rangle$

lemma *enumerate-mono*: $m<n \Longrightarrow infinite\ S \Longrightarrow enumerate\ S\ m < enumerate\ S\ n$
$\langle proof \rangle$

5.4 Miscellaneous

A few trivial lemmas about sets that contain at most one element. These simplify the reasoning about deterministic automata.

definition
atmost-one $::\ 'a\ set \Rightarrow bool$ **where**
$atmost\text{-}one\ S = (\forall x\ y.\ x \in S \wedge y \in S \longrightarrow x = y)$

lemma *atmost-one-empty*: $S = \{\} \Longrightarrow atmost\text{-}one\ S$
$\langle proof \rangle$

lemma *atmost-one-singleton*: $S = \{x\} \Longrightarrow atmost\text{-}one\ S$

⟨*proof*⟩

lemma *atmost-one-unique* [*elim*]: *atmost-one* $S \Longrightarrow x \in S \Longrightarrow y \in S \Longrightarrow y = x$
⟨*proof*⟩

end

6 Definition of local nilpotency and Lemmas 2.2.1 to 2.2.6 in Aransay's memoir

theory *analytic-part-local*
imports
lemma-2-2-14
~~/src/HOL/Library/Infinite-Set
begin

6.1 Definition of local nilpotent element and the bound function

locale *local-nilpotent-term* = *ring-endomorphisms D R* + *var a* + *var bound-funct* +
constrains *bound-funct* :: *'a => nat*
assumes *a-in-R*: $a \in$ *carrier R*
and *a-local-nilpot*: $\forall x \in$*carrier D*. $(a\ (\hat{\ })_R\ (\textit{bound-funct}\ x))\ x = \mathbf{1}_D$
and *bound-is-least*: *bound-funct* x =(*LEAST* n. $(a\ (\hat{\ })_R\ (n{::}\textit{nat}))\ x = \mathbf{1}_D$)

The following lemma maybe could be included in the *Group.thy* file; there is already a lemma called $\mathbf{1}\ (\hat{\ })\ ?n = \mathbf{1}$, about $\mathbf{1}$, but nothing about $x\ (\hat{\ })\ (1{::}'c)$

lemma (**in** *monoid*) *nat-pow-1*: **assumes** x: $x \in$ *carrier G* **shows** $x\ (\hat{\ })_G\ (1{::}\textit{nat}) = x$
⟨*proof*⟩

If the element a is nilpotent, with $(a\ (\hat{\ })_R\ \textit{bound}\ x)\ x = \mathbf{1}$, and *bound-funct* $x \leq m$, then $(a\ (\hat{\ })_R\ m)\ x = \mathbf{1}$

lemma (**in** *local-nilpotent-term*) *a-n-zero-a-m-zero*: **assumes** *bound-le-m*: *bound-funct* $x \leq m$
shows $(a(\hat{\ })_R(m))\ x = \mathbf{1}_D$
⟨*proof*⟩

The following definition is the power series of the local nilpotent endomorphism a in an element of its domain x; the power series is defined as the finite product in the differential group D of the powers $\lambda i.\ (a\ (\hat{\ })_R\ i)\ x$, up to *bound-funct* x

A different solution would be to consider the finite sum in the ring of endomorphisms R of terms *op* $(\hat{\ })_R\ a$ and then apply it to each element of the domain x

The first solution seems to me more coherent with the notion of "local nilpotency" we are dealing with, but both are identical

6.2 Definition of power series and some lemmas

context *local-nilpotent-term*
begin
definition *power-series* x == *finprod* D (λi::*nat*. (a(^)$_R$ i) x) {..*bound-funct* x}
end

Some results about the power series

lemma (**in** *local-nilpotent-term*) *power-pi*: (*op* (^)$_R$ a) $\in$ {..(k::*nat*)} $\rightarrow$ *carrier* R
⟨*proof*⟩

lemma (**in** *local-nilpotent-term*) *power-pi-D*: (λi::*nat*. (a(^)$_R$ i) x) $\in$ {..(k::*nat*)} $\rightarrow$ *carrier* D
⟨*proof*⟩

As we already stated, λx. $\bigotimes i \in$ {..j}. (a (^)$_R$ i) x is equal to *finsum* R (*op* (^)$_R$ a) {..j}

lemma (**in** *local-nilpotent-term*) *finprod-eq-finsum-bound-funct*:
shows *finprod* D (λi::*nat*. (a(^)$_R$ i) x) {..*bound-funct* x} = ((*finsum* R (λi::*nat*. (a(^)$_R$ i)) {..*bound-funct* x}) x)
⟨*proof*⟩

lemma (**in** *local-nilpotent-term*) *power-series-closed*: **shows** ($\bigotimes i \in$ {..m::*nat*}. (a (^)$_R$ i) x) $\in$ *carrier* D
⟨*proof*⟩

The following result is equal to the previous one but for the case of definition of the power series

lemma (**in** *local-nilpotent-term*) *power-series-closed2*: ($\bigotimes i \in$ {..*bound-funct* x}. (a (^)$_R$ i) x) $\in$ *carrier* D
⟨*proof*⟩

lemma (**in** *local-nilpotent-term*) *power-series-extended*: **assumes** *bf-le-m*: *bound-funct* $x \leq m$
shows *power-series* x = *finprod* D (λi::*nat*. (a (^)$_R$ i) x) {..m}
⟨*proof*⟩

The power series is itself an endomorphism of the differential group

lemma (**in** *local-nilpotent-term*) *power-series-in-R*: **shows** *power-series* $\in$ *carrier* R
⟨*proof*⟩

6.3 Some basic operations over finite series

Right distributivity of the product

lemma (**in** *ring*) *finsum-dist-r*: **assumes** *a-in-R*: $a \in$ *carrier R* **and** *b-in-R*: $b \in$ *carrier R*
shows $b \otimes$ *finsum R* (*op* $(\hat{\ })$ *a*) $\{..(m::nat)\} = (\bigoplus i \in \{..(m::nat)\}.\ b \otimes a\ (\hat{\ })\ i)$
⟨*proof*⟩

lemma (**in** *local-nilpotent-term*) *b-power-pi-D*: **assumes** *b-in-R*: $b \in$ *carrier R* **shows** $(\lambda i.\ b\ ((a\ (\hat{\ })_R\ i)\ x)) \in \{..(k::nat)\} \rightarrow$ *carrier D*
⟨*proof*⟩

lemma (**in** *local-nilpotent-term*) *nat-pow-closed-D*: **shows** $(a\ (\hat{\ })_R\ (m::nat))\ x \in$ *carrier D*
⟨*proof*⟩

Left distributivity of the product of a finite sum

lemma (**in** *local-nilpotent-term*) *power-series-dist-l*: **assumes** *b-in-R*: $b \in$ *carrier R*
shows $b\ (\bigotimes i \in \{..(m::nat)\}.\ (a\ (\hat{\ })_R\ i)\ x) = (\bigotimes i \in \{..(m::nat)\}.\ (b\ ((a\ (\hat{\ })_R\ i)\ x)))$
⟨*proof*⟩

lemma (**in** *local-nilpotent-term*) *power-pi-b-D*: **assumes** *b-in-R*: $b \in$ *carrier R* **shows** $(\lambda i.\ (a\ (\hat{\ })_R\ i)\ (b\ x)) \in \{..(k::nat)\} \rightarrow$ *carrier D*
⟨*proof*⟩

lemma (**in** *local-nilpotent-term*) *power-series-dist-r*: **assumes** *b-in-R*: $b \in$ *carrier R*
shows $(\lambda x.\ (\bigotimes i \in \{..m\}.\ (a\ (\hat{\ })_R\ i)\ x))\ (b\ x) = (\bigotimes i \in \{..(m::nat)\}.\ ((a\ (\hat{\ })_R\ i)\ (b\ x)))$ ⟨*proof*⟩

The following lemma showed to be useful in some situations

lemma (**in** *comm-monoid*) *finprod-singleton* [*simp*]:
$f \in \{i::nat\}$ −> *carrier G* ==> *finprod G f* $\{i\} = f\ i$ ⟨*proof*⟩

Finite series can be decomposed in the product of its first element and the remaining part

lemma (**in** *local-nilpotent-term*) *power-series-first-element*:
shows *finprod D* $(\lambda i::nat.\ (a\ (\hat{\ })_R\ i)\ x)\ \{..(i::nat)\} = (a(\hat{\ })_R\ (0::nat))\ x \otimes$ *finprod D* $(\lambda i::nat.\ (a\ (\hat{\ })_R\ i)\ x)\ \{1..(i::nat)\}$
⟨*proof*⟩

Finite series which start in index one can be seen as the product of the generic term and the finite series in index zero

lemma (**in** *local-nilpotent-term*) *power-series-factor*: **shows** $(\bigotimes j \in \{(1::nat)..Suc\ i\}.\ (a\ (\hat{\ })_R\ j)\ x) = a\ (\bigotimes j \in \{..i\}.\ (a\ (\hat{\ })_R\ j)\ x)$
⟨*proof*⟩

If we were able to interpret locales, now the idea would be to interpret the *locale nilpotent-term* with local nilpotent term α, as later defined in locale *alpha-beta*

6.4 Definition and some lemmas of perturbations

Perturbations are a homomorphism of D (not a differential homomorphism!) such that its addition with the differential is again a differential

constdefs (**structure** D)
  pert :: - => ('*a* => '*a*) *set*
  pert D == {δ. $\delta \in$ *hom-completion D D* &
  diff-group (| *carrier* = *carrier D*, *mult* = *mult D*, *one* = *one D*, *diff* = (λx. *if x* $\in$ *carrier D then* ((*differ*) *x*) $\otimes$ (δ *x*) *else* **1**)|) }

locale *diff-group-pert* = *diff-group D* + *var* δ +
  assumes *delta-pert*: $\delta \in$ *pert D*

lemma (**in** *diff-group-pert*) *diff-group-pert-is-diff-group*:
  shows *diff-group* (|*carrier* = *carrier D*, *mult* = *mult D*, *one* = *one D*, *diff* = (λx. *if x* $\in$ *carrier D then* ((*differ*$_D$) *x*) $\otimes_D$ (δ *x*) *else* $\mathbf{1}_D$)|)
  ⟨*proof*⟩

lemma (**in** *diff-group-pert*) *pert-is-hom*: **shows** $\delta \in$ *hom-completion D D*
  ⟨*proof*⟩

lemma (**in** *ring-endomorphisms*) *diff-group-pert-is-diff-group*: **assumes** *delta*: $\delta \in$ *pert D*
  shows *diff-group* (| *carrier* = *carrier D*, *mult* = *mult D*, *one* = *one D*, *diff* = (*differ*$_D$) $\oplus_R$ δ |)
  ⟨*proof*⟩

lemma (**in** *ring-endomorphisms*) *pert-in-R* [*simp*]: **assumes** *delta*: $\delta \in$ *pert D*
shows $\delta \in$ *carrier R*
  ⟨*proof*⟩

lemma (**in** *ring-endomorphisms*) *diff-pert-in-R* [*simp*]: **assumes** *delta*: $\delta \in$ *pert D*
shows (*differ*$_D$) $\oplus_R$ $\delta \in$ *carrier R*
  ⟨*proof*⟩

The reason to introduce α by means of a *defines* command is to get the expected behavior when merging this locale with locale *local-nilpotent-term D R* α *bound-phi* in the definition of locale *local-nilpotent-alpha*

locale *alpha-beta* = *ring-endomorphisms* + *reduction* + *var* δ + *var* α +
  assumes *delta-pert*: $\delta \in$ *pert D*
  defines *alpha-def*: α == $\ominus_R$ ($\delta \otimes_R h$)

context *alpha-beta*
begin

definition *beta-def*: $\beta = \ominus_R (h \otimes_R \delta)$

end

locale *local-nilpotent-alpha* = *alpha-beta* + *local-nilpotent-term D R* α *bound-phi*

The definition of Φ corresponds with the one given in the Basic Perturbation Lemma, Lemma 2.3.1 in Aransay's memoir

context *local-nilpotent-alpha*
begin

definition *phi-def*: Φ == *local-nilpotent-term.power-series D R* α *bound-phi*

end

lemma (**in** *alpha-beta*) *pert-in-R* [*simp*]: **shows** $\delta \in$ *carrier R*
⟨*proof*⟩

lemma (**in** *alpha-beta*) *h-in-R* [*simp*]: **shows** $h \in$ *carrier R*
⟨*proof*⟩

lemma (**in** *alpha-beta*) *alpha-in-R*: **shows** $\alpha \in$ *carrier R*
⟨*proof*⟩

lemma (**in** *alpha-beta*) *beta-in-R*: **shows** $\beta \in$ *carrier R*
⟨*proof*⟩

lemma (**in** *alpha-beta*) *alpha-i-in-R*: **shows** $\alpha(\hat{})_R$ (*i*::*nat*) $\in$ *carrier R*
⟨*proof*⟩

lemma (**in** *alpha-beta*) *beta-i-in-R*: **shows** $\beta(\hat{})_R$ (*i*::*nat*) $\in$ *carrier R*
⟨*proof*⟩

lemma (**in** *ring*) *power-minus-a-b*:
assumes *a*: $a \in$ *carrier R* **and** *b*: $b \in$ *carrier R* **shows** $(\ominus (a \otimes b)) (\hat{})$ *Suc n* $= \ominus a \otimes ((\ominus (b \otimes a)) (\hat{})\ n) \otimes b$
⟨*proof*⟩

The following comment is already obsolete in the *Isabelle-11−Feb−2007* repository version

Comment: At the moment, the "definition" command is not inherited by locales defined from old ones; in the following lemma, there would be two ways of recovering the definition of β. The first one would be to give its long name *local-nilpotent-alpha*.β δ, and the other way is to use abbreviations.

Due to aesthetic reasons, we choose the second solution, while waiting to the "definition" command to be properly inherited

abbreviation (*in local-nilpotent-alpha*) β == *alpha-beta*.β R h δ

The following lemma proves that whenever α is a local nilpotent term, so will be β

lemma (**in** *local-nilpotent-alpha*) *nilp-alpha-nilp-beta*: **shows** *local-nilpotent-term* D R β (λx. (*LEAST* n::*nat*. (β (^)$_R$ n) x = $\mathbf{1}_D$))
⟨*proof*⟩

lemma (**in** *local-nilpotent-alpha*) *bound-psi-exists*: **shows** $\exists$ *bound-psi*. *local-nilpotent-term* D R β *bound-psi*
⟨*proof*⟩

context *local-nilpotent-alpha*
begin

definition *bound-psi* $\equiv$ (λx. (*LEAST* n::*nat*. (β (^)$_R$ n) x = $\mathbf{1}_D$))

The definition of Ψ below is equivalent to the one given in the statement of Lemma 2.3.1 in Aransay's memoir

definition *psi-def*: $\Psi \equiv$ *local-nilpotent-term.power-series* D R β *bound-psi*

end

6.5 Some properties of the endomorphisms Φ, Ψ, α and β

lemma (**in** *local-nilpotent-alpha*) *local-nilpotent-term-alpha*: **shows** *local-nilpotent-term* D R α *bound-phi*
⟨*proof*⟩

lemma (**in** *local-nilpotent-alpha*) *local-nilpotent-term-beta*: **shows** *local-nilpotent-term* D R β *bound-psi*
⟨*proof*⟩

lemma (**in** *local-nilpotent-alpha*) *phi-x-in-D* [*simp*]: **shows** Φ $x \in$ *carrier* D
⟨*proof*⟩

lemma (**in** *local-nilpotent-alpha*) *phi-in-R* [*simp*]: **shows** $\Phi \in$ *carrier* R
⟨*proof*⟩

lemma (**in** *local-nilpotent-alpha*) *phi-in-hom*: **shows** $\Phi \in$ *hom-completion* D D
⟨*proof*⟩

lemma (**in** *local-nilpotent-alpha*) *psi-in-R* [*simp*]: **shows** $\Psi \in$ *carrier* R
⟨*proof*⟩

lemma (**in** *local-nilpotent-alpha*) *psi-in-hom*: **shows** $\Psi \in$ *hom-completion* D D

$\langle proof \rangle$

lemma (**in** *local-nilpotent-alpha*) *psi-x-in-D* [*simp*]: **shows** $\Psi\ x \in carrier\ D$
$\langle proof \rangle$

lemma (**in** *local-nilpotent-alpha*) *h-alpha-eq-beta-h*: $h \otimes_R \alpha(\hat{\ })_R(i{::}nat) = \beta\ (\hat{\ })_R\ i \otimes_R h$
$\langle proof \rangle$

6.6 Lemmas 2.2.1 to 2.2.6

Lemma 2.2.1

lemma (**in** *local-nilpotent-alpha*) *lemma-2-2-1*: **shows** *bound-psi* $(h\ x) \leq$ *bound-phi* x
$\langle proof \rangle$

Lemma 2.2.3 with endomorphisms applied to elements

lemma (**in** *local-nilpotent-alpha*) *lemma-2-2-3-elements*: **shows** $(h \circ \Phi)\ x = (\Psi \circ h)\ x$
$\langle proof \rangle$

Lemma 2.2.3 with endomorphisms

corollary (**in** *local-nilpotent-alpha*) *lemma-2-2-3*: **shows** $(h \circ \Phi) = (\Psi \circ h)$ $\langle proof \rangle$

The following lemma is simple a renaming of the previous one; the idea is to give to the previous result the name it had before as a premise, to keep the files correspoding to the equational part of the proof working

lemma (**in** *local-nilpotent-alpha*) *psi-h-h-phi*: **shows** $\Psi \otimes_R h = h \otimes_R \Phi$ $\langle proof \rangle$

lemma (**in** *local-nilpotent-alpha*) *alpha-delta-eq-delta-beta*: **shows** $\alpha(\hat{\ })_R(i{::}nat) \otimes_R \delta = \delta \otimes_R \beta\ (\hat{\ })_R\ i$
$\langle proof \rangle$

Lemma 2.2.2 in Aransay's memoir

lemma (**in** *local-nilpotent-alpha*) *lemma-2-2-2*: **shows** *bound-phi* $(\delta\ x) \leq$ *bound-psi* x
$\langle proof \rangle$

Lemma 2.2.4 over endomorphisms applied to generic elements

lemma (**in** *local-nilpotent-alpha*) *lemma-2-2-4-elements*: **shows** $(\delta \circ \Psi)\ x = (\Phi \circ \delta)\ x$
$\langle proof \rangle$

Lemma 2.2.4 over endomorphisms

corollary (**in** *local-nilpotent-alpha*) *lemma-2-2-4*: **shows** $(\delta \circ \Psi) = (\Phi \circ \delta)$ $\langle proof \rangle$

The following lemma is simple a renaming of the previous one; the idea is to give to the previous result the name it had before as a premise, to keep the files correspoding to the equational part of the proof working

lemma (**in** *local-nilpotent-alpha*) *delta-psi-phi-delta*: **shows** $\delta \otimes_R \Psi = \Phi \otimes_R \delta$
⟨*proof*⟩

Lemma 2.2.5 over a generic element of the domain

lemma (**in** *local-nilpotent-alpha*) *lemma-2-2-5-elements*: **shows** $\Psi\ x = (\mathbf{1}_R \ominus_R (h \otimes_R \delta \otimes_R \Psi))\ x$ **and** $\Psi\ x = (\mathbf{1}_R \ominus_R (h \otimes_R \Phi \otimes_R \delta))\ x$
and $\Psi\ x = (\mathbf{1}_R \ominus_R (\Psi \otimes_R h \otimes_R \delta))\ x$
⟨*proof*⟩

Lemma 2.2.5 in generic terms

lemma (**in** *local-nilpotent-alpha*) *lemma-2-2-5*: **shows** $\Psi = \mathbf{1}_R \ominus_R (h \otimes_R \delta \otimes_R \Psi)$ **and** $\Psi = \mathbf{1}_R \ominus_R (h \otimes_R \Phi \otimes_R \delta)$
and $\Psi = \mathbf{1}_R \ominus_R (\Psi \otimes_R h \otimes_R \delta)$ ⟨*proof*⟩

The following lemma is simple a renaming of the previous one; the idea is to give to the previous result the name it had before as a premise, to keep the proofs correspoding to the equational part of the proof working

lemma (**in** *local-nilpotent-alpha*) *psi-prop*: **shows** $\Psi = \mathbf{1}_R \ominus_R (h \otimes_R \delta \otimes_R \Psi)$ **and** $\Psi = \mathbf{1}_R \ominus_R (h \otimes_R \Phi \otimes_R \delta)$
and $\Psi = \mathbf{1}_R \ominus_R (\Psi \otimes_R h \otimes_R \delta)$ ⟨*proof*⟩

Lemma 2.2.6 over a generic element of the domain

lemma (**in** *local-nilpotent-alpha*) *lemma-2-2-6-elements*: **shows** $\Phi\ x = (\mathbf{1}_R \ominus_R (\delta \otimes_R h \otimes_R \Phi))\ x$ **and** $\Phi\ x = (\mathbf{1}_R \ominus_R (\delta \otimes_R \Psi \otimes_R h))\ x$
and $\Phi\ x = (\mathbf{1}_R \ominus_R (\Phi \otimes_R \delta \otimes_R h))\ x$
⟨*proof*⟩

Lemma 2.2.6

lemma (**in** *local-nilpotent-alpha*) *lemma-2-2-6*: **shows** $\Phi = (\mathbf{1}_R \ominus_R (\delta \otimes_R h \otimes_R \Phi))$ **and** $\Phi = (\mathbf{1}_R \ominus_R (\delta \otimes_R \Psi \otimes_R h))$
and $\Phi = (\mathbf{1}_R \ominus_R (\Phi \otimes_R \delta \otimes_R h))$ ⟨*proof*⟩

The following lemma is simple a renaming of the previous one; the idea is to give to the previous result the name it had before as a premise, to keep the proofs correspoding to the equational part of the proof working

lemma (**in** *local-nilpotent-alpha*) *phi-prop*: **shows** $\Phi = \mathbf{1}_R \ominus_R (\delta \otimes_R h \otimes_R \Phi)$ **and** $\Phi = \mathbf{1}_R \ominus_R (\delta \otimes_R \Psi \otimes_R h)$
and $\Phi = \mathbf{1}_R \ominus_R (\Phi \otimes_R \delta \otimes_R h)$ ⟨*proof*⟩

end

7 Lemma 2.2.15 in Aransay's memoir

theory *lemma-2-2-15-local-nilpot*

imports
analytic-part-local
begin

We define a locale setting merging the specifications introduced for *lemma 2.2.14* and also the one created for the *local nilpotent term alpha*

A few definitions are also provided in this locale setting

locale *lemma-2-2-15* = *lemma-2-2-14 D R h* + *local-nilpotent-alpha D R C f g h* δ α *bound-phi*

context *lemma-2-2-15*
begin

definition h' **where** $h' == h \otimes_R \Phi$

definition p' **where** $p' == ((\mathit{differ}_D) \oplus_R \delta) \otimes_R h' \oplus_R h' \otimes_R ((\mathit{differ}_D) \oplus_R \delta)$

definition diff' **where** $\mathit{diff}' == \mathit{differ} \oplus_R \delta$

definition D' **where** $D' == (\!| \mathit{carrier} = \mathit{carrier}\ D, \mathit{mult} = \mathit{mult}\ D, \mathit{one} = \mathit{one}\ D, \mathit{diff} = \mathit{differ} \oplus_R \delta |\!)$

definition $\mathit{ker}\text{-}p'$ **where** $\mathit{ker}\text{-}p' == \mathit{kernel}\ (\!| \mathit{carrier} = \mathit{carrier}\ D, \mathit{mult} = \mathit{mult}\ D, \mathit{one} = \mathit{one}\ D, \mathit{diff} = \mathit{differ} \oplus_R \delta |\!)$
$(\!| \mathit{carrier} = \mathit{carrier}\ D, \mathit{mult} = \mathit{mult}\ D, \mathit{one} = \mathit{one}\ D, \mathit{diff} = \mathit{differ} \oplus_R \delta |\!)\ p'$

definition $\mathit{diff}\text{-}\mathit{group}\text{-}\mathit{ker}\text{-}p'$
where $\mathit{diff}\text{-}\mathit{group}\text{-}\mathit{ker}\text{-}p' == (\!|\mathit{carrier} = \mathit{kernel}\ (\!| \mathit{carrier} = \mathit{carrier}\ D, \mathit{mult} = \mathit{mult}\ D, \mathit{one} = \mathit{one}\ D, \mathit{diff} = \mathit{differ} \oplus_R \delta |\!)$
$(\!| \mathit{carrier} = \mathit{carrier}\ D, \mathit{mult} = \mathit{mult}\ D, \mathit{one} = \mathit{one}\ D, \mathit{diff} = \mathit{differ} \oplus_R \delta |\!)\ p', \mathit{mult} = \mathit{mult}\ D,$
$\mathit{one} = \mathit{one}\ D, \mathit{diff} = \mathit{completion}\ ((\!|\mathit{carrier} = \mathit{kernel}\ (\!| \mathit{carrier} = \mathit{carrier}\ D, \mathit{mult} = \mathit{mult}\ D, \mathit{one} = \mathit{one}\ D, \mathit{diff} = \mathit{differ} \oplus_R \delta |\!)$
$(\!| \mathit{carrier} = \mathit{carrier}\ D, \mathit{mult} = \mathit{mult}\ D, \mathit{one} = \mathit{one}\ D, \mathit{diff} = \mathit{differ} \oplus_R \delta |\!)\ p', \mathit{mult} = \mathit{mult}\ D, \mathit{one} = \mathit{one}\ D, \mathit{diff} = \mathit{differ} \oplus_R \delta |\!))$
$(\!| \mathit{carrier} = \mathit{carrier}\ D, \mathit{mult} = \mathit{mult}\ D, \mathit{one} = \mathit{one}\ D, \mathit{diff} = \mathit{differ} \oplus_R \delta |\!)\ (\mathit{differ} \oplus_R \delta)\ |\!)$

definition $\mathit{inc}\text{-}\mathit{ker}\text{-}p'$ **where** $\mathit{inc}\text{-}\mathit{ker}\text{-}p' == (\lambda x.\ \mathit{if}\ x \in \mathit{kernel}\ (\!| \mathit{carrier} = \mathit{carrier}\ D, \mathit{mult} = \mathit{mult}\ D, \mathit{one} = \mathit{one}\ D, \mathit{diff} = \mathit{differ} \oplus_R \delta |\!)$
$(\!| \mathit{carrier} = \mathit{carrier}\ D, \mathit{mult} = \mathit{mult}\ D, \mathit{one} = \mathit{one}\ D, \mathit{diff} = \mathit{differ} \oplus_R \delta |\!)\ \ p'\ \mathit{then}\ x\ \mathit{else}\ \mathbf{1}_{D'})$

end

lemma (**in** *lemma-2-2-15*) h'-*in-R* [*simp*]: **shows** $h' \in \mathit{carrier}\ R\ \langle \mathit{proof} \rangle$

lemma (**in** *lemma-2-2-15*) *pert-in-R* [*simp*]: **shows** $\delta \in \mathit{carrier}\ R\ \langle \mathit{proof} \rangle$

lemma (**in** *lemma-2-2-15*) *p′-in-R* [*simp*]: **shows** $p' \in$ *carrier R* ⟨*proof*⟩

lemma (**in** *lemma-2-2-15*) *diff′-in-R* [*simp*]: **shows** *diff′* $\in$ *carrier R* ⟨*proof*⟩

The endomorphisms (not the differential endomorphisms) over a differential group happen to be the same ones as the homomorphisms over a perturbed version of this differential group

In other words, the definition of homomorphism over a differential group is independient of the differential

In the case of differential homomorphisms, this is not always true

lemma (**in** *ring-endomorphisms*) *hom-completion-eq*: **assumes** $\delta \in$ *pert D*
shows *hom-completion* (|*carrier* = *carrier D*, *mult* = *mult D*, *one* = *one D*, *diff* = *differ* $\oplus_R$ δ|)
(|*carrier* = *carrier D*, *mult* = *mult D*, *one* = *one D*, *diff* = *differ* $\oplus_R$ δ|) = *hom-completion D D*
⟨*proof*⟩

lemma (**in** *ring-endomorphisms*) *ring-endomorphisms-pert*: **assumes** *delta*: $\delta \in$ *pert D*
shows *ring-endomorphisms* (|*carrier* = *carrier D*, *mult* = *mult D*, *one* = *one D*, *diff* = *differ* $\oplus_R$ δ|) *R*
(**is** *ring-endomorphisms ?D′ R*)
⟨*proof*⟩

The two following lemmas prove that $h' \otimes_R h' = \mathbf{0}_R$ and $h' \otimes_R$ *diff′* $\otimes_R h' = h'$; these are the properties that will allow us to introduce *reduction D diff-group-ker-p* $(\mathbf{1}_R \ominus_R p)$ *inc-ker-p h* in order to define the reduction needed for *Lemma 2.2.15*

lemma (**in** *lemma-2-2-15*) *h′-nil*: **shows** $h' \otimes_R h' = \mathbf{0}_R$
⟨*proof*⟩

lemma (**in** *lemma-2-2-15*) *h′d′h′-h′*: **shows** $h' \otimes_R$ *diff′* $\otimes_R h' = h'$

⟨*proof*⟩

The following lemma is an instantiation of *lemma-2-2-14*, where D' = (|*carrier* = *carrier D*, *mult* = *op* $\otimes$, *one* = $\mathbf{1}$, *diff* = *differ* $\oplus_R$ δ|) $R = R$, and finally $h = h \otimes_R \Phi$.

Therefore, the premises of locale *lemma-2-2-14* have to be verified

It is not neccesary to explicitly prove that *diff-group-ker-p′* is a differential group, since it is one of the premises in the definition of reduction

lemma (**in** *lemma-2-2-15*) *lemma-2-2-15*: **shows** *reduction D′ diff-group-ker-p′* $(\mathbf{1}_R \ominus_R p')$ *inc-ker-p′ h′*

⟨*proof*⟩

end

8 Proposition 2.2.16 and Lemma 2.2.17 in Aransay's memoir

theory *lemma-2-2-17-local-nilpot*
  imports
  lemma-2-2-15-local-nilpot
begin

8.1 Previous definitions

Locale *proposition-2-2-16* does not introduce new facts; only some new definitions are given in the locale

locale *proposition-2-2-16* = *lemma-2-2-15*

context *proposition-2-2-16*
begin

definition π **where** $\pi = \mathbf{1}_R \ominus_R p$

definition π' **where** $\pi' = \mathbf{1}_R \ominus_R p'$

end

The following lemma has been extracted from the proof of *Proposition 2.2.16* as stated in the memoir

lemma (**in** *proposition-2-2-16*) *hp′-h* [*simp*]: **shows** $h \otimes_R p' = h$ **and** $p' \otimes_R h = h$

⟨*proof*⟩

Another rewriting step that will be later used

lemma (**in** *lemma-2-2-14*) *ph-h*[*simp*]: **shows** $p \otimes_R h = h$ **and** $h \otimes_R p = h$
⟨*proof*⟩

8.2 Proposition 2.2.16

The following lemma corresponds to the *Proposition 2.2.16* as stated in Aransay's memoir

The previous lemmas $h \otimes_R p' = h$
$p' \otimes_R h = h$ and $p \otimes_R h = h$
$h \otimes_R p = h$ are now used

lemma (**in** *proposition-2-2-16*) *proposition-2-2-16*[*simp*]:
shows *h-π'*: $h \otimes_R \pi' = \mathbf{0}_R$ **and** *π'-h*: $\pi' \otimes_R h = \mathbf{0}_R$ **and** *π-h'*: $\pi \otimes_R h' = \mathbf{0}_R$ **and** *h'-π*: $h' \otimes_R \pi = \mathbf{0}_R$
⟨*proof*⟩

lemma (**in** *proposition-2-2-16*) *p'-projector*: **shows** $p' \otimes_R p' = p'$

⟨*proof*⟩

The following lemmas *π-projector* and *π'-projector* correspond to one of the parts of the proof of Lemma 2.2.17, as stated in the memoir; here they have been extracted as independent results, because later they will be used to get some other results

lemma (**in** *proposition-2-2-16*) *π-in-R* [*simp*]: **shows** $\pi \in$ *carrier R* ⟨*proof*⟩

lemma (**in** *proposition-2-2-16*) *π'-in-R* [*simp*]: **shows** $\pi' \in$ *carrier R* ⟨*proof*⟩

lemma (**in** *proposition-2-2-16*) *π-projector*: **shows** $\pi \otimes_R \pi = \pi$
⟨*proof*⟩

lemma (**in** *proposition-2-2-16*) *π'-projector*: **shows** $\pi' \otimes_R \pi' = \pi'$
⟨*proof*⟩

8.3 Lemma 2.2.17

Lemma 2.2.17 proves the existence of an isomorphism between the differential subgroups *diff-group-im-π* and *diff-group-im-π'*

The isomorphism will be explicitly given

Lemma *im-π-ker-p* corresponds to the first part of the proof of Lemma 2.2.17 in Aransay's memoir; in this part, we prove both *im* π' = *kernel* D' D' p', where D' is the differential group perturbed, i.e., D' = ⦇*carrier* = *carrier* *D*, *mult* = *op* ⊗, *one* = **1**, *diff* = *differ* $\oplus_R$ δ⦈, and also *im* π = *kernel D D p*

The reason to prove these equalities between sets is that later, it will be easier to prove the existence of an isomorphism between *diff-group-im-π* and *diff-group-im-π'* than between the kernel sets

The two following proofs in lemma *im-π-ker-p* are quite similar, but maybe trying to extract the common parts and obtaining both goals just by instantiation of the obtained common lemma would have been even, at least, longer

lemma (**in** *proposition-2-2-16*) *im-π-ker-p*: **shows** *image* π (*carrier D*) = *kernel D D p* **and** *image* π' (*carrier* D') = *kernel* D' D' p'
⟨*proof*⟩

The following definition is similar to the one of isomorphism given in Isabelle, but here we add the premise that the homomorphism has to be also a completion. This is mainly to keep the coherence with the previous work

constdefs
  iso-compl :: - => - => ('*a* => '*b*) *set* (**infixr** $\cong_{compl}$ *60*)
  $D \cong_{compl} C$ == {*h*. *h* ∈ *hom-completion D C* & *bij-betw h* (*carrier D*) (*carrier C*)}

The following is an introduction lemma for isomorphisms between groups; maybe it could be introduced in the *Group.thy* file, avoiding the premise on completions!!

lemma *iso-complI*: **assumes** *closed*: $\bigwedge x.\ x \in carrier\ D \Longrightarrow h\ x \in carrier\ C$
  and *mult*: $\bigwedge x\ y.\ [\![x \in carrier\ D;\ y \in carrier\ D]\!] \Longrightarrow h\ (x \otimes_D y) = h\ x \otimes_C h\ y$
  and *complect*: $\exists\, g.\ h = (\lambda x.\ if\ x \in carrier\ D\ then\ g\ x\ else\ \mathbf{1}_C)$
  and *inj-on*: $\bigwedge x\ y.\ [\![\ x \in carrier\ D;\ \ y \in carrier\ D;\ \ h\ (x) = h\ (y)\ ]\!] \Longrightarrow x{=}y$
  and *image*: $\bigwedge y.\ y{\in}carrier\ C \Longrightarrow \exists\, x \in carrier\ D.\ y = h\ (x)$
  shows $h \in D \cong_{compl} C$
  ⟨*proof*⟩

Lemmas $\pi\pi'\pi$-π and $\pi'\pi\pi'$-π have been also extracted from the proof of *Lemma 2.2.17* as stated in the memoir

They are used in order to prove injectivity and surjection of π and π'

lemma (**in** *proposition-2-2-16*) $\pi\pi'\pi$-π: **shows** $\pi \otimes_R \pi' \otimes_R \pi = \pi$
⟨*proof*⟩

lemma (**in** *proposition-2-2-16*) $\pi'\pi\pi'$-π': **shows** $\pi' \otimes_R \pi \otimes_R \pi' = \pi'$
⟨*proof*⟩

The following locale definition only introduces some new definitions of constants; they will improve the presentation of the results

locale *lemma-2-2-17* = *proposition-2-2-16*

context *lemma-2-2-17*
begin

definition *im*-π **where** *im*-π == *image* π (*carrier D*)

definition *im*-π' **where** *im*-π' == *image* π' (*carrier D'*)

definition *diff-group-im*-π **where** *diff-group-im*-π == (|*carrier* = *image* π (*carrier D*), *mult* = *mult D*, *one* = *one D*,
  diff = *completion* (|*carrier* = *image* π (*carrier D*), *mult* = *mult D*, *one* = *one D*, *diff* = *diff D*|) *D* (*diff D*)|)

definition *diff-group-im*-π' **where** *diff-group-im*-π' == (|*carrier* = *image* π' (*carrier D'*), *mult* = *mult D*, *one* = *one D*,

$diff = completion$ $(\!|carrier = image\ \pi'\ (carrier\ D'),\ mult = mult\ D,\ one = one\ D,\ diff = (differ \oplus_R \delta)|\!)$
$(\!|carrier = carrier\ D,\ mult = mult\ D,\ one = one\ D,\ diff = (differ \oplus_R \delta)|\!)$ $(differ \oplus_R \delta)|\!)$

definition *diff-im-π-def*: $diff\text{-}im\text{-}\pi == completion$ $(\!|carrier = image\ \pi\ (carrier\ D),\ mult = mult\ D,\ one = one\ D,\ diff = diff\ D|\!)$ D $(diff\ D)$

definition *diff-im-π′-def*: $diff\text{-}im\text{-}\pi' == completion$ $(\!|carrier = image\ \pi'\ (carrier\ D'),\ mult = mult\ D,\ one = one\ D,$
$diff =(differ \oplus_R \delta)\ |\!)$ $(\!|\ carrier = carrier\ D,\ mult = mult\ D,\ one = one\ D,\ diff = differ \oplus_R \delta|\!)$ $(differ \oplus_R \delta)$

definition τ **where** $\tau == completion$
$(\!|\ carrier = image\ \pi\ (carrier\ D),\ mult = mult\ D,\ one = one\ D,$
$diff = completion$ $(\!|carrier = image\ \pi\ (carrier\ D),\ mult = mult\ D,\ one = one\ D,\ diff = diff\ D|\!)$ D $(diff\ D)|\!)$
$(\!|carrier = image\ \pi'\ (carrier\ D'),\ mult = mult\ D,\ one = one\ D,$
$diff = completion$ $(\!|carrier = image\ \pi'\ (carrier\ D'),\ mult = mult\ D,\ one = one\ D,\ diff = differ \oplus_R \delta|\!)$
$(\!|carrier = carrier\ D,\ mult = mult\ D,\ one = one\ D,\ diff = differ \oplus_R \delta|\!)$ $(differ \oplus_R \delta)\ |\!)$ π'

The following definition of τ' corresponds to the inverse of τ

definition
τ' **where** $\tau' == completion$
$(\!|carrier = image\ \pi'\ (carrier\ D'),\ mult = mult\ D,\ one = one\ D,$
$diff = completion$ $(\!|carrier = image\ \pi'\ (carrier\ D'),\ mult = mult\ D,\ one = one\ D,\ diff = differ \oplus_R \delta|\!)$
$(\!|carrier = carrier\ D,\ mult = mult\ D,\ one = one\ D,\ diff = differ \oplus_R \delta|\!)$ $(differ \oplus_R \delta)\ |\!)$
$(\!|\ carrier = image\ \pi\ (carrier\ D),\ mult = mult\ D,\ one = one\ D,$
$diff = completion$ $(\!|carrier = image\ \pi\ (carrier\ D),\ mult = mult\ D,\ one = one\ D,\ diff = diff\ D|\!)$ D $(diff\ D)|\!)$ π

end

As with *Lemma 2.2.14*, we divide the proof of *Lemma 2.2.17* in four parts. First we prove that there are two homomorphisms, one in each direction, satisfying that they are isomorphisms. Then, in other two lemmas, we prove that their compositions, also in both directions, are equal to the corresponding identities

lemma (**in** *lemma-2-2-17*) *lemma-2-2-17-first-part*: **shows** $\tau \in (diff\text{-}group\text{-}im\text{-}\pi \cong_{compl} diff\text{-}group\text{-}im\text{-}\pi')$
⟨*proof*⟩

lemma-2-2-17-second-part proves that $\tau' \in diff\text{-}group\text{-}im\text{-}\pi' \cong_{compl} diff\text{-}group\text{-}im\text{-}\pi$

lemma (**in** *lemma-2-2-17*) *lemma-2-2-17-second-part*: **shows** $\tau' \in (diff\text{-}group\text{-}im\text{-}\pi' \cong_{compl} diff\text{-}group\text{-}im\text{-}\pi)$

(**is** $\tau' \in$ (*?IM-*π' $\cong_{compl}$ *?IM-*π))
⟨*proof*⟩

In *lemma-2-2-17-first-part* and *lemma-2-2-17-second-part* we have proved the isomorphism between *diff-group-im-*π and *diff-group-im-*π'; now, with the help of π ' *carrier D* = *kernel D D p*
π' ' *carrier D'* = *kernel D' D' p'*, where we have proved both that *im* π = *ker p* and also that *im* π' = *ker p'*, we prove that *ker p* and *ker p'* are also isomorphic. Then we obtain the statement as it is presented in *Lemma 2.2.17* in Aransay's memoir

lemma (**in** *lemma-2-2-17*) *lemma-2-2-17-kernel*: **shows** $\tau \in$ (*diff-group-ker-p* $\cong_{compl}$ *diff-group-ker-p'*)
and $\tau' \in$ (*diff-group-ker-p'* $\cong_{compl}$ *diff-group-ker-p*)
⟨*proof*⟩

lemma (**in** *lemma-2-2-17*) *lemma-2-2-17-third-part*:
shows $\tau \circ \tau' = (\lambda x.$ *if* $x \in$ *carrier diff-group-im-*π' *then id x else* $\mathbf{1}_{diff\text{-}group\text{-}im\text{-}\pi'}$)
(**is** $\tau \circ \tau'$ = *?id-image-*π')
⟨*proof*⟩

lemma (**in** *lemma-2-2-17*) *lemma-2-2-17-fourth-part*:
shows $\tau' \circ \tau = (\lambda x.$ *if* $x \in$ *carrier diff-group-im-*π *then id x else* $\mathbf{1}_{diff\text{-}group\text{-}im\text{-}\pi}$)
(**is** $\tau' \circ \tau$ = *?id-image-*π)
⟨*proof*⟩

In the following lemma, again we transfer the result obtained in $\tau \circ \tau'$ = ($\lambda x.$ *if* $x \in$ *carrier diff-group-im-*π' *then id x else* $\mathbf{1}_{diff\text{-}group\text{-}im\text{-}\pi'}$) and $\tau' \circ \tau$ = ($\lambda x.$ *if* $x \in$ *carrier diff-group-im-*π *then id x else* $\mathbf{1}_{diff\text{-}group\text{-}im\text{-}\pi}$) from the image sets to the kernel sets

lemma (**in** *lemma-2-2-17*) *lemma-2-2-17-identities*: **shows** $\tau' \circ \tau = (\lambda x.$ *if* $x \in$ *carrier diff-group-ker-p then id x else* $\mathbf{1}_{diff\text{-}group\text{-}ker\text{-}p}$)
and $\tau \circ \tau' = (\lambda x.$ *if* $x \in$ *carrier diff-group-ker-p' then id x else* $\mathbf{1}_{diff\text{-}group\text{-}ker\text{-}p'}$)
⟨*proof*⟩

We now define what we consider inverse isomorphisms between differential groups (actually the definition also holds for monoids) by means of homomorphism

The previous definition, *op* $\cong_{invdiff}$, defined an isomorphism by means of *differential* homomorphisms

constdefs
iso-inv-compl :: ('*a*, '*c*) *monoid-scheme* => ('*b*, '*d*) *monoid-scheme* => (('*a* => '*b*) × ('*b* => '*a*)) *set* (**infixr** $\cong_{invcompl}$ *60*)

$D \cong_{invcompl} C == \{(f, g).\ f \in (D \cong_{compl} C)\ \&\ g \in (C \cong_{compl} D)\ \&\ (f \circ g = \mathit{completion}\ C\ C\ id)\ \&\ (g \circ f = \mathit{completion}\ D\ D\ id)\}$

lemma *iso-inv-complI*: **assumes** *f*: $f \in (D \cong_{compl} C)$ **and** *g*: $g \in (C \cong_{compl} D)$ **and** *fg-id*: $(f \circ g = \mathit{completion}\ C\ C\ id)$
and *gf-id*: $(g \circ f = \mathit{completion}\ D\ D\ id)$ **shows** $(f, g) \in (D \cong_{invcompl} C)$
⟨*proof*⟩

lemma *iso-inv-diff-impl-iso-inv-compl*: **assumes** *f-g*: $(f, g) \in (D \cong_{invdiff} C)$ **shows** $(f, g) \in (D \cong_{invcompl} C)$
⟨*proof*⟩

lemma *iso-inv-compl-iso-compl*: **assumes** *f-f'*: $(f, f') \in (D \cong_{invcompl} C)$ **shows** $f \in (D \cong_{compl} C)$
⟨*proof*⟩

lemma *iso-inv-compl-iso-compl2*: **assumes** *f-f'*: $(f, f') \in (D \cong_{invcompl} C)$ **shows** $f' \in (C \cong_{compl} D)$
⟨*proof*⟩

lemma *iso-inv-compl-id*: **assumes** *f-f'*: $(f, f') \in (D \cong_{invcompl} C)$ **shows** $f' \circ f = \mathit{completion}\ D\ D\ id$
⟨*proof*⟩

lemma *iso-inv-compl-id2*: **assumes** *f-f'*: $(f, f') \in (D \cong_{invcompl} C)$ **shows** $f \circ f' = \mathit{completion}\ C\ C\ id$
⟨*proof*⟩

lemma (**in** *lemma-2-2-17*) *lemma-2-2-17*: **shows** $(\tau, \tau') \in (\mathit{diff\text{-}group\text{-}ker\text{-}p} \cong_{invcompl} \mathit{diff\text{-}group\text{-}ker\text{-}p'})$
⟨*proof*⟩

end

9 Lemma 2.2.18 in Aransay's memoir

theory *lemma-2-2-18-local-nilpot*
imports
lemma-2-2-17-local-nilpot
begin

Lemma 2.2.18 is generic, in the sense that the previous definitions and premises from locales *lemma-2-2-11* to *lemma-2-2-17* are not needed. Only the notion of differential groups and isomorphism of abelian groups are introduced.

As far as we are in a generic setting, with homomorphisms instead of endomorphisms, the automation of the ring of endomorphisms is lost, and proofs become a bit more obscure

Composition of completions is again a completion

lemma *hom-completion-comp-closed*: **includes** *group* A + *group* B + *group* C
assumes f: $f \in$ *hom-completion* A B **and** g: $g \in$ *hom-completion* B C
shows $g \circ f \in$ *hom-completion* A C
⟨*proof*⟩

lemma *iso-inv-compl-coherent-iso-inv-diff*: **assumes** *fg*: $(f, g) \in (F \cong_{invcompl} G)$
and *f-coherent*: $f \circ$ *diff* F = *diff* $G \circ f$
and *g-coherent*: $g \circ$ *diff* G = *diff* $F \circ g$ **shows** $(f, g) \in (F \cong_{invdiff} G)$
⟨*proof*⟩

9.1 Lemma 2.2.18

The following lemma corresponds to *Lemma 2.2.18* in the memoir

It illustrates quite precisely the difficulties of proving facts about homomorphisms and endomorphisms when we loose the automation supplied in the previous lemmas

The difficulties are due to the neccesity of operating with endomorphisms and homomorphisms between different domains, A and B

A suitable environment would be the one defined by the ring *End* (A), the ring *End* (B), the commutative group *hom* (A, B) and the commutative group *hom* (A, B), but then the question would be how to supply this structure with any automation

In my opinion, the definition *comm-group* $?G \equiv$ *comm-monoid* $?G \wedge$ *group* $?G$ should be relaxed; in its actual version, when unfolded, the characterization *group* $G \wedge$ *comm-monoid* G is obtained, which unfolded again produces *group-axioms* $G \wedge$ *monoid* $G \wedge$ *comm-monoid-axioms* $G \wedge$ *monoid* G, which is redundant. Two possible solutions would be to define *comm-group* G = *group* $G \wedge$ *comm-monoid-axioms* G or also *comm-group* G = *group-axioms* $G \wedge$ *comm-monoid* G

lemma *lemma-2-2-18*: **assumes** A: *diff-group* A **and** B: *comm-group* B **and** F-F': $(F, F') \in (A \cong_{invcompl} B)$
shows *diff-group* (|*carrier* = *carrier* B, *mult* = *mult* B, *one* = *one* B, *diff* = $F \circ$ (*diff* A) $\circ F'$|)
(**is** *diff-group* $?B'$)
and $(F, F') \in (A \cong_{invdiff}$ (|*carrier* = *carrier* B, *mult* = *mult* B, *one* = *one* B, *diff* = $F \circ$ (*diff* A) $\circ F'$|))
(**is** - $\in A \cong_{invdiff} ?B'$)
⟨*proof*⟩

end

10 Lemma 2.2.19 in Aransay's memoir

theory *lemma-2-2-19-local-nilpot*
imports
lemma-2-2-18-local-nilpot
begin

Lemma 2.2.19, as well as *Lemma 2.2.18*, is generic in the sense that the previous definitions and premises from locales *lemma-2-2-11* to *lemma-2-2-17* are not needed. Only the definition of reduction is used

lemma (**in** *diff-group*) *diff-group-is-group*: **shows** *group D* ⟨*proof*⟩

lemma *hom-diffs-comp-closed*: **includes** *diff-group A* **includes** *diff-group B* **includes** *diff-group C*
assumes *f*: $f \in$ *hom-diff A B* **and** *g*: $g \in$ *hom-diff B C*
shows $g \circ f \in$ *hom-diff A C*
⟨*proof*⟩

10.1 Lemma 2.2.19

The following lemma corresponds to *Lemma 2.2.19* as stated in Aransay's memoir

lemma (**in** *reduction*) *lemma-2-2-19*: **assumes** *B*: *diff-group B* **and** *F-F′-isom*: $(F, F') \in (C \cong_{invdiff} B)$
shows *reduction D B* $(F \circ f)$ $(g \circ F')$ *h*
⟨*proof*⟩

end

11 Proof of the Basic Perturbation Lemma

theory *Basic-Perturbation-Lemma-local-nilpot*
imports
lemma-2-2-19-local-nilpot
begin

In the following locale we define an abbreviation that we will use later in proofs, and we also join the results obtained in locale *lemma-2-2-17* with the ones reached in *lemma-2-2-11.* The combination of both locales give us the set of premises in the Basic Perturbation Lemma (from now on, BPL) statement

locale *BPL* = *lemma-2-2-17* + *lemma-2-2-11*

context *BPL*
begin

definition f' **where** $f' ==$ (*completion*
(| *carrier* = *kernel D D p*, *mult* = *op* $\otimes$, *one* = $\mathbf{1}$,
diff = *completion* (| *carrier* = *kernel D D p*, *mult* = *op* $\otimes$, *one* = $\mathbf{1}$, *diff* = *differ* |)
D (*differ*) |) *C f*)

end

lemma (**in** *BPL*) π-*gf*: **shows** $g \circ f = \pi$
⟨*proof*⟩

11.1 BPL proof

The following lemma corresponds to the first part of *Lemma 2.2.20* (i.e., the BPL) in Aransay's memoir

The proof of the BPL is divided into two parts, as it is also in Aransay's memoir.

In the first part, proved in *BPL-reduction*, from the given premises, we buid a new reduction from $D' =$ (| *carrier* = *carrier D*, ..., *diff* = $\mathit{differ}_D \oplus_R \delta$ |) into C', where $C' =$ (| *carrier* = *carrier C*, ... , *diff* = $f' \circ (\tau' \circ \mathit{differ}_{\mathit{diff\text{-}group\text{-}ker\text{-}p'}} \circ \tau) \circ g$ |) $(f' \circ (\tau' \circ (\mathbf{1}_R \ominus_R p')))$

The reduction is given by the triple $f' \circ (\tau' \circ \mathbf{1}_R \ominus_R p')$, *inc-ker-p'* $\circ\ \tau \circ g$, h'

In the second part of the proof of the BPL, here stored in lemma *BPL-simplifications*, the expressions $f' \circ (\tau' \circ \mathbf{1}_R \ominus_R p')$, *inc-ker-p'* $\circ\ \tau \circ g$ and $f' \circ (\tau' \circ \mathit{differ}_{\mathit{diff\text{-}group\text{-}ker\text{-}p'}} \circ \tau) \circ g$ are simplified, obtaining the ones in the BPL statement

By finally joining *BPL-reduction* and *BPL-simplifications*, we complete the proof of the BPL

11.2 Existence of a reduction

lemma (**in** *BPL*) *BPL-reduction*:
shows *reduction* D'
(| *carrier* = *carrier C*, *mult* = *mult C*, *one* = *one C*, *diff* = $f' \circ (\tau' \circ \mathit{differ}_{\mathit{diff\text{-}group\text{-}ker\text{-}p'}} \circ \tau) \circ g$ |) $(f' \circ (\tau' \circ (\mathbf{1}_R \ominus_R p')))$
(*inc-ker-p'* $\circ\ \tau \circ g$) h'
⟨*proof*⟩

11.3 BPL previous simplifications

In order to prove the simplifications required in the second part of the proof, i.e. lemma *BPL-simplifications*, we first have to prove some results concerning the composition of some of the homomorphisms and endomorphisms we have already introduced.

Therefore, we have the ring R and we prove that it behaves as expected with some homomorphisms from *Hom* $(D\ C)$ and *Hom* $(C\ D)$, where the operation to relate them is the composition

We will prove some properties such as distributivity of composition with respect to addition of endomorphisms and the like

The results are stated in generic terms

lemma (**in** *ring-endomorphisms*) *add-dist-comp*: **assumes** C: *diff-group* C **and** g: $g \in$ *hom-completion* $C\ D$ **and** a: $a \in$ *carrier* R
and b: $b \in$ *carrier* R **shows** $(a \oplus_R b) \circ g = (\lambda x.$ *if* $x \in$ *carrier* C *then* $(a \circ g)\ x \otimes (b \circ g)\ x$ *else* $\mathbf{1})$
⟨*proof*⟩

lemma (**in** *ring-endomorphisms*) *comp-hom-compl*: **assumes** C: *diff-group* C **and** g: $g \in$ *hom-completion* $C\ D$ **and** a: $a \in$ *carrier* R
shows $a \circ g = (\lambda x.$ *if* $x \in$ *carrier* C *then* $(a \circ g)\ x$ *else* $\mathbf{1})$
⟨*proof*⟩

lemma (**in** *ring-endomorphisms*) *one-comp-g*: **assumes** C: *diff-group* C **and** g: $g \in$ *hom-completion* $C\ D$
shows $\mathbf{1}_R \circ g = g$
⟨*proof*⟩

lemma (**in** *ring-endomorphisms*) *minus-dist-comp*: **assumes** C: *diff-group* C **and** g: $g \in$ *hom-completion* $C\ D$ **and** a: $a \in$ *carrier* R
and b: $b \in$ *carrier* R **shows** $(a \ominus_R b) \circ g = (\lambda x.$ *if* $x \in$ *carrier* C *then* $(a \circ g)\ x \otimes ((\ominus_R b) \circ g)\ x$ *else* $\mathbf{1})$
⟨*proof*⟩

lemma (**in** *ring-endomorphisms*) *minus-comp-g*: **assumes** C: *diff-group* C **and** g: $g \in$ *hom-completion* $C\ D$ **and** a: $a \in$ *carrier* R
and b: $b \in$ *carrier* R **and** *a-eq-b*: $a = b$ **shows** $(\ominus_R a) \circ g = (\ominus_R b) \circ g$
⟨*proof*⟩

lemma (**in** *ring-endomorphisms*) *minus-comp-g2*: **assumes** C: *diff-group* C **and** g: $g \in$ *hom-completion* $C\ D$ **and** a: $a \in$ *carrier* R
and b: $b \in$ *carrier* R **and** *a-eq-b*: $a \circ g = b \circ g$ **shows** $(\ominus_R a) \circ g = (\ominus_R b) \circ g$
⟨*proof*⟩

lemma (**in** *ring-endomorphisms*) *l-add-dist-comp*: **includes** *diff-group* C **assumes** f: $f \in$ *hom-completion* $D\ C$ **and** a: $a \in$ *carrier* R
and b: $b \in$ *carrier* R **shows** $f \circ (a \oplus_R b) = (\lambda x.$ *if* $x \in$ *carrier* D *then* $(f \circ a)\ x \otimes_C (f \circ b)\ x$ *else* $\mathbf{1}_C)$
⟨*proof*⟩

lemma (**in** *ring-endomorphisms*) *l-comp-hom-compl*: **assumes** C: *diff-group* C **and** f: $f \in$ *hom-completion* D C **and** a: $a \in$ *carrier* R
shows $f \circ a = (\lambda x.\ \mathit{if}\ x \in \mathit{carrier}\ D\ \mathit{then}\ (f \circ a)\ x\ \mathit{else}\ \mathbf{1}_C)$
⟨*proof*⟩

lemma (**in** *ring-endomorphisms*) *l-minus-dist-comp*: **includes** *diff-group* C **assumes** f: $f \in$ *hom-completion* D C **and** a: $a \in$ *carrier* R
and b: $b \in$ *carrier* R **shows** $f \circ (a \ominus_R b) = (\lambda x.\ \mathit{if}\ x \in \mathit{carrier}\ D\ \mathit{then}\ (f \circ a)\ x \otimes_C (f \circ (\ominus_R\ b))\ x\ \mathit{else}\ \mathbf{1}_C)$
⟨*proof*⟩

lemma (**in** *ring-endomorphisms*) *l-minus-comp-f*: **assumes** C: *diff-group* C **and** f: $f \in$ *hom-completion* D C **and** a: $a \in$ *carrier* R
and b: $b \in$ *carrier* R **and** *a-eq-b*: $f \circ a = f \circ b$ **shows** $f \circ (\ominus_R\ a) = f \circ (\ominus_R\ b)$
⟨*proof*⟩

The following properties are used later in lemma *BPL-simplifications*; just in order to make the proof of *BPL-simplification* shorter, we have extracted them, as far as they are not generic properties that can be used in other different settings

lemma (**in** *BPL*) *inc-ker-pτ-eq-τ*: **shows** $\mathit{inc\text{-}ker\text{-}p}' \circ \tau = \tau$
⟨*proof*⟩

lemma (**in** *BPL*) *τg-eq-π′g*: **shows** $\tau \circ g = \pi' \circ g$
⟨*proof*⟩

lemma (**in** *BPL*) *diff′h′g-eq-zero*: **shows** $(\mathit{diff}' \otimes_R h') \circ g = (\lambda x.\ \mathit{if}\ x \in \mathit{carrier}\ C\ \mathit{then}\ \mathbf{1}\ \mathit{else}\ \mathbf{1})$
⟨*proof*⟩

lemma (**in** *BPL*) *h′diff′g-eq-psihdeltag*: **shows** $(h' \otimes_R \mathit{diff}') \circ g = (\Psi \otimes_R h \otimes_R \delta) \circ g$
⟨*proof*⟩

lemma (**in** *BPL*) *p′g-eq-psihdeltag*: **shows** $p' \circ g = (\Psi \otimes_R h \otimes_R \delta) \circ g$
⟨*proof*⟩

lemma (**in** *BPL*) *τ′π′-eq-ππ′*: **shows** $\tau' \circ \pi' = \pi \circ \pi'$
⟨*proof*⟩

lemma (**in** *BPL*) *f′π-eq-fπ*: **shows** $f' \circ \pi = f \circ \pi$
⟨*proof*⟩

lemma (**in** *BPL*) *fπ-eq-f*: **shows** $f \circ \pi = f$
⟨*proof*⟩

lemma (**in** *BPL*) *fh′diff′-eq-zero*: **shows** $f \circ (h' \otimes_R \mathit{diff}') = (\lambda x.\ \mathit{if}\ x \in \mathit{carrier}\ D\ \mathit{then}\ \mathbf{1}_C\ \mathit{else}\ \mathbf{1}_C)$
⟨*proof*⟩

lemma (**in** *BPL*) *fdiff′h′-eq-fdeltahphi*: **shows** $f \circ (\mathit{diff}' \otimes_R h') = f \circ \delta \otimes_R h \otimes_R \Phi$
⟨*proof*⟩

lemma (**in** *BPL*) *fp′-eq-fdeltahphi*: **shows** $f \circ p' = f \circ \delta \otimes_R h \otimes_R \Phi$
⟨*proof*⟩

lemma (**in** *BPL*) *diff-ker-p′π′-eq-diff′π′*: **shows** $\mathit{differ}_{\mathit{diff\text{-}group\text{-}ker\text{-}p}'} \circ \pi' = \mathit{diff}' \circ \pi'$
⟨*proof*⟩

lemma (**in** *BPL*) *τ′diff′-eq-πdiff′*: **shows** $\tau' \circ \mathit{differ}_{\mathit{diff\text{-}group\text{-}ker\text{-}p}'} = \pi \circ \mathit{differ}_{\mathit{diff\text{-}group\text{-}ker\text{-}p}'}$
⟨*proof*⟩

lemma (**in** *BPL*) *fπ′g-eq-id*: **shows** $f \circ \pi' \circ g = (\lambda x.\ \mathit{if}\ x \in \mathit{carrier}\ C\ \mathit{then}\ \mathit{id}\ x\ \mathit{else}\ \mathbf{1}_C)$
⟨*proof*⟩

lemma (**in** *BPL*) *π′g-eq-psig*: **shows** $\pi' \circ g = \Psi \circ g$
⟨*proof*⟩

11.4 BPL simplification

Now we can prove the simplifications of the terms in the reduction; these simplification processes correspond to the ones in pages 56 and 57 of Aransay's memoir

lemma (**in** *BPL*) *BPL-simplifications*: **shows** *f*: $(f' \circ (\tau' \circ \mathbf{1}_R \ominus_R p')) = f \circ \Phi$
and *g*: $(\mathit{inc\text{-}ker\text{-}p}' \circ \tau \circ g) = \Psi \circ g$
and *diff-C*: $f' \circ (\tau' \circ \mathit{differ}_{\mathit{diff\text{-}group\text{-}ker\text{-}p}'} \circ \tau) \circ g = (\lambda x.\ \mathit{if}\ x \in \mathit{carrier}\ C\ \mathit{then}\ (\mathit{differ}_C)\ x \otimes_C (f \circ \delta \circ \Psi \circ g)\ x\ \mathit{else}\ \mathbf{1}_C)$
⟨*proof*⟩

By joining $\mathit{reduction}\ D'\ (\!|\mathit{carrier} = \mathit{carrier}\ C,\ \mathit{mult} = \mathit{op} \otimes_C,\ \mathit{one} = \mathbf{1}_C,\ \mathit{diff} = f' \circ (\tau' \circ \mathit{differ}_{\mathit{diff\text{-}group\text{-}ker\text{-}p}'} \circ \tau) \circ g|\!)\ (f' \circ (\tau' \circ \mathbf{1}_R \ominus_R p'))\ (\mathit{inc\text{-}ker\text{-}p}' \circ \tau \circ g)\ h'$ and $f' \circ (\tau' \circ \mathbf{1}_R \ominus_R p') = f \circ \Phi$

$\mathit{inc\text{-}ker\text{-}p}' \circ \tau \circ g = \Psi \circ g$

$f' \circ (\tau' \circ \mathit{differ}_{\mathit{diff\text{-}group\text{-}ker\text{-}p}'} \circ \tau) \circ g = (\lambda x.\ \mathit{if}\ x \in \mathit{carrier}\ C\ \mathit{then}\ (\mathit{differ}_C)\ x \otimes_C (f \circ \delta \circ \Psi \circ g)\ x\ \mathit{else}\ \mathbf{1}_C)$ we get the proof of the BPL, stated as in *Lemma 2.2.20* in Aransay's memoir

lemma (**in** *BPL*) *BPL*: **shows** *reduction D′*

(| *carrier* = *carrier* C, *mult* = *mult* C, *one* = *one* C, *diff* = (λx. *if* $x \in$ *carrier* C *then* (*differ* $_C$) $x \otimes_C (f \circ \delta \circ \Psi \circ g)$ x *else* $\mathbf{1}_C$)|)
$(f \circ \Phi)$ $(\Psi \circ g)$ h'
⟨*proof*⟩

end

12 Definitions of Upper Bounds and Least Upper Bounds

theory *Lubs*
imports *Main*
begin

Thanks to suggestions by James Margetson

definition
setle :: ['*a set*, '*a*::*ord*] => *bool* (**infixl** *<= *70*) **where**
S *<= *x* = (*ALL* *y*: *S*. *y* <= *x*)

definition
setge :: ['*a*::*ord*, '*a set*] => *bool* (**infixl** <=* *70*) **where**
x <=* *S* = (*ALL* *y*: *S*. *x* <= *y*)

definition
leastP :: ['*a* =>*bool*,'*a*::*ord*] => *bool* **where**
leastP P x = (*P x* & *x* <=* *Collect P*)

definition
isUb :: ['*a set*, '*a set*, '*a*::*ord*] => *bool* **where**
isUb R S x = (*S* *<= *x* & *x*: *R*)

definition
isLub :: ['*a set*, '*a set*, '*a*::*ord*] => *bool* **where**
isLub R S x = *leastP* (*isUb R S*) *x*

definition
ubs :: ['*a set*, '*a*::*ord set*] => '*a set* **where**
ubs R S = *Collect* (*isUb R S*)

12.1 Rules for the Relations *<= and <=*

lemma *setleI*: *ALL* *y*: *S*. *y* <= *x* ==> *S* *<= *x*
⟨*proof*⟩

lemma *setleD*: [| *S* *<= *x*; *y*: *S* |] ==> *y* <= *x*
⟨*proof*⟩

lemma *setgeI*: *ALL* *y*: *S*. *x*<= *y* ==> *x* <=* *S*

⟨*proof*⟩

lemma *setgeD*: [| x <=* S; y: S |] ==> x <= y
⟨*proof*⟩

12.2 Rules about the Operators *leastP*, *ub* and *lub*

lemma *leastPD1*: *leastP* P x ==> P x
⟨*proof*⟩

lemma *leastPD2*: *leastP* P x ==> x <=* *Collect* P
⟨*proof*⟩

lemma *leastPD3*: [| *leastP* P x; y: *Collect* P |] ==> x <= y
⟨*proof*⟩

lemma *isLubD1*: *isLub* R S x ==> S *<= x
⟨*proof*⟩

lemma *isLubD1a*: *isLub* R S x ==> x: R
⟨*proof*⟩

lemma *isLub-isUb*: *isLub* R S x ==> *isUb* R S x
⟨*proof*⟩

lemma *isLubD2*: [| *isLub* R S x; y : S |] ==> y <= x
⟨*proof*⟩

lemma *isLubD3*: *isLub* R S x ==> *leastP*(*isUb* R S) x
⟨*proof*⟩

lemma *isLubI1*: *leastP*(*isUb* R S) x ==> *isLub* R S x
⟨*proof*⟩

lemma *isLubI2*: [| *isUb* R S x; x <=* *Collect* (*isUb* R S) |] ==> *isLub* R S x
⟨*proof*⟩

lemma *isUbD*: [| *isUb* R S x; y : S |] ==> y <= x
⟨*proof*⟩

lemma *isUbD2*: *isUb* R S x ==> S *<= x
⟨*proof*⟩

lemma *isUbD2a*: *isUb* R S x ==> x: R
⟨*proof*⟩

lemma *isUbI*: [| S *<= x; x: R |] ==> *isUb* R S x
⟨*proof*⟩

lemma *isLub-le-isUb*: [| *isLub R S x*; *isUb R S y* |] ==> *x* <= *y*
⟨*proof*⟩

lemma *isLub-ubs*: *isLub R S x* ==> *x* <=∗ *ubs R S*
⟨*proof*⟩

end

13 Abstract rational numbers

theory *Abstract-Rat*
imports *GCD*
begin

types $Num = int \times int$

abbreviation
  $Num0\text{-}syn :: Num\ (0_N)$
where $0_N \equiv (0,\ 0)$

abbreviation
  $Numi\text{-}syn :: int \Rightarrow Num\ (\text{-}_N)$
where $i_N \equiv (i,\ 1)$

definition
  $isnormNum :: Num \Rightarrow bool$
where
  $isnormNum = (\lambda(a,b).\ (if\ a = 0\ then\ b = 0\ else\ b > 0 \wedge igcd\ a\ b = 1))$

definition
  $normNum :: Num \Rightarrow Num$
where
  $normNum = (\lambda(a,b).\ (if\ a{=}0 \vee b = 0\ then\ (0,0)\ else$
  $(let\ g = igcd\ a\ b$
   $in\ if\ b > 0\ then\ (a\ div\ g,\ b\ div\ g)\ else\ (-\ (a\ div\ g),\ -\ (b\ div\ g)))))$

lemma *normNum-isnormNum* [*simp*]: *isnormNum* (*normNum x*)
⟨*proof*⟩

Arithmetic over Num

definition
  $Nadd :: Num \Rightarrow Num \Rightarrow Num$ (**infixl** $+_N$ *60*)
where
  $Nadd = (\lambda(a,b)\ (a',b').\ if\ a = 0 \vee b = 0\ then\ normNum(a',b')$
    $else\ if\ a'{=}0 \vee b' = 0\ then\ normNum(a,b)$
    $else\ normNum(a*b' + b*a',\ b*b'))$

definition

$Nmul$ :: $Num \Rightarrow Num \Rightarrow Num$ (**infixl** $*_N$ 60)
where
$Nmul = (\lambda(a,b)\ (a',b').\ let\ g = igcd\ (a*a')\ (b*b')$
$in\ (a*a'\ div\ g,\ b*b'\ div\ g))$

definition
$Nneg$ :: $Num \Rightarrow Num$ ($\sim_N$)
where
$Nneg \equiv (\lambda(a,b).\ (-a,b))$

definition
$Nsub$ :: $Num \Rightarrow Num \Rightarrow Num$ (**infixl** $-_N$ 60)
where
$Nsub = (\lambda a\ b.\ a +_N \sim_N b)$

definition
$Ninv$ :: $Num \Rightarrow Num$
where
$Ninv \equiv \lambda(a,b).\ if\ a < 0\ then\ (-b,\ |a|)\ else\ (b,a)$

definition
$Ndiv$ :: $Num \Rightarrow Num \Rightarrow Num$ (**infixl** $\div_N$ 60)
where
$Ndiv \equiv \lambda a\ b.\ a *_N Ninv\ b$

lemma *Nneg-normN*[*simp*]: $isnormNum\ x \Longrightarrow isnormNum\ (\sim_N x)$
⟨*proof*⟩
lemma *Nadd-normN*[*simp*]: $isnormNum\ (x +_N y)$
⟨*proof*⟩
lemma *Nsub-normN*[*simp*]: $[\![\ isnormNum\ y]\!] \Longrightarrow isnormNum\ (x -_N y)$
⟨*proof*⟩
lemma *Nmul-normN*[*simp*]: **assumes** *xn*:$isnormNum\ x$ **and** *yn*: $isnormNum\ y$
shows $isnormNum\ (x *_N y)$
⟨*proof*⟩

lemma *Ninv-normN*[*simp*]: $isnormNum\ x \Longrightarrow isnormNum\ (Ninv\ x)$
⟨*proof*⟩

lemma *isnormNum-int*[*simp*]:
$isnormNum\ 0_N\ isnormNum\ (1::int)_N\ i \neq 0 \Longrightarrow isnormNum\ i_N$
⟨*proof*⟩

Relations over Num

definition
$Nlt0$:: $Num \Rightarrow bool$ ($0>_N$)
where
$Nlt0 = (\lambda(a,b).\ a < 0)$

definition

Nle0:: *Num* ⇒ *bool* ($0 \geq_N$)
where
Nle0 = (λ(*a*,*b*). *a* ≤ *0*)

definition
Ngt0:: *Num* ⇒ *bool* ($0 <_N$)
where
Ngt0 = (λ(*a*,*b*). *a* > *0*)

definition
Nge0:: *Num* ⇒ *bool* ($0 \leq_N$)
where
Nge0 = (λ(*a*,*b*). *a* ≥ *0*)

definition
Nlt :: *Num* ⇒ *Num* ⇒ *bool* (**infix** $<_N$ *55*)
where
Nlt = (λ*a b*. $0 >_N$ ($a -_N b$))

definition
Nle :: *Num* ⇒ *Num* ⇒ *bool* (**infix** $\leq_N$ *55*)
where
Nle = (λ*a b*. $0 \geq_N$ ($a -_N b$))

definition
INum = (λ(*a*,*b*). *of-int a* / *of-int b*)

lemma *INum-int* [*simp*]: *INum* i_N = ((*of-int i*) ::′*a*::*field*) *INum* 0_N = (*0*::′*a*::*field*)
⟨*proof*⟩

lemma *isnormNum-unique*[*simp*]:
assumes *na*: *isnormNum x* **and** *nb*: *isnormNum y*
shows ((*INum x* ::′*a*::{*ring-char-0*,*field*, *division-by-zero*}) = *INum y*) = (*x* = *y*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *isnormNum0*[*simp*]: *isnormNum x* ⟹ (*INum x* = (*0*::′*a*::{*ring-char-0*, *field*,*division-by-zero*})) = (*x* = 0_N)
⟨*proof*⟩

lemma *of-int-div-aux*: *d* ~= *0* ==> ((*of-int x*)::′*a*::{*field*, *ring-char-0*}) / (*of-int d*) =
of-int (*x div d*) + (*of-int* (*x mod d*)) / ((*of-int d*)::′*a*)
⟨*proof*⟩

lemma *of-int-div*: (*d*::*int*) ~= *0* ==> *d dvd n* ==>
(*of-int*(*n div d*)::′*a*::{*field*, *ring-char-0*}) = *of-int n* / *of-int d*
⟨*proof*⟩

lemma *normNum*[*simp*]: *INum* (*normNum* x) = (*INum* x :: $'a$::{*ring-char-0*,*field*, *division-by-zero*})
⟨*proof*⟩

lemma *INum-normNum-iff*: (*INum* x ::$'a$::{*field*, *division-by-zero*, *ring-char-0*}) = *INum* y $\longleftrightarrow$ *normNum* x = *normNum* y (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *Nadd*[*simp*]: *INum* (x $+_N$ y) = *INum* x + (*INum* y :: $'a$ :: {*ring-char-0*,*division-by-zero*,*field*})
⟨*proof*⟩

lemma *Nmul*[*simp*]: *INum* (x $*_N$ y) = *INum* x * (*INum* y:: $'a$ :: {*ring-char-0*,*division-by-zero*,*field*})

⟨*proof*⟩

lemma *Nneg*[*simp*]: *INum* ($\sim_N$ x) = − (*INum* x ::$'a$:: *field*)
⟨*proof*⟩

lemma *Nsub*[*simp*]: **shows** *INum* (x $-_N$ y) = *INum* x − (*INum* y:: $'a$ :: {*ring-char-0*,*division-by-zero*,*field*})
⟨*proof*⟩

lemma *Ninv*[*simp*]: *INum* (*Ninv* x) = (*1*::$'a$ :: {*division-by-zero*,*field*}) / (*INum* x)
⟨*proof*⟩

lemma *Ndiv*[*simp*]: *INum* (x $\div_N$ y) = *INum* x / (*INum* y ::$'a$ :: {*ring-char-0*, *division-by-zero*,*field*}) ⟨*proof*⟩

lemma *Nlt0-iff*[*simp*]: **assumes** *nx*: *isnormNum* x
shows ((*INum* x :: $'a$ :: {*ring-char-0*,*division-by-zero*,*ordered-field*})< *0*) = *0*$>_N$ x
⟨*proof*⟩

lemma *Nle0-iff*[*simp*]:**assumes** *nx*: *isnormNum* x
shows ((*INum* x :: $'a$ :: {*ring-char-0*,*division-by-zero*,*ordered-field*}) ≤ *0*) = *0*$\geq_N$ x
⟨*proof*⟩

lemma *Ngt0-iff*[*simp*]:**assumes** *nx*: *isnormNum* x **shows** ((*INum* x :: $'a$ :: {*ring-char-0*,*division-by-zero*,*ordere*
0) = *0*$<_N$ x
⟨*proof*⟩
lemma *Nge0-iff*[*simp*]:**assumes** *nx*: *isnormNum* x
shows ((*INum* x :: $'a$ :: {*ring-char-0*,*division-by-zero*,*ordered-field*}) ≥ *0*) = *0*$\leq_N$ x
⟨*proof*⟩

lemma *Nlt-iff*[*simp*]: **assumes** *nx*: *isnormNum* x **and** *ny*: *isnormNum* y

shows $((INum\ x :: 'a :: \{ring\text{-}char\text{-}0, division\text{-}by\text{-}zero, ordered\text{-}field\}) < INum\ y)$ $= (x <_N y)$
$\langle proof \rangle$

lemma *Nle-iff* [*simp*]: **assumes** *nx*: *isnormNum x* **and** *ny*: *isnormNum y*
shows $((INum\ x :: 'a :: \{ring\text{-}char\text{-}0, division\text{-}by\text{-}zero, ordered\text{-}field\}) \leq INum\ y)$ $= (x \leq_N y)$
$\langle proof \rangle$

lemma *Nadd-commute*: $x +_N y = y +_N x$
$\langle proof \rangle$

lemma[*simp*]: $(0,\ b) +_N y = normNum\ y$ $(a,\ 0) +_N y = normNum\ y$
$x +_N (0,\ b) = normNum\ x$ $x +_N (a,\ 0) = normNum\ x$
$\langle proof \rangle$

lemma *normNum-nilpotent-aux* [*simp*]: **assumes** *nx*: *isnormNum x*
shows $normNum\ x = x$
$\langle proof \rangle$

lemma *normNum-nilpotent* [*simp*]: $normNum\ (normNum\ x) = normNum\ x$
$\langle proof \rangle$
lemma *normNum0* [*simp*]: $normNum\ (0,b) = 0_N$ $normNum\ (a,0) = 0_N$
$\langle proof \rangle$
lemma *normNum-Nadd*: $normNum\ (x +_N y) = x +_N y$ $\langle proof \rangle$
lemma *Nadd-normNum1* [*simp*]: $normNum\ x +_N y = x +_N y$
$\langle proof \rangle$
lemma *Nadd-normNum2* [*simp*]: $x +_N normNum\ y = x +_N y$
$\langle proof \rangle$

lemma *Nadd-assoc*: $x +_N y +_N z = x +_N (y +_N z)$
$\langle proof \rangle$

lemma *Nmul-commute*: $isnormNum\ x \Longrightarrow isnormNum\ y \Longrightarrow x *_N y = y *_N x$
$\langle proof \rangle$

lemma *Nmul-assoc*: **assumes** *nx*: *isnormNum x* **and** *ny*:*isnormNum y* **and** *nz*:*isnormNum z*
shows $x *_N y *_N z = x *_N (y *_N z)$
$\langle proof \rangle$

lemma *Nsub0*: **assumes** *x*: *isnormNum x* **and** *y*:*isnormNum y* **shows** $(x -_N y = 0_N) = (x = y)$
$\langle proof \rangle$

lemma *Nmul0* [*simp*]: $c *_N 0_N = 0_N$ $0_N *_N c = 0_N$
$\langle proof \rangle$

lemma *Nmul-eq0* [*simp*]: **assumes** *nx*:*isnormNum x* **and** *ny*: *isnormNum y*

shows $(x *_N\ y = 0_N) = (x = 0_N \lor y = 0_N)$
⟨*proof*⟩
lemma *Nneg-Nneg*[*simp*]: $\sim_N\ (\sim_N\ c) = c$
⟨*proof*⟩

lemma *Nmul1*[*simp*]:
isnormNum $c \Longrightarrow 1_N *_N\ c = c$
isnormNum $c \Longrightarrow c *_N\ 1_N\ = c$
⟨*proof*⟩

end

14 Rational numbers

theory *Rational*
imports ~~/*src*/*HOL*/*Library*/*Abstract-Rat*
uses (*rat-arith.ML*)
begin

14.1 Rational numbers

14.1.1 Equivalence of fractions

definition
fraction :: (*int* × *int*) *set* **where**
fraction = $\{x.\ snd\ x \neq 0\}$

definition
ratrel :: ((*int* × *int*) × (*int* × *int*)) *set* **where**
ratrel = $\{(x,y).\ snd\ x \neq 0 \land snd\ y \neq 0 \land fst\ x * snd\ y = fst\ y * snd\ x\}$

lemma *fraction-iff* [*simp*]: $(x \in fraction) = (snd\ x \neq 0)$
⟨*proof*⟩

lemma *ratrel-iff* [*simp*]:
$((x,y) \in ratrel) =$
$(snd\ x \neq 0 \land snd\ y \neq 0 \land fst\ x * snd\ y = fst\ y * snd\ x)$
⟨*proof*⟩

lemma *refl-ratrel*: *refl fraction ratrel*
⟨*proof*⟩

lemma *sym-ratrel*: *sym ratrel*
⟨*proof*⟩

lemma *trans-ratrel-lemma*:
assumes *1*: $a * b' = a' * b$
assumes *2*: $a' * b'' = a'' * b'$
assumes *3*: $b' \neq (0{::}int)$

shows $a * b'' = a'' * b$
⟨*proof*⟩

lemma *trans-ratrel*: *trans ratrel*
⟨*proof*⟩

lemma *equiv-ratrel*: *equiv fraction ratrel*
⟨*proof*⟩

lemmas *equiv-ratrel-iff* [*iff*] = *eq-equiv-class-iff* [*OF equiv-ratrel*]

lemma *equiv-ratrel-iff2*:
$[\![snd\ x \neq 0;\ snd\ y \neq 0]\!]$
$\Longrightarrow (ratrel\ ``\ \{x\} = ratrel\ ``\ \{y\}) = ((x,y) \in ratrel)$
⟨*proof*⟩

14.1.2 The type of rational numbers

typedef (*Rat*) *rat* = *fraction*//*ratrel*
⟨*proof*⟩

lemma *ratrel-in-Rat* [*simp*]: $snd\ x \neq 0 \Longrightarrow ratrel``\{x\} \in Rat$
⟨*proof*⟩

declare *Abs-Rat-inject* [*simp*] *Abs-Rat-inverse* [*simp*]

definition
Fract :: *int* ⇒ *int* ⇒ *rat* **where**
[*code func del*]: $Fract\ a\ b = Abs\text{-}Rat\ (ratrel``\{(a,b)\})$

lemma *Fract-zero*:
Fract k 0 = *Fract l 0*
⟨*proof*⟩

theorem *Rat-cases* [*case-names Fract*, *cases type*: *rat*]:
$(!!a\ b.\ q = Fract\ a\ b ==> b \neq 0 ==> C) ==> C$
⟨*proof*⟩

theorem *Rat-induct* [*case-names Fract*, *induct type*: *rat*]:
$(!!a\ b.\ b \neq 0 ==> P\ (Fract\ a\ b)) ==> P\ q$
⟨*proof*⟩

14.1.3 Congruence lemmas

lemma *add-congruent2*:
$(\lambda x\ y.\ ratrel``\{(fst\ x * snd\ y + fst\ y * snd\ x,\ snd\ x * snd\ y)\})$
respects2 ratrel
⟨*proof*⟩

lemma *minus-congruent*:
$(\lambda x.\ ratrel``\{(-\ fst\ x,\ snd\ x)\})$ *respects ratrel*
⟨*proof*⟩

lemma *mult-congruent2*:
$(\lambda x\ y.\ ratrel``\{(fst\ x * fst\ y,\ snd\ x * snd\ y)\})$ *respects2 ratrel*
⟨*proof*⟩

lemma *inverse-congruent*:
$(\lambda x.\ ratrel``\{if\ fst\ x{=}0\ then\ (0,1)\ else\ (snd\ x,\ fst\ x)\})$ *respects ratrel*
⟨*proof*⟩

lemma *le-congruent2*:
$(\lambda x\ y.\ \{(fst\ x * snd\ y){*}(snd\ x * snd\ y) \leq (fst\ y * snd\ x){*}(snd\ x * snd\ y)\})$
respects2 ratrel
⟨*proof*⟩

lemmas *UN-ratrel* = *UN-equiv-class* [*OF equiv-ratrel*]
lemmas *UN-ratrel2* = *UN-equiv-class2* [*OF equiv-ratrel equiv-ratrel*]

14.1.4 Standard operations on rational numbers

instantiation *rat* :: {*zero*, *one*, *plus*, *minus*, *uminus*, *times*, *inverse*, *ord*, *abs*, *sgn*}
begin

definition
Zero-rat-def [*code func del*]: $0 = Fract\ 0\ 1$

definition
One-rat-def [*code func del*]: $1 = Fract\ 1\ 1$

definition
add-rat-def [*code func del*]:
$q + r =$
$Abs\text{-}Rat\ (\bigcup x \in Rep\text{-}Rat\ q.\ \bigcup y \in Rep\text{-}Rat\ r.$
$ratrel``\{(fst\ x * snd\ y + fst\ y * snd\ x,\ snd\ x * snd\ y)\})$

definition
minus-rat-def [*code func del*]:
$-\ q = Abs\text{-}Rat\ (\bigcup x \in Rep\text{-}Rat\ q.\ ratrel``\{(-\ fst\ x,\ snd\ x)\})$

definition
diff-rat-def [*code func del*]: $q - r = q + -\ (r{::}rat)$

definition
mult-rat-def [*code func del*]:
$q * r =$
$Abs\text{-}Rat\ (\bigcup x \in Rep\text{-}Rat\ q.\ \bigcup y \in Rep\text{-}Rat\ r.$

$ratrel``\{(fst\ x * fst\ y,\ snd\ x * snd\ y)\})$

definition
inverse-rat-def [*code func del*]:
$inverse\ q =$
$Abs\text{-}Rat\ (\bigcup x \in Rep\text{-}Rat\ q.$
$ratrel``\{if\ fst\ x{=}0\ then\ (0,1)\ else\ (snd\ x,\ fst\ x)\})$

definition
divide-rat-def [*code func del*]: $q\ /\ r = q * inverse\ (r{::}rat)$

definition
le-rat-def [*code func del*]:
$q \le r \longleftrightarrow contents\ (\bigcup x \in Rep\text{-}Rat\ q.\ \bigcup y \in Rep\text{-}Rat\ r.$
$\{(fst\ x * snd\ y)*(snd\ x * snd\ y) \le (fst\ y * snd\ x)*(snd\ x * snd\ y)\})$

definition
less-rat-def [*code func del*]: $z < (w{::}rat) \longleftrightarrow z \le w \wedge z \ne w$

definition
abs-rat-def: $|q| = (if\ q < 0\ then\ -q\ else\ (q{::}rat))$

definition
sgn-rat-def: $sgn\ (q{::}rat) = (if\ q{=}0\ then\ 0\ else\ if\ 0{<}q\ then\ 1\ else\ -\ 1)$

instance ⟨*proof*⟩

end

instantiation *rat* :: *power*
begin

primrec *power-rat*
where
rat-power-0: $q\ \hat{}\ 0 = (1{::}rat)$
| *rat-power-Suc*: $q\ \hat{}\ (Suc\ n) = (q{::}rat) * (q\ \hat{}\ n)$

instance ⟨*proof*⟩

end

theorem *eq-rat*: $b \ne 0 ==> d \ne 0 ==>$
$(Fract\ a\ b = Fract\ c\ d) = (a * d = c * b)$
⟨*proof*⟩

theorem *add-rat*: $b \ne 0 ==> d \ne 0 ==>$
$Fract\ a\ b + Fract\ c\ d = Fract\ (a * d + c * b)\ (b * d)$
⟨*proof*⟩

theorem *minus-rat*: $b \neq 0 ==> -(Fract\ a\ b) = Fract\ (-a)\ b$
⟨*proof*⟩

theorem *diff-rat*: $b \neq 0 ==> d \neq 0 ==>$
$Fract\ a\ b - Fract\ c\ d = Fract\ (a * d - c * b)\ (b * d)$
⟨*proof*⟩

theorem *mult-rat*: $b \neq 0 ==> d \neq 0 ==>$
$Fract\ a\ b * Fract\ c\ d = Fract\ (a * c)\ (b * d)$
⟨*proof*⟩

theorem *inverse-rat*: $a \neq 0 ==> b \neq 0 ==>$
$inverse\ (Fract\ a\ b) = Fract\ b\ a$
⟨*proof*⟩

theorem *divide-rat*: $c \neq 0 ==> b \neq 0 ==> d \neq 0 ==>$
$Fract\ a\ b\ /\ Fract\ c\ d = Fract\ (a * d)\ (b * c)$
⟨*proof*⟩

theorem *le-rat*: $b \neq 0 ==> d \neq 0 ==>$
$(Fract\ a\ b \leq Fract\ c\ d) = ((a * d) * (b * d) \leq (c * b) * (b * d))$
⟨*proof*⟩

theorem *less-rat*: $b \neq 0 ==> d \neq 0 ==>$
$(Fract\ a\ b < Fract\ c\ d) = ((a * d) * (b * d) < (c * b) * (b * d))$
⟨*proof*⟩

theorem *abs-rat*: $b \neq 0 ==> |Fract\ a\ b| = Fract\ |a|\ |b|$
⟨*proof*⟩

14.1.5 The ordered field of rational numbers

instance *rat* :: *field*
⟨*proof*⟩

instance *rat* :: *linorder*
⟨*proof*⟩

instantiation *rat* :: *distrib-lattice*
begin

definition
$(inf :: rat \Rightarrow rat \Rightarrow rat) = min$

definition
$(sup :: rat \Rightarrow rat \Rightarrow rat) = max$

instance
⟨*proof*⟩

end

instance *rat* :: *ordered-field*
⟨*proof*⟩

instance *rat* :: *division-by-zero*
⟨*proof*⟩

instance *rat* :: *recpower*
⟨*proof*⟩

14.2 Various Other Results

lemma *minus-rat-cancel* [*simp*]: $b \neq 0 ==> Fract\ (-a)\ (-b) = Fract\ a\ b$
⟨*proof*⟩

theorem *Rat-induct-pos* [*case-names Fract*, *induct type*: *rat*]:
  assumes *step*: $!!a\ b.\ 0 < b ==> P\ (Fract\ a\ b)$
    shows $P\ q$
⟨*proof*⟩

lemma *zero-less-Fract-iff*:
   $0 < b ==> (0 < Fract\ a\ b) = (0 < a)$
⟨*proof*⟩

lemma *Fract-add-one*: $n \neq 0 ==> Fract\ (m + n)\ n = Fract\ m\ n + 1$
⟨*proof*⟩

lemma *of-nat-rat*: $of\text{-}nat\ k = Fract\ (of\text{-}nat\ k)\ 1$
⟨*proof*⟩

lemma *of-int-rat*: $of\text{-}int\ k = Fract\ k\ 1$
⟨*proof*⟩

lemma *Fract-of-nat-eq*: $Fract\ (of\text{-}nat\ k)\ 1 = of\text{-}nat\ k$
⟨*proof*⟩

lemma *Fract-of-int-eq*: $Fract\ k\ 1 = of\text{-}int\ k$
⟨*proof*⟩

lemma *Fract-of-int-quotient*: $Fract\ k\ l = (if\ l = 0\ then\ Fract\ 1\ 0\ else\ of\text{-}int\ k\ /\ of\text{-}int\ l)$
⟨*proof*⟩

14.3 Numerals and Arithmetic

instantiation *rat* :: *number-ring*
begin

definition
rat-number-of-def [*code func del*]: *number-of* w = (*of-int* w :: *rat*)

instance
⟨*proof*⟩

end

⟨*ML*⟩

14.4 Embedding from Rationals to other Fields

class *field-char-0* = *field* + *ring-char-0*

instance *ordered-field* < *field-char-0* ⟨*proof*⟩

definition
of-rat :: *rat* ⇒ *'a::field-char-0*
where
[*code func del*]: *of-rat* q = *contents* ($\bigcup$(*a*,*b*) ∈ *Rep-Rat* q. {*of-int* a / *of-int* b})

lemma *of-rat-congruent*:
(λ(*a*, *b*). {*of-int* a / *of-int* *b::'a::field-char-0*}) *respects ratrel*
⟨*proof*⟩

lemma *of-rat-rat*:
$b \neq 0 \Longrightarrow$ *of-rat* (*Fract* a b) = *of-int* a / *of-int* b
⟨*proof*⟩

lemma *of-rat-0* [*simp*]: *of-rat* 0 = 0
⟨*proof*⟩

lemma *of-rat-1* [*simp*]: *of-rat* 1 = 1
⟨*proof*⟩

lemma *of-rat-add*: *of-rat* (a + b) = *of-rat* a + *of-rat* b
⟨*proof*⟩

lemma *of-rat-minus*: *of-rat* (− a) = − *of-rat* a
⟨*proof*⟩

lemma *of-rat-diff*: *of-rat* (a − b) = *of-rat* a − *of-rat* b
⟨*proof*⟩

lemma *of-rat-mult*: *of-rat* (a ∗ b) = *of-rat* a ∗ *of-rat* b
⟨*proof*⟩

lemma *nonzero-of-rat-inverse*:
$a \neq 0 \Longrightarrow$ *of-rat* (*inverse* a) = *inverse* (*of-rat* a)

$\langle proof \rangle$

lemma *of-rat-inverse*:
$(\textit{of-rat}\ (\textit{inverse}\ a)::'a::\{\textit{field-char-0},\textit{division-by-zero}\}) =$
$\textit{inverse}\ (\textit{of-rat}\ a)$
$\langle proof \rangle$

lemma *nonzero-of-rat-divide*:
$b \neq 0 \Longrightarrow \textit{of-rat}\ (a\ /\ b) = \textit{of-rat}\ a\ /\ \textit{of-rat}\ b$
$\langle proof \rangle$

lemma *of-rat-divide*:
$(\textit{of-rat}\ (a\ /\ b)::'a::\{\textit{field-char-0},\textit{division-by-zero}\})$
$= \textit{of-rat}\ a\ /\ \textit{of-rat}\ b$
$\langle proof \rangle$

lemma *of-rat-power*:
$(\textit{of-rat}\ (a\ \hat{}\ n)::'a::\{\textit{field-char-0},\textit{recpower}\}) = \textit{of-rat}\ a\ \hat{}\ n$
$\langle proof \rangle$

lemma *of-rat-eq-iff* [*simp*]: $(\textit{of-rat}\ a = \textit{of-rat}\ b) = (a = b)$
$\langle proof \rangle$

lemmas *of-rat-eq-0-iff* [*simp*] = *of-rat-eq-iff* [*of - 0, simplified*]

lemma *of-rat-eq-id* [*simp*]: $\textit{of-rat} = (\textit{id} :: \textit{rat} \Rightarrow \textit{rat})$
$\langle proof \rangle$

Collapse nested embeddings

lemma *of-rat-of-nat-eq* [*simp*]: $\textit{of-rat}\ (\textit{of-nat}\ n) = \textit{of-nat}\ n$
$\langle proof \rangle$

lemma *of-rat-of-int-eq* [*simp*]: $\textit{of-rat}\ (\textit{of-int}\ z) = \textit{of-int}\ z$
$\langle proof \rangle$

lemma *of-rat-number-of-eq* [*simp*]:
$\textit{of-rat}\ (\textit{number-of}\ w) = (\textit{number-of}\ w :: 'a::\{\textit{number-ring},\textit{field-char-0}\})$
$\langle proof \rangle$

lemmas *zero-rat* = *Zero-rat-def*
lemmas *one-rat* = *One-rat-def*

abbreviation
$\textit{rat-of-nat} :: \textit{nat} \Rightarrow \textit{rat}$
where
$\textit{rat-of-nat} \equiv \textit{of-nat}$

abbreviation
$\textit{rat-of-int} :: \textit{int} \Rightarrow \textit{rat}$

where
  rat-of-int $\equiv$ *of-int*

14.5 Implementation of rational numbers as pairs of integers

definition
  $Rational :: int \times int \Rightarrow rat$
where
  $Rational = INum$

code-datatype *Rational*

lemma *Rational-simp*:
  $Rational\ (k,\ l) = rat\text{-}of\text{-}int\ k\ /\ rat\text{-}of\text{-}int\ l$
  ⟨*proof*⟩

lemma *Rational-zero* [*simp*]: $Rational\ 0_N = 0$
  ⟨*proof*⟩

lemma *Rational-lit* [*simp*]: $Rational\ i_N = rat\text{-}of\text{-}int\ i$
  ⟨*proof*⟩

lemma *zero-rat-code* [*code*, *code unfold*]:
  $0 = Rational\ 0_N$ ⟨*proof*⟩
declare *zero-rat-code* [*symmetric*, *code post*]

lemma *one-rat-code* [*code*, *code unfold*]:
  $1 = Rational\ 1_N$ ⟨*proof*⟩
declare *one-rat-code* [*symmetric*, *code post*]

lemma [*code unfold*, *symmetric*, *code post*]:
  $number\text{-}of\ k = rat\text{-}of\text{-}int\ (number\text{-}of\ k)$
  ⟨*proof*⟩

definition
  [*code func del*]: $Fract'\ (b{::}bool)\ k\ l = Fract\ k\ l$

lemma [*code*]:
  $Fract\ k\ l = Fract'\ (l \neq 0)\ k\ l$
  ⟨*proof*⟩

lemma [*code*]:
  $Fract'\ True\ k\ l = (if\ l \neq 0\ then\ Rational\ (k,\ l)\ else\ Fract\ 1\ 0)$
  ⟨*proof*⟩

lemma [*code*]:
  $of\text{-}rat\ (Rational\ (k,\ l)) = (if\ l \neq 0\ then\ of\text{-}int\ k\ /\ of\text{-}int\ l\ else\ 0)$
  ⟨*proof*⟩

instantiation *rat* :: *eq*
begin

definition [*code func del*]: *eq-class.eq* (*r*::*rat*) *s* $\longleftrightarrow$ *r* = *s*

instance ⟨*proof*⟩

lemma *rat-eq-code* [*code*]: *eq-class.eq* (*Rational x*) (*Rational y*) $\longleftrightarrow$ *eq-class.eq* (*normNum x*) (*normNum y*)
⟨*proof*⟩

end

lemma *rat-less-eq-code* [*code*]: *Rational x* $\leq$ *Rational y* $\longleftrightarrow$ *normNum x* $\leq_N$ *normNum y*
⟨*proof*⟩

lemma *rat-less-code* [*code*]: *Rational x* < *Rational y* $\longleftrightarrow$ *normNum x* $<_N$ *normNum y*
⟨*proof*⟩

lemma *rat-add-code* [*code*]: *Rational x* + *Rational y* = *Rational* (*x* $+_N$ *y*)
⟨*proof*⟩

lemma *rat-mul-code* [*code*]: *Rational x* * *Rational y* = *Rational* (*x* $*_N$ *y*)
⟨*proof*⟩

lemma *rat-neg-code* [*code*]: − *Rational x* = *Rational* ($\sim_N$ *x*)
⟨*proof*⟩

lemma *rat-sub-code* [*code*]: *Rational x* − *Rational y* = *Rational* (*x* $-_N$ *y*)
⟨*proof*⟩

lemma *rat-inv-code* [*code*]: *inverse* (*Rational x*) = *Rational* (*Ninv x*)
⟨*proof*⟩

lemma *rat-div-code* [*code*]: *Rational x* / *Rational y* = *Rational* (*x* $\div_N$ *y*)
⟨*proof*⟩

Setup for SML code generator

types-code
rat ((*int* */ *int*))
attach (*term-of*) ⟪
fun term-of-rat (*p*, *q*) =
let
val rT = *Type* (*Rational.rat*, [])
in
if q = *1 orelse p* = *0 then HOLogic.mk-number rT p*
else @{*term op* / :: *rat* ⇒ *rat* ⇒ *rat*} $

```
    HOLogic.mk-number rT p $ HOLogic.mk-number rT q
  end;
»
attach (test) «
fun gen-rat i =
  let
    val p = random-range 0 i;
    val q = random-range 1 (i + 1);
    val g = Integer.gcd p q;
    val p' = p div g;
    val q' = q div g;
    val r = (if one-of [true, false] then p' else ~ p',
      if p' = 0 then 0 else q')
  in
    (r, fn () => term-of-rat r)
  end;
»

consts-code
  Rational ((-))

consts-code
  of-int :: int ⇒ rat (⟨module⟩rat'-of'-int)
attach «
fun rat-of-int 0 = (0, 0)
  | rat-of-int i = (i, 1);
»

end
```

15 Positive real numbers

theory *PReal*
imports *Rational*
begin

Could be generalized and moved to *Ring-and-Field*

lemma *add-eq-exists*: $\exists x.\ a+x = (b::rat)$
⟨*proof*⟩

definition
  cut :: *rat set => bool* **where**
  cut A = ({} ⊂ *A* &
        A < {*r*. *0* < *r*} &
        (∀ *y* ∈ *A*. ((∀ *z*. *0*<*z* & *z* < *y* --> *z* ∈ *A*) & (∃ *u* ∈ *A*. *y* < *u*))))

lemma *cut-of-rat*:
  assumes *q*: *0* < *q* **shows** *cut* {*r*::*rat*. *0* < *r* & *r* < *q*} (**is** *cut ?A*)

⟨*proof*⟩

typedef *preal* = $\{A.\ cut\ A\}$
⟨*proof*⟩

definition
preal-of-rat :: *rat* => *preal* **where**
preal-of-rat q = *Abs-preal* $\{x::rat.\ 0 < x\ \&\ x < q\}$

definition
psup :: *preal set* => *preal* **where**
psup P = *Abs-preal* $(\bigcup X \in P.\ Rep\text{-}preal\ X)$

definition
add-set :: [*rat set*,*rat set*] => *rat set* **where**
add-set $A\ B = \{w.\ \exists\, x \in A.\ \exists\, y \in B.\ w = x + y\}$

definition
diff-set :: [*rat set*,*rat set*] => *rat set* **where**
diff-set $A\ B = \{w.\ \exists\, x.\ 0 < w\ \&\ 0 < x\ \&\ x \notin B\ \&\ x + w \in A\}$

definition
mult-set :: [*rat set*,*rat set*] => *rat set* **where**
mult-set $A\ B = \{w.\ \exists\, x \in A.\ \exists\, y \in B.\ w = x * y\}$

definition
inverse-set :: *rat set* => *rat set* **where**
inverse-set $A = \{x.\ \exists\, y.\ 0 < x\ \&\ x < y\ \&\ inverse\ y \notin A\}$

instantiation *preal* :: {*ord*, *plus*, *minus*, *times*, *inverse*, *one*}
begin

definition
preal-less-def:
$R < S$ == *Rep-preal* R < *Rep-preal* S

definition
preal-le-def:
$R \le S$ == *Rep-preal* $R \subseteq$ *Rep-preal* S

definition
preal-add-def:
$R + S$ == *Abs-preal* (*add-set* (*Rep-preal* R) (*Rep-preal* S))

definition
preal-diff-def:
$R - S$ == *Abs-preal* (*diff-set* (*Rep-preal* R) (*Rep-preal* S))

definition
preal-mult-def:
$R * S == Abs\text{-}preal\ (mult\text{-}set\ (Rep\text{-}preal\ R)\ (Rep\text{-}preal\ S))$

definition
preal-inverse-def:
$inverse\ R == Abs\text{-}preal\ (inverse\text{-}set\ (Rep\text{-}preal\ R))$

definition $R\ /\ S = R * inverse\ (S::preal)$

definition
preal-one-def:
$1 == preal\text{-}of\text{-}rat\ 1$

instance ⟨*proof*⟩

end

Reduces equality on abstractions to equality on representatives

declare *Abs-preal-inject* [*simp*]
declare *Abs-preal-inverse* [*simp*]

lemma *rat-mem-preal*: $0 < q ==> \{r::rat.\ 0 < r\ \&\ r < q\} \in preal$
⟨*proof*⟩

lemma *preal-nonempty*: $A \in preal ==> \exists x{\in}A.\ 0 < x$
⟨*proof*⟩

lemma *preal-Ex-mem*: $A \in preal \Longrightarrow \exists x.\ x \in A$
⟨*proof*⟩

lemma *preal-imp-psubset-positives*: $A \in preal ==> A < \{r.\ 0 < r\}$
⟨*proof*⟩

lemma *preal-exists-bound*: $A \in preal ==> \exists x.\ 0 < x\ \&\ x \notin A$
⟨*proof*⟩

lemma *preal-exists-greater*: $[|\ A \in preal;\ y \in A\ |] ==> \exists u \in A.\ y < u$
⟨*proof*⟩

lemma *preal-downwards-closed*: $[|\ A \in preal;\ y \in A;\ 0 < z;\ z < y\ |] ==> z \in A$
⟨*proof*⟩

Relaxing the final premise

lemma *preal-downwards-closed'*:
$[|\ A \in preal;\ y \in A;\ 0 < z;\ z \le y\ |] ==> z \in A$
⟨*proof*⟩

A positive fraction not in a positive real is an upper bound. Gleason p. 122

- Remark (1)

lemma *not-in-preal-ub*:
  assumes *A*: $A \in preal$
    and *notx*: $x \notin A$
    and *y*: $y \in A$
    and *pos*: $0 < x$
  shows $y < x$
⟨*proof*⟩

preal lemmas instantiated to *Rep-preal X*

lemma *mem-Rep-preal-Ex*: $\exists x.\ x \in Rep\text{-}preal\ X$
⟨*proof*⟩

lemma *Rep-preal-exists-bound*: $\exists x>0.\ x \notin Rep\text{-}preal\ X$
⟨*proof*⟩

lemmas *not-in-Rep-preal-ub* = *not-in-preal-ub* [*OF Rep-preal*]

15.1 *preal-of-prat*: the Injection from prat to preal

lemma *rat-less-set-mem-preal*: $0 < y ==> \{u::rat.\ 0 < u\ \&\ u < y\} \in preal$
⟨*proof*⟩

lemma *rat-subset-imp-le*:
  $[|\{u::rat.\ 0 < u\ \&\ u < x\} \subseteq \{u.\ 0 < u\ \&\ u < y\};\ 0<x|] ==> x \leq y$
⟨*proof*⟩

lemma *rat-set-eq-imp-eq*:
  $[|\{u::rat.\ 0 < u\ \&\ u < x\} = \{u.\ 0 < u\ \&\ u < y\};$
    $0 < x;\ 0 < y|] ==> x = y$
⟨*proof*⟩

15.2 Properties of Ordering

lemma *preal-le-refl*: $w \leq (w::preal)$
⟨*proof*⟩

lemma *preal-le-trans*: $[|\ i \leq j;\ j \leq k\ |] ==> i \leq (k::preal)$
⟨*proof*⟩

lemma *preal-le-anti-sym*: $[|\ z \leq w;\ w \leq z\ |] ==> z = (w::preal)$
⟨*proof*⟩

lemma *preal-less-le*: $((w::preal) < z) = (w \leq z\ \&\ w \neq z)$
⟨*proof*⟩

instance *preal* :: *order*
  ⟨*proof*⟩

lemma *preal-imp-pos*: $[|A \in \mathit{preal}; r \in A|] ==> 0 < r$
⟨*proof*⟩

lemma *preal-le-linear*: $x <= y \mid y <= (x{::}\mathit{preal})$
⟨*proof*⟩

instance *preal* :: *linorder*
⟨*proof*⟩

instantiation *preal* :: *distrib-lattice*
begin

definition
$(\mathit{inf} :: \mathit{preal} \Rightarrow \mathit{preal} \Rightarrow \mathit{preal}) = \mathit{min}$

definition
$(\mathit{sup} :: \mathit{preal} \Rightarrow \mathit{preal} \Rightarrow \mathit{preal}) = \mathit{max}$

instance
⟨*proof*⟩

end

15.3 Properties of Addition

lemma *preal-add-commute*: $(x{::}\mathit{preal}) + y = y + x$
⟨*proof*⟩

Lemmas for proving that addition of two positive reals gives a positive real

lemma *empty-psubset-nonempty*: $a \in A ==> \{\} \subset A$
⟨*proof*⟩

Part 1 of Dedekind sections definition

lemma *add-set-not-empty*:
$[|A \in \mathit{preal}; B \in \mathit{preal}|] ==> \{\} \subset \mathit{add\text{-}set}\ A\ B$
⟨*proof*⟩

Part 2 of Dedekind sections definition. A structured version of this proof is *preal-not-mem-mult-set-Ex* below.

lemma *preal-not-mem-add-set-Ex*:
$[|A \in \mathit{preal}; B \in \mathit{preal}|] ==> \exists\, q>0.\ q \notin \mathit{add\text{-}set}\ A\ B$
⟨*proof*⟩

lemma *add-set-not-rat-set*:
assumes A: $A \in \mathit{preal}$
and B: $B \in \mathit{preal}$
shows $\mathit{add\text{-}set}\ A\ B < \{r.\ 0 < r\}$
⟨*proof*⟩

Part 3 of Dedekind sections definition

lemma *add-set-lemma3*:
$[|A \in preal; B \in preal; u \in add\text{-}set\ A\ B; 0 < z; z < u|]$
$==> z \in add\text{-}set\ A\ B$
$\langle proof \rangle$

Part 4 of Dedekind sections definition

lemma *add-set-lemma4*:
$[|A \in preal; B \in preal; y \in add\text{-}set\ A\ B|] ==> \exists\, u \in add\text{-}set\ A\ B.\ y < u$
$\langle proof \rangle$

lemma *mem-add-set*:
$[|A \in preal; B \in preal|] ==> add\text{-}set\ A\ B \in preal$
$\langle proof \rangle$

lemma *preal-add-assoc*: $((x::preal) + y) + z = x + (y + z)$
$\langle proof \rangle$

instance *preal* :: *ab-semigroup-add*
$\langle proof \rangle$

lemma *preal-add-left-commute*: $x + (y + z) = y + ((x + z)::preal)$
$\langle proof \rangle$

Positive Real addition is an AC operator

lemmas *preal-add-ac = preal-add-assoc preal-add-commute preal-add-left-commute*

15.4 Properties of Multiplication

Proofs essentially same as for addition

lemma *preal-mult-commute*: $(x::preal) * y = y * x$
$\langle proof \rangle$

Multiplication of two positive reals gives a positive real.

Lemmas for proving positive reals multiplication set in *preal*

Part 1 of Dedekind sections definition

lemma *mult-set-not-empty*:
$[|A \in preal; B \in preal|] ==> \{\} \subset mult\text{-}set\ A\ B$
$\langle proof \rangle$

Part 2 of Dedekind sections definition

lemma *preal-not-mem-mult-set-Ex*:
assumes A: $A \in preal$
and B: $B \in preal$
shows $\exists\, q.\ 0 < q\ \&\ q \notin mult\text{-}set\ A\ B$
$\langle proof \rangle$

lemma *mult-set-not-rat-set*:
  assumes *A*: $A \in preal$
    and *B*: $B \in preal$
  shows $mult\text{-}set\ A\ B < \{r.\ 0 < r\}$
⟨*proof*⟩

Part 3 of Dedekind sections definition

lemma *mult-set-lemma3*:
    $[|A \in preal;\ B \in preal;\ u \in mult\text{-}set\ A\ B;\ 0 < z;\ z < u|]$
    $==> z \in mult\text{-}set\ A\ B$
⟨*proof*⟩

Part 4 of Dedekind sections definition

lemma *mult-set-lemma4*:
    $[|A \in preal;\ B \in preal;\ y \in mult\text{-}set\ A\ B|] ==> \exists u \in mult\text{-}set\ A\ B.\ y < u$
⟨*proof*⟩

lemma *mem-mult-set*:
    $[|A \in preal;\ B \in preal|] ==> mult\text{-}set\ A\ B \in preal$
⟨*proof*⟩

lemma *preal-mult-assoc*: $((x::preal) * y) * z = x * (y * z)$
⟨*proof*⟩

instance *preal* :: *ab-semigroup-mult*
⟨*proof*⟩

lemma *preal-mult-left-commute*: $x * (y * z) = y * ((x * z)::preal)$
⟨*proof*⟩

Positive Real multiplication is an AC operator

lemmas *preal-mult-ac* =
    preal-mult-assoc preal-mult-commute preal-mult-left-commute

Positive real 1 is the multiplicative identity element

lemma *preal-mult-1*: $(1::preal) * z = z$
⟨*proof*⟩

instance *preal* :: *comm-monoid-mult*
⟨*proof*⟩

lemma *preal-mult-1-right*: $z * (1::preal) = z$
⟨*proof*⟩

15.5 Distribution of Multiplication across Addition

lemma *mem-Rep-preal-add-iff*:

$(z \in \mathit{Rep\text{-}preal}(R{+}S)) = (\exists\, x \in \mathit{Rep\text{-}preal}\ R.\ \exists\, y \in \mathit{Rep\text{-}preal}\ S.\ z = x + y)$
⟨*proof*⟩

lemma *mem-Rep-preal-mult-iff*:
$(z \in \mathit{Rep\text{-}preal}(R{*}S)) = (\exists\, x \in \mathit{Rep\text{-}preal}\ R.\ \exists\, y \in \mathit{Rep\text{-}preal}\ S.\ z = x * y)$
⟨*proof*⟩

lemma *distrib-subset1*:
$\mathit{Rep\text{-}preal}\ (w * (x + y)) \subseteq \mathit{Rep\text{-}preal}\ (w * x + w * y)$
⟨*proof*⟩

lemma *preal-add-mult-distrib-mean*:
assumes a: $a \in \mathit{Rep\text{-}preal}\ w$
and b: $b \in \mathit{Rep\text{-}preal}\ w$
and d: $d \in \mathit{Rep\text{-}preal}\ x$
and e: $e \in \mathit{Rep\text{-}preal}\ y$
shows $\exists\, c \in \mathit{Rep\text{-}preal}\ w.\ a * d + b * e = c * (d + e)$
⟨*proof*⟩

lemma *distrib-subset2*:
$\mathit{Rep\text{-}preal}\ (w * x + w * y) \subseteq \mathit{Rep\text{-}preal}\ (w * (x + y))$
⟨*proof*⟩

lemma *preal-add-mult-distrib2*: $(w * ((x{::}\mathit{preal}) + y)) = (w * x) + (w * y)$
⟨*proof*⟩

lemma *preal-add-mult-distrib*: $(((x{::}\mathit{preal}) + y) * w) = (x * w) + (y * w)$
⟨*proof*⟩

instance *preal* :: *comm-semiring*
⟨*proof*⟩

15.6 Existence of Inverse, a Positive Real

lemma *mem-inv-set-ex*:
assumes A: $A \in \mathit{preal}$ **shows** $\exists\, x\ y.\ 0 < x\ \&\ x < y\ \&\ \mathit{inverse}\ y \notin A$
⟨*proof*⟩

Part 1 of Dedekind sections definition

lemma *inverse-set-not-empty*:
$A \in \mathit{preal} \Longrightarrow \{\} \subset \mathit{inverse\text{-}set}\ A$
⟨*proof*⟩

Part 2 of Dedekind sections definition

lemma *preal-not-mem-inverse-set-Ex*:
assumes A: $A \in \mathit{preal}$ **shows** $\exists\, q.\ 0 < q\ \&\ q \notin \mathit{inverse\text{-}set}\ A$
⟨*proof*⟩

lemma *inverse-set-not-rat-set*:

assumes A: $A \in preal$ **shows** $inverse\text{-}set\ A < \{r.\ 0 < r\}$
⟨*proof*⟩

Part 3 of Dedekind sections definition

lemma *inverse-set-lemma3*:
$[|A \in preal;\ u \in inverse\text{-}set\ A;\ 0 < z;\ z < u|]$
$==> z \in inverse\text{-}set\ A$
⟨*proof*⟩

Part 4 of Dedekind sections definition

lemma *inverse-set-lemma4*:
$[|A \in preal;\ y \in inverse\text{-}set\ A|] ==> \exists u \in inverse\text{-}set\ A.\ y < u$
⟨*proof*⟩

lemma *mem-inverse-set*:
$A \in preal ==> inverse\text{-}set\ A \in preal$
⟨*proof*⟩

15.7 Gleason's Lemma 9-3.4, page 122

lemma *Gleason9-34-exists*:
assumes A: $A \in preal$
and $\forall x{\in}A.\ x + u \in A$
and $0 \le z$
shows $\exists b{\in}A.\ b + (of\text{-}int\ z) * u \in A$
⟨*proof*⟩

lemma *Gleason9-34-contra*:
assumes A: $A \in preal$
shows $[|\forall x{\in}A.\ x + u \in A;\ 0 < u;\ 0 < y;\ y \notin A|] ==> False$
⟨*proof*⟩

lemma *Gleason9-34*:
assumes A: $A \in preal$
and *upos*: $0 < u$
shows $\exists r \in A.\ r + u \notin A$
⟨*proof*⟩

15.8 Gleason's Lemma 9-3.6

lemma *lemma-gleason9-36*:
assumes A: $A \in preal$
and x: $1 < x$
shows $\exists r \in A.\ r{*}x \notin A$
⟨*proof*⟩

15.9 Existence of Inverse: Part 2

lemma *mem-Rep-preal-inverse-iff*:
$(z \in \mathit{Rep\text{-}preal}(\mathit{inverse}\ R)) =$
$(0 < z \wedge (\exists\, y.\ z < y \wedge \mathit{inverse}\ y \notin \mathit{Rep\text{-}preal}\ R))$
⟨*proof*⟩

lemma *Rep-preal-of-rat*:
$0 < q ==> \mathit{Rep\text{-}preal}\ (\mathit{preal\text{-}of\text{-}rat}\ q) = \{x.\ 0 < x \wedge x < q\}$
⟨*proof*⟩

lemma *subset-inverse-mult-lemma*:
assumes *xpos*: $0 < x$ **and** *xless*: $x < 1$
shows $\exists\, r\ u\ y.\ 0 < r\ \&\ r < y\ \&\ \mathit{inverse}\ y \notin \mathit{Rep\text{-}preal}\ R\ \&$
$u \in \mathit{Rep\text{-}preal}\ R\ \&\ x = r * u$
⟨*proof*⟩

lemma *subset-inverse-mult*:
$\mathit{Rep\text{-}preal}(\mathit{preal\text{-}of\text{-}rat}\ 1) \subseteq \mathit{Rep\text{-}preal}(\mathit{inverse}\ R * R)$
⟨*proof*⟩

lemma *inverse-mult-subset-lemma*:
assumes *rpos*: $0 < r$
and *rless*: $r < y$
and *notin*: $\mathit{inverse}\ y \notin \mathit{Rep\text{-}preal}\ R$
and *q*: $q \in \mathit{Rep\text{-}preal}\ R$
shows $r{*}q < 1$
⟨*proof*⟩

lemma *inverse-mult-subset*:
$\mathit{Rep\text{-}preal}(\mathit{inverse}\ R * R) \subseteq \mathit{Rep\text{-}preal}(\mathit{preal\text{-}of\text{-}rat}\ 1)$
⟨*proof*⟩

lemma *preal-mult-inverse*: $\mathit{inverse}\ R * R = (1{::}\mathit{preal})$
⟨*proof*⟩

lemma *preal-mult-inverse-right*: $R * \mathit{inverse}\ R = (1{::}\mathit{preal})$
⟨*proof*⟩

Theorems needing *Gleason9-34*

lemma *Rep-preal-self-subset*: $\mathit{Rep\text{-}preal}\ (R) \subseteq \mathit{Rep\text{-}preal}(R + S)$
⟨*proof*⟩

lemma *Rep-preal-sum-not-subset*: $\sim\ \mathit{Rep\text{-}preal}\ (R + S) \subseteq \mathit{Rep\text{-}preal}(R)$
⟨*proof*⟩

lemma *Rep-preal-sum-not-eq*: $\mathit{Rep\text{-}preal}\ (R + S) \neq \mathit{Rep\text{-}preal}(R)$
⟨*proof*⟩

at last, Gleason prop. 9-3.5(iii) page 123

lemma *preal-self-less-add-left*: $(R::preal) < R + S$
$\langle proof \rangle$

lemma *preal-self-less-add-right*: $(R::preal) < S + R$
$\langle proof \rangle$

lemma *preal-not-eq-self*: $x \neq x + (y::preal)$
$\langle proof \rangle$

15.10 Subtraction for Positive Reals

Gleason prop. 9-3.5(iv), page 123: proving $A < B \Longrightarrow \exists D.\ A + D = B$. We define the claimed D and show that it is a positive real

Part 1 of Dedekind sections definition

lemma *diff-set-not-empty*:
$R < S ==> \{\} \subset \textit{diff-set}\ (\textit{Rep-preal}\ S)\ (\textit{Rep-preal}\ R)$
$\langle proof \rangle$

Part 2 of Dedekind sections definition

lemma *diff-set-nonempty*:
$\exists q.\ 0 < q\ \&\ q \notin \textit{diff-set}\ (\textit{Rep-preal}\ S)\ (\textit{Rep-preal}\ R)$
$\langle proof \rangle$

lemma *diff-set-not-rat-set*:
$\textit{diff-set}\ (\textit{Rep-preal}\ S)\ (\textit{Rep-preal}\ R) < \{r.\ 0 < r\}$ (**is** $?lhs < ?rhs$)
$\langle proof \rangle$

Part 3 of Dedekind sections definition

lemma *diff-set-lemma3*:
$[|R < S;\ u \in \textit{diff-set}\ (\textit{Rep-preal}\ S)\ (\textit{Rep-preal}\ R);\ 0 < z;\ z < u|]$
$==> z \in \textit{diff-set}\ (\textit{Rep-preal}\ S)\ (\textit{Rep-preal}\ R)$
$\langle proof \rangle$

Part 4 of Dedekind sections definition

lemma *diff-set-lemma4*:
$[|R < S;\ y \in \textit{diff-set}\ (\textit{Rep-preal}\ S)\ (\textit{Rep-preal}\ R)|]$
$==> \exists u \in \textit{diff-set}\ (\textit{Rep-preal}\ S)\ (\textit{Rep-preal}\ R).\ y < u$
$\langle proof \rangle$

lemma *mem-diff-set*:
$R < S ==> \textit{diff-set}\ (\textit{Rep-preal}\ S)\ (\textit{Rep-preal}\ R) \in \textit{preal}$
$\langle proof \rangle$

lemma *mem-Rep-preal-diff-iff*:
$R < S ==>$
$(z \in \textit{Rep-preal}(S-R)) =$
$(\exists x.\ 0 < x\ \&\ 0 < z\ \&\ x \notin \textit{Rep-preal}\ R\ \&\ x + z \in \textit{Rep-preal}\ S)$

⟨*proof*⟩

proving that $R + D \leq S$

lemma *less-add-left-lemma*:
  assumes *Rless*: $R < S$
    and *a*: $a \in$ *Rep-preal* R
    and *cb*: $c + b \in$ *Rep-preal* S
    and $c \notin$ *Rep-preal* R
    and $0 < b$
    and $0 < c$
  shows $a + b \in$ *Rep-preal* S
⟨*proof*⟩

lemma *less-add-left-le1*:
    $R < (S{::}preal) ==> R + (S{-}R) \leq S$
⟨*proof*⟩

15.11 proving that $S \leq R + D$ — trickier

lemma *lemma-sum-mem-Rep-preal-ex*:
   $x \in$ *Rep-preal* $S ==> \exists\, e.\ 0 < e\ \&\ x + e \in$ *Rep-preal* S
⟨*proof*⟩

lemma *less-add-left-lemma2*:
  assumes *Rless*: $R < S$
    and *x*: $x \in$ *Rep-preal* S
    and *xnot*: $x \notin$ *Rep-preal* R
  shows $\exists\, u\ v\ z.\ 0 < v\ \&\ 0 < z\ \&\ u \in$ *Rep-preal* $R\ \&\ z \notin$ *Rep-preal* $R\ \&$
        $z + v \in$ *Rep-preal* $S\ \&\ x = u + v$
⟨*proof*⟩

lemma *less-add-left-le2*: $R < (S{::}preal) ==> S \leq R + (S{-}R)$
⟨*proof*⟩

lemma *less-add-left*: $R < (S{::}preal) ==> R + (S{-}R) = S$
⟨*proof*⟩

lemma *less-add-left-Ex*: $R < (S{::}preal) ==> \exists\, D.\ R + D = S$
⟨*proof*⟩

lemma *preal-add-less2-mono1*: $R < (S{::}preal) ==> R + T < S + T$
⟨*proof*⟩

lemma *preal-add-less2-mono2*: $R < (S{::}preal) ==> T + R < T + S$
⟨*proof*⟩

lemma *preal-add-right-less-cancel*: $R + T < S + T ==> R < (S{::}preal)$
⟨*proof*⟩

lemma *preal-add-left-less-cancel*: $T + R < T + S ==> R < (S::preal)$
⟨*proof*⟩

lemma *preal-add-less-cancel-right*: $((R::preal) + T < S + T) = (R < S)$
⟨*proof*⟩

lemma *preal-add-less-cancel-left*: $(T + (R::preal) < T + S) = (R < S)$
⟨*proof*⟩

lemma *preal-add-le-cancel-right*: $((R::preal) + T \leq S + T) = (R \leq S)$
⟨*proof*⟩

lemma *preal-add-le-cancel-left*: $(T + (R::preal) \leq T + S) = (R \leq S)$
⟨*proof*⟩

lemma *preal-add-less-mono*:
$[| x1 < y1; x2 < y2 |] ==> x1 + x2 < y1 + (y2::preal)$
⟨*proof*⟩

lemma *preal-add-right-cancel*: $(R::preal) + T = S + T ==> R = S$
⟨*proof*⟩

lemma *preal-add-left-cancel*: $C + A = C + B ==> A = (B::preal)$
⟨*proof*⟩

lemma *preal-add-left-cancel-iff*: $(C + A = C + B) = ((A::preal) = B)$
⟨*proof*⟩

lemma *preal-add-right-cancel-iff*: $(A + C = B + C) = ((A::preal) = B)$
⟨*proof*⟩

lemmas *preal-cancels* =
preal-add-less-cancel-right preal-add-less-cancel-left
preal-add-le-cancel-right preal-add-le-cancel-left
preal-add-left-cancel-iff preal-add-right-cancel-iff

instance *preal* :: *ordered-cancel-ab-semigroup-add*
⟨*proof*⟩

15.12 Completeness of type *preal*

Prove that supremum is a cut

Part 1 of Dedekind sections definition

lemma *preal-sup-set-not-empty*:
$P \neq \{\} ==> \{\} \subset (\bigcup X \in P.\ Rep\text{-}preal(X))$
⟨*proof*⟩

Part 2 of Dedekind sections definition

lemma *preal-sup-not-exists*:
$\forall X \in P.\ X \leq Y ==> \exists q.\ 0 < q \ \&\ q \notin (\bigcup X \in P.\ Rep\text{-}preal(X))$
$\langle proof \rangle$

lemma *preal-sup-set-not-rat-set*:
$\forall X \in P.\ X \leq Y ==> (\bigcup X \in P.\ Rep\text{-}preal(X)) < \{r.\ 0 < r\}$
$\langle proof \rangle$

Part 3 of Dedekind sections definition

lemma *preal-sup-set-lemma3*:
$[|P \neq \{\}; \forall X \in P.\ X \leq Y;\ u \in (\bigcup X \in P.\ Rep\text{-}preal(X));\ 0 < z;\ z < u|]$
$==> z \in (\bigcup X \in P.\ Rep\text{-}preal(X))$
$\langle proof \rangle$

Part 4 of Dedekind sections definition

lemma *preal-sup-set-lemma4*:
$[|P \neq \{\}; \forall X \in P.\ X \leq Y;\ y \in (\bigcup X \in P.\ Rep\text{-}preal(X))\ |]$
$==> \exists u \in (\bigcup X \in P.\ Rep\text{-}preal(X)).\ y < u$
$\langle proof \rangle$

lemma *preal-sup*:
$[|P \neq \{\}; \forall X \in P.\ X \leq Y|] ==> (\bigcup X \in P.\ Rep\text{-}preal(X)) \in preal$
$\langle proof \rangle$

lemma *preal-psup-le*:
$[|\ \forall X \in P.\ X \leq Y;\ \ x \in P\ |] ==> x \leq psup\ P$
$\langle proof \rangle$

lemma *psup-le-ub*: $[|\ P \neq \{\}; \forall X \in P.\ X \leq Y\ |] ==> psup\ P \leq Y$
$\langle proof \rangle$

Supremum property

lemma *preal-complete*:
$[|\ P \neq \{\}; \forall X \in P.\ X \leq Y\ |] ==> (\exists X \in P.\ Z < X) = (Z < psup\ P)$
$\langle proof \rangle$

15.13 The Embedding from *rat* into *preal*

lemma *preal-of-rat-add-lemma1*:
$[|x < y + z;\ 0 < x;\ 0 < y|] ==> x * y * inverse\ (y + z) < (y::rat)$
$\langle proof \rangle$

lemma *preal-of-rat-add-lemma2*:
assumes $u < x + y$
and $0 < x$
and $0 < y$
and $0 < u$
shows $\exists v\ w::rat.\ w < y \ \&\ 0 < v \ \&\ v < x \ \&\ 0 < w \ \&\ u = v + w$
$\langle proof \rangle$

lemma *preal-of-rat-add*:
  $[| \ 0 < x; \ 0 < y|]$
  $==> \mathit{preal\text{-}of\text{-}rat} \ ((x::rat) + y) = \mathit{preal\text{-}of\text{-}rat} \ x + \mathit{preal\text{-}of\text{-}rat} \ y$
⟨*proof*⟩

lemma *preal-of-rat-mult-lemma1*:
  $[|x < y; \ 0 < x; \ 0 < z|] ==> x * z * \mathit{inverse} \ y < (z::rat)$
⟨*proof*⟩

lemma *preal-of-rat-mult-lemma2*:
 assumes *xless*: $x < y * z$
  and *xpos*: $0 < x$
  and *ypos*: $0 < y$
 shows $x * z * \mathit{inverse} \ y * \mathit{inverse} \ z < (z::rat)$
⟨*proof*⟩

lemma *preal-of-rat-mult-lemma3*:
 assumes *uless*: $u < x * y$
  and $0 < x$
  and $0 < y$
  and $0 < u$
 shows $\exists v \ w::rat. \ v < x \ \& \ w < y \ \& \ 0 < v \ \& \ 0 < w \ \& \ u = v * w$
⟨*proof*⟩

lemma *preal-of-rat-mult*:
  $[| \ 0 < x; \ 0 < y|]$
  $==> \mathit{preal\text{-}of\text{-}rat} \ ((x::rat) * y) = \mathit{preal\text{-}of\text{-}rat} \ x * \mathit{preal\text{-}of\text{-}rat} \ y$
⟨*proof*⟩

lemma *preal-of-rat-less-iff*:
  $[| \ 0 < x; \ 0 < y|] ==> (\mathit{preal\text{-}of\text{-}rat} \ x < \mathit{preal\text{-}of\text{-}rat} \ y) = (x < y)$
⟨*proof*⟩

lemma *preal-of-rat-le-iff*:
  $[| \ 0 < x; \ 0 < y|] ==> (\mathit{preal\text{-}of\text{-}rat} \ x \le \mathit{preal\text{-}of\text{-}rat} \ y) = (x \le y)$
⟨*proof*⟩

lemma *preal-of-rat-eq-iff*:
  $[| \ 0 < x; \ 0 < y|] ==> (\mathit{preal\text{-}of\text{-}rat} \ x = \mathit{preal\text{-}of\text{-}rat} \ y) = (x = y)$
⟨*proof*⟩

end

16 Defining the Reals from the Positive Reals

theory *RealDef*
imports *PReal*

uses (*real-arith.ML*)
begin

definition
realrel :: $((preal * preal) * (preal * preal))$ *set* **where**
realrel = $\{p.\ \exists x1\ y1\ x2\ y2.\ p = ((x1,y1),(x2,y2))\ \&\ x1+y2 = x2+y1\}$

typedef (*Real*) *real* = *UNIV*//*realrel*
⟨*proof*⟩

definition

real-of-preal :: *preal* => *real* **where**
real-of-preal m = *Abs-Real*(*realrel*‘‘{(*m* + *1*, *1*)})

instantiation *real* :: {*zero*, *one*, *plus*, *minus*, *uminus*, *times*, *inverse*, *ord*, *abs*, *sgn*}
begin

definition
real-zero-def [*code func del*]: *0* = *Abs-Real*(*realrel*‘‘{(*1*, *1*)})

definition
real-one-def [*code func del*]: *1* = *Abs-Real*(*realrel*‘‘{(*1* + *1*, *1*)})

definition
real-add-def [*code func del*]: *z* + *w* =
contents ($\bigcup$(*x*,*y*) ∈ *Rep-Real*(*z*). $\bigcup$(*u*,*v*) ∈ *Rep-Real*(*w*).
{ *Abs-Real*(*realrel*‘‘{(*x*+*u*, *y*+*v*)}) })

definition
real-minus-def [*code func del*]: − *r* = *contents* ($\bigcup$(*x*,*y*) ∈ *Rep-Real*(*r*). { *Abs-Real*(*realrel*‘‘{(*y*,*x*)}) })

definition
real-diff-def [*code func del*]: *r* − (*s*::*real*) = *r* + − *s*

definition
real-mult-def [*code func del*]:
z ∗ *w* =
contents ($\bigcup$(*x*,*y*) ∈ *Rep-Real*(*z*). $\bigcup$(*u*,*v*) ∈ *Rep-Real*(*w*).
{ *Abs-Real*(*realrel*‘‘{(*x*∗*u* + *y*∗*v*, *x*∗*v* + *y*∗*u*)}) })

definition
real-inverse-def [*code func del*]: *inverse* (*R*::*real*) = (*THE S*. (*R* = *0* & *S* = *0*) | *S* ∗ *R* = *1*)

definition
real-divide-def [*code func del*]: *R* / (*S*::*real*) = *R* ∗ *inverse S*

definition
real-le-def [*code func del*]: $z \le (w{::}real) \longleftrightarrow$
$(\exists\, x\ y\ u\ v.\ x{+}v \le u{+}y\ \&\ (x{,}y) \in \mathit{Rep\text{-}Real}\ z\ \&\ (u{,}v) \in \mathit{Rep\text{-}Real}\ w)$

definition
real-less-def [*code func del*]: $x < (y{::}real) \longleftrightarrow x \le y \wedge x \neq y$

definition
real-abs-def: $abs\ (r{::}real) = (if\ r < 0\ then - r\ else\ r)$

definition
real-sgn-def: $sgn\ (x{::}real) = (if\ x{=}0\ then\ 0\ else\ if\ 0{<}x\ then\ 1\ else - 1)$

instance ⟨*proof*⟩

end

16.1 Equivalence relation over positive reals

lemma *preal-trans-lemma*:
assumes $x + y1 = x1 + y$
and $x + y2 = x2 + y$
shows $x1 + y2 = x2 + (y1{::}preal)$
⟨*proof*⟩

lemma *realrel-iff* [*simp*]: $(((x1{,}y1){,}(x2{,}y2)) \in realrel) = (x1 + y2 = x2 + y1)$
⟨*proof*⟩

lemma *equiv-realrel*: *equiv UNIV realrel*
⟨*proof*⟩

Reduces equality of equivalence classes to the *realrel* relation: $(realrel\ ``\ \{x\} = realrel\ ``\ \{y\}) = ((x,\ y) \in realrel)$

lemmas *equiv-realrel-iff* =
eq-equiv-class-iff [*OF equiv-realrel UNIV-I UNIV-I*]

declare *equiv-realrel-iff* [*simp*]

lemma *realrel-in-real* [*simp*]: $realrel``\{(x{,}y)\}$: *Real*
⟨*proof*⟩

declare *Abs-Real-inject* [*simp*]
declare *Abs-Real-inverse* [*simp*]

Case analysis on the representation of a real number as an equivalence class of pairs of positive reals.

lemma *eq-Abs-Real* [*case-names Abs-Real*, *cases type*: *real*]:
 (!!*x y*. *z* = *Abs-Real*(*realrel*``{(*x*,*y*)}) ==> *P*) ==> *P*
⟨*proof*⟩

16.2 Addition and Subtraction

lemma *real-add-congruent2-lemma*:
 [|*a* + *ba* = *aa* + *b*; *ab* + *bc* = *ac* + *bb*|]
 ==> *a* + *ab* + (*ba* + *bc*) = *aa* + *ac* + (*b* + (*bb*::*preal*))
⟨*proof*⟩

lemma *real-add*:
 Abs-Real (*realrel*``{(*x*,*y*)}) + *Abs-Real* (*realrel*``{(*u*,*v*)}) =
 Abs-Real (*realrel*``{(*x*+*u*, *y*+*v*)})
⟨*proof*⟩

lemma *real-minus*: − *Abs-Real*(*realrel*``{(*x*,*y*)}) = *Abs-Real*(*realrel* `` {(*y*,*x*)})
⟨*proof*⟩

instance *real* :: *ab-group-add*
⟨*proof*⟩

16.3 Multiplication

lemma *real-mult-congruent2-lemma*:
 !!(*x1*::*preal*). [| *x1* + *y2* = *x2* + *y1* |] ==>
  x ∗ *x1* + *y* ∗ *y1* + (*x* ∗ *y2* + *y* ∗ *x2*) =
  x ∗ *x2* + *y* ∗ *y2* + (*x* ∗ *y1* + *y* ∗ *x1*)
⟨*proof*⟩

lemma *real-mult-congruent2*:
 (%*p1 p2*.
  (%(*x1*,*y1*). (%(*x2*,*y2*).
   { *Abs-Real* (*realrel*``{(*x1*∗*x2* + *y1*∗*y2*, *x1*∗*y2*+*y1*∗*x2*)}) }) *p2*) *p1*)
 respects2 realrel
⟨*proof*⟩

lemma *real-mult*:
 Abs-Real((*realrel*``{(*x1*,*y1*)})) ∗ *Abs-Real*((*realrel*``{(*x2*,*y2*)})) =
 Abs-Real(*realrel* `` {(*x1*∗*x2*+*y1*∗*y2*,*x1*∗*y2*+*y1*∗*x2*)})
⟨*proof*⟩

lemma *real-mult-commute*: (*z*::*real*) ∗ *w* = *w* ∗ *z*
⟨*proof*⟩

lemma *real-mult-assoc*: ((*z1*::*real*) ∗ *z2*) ∗ *z3* = *z1* ∗ (*z2* ∗ *z3*)
⟨*proof*⟩

lemma *real-mult-1*: (*1*::*real*) ∗ *z* = *z*
⟨*proof*⟩

lemma *real-add-mult-distrib*: $((z1::real) + z2) * w = (z1 * w) + (z2 * w)$
⟨*proof*⟩

one and zero are distinct

lemma *real-zero-not-eq-one*: $0 \neq (1::real)$
⟨*proof*⟩

instance *real* :: *comm-ring-1*
⟨*proof*⟩

16.4 Inverse and Division

lemma *real-zero-iff*: $Abs\text{-}Real\ (realrel\ ``\ \{(x, x)\}) = 0$
⟨*proof*⟩

Instead of using an existential quantifier and constructing the inverse within the proof, we could define the inverse explicitly.

lemma *real-mult-inverse-left-ex*: $x \neq 0 ==> \exists\, y.\ y*x = (1::real)$
⟨*proof*⟩

lemma *real-mult-inverse-left*: $x \neq 0 ==> inverse(x)*x = (1::real)$
⟨*proof*⟩

16.5 The Real Numbers form a Field

instance *real* :: *field*
⟨*proof*⟩

Inverse of zero! Useful to simplify certain equations

lemma *INVERSE-ZERO*: $inverse\ 0 = (0::real)$
⟨*proof*⟩

instance *real* :: *division-by-zero*
⟨*proof*⟩

16.6 The $\leq$ Ordering

lemma *real-le-refl*: $w \leq (w::real)$
⟨*proof*⟩

The arithmetic decision procedure is not set up for type preal. This lemma is currently unused, but it could simplify the proofs of the following two lemmas.

lemma *preal-eq-le-imp-le*:
  assumes *eq*: $a+b = c+d$ **and** *le*: $c \leq a$
  shows $b \leq (d::preal)$
⟨*proof*⟩

lemma *real-le-lemma*:
  assumes *l*: $u1 + v2 \leq u2 + v1$
    and $x1 + v1 = u1 + y1$
    and $x2 + v2 = u2 + y2$
  shows $x1 + y2 \leq x2 + (y1::preal)$
⟨*proof*⟩

lemma *real-le*:
  $(Abs\text{-}Real(realrel``\{(x1,y1)\}) \leq Abs\text{-}Real(realrel``\{(x2,y2)\})) =$
  $(x1 + y2 \leq x2 + y1)$
⟨*proof*⟩

lemma *real-le-anti-sym*: $[| z \leq w; w \leq z |] ==> z = (w::real)$
⟨*proof*⟩

lemma *real-trans-lemma*:
  assumes $x + v \leq u + y$
    and $u + v' \leq u' + v$
    and $x2 + v2 = u2 + y2$
  shows $x + v' \leq u' + (y::preal)$
⟨*proof*⟩

lemma *real-le-trans*: $[| i \leq j; j \leq k |] ==> i \leq (k::real)$
⟨*proof*⟩

lemma *real-less-le*: $((w::real) < z) = (w \leq z \ \& \ w \neq z)$
⟨*proof*⟩

instance *real* :: *order*
⟨*proof*⟩

lemma *real-le-linear*: $(z::real) \leq w \mid w \leq z$
⟨*proof*⟩

instance *real* :: *linorder*
  ⟨*proof*⟩

lemma *real-le-eq-diff*: $(x \leq y) = (x-y \leq (0::real))$
⟨*proof*⟩

lemma *real-add-left-mono*:
  assumes *le*: $x \leq y$ **shows** $z + x \leq z + (y::real)$
⟨*proof*⟩

lemma *real-sum-gt-zero-less*: $(0 < S + (-W::real)) ==> (W < S)$
⟨*proof*⟩

lemma *real-less-sum-gt-zero*: $(W < S) ==> (0 < S + (-W::real))$
⟨*proof*⟩

lemma *real-mult-order*: $[| \ 0 < x; \ 0 < y \ |] ==> (0::real) < x * y$
⟨*proof*⟩

lemma *real-mult-less-mono2*: $[| \ (0::real) < z; \ x < y \ |] ==> z * x < z * y$
⟨*proof*⟩

instantiation *real* :: *distrib-lattice*
begin

definition
  $(inf :: real \Rightarrow real \Rightarrow real) = min$

definition
  $(sup :: real \Rightarrow real \Rightarrow real) = max$

instance
  ⟨*proof*⟩

end

16.7 The Reals Form an Ordered Field

instance *real* :: *ordered-field*
⟨*proof*⟩

instance *real* :: *lordered-ab-group-add* ⟨*proof*⟩

The function *real-of-preal* requires many proofs, but it seems to be essential for proving completeness of the reals from that of the positive reals.

lemma *real-of-preal-add*:
  $real\text{-}of\text{-}preal \ ((x::preal) + y) = real\text{-}of\text{-}preal \ x + real\text{-}of\text{-}preal \ y$
⟨*proof*⟩

lemma *real-of-preal-mult*:
  $real\text{-}of\text{-}preal \ ((x::preal) * y) = real\text{-}of\text{-}preal \ x * real\text{-}of\text{-}preal \ y$
⟨*proof*⟩

Gleason prop 9-4.4 p 127

lemma *real-of-preal-trichotomy*:
  $\exists m. \ (x::real) = real\text{-}of\text{-}preal \ m \mid x = 0 \mid x = -(real\text{-}of\text{-}preal \ m)$
⟨*proof*⟩

lemma *real-of-preal-leD*:

real-of-preal m1 ≤ real-of-preal m2 ==> m1 ≤ m2
⟨*proof*⟩

lemma *real-of-preal-lessI*: *m1 < m2 ==> real-of-preal m1 < real-of-preal m2*
⟨*proof*⟩

lemma *real-of-preal-lessD*:
real-of-preal m1 < real-of-preal m2 ==> m1 < m2
⟨*proof*⟩

lemma *real-of-preal-less-iff* [*simp*]:
(*real-of-preal m1 < real-of-preal m2*) = (*m1 < m2*)
⟨*proof*⟩

lemma *real-of-preal-le-iff*:
(*real-of-preal m1 ≤ real-of-preal m2*) = (*m1 ≤ m2*)
⟨*proof*⟩

lemma *real-of-preal-zero-less*: *0 < real-of-preal m*
⟨*proof*⟩

lemma *real-of-preal-minus-less-zero*: − *real-of-preal m < 0*
⟨*proof*⟩

lemma *real-of-preal-not-minus-gt-zero*: ~ *0 <* − *real-of-preal m*
⟨*proof*⟩

16.8 Theorems About the Ordering

lemma *real-gt-zero-preal-Ex*: (*0 < x*) = (∃ *y. x = real-of-preal y*)
⟨*proof*⟩

lemma *real-gt-preal-preal-Ex*:
real-of-preal z < x ==> ∃ *y. x = real-of-preal y*
⟨*proof*⟩

lemma *real-ge-preal-preal-Ex*:
real-of-preal z ≤ x ==> ∃ *y. x = real-of-preal y*
⟨*proof*⟩

lemma *real-less-all-preal*: *y ≤ 0 ==>* ∀ *x. y < real-of-preal x*
⟨*proof*⟩

lemma *real-less-all-real2*: ~ *0 < y ==>* ∀ *x. y < real-of-preal x*
⟨*proof*⟩

16.9 More Lemmas

lemma *real-mult-left-cancel*: (*c::real*) ≠ *0 ==>* (*c*∗*a=c*∗*b*) = (*a=b*)
⟨*proof*⟩

lemma *real-mult-right-cancel*: $(c::real) \neq 0 ==> (a*c=b*c) = (a=b)$
⟨*proof*⟩

lemma *real-mult-less-iff1* [*simp*]: $(0::real) < z ==> (x*z < y*z) = (x < y)$
⟨*proof*⟩

lemma *real-mult-le-cancel-iff1* [*simp*]: $(0::real) < z ==> (x*z \leq y*z) = (x \leq y)$
⟨*proof*⟩

lemma *real-mult-le-cancel-iff2* [*simp*]: $(0::real) < z ==> (z*x \leq z*y) = (x \leq y)$
⟨*proof*⟩

lemma *real-inverse-gt-one*: $[| (0::real) < x; x < 1 |] ==> 1 < inverse\ x$
⟨*proof*⟩

16.10 Embedding numbers into the Reals

abbreviation
real-of-nat :: *nat* ⇒ *real*
where
real-of-nat ≡ *of-nat*

abbreviation
real-of-int :: *int* ⇒ *real*
where
real-of-int ≡ *of-int*

abbreviation
real-of-rat :: *rat* ⇒ *real*
where
real-of-rat ≡ *of-rat*

consts

real :: *'a* => *real*

defs (overloaded)
real-of-nat-def [*code inline*]: *real* == *real-of-nat*
real-of-int-def [*code inline*]: *real* == *real-of-int*

lemma *real-eq-of-nat*: *real* = *of-nat*
⟨*proof*⟩

lemma *real-eq-of-int*: *real* = *of-int*
⟨*proof*⟩

lemma *real-of-int-zero* [*simp*]: *real* $(0::int) = 0$
⟨*proof*⟩

lemma *real-of-one* [*simp*]: *real* (*1*::*int*) = (*1*::*real*)
⟨*proof*⟩

lemma *real-of-int-add* [*simp*]: *real*(*x* + *y*) = *real* (*x*::*int*) + *real y*
⟨*proof*⟩

lemma *real-of-int-minus* [*simp*]: *real*(−*x*) = −*real* (*x*::*int*)
⟨*proof*⟩

lemma *real-of-int-diff* [*simp*]: *real*(*x* − *y*) = *real* (*x*::*int*) − *real y*
⟨*proof*⟩

lemma *real-of-int-mult* [*simp*]: *real*(*x* ∗ *y*) = *real* (*x*::*int*) ∗ *real y*
⟨*proof*⟩

lemma *real-of-int-setsum* [*simp*]: *real* ((*SUM x*:*A*. *f x*)::*int*) = (*SUM x*:*A*. *real*(*f x*))
⟨*proof*⟩

lemma *real-of-int-setprod* [*simp*]: *real* ((*PROD x*:*A*. *f x*)::*int*) =
(*PROD x*:*A*. *real*(*f x*))
⟨*proof*⟩

lemma *real-of-int-zero-cancel* [*simp*]: (*real x* = *0*) = (*x* = (*0*::*int*))
⟨*proof*⟩

lemma *real-of-int-inject* [*iff*]: (*real* (*x*::*int*) = *real y*) = (*x* = *y*)
⟨*proof*⟩

lemma *real-of-int-less-iff* [*iff*]: (*real* (*x*::*int*) < *real y*) = (*x* < *y*)
⟨*proof*⟩

lemma *real-of-int-le-iff* [*simp*]: (*real* (*x*::*int*) ≤ *real y*) = (*x* ≤ *y*)
⟨*proof*⟩

lemma *real-of-int-gt-zero-cancel-iff* [*simp*]: (*0* < *real* (*n*::*int*)) = (*0* < *n*)
⟨*proof*⟩

lemma *real-of-int-ge-zero-cancel-iff* [*simp*]: (*0* <= *real* (*n*::*int*)) = (*0* <= *n*)
⟨*proof*⟩

lemma *real-of-int-lt-zero-cancel-iff* [*simp*]: (*real* (*n*::*int*) < *0*) = (*n* < *0*)
⟨*proof*⟩

lemma *real-of-int-le-zero-cancel-iff* [*simp*]: (*real* (*n*::*int*) <= *0*) = (*n* <= *0*)
⟨*proof*⟩

lemma *real-of-int-abs* [*simp*]: *real* (*abs x*) = *abs*(*real* (*x*::*int*))

⟨*proof*⟩

lemma *int-less-real-le*: $((n::int) < m) = (real\ n + 1 <= real\ m)$
⟨*proof*⟩

lemma *int-le-real-less*: $((n::int) <= m) = (real\ n < real\ m + 1)$
⟨*proof*⟩

lemma *real-of-int-div-aux*: $d\ \tilde{}= 0 ==> (real\ (x::int)) / (real\ d) = real\ (x\ div\ d) + (real\ (x\ mod\ d)) / (real\ d)$
⟨*proof*⟩

lemma *real-of-int-div*: $(d::int)\ \tilde{}= 0 ==> d\ dvd\ n ==> real(n\ div\ d) = real\ n / real\ d$
⟨*proof*⟩

lemma *real-of-int-div2*:
$0 <= real\ (n::int) / real\ (x) - real\ (n\ div\ x)$
⟨*proof*⟩

lemma *real-of-int-div3*:
$real\ (n::int) / real\ (x) - real\ (n\ div\ x) <= 1$
⟨*proof*⟩

lemma *real-of-int-div4*: $real\ (n\ div\ x) <= real\ (n::int) / real\ x$
⟨*proof*⟩

16.11 Embedding the Naturals into the Reals

lemma *real-of-nat-zero* [*simp*]: $real\ (0::nat) = 0$
⟨*proof*⟩

lemma *real-of-nat-one* [*simp*]: $real\ (Suc\ 0) = (1::real)$
⟨*proof*⟩

lemma *real-of-nat-add* [*simp*]: $real\ (m + n) = real\ (m::nat) + real\ n$
⟨*proof*⟩

lemma *real-of-nat-Suc*: $real\ (Suc\ n) = real\ n + (1::real)$
⟨*proof*⟩

lemma *real-of-nat-less-iff* [*iff*]:
$(real\ (n::nat) < real\ m) = (n < m)$
⟨*proof*⟩

lemma *real-of-nat-le-iff* [*iff*]: $(real\ (n::nat) \leq real\ m) = (n \leq m)$
⟨*proof*⟩

lemma *real-of-nat-ge-zero* [*iff*]: $0 \leq real\ (n::nat)$
⟨*proof*⟩

lemma *real-of-nat-Suc-gt-zero*: $0 < real\ (Suc\ n)$
⟨*proof*⟩

lemma *real-of-nat-mult* [*simp*]: $real\ (m * n) = real\ (m::nat) * real\ n$
⟨*proof*⟩

lemma *real-of-nat-setsum* [*simp*]: $real\ ((SUM\ x{:}A.\ f\ x)::nat) =$
$(SUM\ x{:}A.\ real(f\ x))$
⟨*proof*⟩

lemma *real-of-nat-setprod* [*simp*]: $real\ ((PROD\ x{:}A.\ f\ x)::nat) =$
$(PROD\ x{:}A.\ real(f\ x))$
⟨*proof*⟩

lemma *real-of-card*: $real\ (card\ A) = setsum\ (\%x.1)\ A$
⟨*proof*⟩

lemma *real-of-nat-inject* [*iff*]: $(real\ (n::nat) = real\ m) = (n = m)$
⟨*proof*⟩

lemma *real-of-nat-zero-iff* [*iff*]: $(real\ (n::nat) = 0) = (n = 0)$
⟨*proof*⟩

lemma *real-of-nat-diff*: $n \leq m ==> real\ (m - n) = real\ (m::nat) - real\ n$
⟨*proof*⟩

lemma *real-of-nat-gt-zero-cancel-iff* [*simp*]: $(0 < real\ (n::nat)) = (0 < n)$
⟨*proof*⟩

lemma *real-of-nat-le-zero-cancel-iff* [*simp*]: $(real\ (n::nat) \leq 0) = (n = 0)$
⟨*proof*⟩

lemma *not-real-of-nat-less-zero* [*simp*]: $\sim real\ (n::nat) < 0$
⟨*proof*⟩

lemma *real-of-nat-ge-zero-cancel-iff* [*simp*]: $(0 \leq real\ (n::nat))$
⟨*proof*⟩

lemma *nat-less-real-le*: $((n::nat) < m) = (real\ n + 1 <= real\ m)$
⟨*proof*⟩

lemma *nat-le-real-less*: $((n::nat) <= m) = (real\ n < real\ m + 1)$
⟨*proof*⟩

lemma *real-of-nat-div-aux*: $0 < d ==> (real\ (x::nat))\ /\ (real\ d) =$
$real\ (x\ div\ d) + (real\ (x\ mod\ d))\ /\ (real\ d)$

⟨*proof*⟩

lemma *real-of-nat-div*: *0 < (d::nat) ==> d dvd n ==>*
real(n div d) = real n / real d
⟨*proof*⟩

lemma *real-of-nat-div2*:
0 <= real (n::nat) / real (x) − real (n div x)
⟨*proof*⟩

lemma *real-of-nat-div3*:
real (n::nat) / real (x) − real (n div x) <= 1
⟨*proof*⟩

lemma *real-of-nat-div4*: *real (n div x) <= real (n::nat) / real x*
⟨*proof*⟩

lemma *real-of-int-real-of-nat*: *real (int n) = real n*
⟨*proof*⟩

lemma *real-of-int-of-nat-eq* [*simp*]: *real (of-nat n :: int) = real n*
⟨*proof*⟩

lemma *real-nat-eq-real* [*simp*]: *0 <= x ==> real(nat x) = real x*
⟨*proof*⟩

16.12 Numerals and Arithmetic

instantiation *real* :: *number-ring*
begin

definition
real-number-of-def [*code func del*]: *number-of w = real-of-int w*

instance
⟨*proof*⟩

end

lemma [*code unfold*, *symmetric*, *code post*]:
number-of k = real-of-int (number-of k)
⟨*proof*⟩

Collapse applications of *real* to *number-of*

lemma *real-number-of* [*simp*]: *real (number-of v :: int) = number-of v*
⟨*proof*⟩

lemma *real-of-nat-number-of* [*simp*]:
real (number-of v :: nat) =

(if neg (number-of v :: int) then 0
else (number-of v :: real))
⟨*proof*⟩

⟨*ML*⟩

16.13 Simprules combining x+y and 0: ARE THEY NEEDED?

Needed in this non-standard form by Hyperreal/Transcendental

lemma *real-0-le-divide-iff*:
$((0::real) \le x/y) = ((x \le 0 \mid 0 \le y) \ \& \ (0 \le x \mid y \le 0))$
⟨*proof*⟩

lemma *real-add-minus-iff* [*simp*]: $(x + - a = (0::real)) = (x{=}a)$
⟨*proof*⟩

lemma *real-add-eq-0-iff*: $(x{+}y = (0::real)) = (y = -x)$
⟨*proof*⟩

lemma *real-add-less-0-iff*: $(x{+}y < (0::real)) = (y < -x)$
⟨*proof*⟩

lemma *real-0-less-add-iff*: $((0::real) < x{+}y) = (-x < y)$
⟨*proof*⟩

lemma *real-add-le-0-iff*: $(x{+}y \le (0::real)) = (y \le -x)$
⟨*proof*⟩

lemma *real-0-le-add-iff*: $((0::real) \le x{+}y) = (-x \le y)$
⟨*proof*⟩

16.13.1 Density of the Reals

lemma *real-lbound-gt-zero*:
$[| \ (0::real) < d1; \ 0 < d2 \ |] ==> \exists e. \ 0 < e \ \& \ e < d1 \ \& \ e < d2$
⟨*proof*⟩

Similar results are proved in *Ring-and-Field*

lemma *real-less-half-sum*: $x < y ==> x < (x{+}y) \ / \ (2::real)$
⟨*proof*⟩

lemma *real-gt-half-sum*: $x < y ==> (x{+}y)/(2::real) < y$
⟨*proof*⟩

16.14 Absolute Value Function for the Reals

lemma *abs-minus-add-cancel*: $abs(x + (-y)) = abs \ (y + (-(x::real)))$
⟨*proof*⟩

lemma *abs-le-interval-iff*: $(abs\ x \le r) = (-r \le x\ \&\ x \le (r::real))$
⟨*proof*⟩

lemma *abs-add-one-gt-zero* [*simp*]: $(0::real) < 1 + abs(x)$
⟨*proof*⟩

lemma *abs-real-of-nat-cancel* [*simp*]: $abs\ (real\ x) = real\ (x::nat)$
⟨*proof*⟩

lemma *abs-add-one-not-less-self* [*simp*]: $\sim\ abs(x) + (1::real) < x$
⟨*proof*⟩

lemma *abs-sum-triangle-ineq*: $abs\ ((x::real) + y + (-l + -m)) \le abs(x + -l) + abs(y + -m)$
⟨*proof*⟩

instance *real* :: *lordered-ring*
⟨*proof*⟩

16.15 Implementation of rational real numbers as pairs of integers

definition
Ratreal :: $int \times int \Rightarrow real$
where
$Ratreal = INum$

code-datatype *Ratreal*

lemma *Ratreal-simp*:
$Ratreal\ (k,\ l) = real\text{-}of\text{-}int\ k\ /\ real\text{-}of\text{-}int\ l$
⟨*proof*⟩

lemma *Ratreal-zero* [*simp*]: $Ratreal\ 0_N = 0$
⟨*proof*⟩

lemma *Ratreal-lit* [*simp*]: $Ratreal\ i_N = real\text{-}of\text{-}int\ i$
⟨*proof*⟩

lemma *zero-real-code* [*code*, *code unfold*]:
$0 = Ratreal\ 0_N$ ⟨*proof*⟩
declare *zero-real-code* [*symmetric*, *code post*]

lemma *one-real-code* [*code*, *code unfold*]:
$1 = Ratreal\ 1_N$ ⟨*proof*⟩
declare *one-real-code* [*symmetric*, *code post*]

instantiation *real* :: *eq*
begin

definition *eq-class.eq* (*x*::*real*) *y* $\longleftrightarrow$ *x* = *y*

instance ⟨*proof*⟩

lemma *real-eq-code* [*code*]: *eq-class.eq* (*Ratreal x*) (*Ratreal y*) $\longleftrightarrow$ *eq-class.eq* (*normNum x*) (*normNum y*)
⟨*proof*⟩

end

lemma *real-less-eq-code* [*code*]: *Ratreal x* $\leq$ *Ratreal y* $\longleftrightarrow$ *normNum x* $\leq_N$ *normNum y*
⟨*proof*⟩

lemma *real-less-code* [*code*]: *Ratreal x* < *Ratreal y* $\longleftrightarrow$ *normNum x* $<_N$ *normNum y*
⟨*proof*⟩

lemma *real-add-code* [*code*]: *Ratreal x* + *Ratreal y* = *Ratreal* (*x* $+_N$ *y*)
⟨*proof*⟩

lemma *real-mul-code* [*code*]: *Ratreal x* $*$ *Ratreal y* = *Ratreal* (*x* $*_N$ *y*)
⟨*proof*⟩

lemma *real-neg-code* [*code*]: − *Ratreal x* = *Ratreal* ($\sim_N$ *x*)
⟨*proof*⟩

lemma *real-sub-code* [*code*]: *Ratreal x* − *Ratreal y* = *Ratreal* (*x* $-_N$ *y*)
⟨*proof*⟩

lemma *real-inv-code* [*code*]: *inverse* (*Ratreal x*) = *Ratreal* (*Ninv x*)
⟨*proof*⟩

lemma *real-div-code* [*code*]: *Ratreal x* / *Ratreal y* = *Ratreal* (*x* $\div_N$ *y*)
⟨*proof*⟩

Setup for SML code generator

types-code
real ((*int* $*$/ *int*))
attach (*term-of*) ⟪
fun term-of-real (*p*, *q*) =
let
val rT = *HOLogic.realT*
in
if q = *1 orelse p* = *0 then HOLogic.mk-number rT p*
else @{*term op* / :: *real* ⇒ *real* ⇒ *real*} $

```
    HOLogic.mk-number rT p $ HOLogic.mk-number rT q
  end;
»
attach (test) «
fun gen-real i =
  let
    val p = random-range 0 i;
    val q = random-range 1 (i + 1);
    val g = Integer.gcd p q;
    val p' = p div g;
    val q' = q div g;
    val r = (if one-of [true, false] then p' else ~ p',
      if p' = 0 then 0 else q')
  in
    (r, fn () => term-of-real r)
  end;
»
```

consts-code
Ratreal ((-))

consts-code
of-int :: *int* ⇒ *real* (⟨**module**⟩*real'-of'-int*)
attach «
```
fun real-of-int 0 = (0, 0)
  | real-of-int i = (i, 1);
»
```

declare *real-of-int-of-nat-eq* [*symmetric*, *code*]

end

17 Completeness of the Reals; Floor and Ceiling Functions

theory *RComplete*
imports *Lubs RealDef*
begin

lemma *real-sum-of-halves*: $x/2 + x/2 = (x::real)$
⟨*proof*⟩

17.1 Completeness of Positive Reals

Supremum property for the set of positive reals

Let P be a non-empty set of positive reals, with an upper bound y. Then P has a least upper bound (written S).

FIXME: Can the premise be weakened to $\forall x \in P.\ x \le y$?

lemma *posreal-complete*:
  assumes *positive-P*: $\forall x \in P.\ (0::real) < x$
    and *not-empty-P*: $\exists x.\ x \in P$
    and *upper-bound-Ex*: $\exists y.\ \forall x \in P.\ x<y$
  shows $\exists S.\ \forall y.\ (\exists x \in P.\ y < x) = (y < S)$
⟨*proof*⟩

Completeness properties using *isUb*, *isLub* etc.

lemma *real-isLub-unique*: $[|\ isLub\ R\ S\ x;\ isLub\ R\ S\ y\ |] ==> x = (y::real)$
  ⟨*proof*⟩

Completeness theorem for the positive reals (again).

lemma *posreals-complete*:
  assumes *positive-S*: $\forall x \in S.\ 0 < x$
    and *not-empty-S*: $\exists x.\ x \in S$
    and *upper-bound-Ex*: $\exists u.\ isUb\ (UNIV::real\ set)\ S\ u$
  shows $\exists t.\ isLub\ (UNIV::real\ set)\ S\ t$
⟨*proof*⟩

reals Completeness (again!)

lemma *reals-complete*:
  assumes *notempty-S*: $\exists X.\ X \in S$
    and *exists-Ub*: $\exists Y.\ isUb\ (UNIV::real\ set)\ S\ Y$
  shows $\exists t.\ isLub\ (UNIV :: real\ set)\ S\ t$
⟨*proof*⟩

17.2 The Archimedean Property of the Reals

theorem *reals-Archimedean*:
  assumes *x-pos*: $0 < x$
  shows $\exists n.\ inverse\ (real\ (Suc\ n)) < x$
⟨*proof*⟩

There must be other proofs, e.g. *Suc* of the largest integer in the cut representing x.

lemma *reals-Archimedean2*: $\exists n.\ (x::real) < real\ (n::nat)$
⟨*proof*⟩

lemma *reals-Archimedean3*:
  assumes *x-greater-zero*: $0 < x$
  shows $\forall (y::real).\ \exists (n::nat).\ y < real\ n * x$
⟨*proof*⟩

lemma *reals-Archimedean6*:
    $0 \le r ==> \exists (n::nat).\ real\ (n - 1) \le r\ \&\ r < real\ (n)$

⟨*proof*⟩

lemma *reals-Archimedean6a*: $0 \le r ==> \exists n.\ real\ (n) \le r\ \&\ r < real\ (Suc\ n)$
⟨*proof*⟩

lemma *reals-Archimedean-6b-int*:
$0 \le r ==> \exists n::int.\ real\ n \le r\ \&\ r < real\ (n+1)$
⟨*proof*⟩

lemma *reals-Archimedean-6c-int*:
$r < 0 ==> \exists n::int.\ real\ n \le r\ \&\ r < real\ (n+1)$
⟨*proof*⟩

17.3 Floor and Ceiling Functions from the Reals to the Integers

definition
floor :: *real* => *int* **where**
$floor\ r = (LEAST\ n::int.\ r < real\ (n+1))$

definition
ceiling :: *real* => *int* **where**
$ceiling\ r = -\ floor\ (-\ r)$

notation (*xsymbols*)
floor (⌊-⌋) **and**
ceiling (⌈-⌉)

notation (*HTML* **output**)
floor (⌊-⌋) **and**
ceiling (⌈-⌉)

lemma *number-of-less-real-of-int-iff* [*simp*]:
$((number\text{-}of\ n) < real\ (m::int)) = (number\text{-}of\ n < m)$
⟨*proof*⟩

lemma *number-of-less-real-of-int-iff2* [*simp*]:
$(real\ (m::int) < (number\text{-}of\ n)) = (m < number\text{-}of\ n)$
⟨*proof*⟩

lemma *number-of-le-real-of-int-iff* [*simp*]:
$((number\text{-}of\ n) \le real\ (m::int)) = (number\text{-}of\ n \le m)$
⟨*proof*⟩

lemma *number-of-le-real-of-int-iff2* [*simp*]:
$(real\ (m::int) \le (number\text{-}of\ n)) = (m \le number\text{-}of\ n)$
⟨*proof*⟩

lemma *floor-zero* [*simp*]: *floor 0* = *0*
⟨*proof*⟩

lemma *floor-real-of-nat-zero* [*simp*]: *floor* (*real* (*0*::*nat*)) = *0*
⟨*proof*⟩

lemma *floor-real-of-nat* [*simp*]: *floor* (*real* (*n*::*nat*)) = *int n*
⟨*proof*⟩

lemma *floor-minus-real-of-nat* [*simp*]: *floor* (− *real* (*n*::*nat*)) = − *int n*
⟨*proof*⟩

lemma *floor-real-of-int* [*simp*]: *floor* (*real* (*n*::*int*)) = *n*
⟨*proof*⟩

lemma *floor-minus-real-of-int* [*simp*]: *floor* (− *real* (*n*::*int*)) = − *n*
⟨*proof*⟩

lemma *real-lb-ub-int*: ∃ *n*::*int*. *real n* ≤ *r* & *r* < *real* (*n*+*1*)
⟨*proof*⟩

lemma *lemma-floor*:
  assumes *a1*: *real m* ≤ *r* **and** *a2*: *r* < *real n* + *1*
  shows *m* ≤ (*n*::*int*)
⟨*proof*⟩

lemma *real-of-int-floor-le* [*simp*]: *real* (*floor r*) ≤ *r*
⟨*proof*⟩

lemma *floor-mono*: *x* < *y* ==> *floor x* ≤ *floor y*
⟨*proof*⟩

lemma *floor-mono2*: *x* ≤ *y* ==> *floor x* ≤ *floor y*
⟨*proof*⟩

lemma *lemma-floor2*: *real n* < *real* (*x*::*int*) + *1* ==> *n* ≤ *x*
⟨*proof*⟩

lemma *real-of-int-floor-cancel* [*simp*]:
    (*real* (*floor x*) = *x*) = (∃ *n*::*int*. *x* = *real n*)
⟨*proof*⟩

lemma *floor-eq*: [| *real n* < *x*; *x* < *real n* + *1* |] ==> *floor x* = *n*
⟨*proof*⟩

lemma *floor-eq2*: [| *real n* ≤ *x*; *x* < *real n* + *1* |] ==> *floor x* = *n*
⟨*proof*⟩

lemma *floor-eq3*: [| *real n* < *x*; *x* < *real* (*Suc n*) |] ==> *nat*(*floor x*) = *n*

⟨*proof*⟩

lemma *floor-eq4*: $[| \ real\ n \leq x;\ x < real\ (Suc\ n)\ |] ==> nat(floor\ x) = n$
⟨*proof*⟩

lemma *floor-number-of-eq* [*simp*]:
$floor(number\text{-}of\ n :: real) = (number\text{-}of\ n :: int)$
⟨*proof*⟩

lemma *floor-one* [*simp*]: $floor\ 1 = 1$
⟨*proof*⟩

lemma *real-of-int-floor-ge-diff-one* [*simp*]: $r - 1 \leq real(floor\ r)$
⟨*proof*⟩

lemma *real-of-int-floor-gt-diff-one* [*simp*]: $r - 1 < real(floor\ r)$
⟨*proof*⟩

lemma *real-of-int-floor-add-one-ge* [*simp*]: $r \leq real(floor\ r) + 1$
⟨*proof*⟩

lemma *real-of-int-floor-add-one-gt* [*simp*]: $r < real(floor\ r) + 1$
⟨*proof*⟩

lemma *le-floor*: $real\ a <= x ==> a <= floor\ x$
⟨*proof*⟩

lemma *real-le-floor*: $a <= floor\ x ==> real\ a <= x$
⟨*proof*⟩

lemma *le-floor-eq*: $(a <= floor\ x) = (real\ a <= x)$
⟨*proof*⟩

lemma *le-floor-eq-number-of* [*simp*]:
$(number\text{-}of\ n <= floor\ x) = (number\text{-}of\ n <= x)$
⟨*proof*⟩

lemma *le-floor-eq-zero* [*simp*]: $(0 <= floor\ x) = (0 <= x)$
⟨*proof*⟩

lemma *le-floor-eq-one* [*simp*]: $(1 <= floor\ x) = (1 <= x)$
⟨*proof*⟩

lemma *floor-less-eq*: $(floor\ x < a) = (x < real\ a)$
⟨*proof*⟩

lemma *floor-less-eq-number-of* [*simp*]:
$(floor\ x < number\text{-}of\ n) = (x < number\text{-}of\ n)$
⟨*proof*⟩

lemma *floor-less-eq-zero* [*simp*]: (*floor* x < *0*) = (x < *0*)
⟨*proof*⟩

lemma *floor-less-eq-one* [*simp*]: (*floor* x < *1*) = (x < *1*)
⟨*proof*⟩

lemma *less-floor-eq*: (a < *floor* x) = (*real* a + *1* <= x)
⟨*proof*⟩

lemma *less-floor-eq-number-of* [*simp*]:
(*number-of* n < *floor* x) = (*number-of* n + *1* <= x)
⟨*proof*⟩

lemma *less-floor-eq-zero* [*simp*]: (*0* < *floor* x) = (*1* <= x)
⟨*proof*⟩

lemma *less-floor-eq-one* [*simp*]: (*1* < *floor* x) = (*2* <= x)
⟨*proof*⟩

lemma *floor-le-eq*: (*floor* x <= a) = (x < *real* a + *1*)
⟨*proof*⟩

lemma *floor-le-eq-number-of* [*simp*]:
(*floor* x <= *number-of* n) = (x < *number-of* n + *1*)
⟨*proof*⟩

lemma *floor-le-eq-zero* [*simp*]: (*floor* x <= *0*) = (x < *1*)
⟨*proof*⟩

lemma *floor-le-eq-one* [*simp*]: (*floor* x <= *1*) = (x < *2*)
⟨*proof*⟩

lemma *floor-add* [*simp*]: *floor* (x + *real* a) = *floor* x + a
⟨*proof*⟩

lemma *floor-add-number-of* [*simp*]:
floor (x + *number-of* n) = *floor* x + *number-of* n
⟨*proof*⟩

lemma *floor-add-one* [*simp*]: *floor* (x + *1*) = *floor* x + *1*
⟨*proof*⟩

lemma *floor-subtract* [*simp*]: *floor* (x − *real* a) = *floor* x − a
⟨*proof*⟩

lemma *floor-subtract-number-of* [*simp*]: *floor* (x − *number-of* n) =
floor x − *number-of* n
⟨*proof*⟩

lemma *floor-subtract-one* [*simp*]: *floor* $(x - 1) =$ *floor* $x - 1$
⟨*proof*⟩

lemma *ceiling-zero* [*simp*]: *ceiling* $0 = 0$
⟨*proof*⟩

lemma *ceiling-real-of-nat* [*simp*]: *ceiling* (*real* (*n*::*nat*)) = *int n*
⟨*proof*⟩

lemma *ceiling-real-of-nat-zero* [*simp*]: *ceiling* (*real* (*0*::*nat*)) = *0*
⟨*proof*⟩

lemma *ceiling-floor* [*simp*]: *ceiling* (*real* (*floor r*)) = *floor r*
⟨*proof*⟩

lemma *floor-ceiling* [*simp*]: *floor* (*real* (*ceiling r*)) = *ceiling r*
⟨*proof*⟩

lemma *real-of-int-ceiling-ge* [*simp*]: $r \leq$ *real* (*ceiling r*)
⟨*proof*⟩

lemma *ceiling-mono*: $x < y$ ==> *ceiling* $x \leq$ *ceiling* y
⟨*proof*⟩

lemma *ceiling-mono2*: $x \leq y$ ==> *ceiling* $x \leq$ *ceiling* y
⟨*proof*⟩

lemma *real-of-int-ceiling-cancel* [*simp*]:
(*real* (*ceiling* x) = x) = ($\exists$ *n*::*int*. $x =$ *real n*)
⟨*proof*⟩

lemma *ceiling-eq*: [| *real* $n < x$; $x <$ *real* $n + 1$ |] ==> *ceiling* $x = n + 1$
⟨*proof*⟩

lemma *ceiling-eq2*: [| *real* $n < x$; $x \leq$ *real* $n + 1$ |] ==> *ceiling* $x = n + 1$
⟨*proof*⟩

lemma *ceiling-eq3*: [| *real* $n - 1 < x$; $x \leq$ *real* n |] ==> *ceiling* $x = n$
⟨*proof*⟩

lemma *ceiling-real-of-int* [*simp*]: *ceiling* (*real* (*n*::*int*)) = *n*
⟨*proof*⟩

lemma *ceiling-number-of-eq* [*simp*]:
ceiling (*number-of n* :: *real*) = (*number-of n*)
⟨*proof*⟩

lemma *ceiling-one* [*simp*]: *ceiling* $1 = 1$

⟨*proof*⟩

lemma *real-of-int-ceiling-diff-one-le* [*simp*]: *real* (*ceiling* r) − *1* ≤ r
⟨*proof*⟩

lemma *real-of-int-ceiling-le-add-one* [*simp*]: *real* (*ceiling* r) ≤ r + *1*
⟨*proof*⟩

lemma *ceiling-le*: *x* <= *real a* ==> *ceiling x* <= *a*
⟨*proof*⟩

lemma *ceiling-le-real*: *ceiling x* <= *a* ==> *x* <= *real a*
⟨*proof*⟩

lemma *ceiling-le-eq*: (*ceiling x* <= *a*) = (*x* <= *real a*)
⟨*proof*⟩

lemma *ceiling-le-eq-number-of* [*simp*]:
(*ceiling x* <= *number-of n*) = (*x* <= *number-of n*)
⟨*proof*⟩

lemma *ceiling-le-zero-eq* [*simp*]: (*ceiling x* <= *0*) = (*x* <= *0*)
⟨*proof*⟩

lemma *ceiling-le-eq-one* [*simp*]: (*ceiling x* <= *1*) = (*x* <= *1*)
⟨*proof*⟩

lemma *less-ceiling-eq*: (*a* < *ceiling x*) = (*real a* < *x*)
⟨*proof*⟩

lemma *less-ceiling-eq-number-of* [*simp*]:
(*number-of n* < *ceiling x*) = (*number-of n* < *x*)
⟨*proof*⟩

lemma *less-ceiling-eq-zero* [*simp*]: (*0* < *ceiling x*) = (*0* < *x*)
⟨*proof*⟩

lemma *less-ceiling-eq-one* [*simp*]: (*1* < *ceiling x*) = (*1* < *x*)
⟨*proof*⟩

lemma *ceiling-less-eq*: (*ceiling x* < *a*) = (*x* <= *real a* − *1*)
⟨*proof*⟩

lemma *ceiling-less-eq-number-of* [*simp*]:
(*ceiling x* < *number-of n*) = (*x* <= *number-of n* − *1*)
⟨*proof*⟩

lemma *ceiling-less-eq-zero* [*simp*]: (*ceiling x* < *0*) = (*x* <= −*1*)
⟨*proof*⟩

lemma *ceiling-less-eq-one* [*simp*]: (*ceiling* x < *1*) = (x <= *0*)
⟨*proof*⟩

lemma *le-ceiling-eq*: (a <= *ceiling* x) = (*real* a − *1* < x)
⟨*proof*⟩

lemma *le-ceiling-eq-number-of* [*simp*]:
(*number-of* n <= *ceiling* x) = (*number-of* n − *1* < x)
⟨*proof*⟩

lemma *le-ceiling-eq-zero* [*simp*]: (*0* <= *ceiling* x) = (−*1* < x)
⟨*proof*⟩

lemma *le-ceiling-eq-one* [*simp*]: (*1* <= *ceiling* x) = (*0* < x)
⟨*proof*⟩

lemma *ceiling-add* [*simp*]: *ceiling* (x + *real* a) = *ceiling* x + a
⟨*proof*⟩

lemma *ceiling-add-number-of* [*simp*]: *ceiling* (x + *number-of* n) =
ceiling x + *number-of* n
⟨*proof*⟩

lemma *ceiling-add-one* [*simp*]: *ceiling* (x + *1*) = *ceiling* x + *1*
⟨*proof*⟩

lemma *ceiling-subtract* [*simp*]: *ceiling* (x − *real* a) = *ceiling* x − a
⟨*proof*⟩

lemma *ceiling-subtract-number-of* [*simp*]: *ceiling* (x − *number-of* n) =
ceiling x − *number-of* n
⟨*proof*⟩

lemma *ceiling-subtract-one* [*simp*]: *ceiling* (x − *1*) = *ceiling* x − *1*
⟨*proof*⟩

17.4 Versions for the natural numbers

definition
natfloor :: *real* => *nat* **where**
natfloor x = *nat*(*floor* x)

definition
natceiling :: *real* => *nat* **where**
natceiling x = *nat*(*ceiling* x)

lemma *natfloor-zero* [*simp*]: *natfloor* *0* = *0*
⟨*proof*⟩

lemma *natfloor-one* [*simp*]: *natfloor 1* = *1*
⟨*proof*⟩

lemma *zero-le-natfloor* [*simp*]: *0* <= *natfloor x*
⟨*proof*⟩

lemma *natfloor-number-of-eq* [*simp*]: *natfloor* (*number-of n*) = *number-of n*
⟨*proof*⟩

lemma *natfloor-real-of-nat* [*simp*]: *natfloor*(*real n*) = *n*
⟨*proof*⟩

lemma *real-natfloor-le*: *0* <= *x* ==> *real*(*natfloor x*) <= *x*
⟨*proof*⟩

lemma *natfloor-neg*: *x* <= *0* ==> *natfloor x* = *0*
⟨*proof*⟩

lemma *natfloor-mono*: *x* <= *y* ==> *natfloor x* <= *natfloor y*
⟨*proof*⟩

lemma *le-natfloor*: *real x* <= *a* ==> *x* <= *natfloor a*
⟨*proof*⟩

lemma *le-natfloor-eq*: *0* <= *x* ==> (*a* <= *natfloor x*) = (*real a* <= *x*)
⟨*proof*⟩

lemma *le-natfloor-eq-number-of* [*simp*]:
~ *neg*((*number-of n*)::*int*) ==> *0* <= *x* ==>
(*number-of n* <= *natfloor x*) = (*number-of n* <= *x*)
⟨*proof*⟩

lemma *le-natfloor-eq-one* [*simp*]: (*1* <= *natfloor x*) = (*1* <= *x*)
⟨*proof*⟩

lemma *natfloor-eq*: *real n* <= *x* ==> *x* < *real n* + *1* ==> *natfloor x* = *n*
⟨*proof*⟩

lemma *real-natfloor-add-one-gt*: *x* < *real*(*natfloor x*) + *1*
⟨*proof*⟩

lemma *real-natfloor-gt-diff-one*: *x* − *1* < *real*(*natfloor x*)
⟨*proof*⟩

lemma *ge-natfloor-plus-one-imp-gt*: *natfloor z* + *1* <= *n* ==> *z* < *real n*
⟨*proof*⟩

lemma *natfloor-add* [*simp*]: *0* <= *x* ==> *natfloor* (*x* + *real a*) = *natfloor x* + *a*

⟨*proof*⟩

lemma *natfloor-add-number-of* [*simp*]:
  ~*neg* ((*number-of* *n*)::*int*) ==> *0* <= *x* ==>
    natfloor (*x* + *number-of* *n*) = *natfloor* *x* + *number-of* *n*
⟨*proof*⟩

lemma *natfloor-add-one*: *0* <= *x* ==> *natfloor*(*x* + *1*) = *natfloor* *x* + *1*
⟨*proof*⟩

lemma *natfloor-subtract* [*simp*]: *real* *a* <= *x* ==>
  natfloor(*x* − *real* *a*) = *natfloor* *x* − *a*
⟨*proof*⟩

lemma *natceiling-zero* [*simp*]: *natceiling* *0* = *0*
⟨*proof*⟩

lemma *natceiling-one* [*simp*]: *natceiling* *1* = *1*
⟨*proof*⟩

lemma *zero-le-natceiling* [*simp*]: *0* <= *natceiling* *x*
⟨*proof*⟩

lemma *natceiling-number-of-eq* [*simp*]: *natceiling* (*number-of* *n*) = *number-of* *n*
⟨*proof*⟩

lemma *natceiling-real-of-nat* [*simp*]: *natceiling*(*real* *n*) = *n*
⟨*proof*⟩

lemma *real-natceiling-ge*: *x* <= *real*(*natceiling* *x*)
⟨*proof*⟩

lemma *natceiling-neg*: *x* <= *0* ==> *natceiling* *x* = *0*
⟨*proof*⟩

lemma *natceiling-mono*: *x* <= *y* ==> *natceiling* *x* <= *natceiling* *y*
⟨*proof*⟩

lemma *natceiling-le*: *x* <= *real* *a* ==> *natceiling* *x* <= *a*
⟨*proof*⟩

lemma *natceiling-le-eq*: *0* <= *x* ==> (*natceiling* *x* <= *a*) = (*x* <= *real* *a*)
⟨*proof*⟩

lemma *natceiling-le-eq-number-of* [*simp*]:
  ~ *neg*((*number-of* *n*)::*int*) ==> *0* <= *x* ==>
    (*natceiling* *x* <= *number-of* *n*) = (*x* <= *number-of* *n*)
⟨*proof*⟩

lemma *natceiling-le-eq-one*: (*natceiling x* <= *1*) = (*x* <= *1*)
⟨*proof*⟩

lemma *natceiling-eq*: *real n* < *x* ==> *x* <= *real n* + *1* ==> *natceiling x* = *n* + *1*
⟨*proof*⟩

lemma *natceiling-add* [*simp*]: *0* <= *x* ==>
natceiling (*x* + *real a*) = *natceiling x* + *a*
⟨*proof*⟩

lemma *natceiling-add-number-of* [*simp*]:
~ *neg* ((*number-of n*)::*int*) ==> *0* <= *x* ==>
natceiling (*x* + *number-of n*) = *natceiling x* + *number-of n*
⟨*proof*⟩

lemma *natceiling-add-one*: *0* <= *x* ==> *natceiling*(*x* + *1*) = *natceiling x* + *1*
⟨*proof*⟩

lemma *natceiling-subtract* [*simp*]: *real a* <= *x* ==>
natceiling(*x* − *real a*) = *natceiling x* − *a*
⟨*proof*⟩

lemma *natfloor-div-nat*: *1* <= *x* ==> *y* > *0* ==>
natfloor (*x* / *real y*) = *natfloor x div y*
⟨*proof*⟩

end

18 Non-denumerability of the Continuum.

theory *ContNotDenum*
imports *RComplete*
begin

18.1 Abstract

The following document presents a proof that the Continuum is uncountable. It is formalised in the Isabelle/Isar theorem proving system.

Theorem: The Continuum $\mathbb{R}$ is not denumerable. In other words, there does not exist a function $f:\mathbb{N}\Rightarrow\mathbb{R}$ such that f is surjective.

Outline: An elegant informal proof of this result uses Cantor's Diagonalisation argument. The proof presented here is not this one. First we formalise some properties of closed intervals, then we prove the Nested Interval Property. This property relies on the completeness of the Real numbers and is the foundation for our argument. Informally it states that an intersection of countable closed intervals (where each successive interval is a subset of the last) is non-empty. We then assume a surjective function f:$\mathbb{N} \Rightarrow \mathbb{R}$ exists and find a real x such that x is not in the range of f by generating a sequence of closed intervals then using the NIP.

18.2 Closed Intervals

This section formalises some properties of closed intervals.

18.2.1 Definition

definition
closed-int :: *real* $\Rightarrow$ *real* $\Rightarrow$ *real set* **where**
closed-int $x\ y = \{z.\ x \leq z \wedge z \leq y\}$

18.2.2 Properties

lemma *closed-int-subset*:
assumes *xy*: $x1 \geq x0$ $y1 \leq y0$
shows *closed-int* $x1\ y1 \subseteq$ *closed-int* $x0\ y0$
⟨*proof*⟩

lemma *closed-int-least*:
assumes *a*: $a \leq b$
shows $a \in$ *closed-int* $a\ b \wedge (\forall x \in$ *closed-int* $a\ b.\ a \leq x)$
⟨*proof*⟩

lemma *closed-int-most*:
assumes *a*: $a \leq b$
shows $b \in$ *closed-int* $a\ b \wedge (\forall x \in$ *closed-int* $a\ b.\ x \leq b)$
⟨*proof*⟩

lemma *closed-not-empty*:
shows $a \leq b \Longrightarrow \exists x.\ x \in$ *closed-int* $a\ b$
⟨*proof*⟩

lemma *closed-mem*:
assumes $a \leq c$ **and** $c \leq b$
shows $c \in$ *closed-int* $a\ b$
⟨*proof*⟩

lemma *closed-subset*:
assumes *ac*: $a \leq b$ $c \leq d$
assumes *closed*: *closed-int* $a\ b \subseteq$ *closed-int* $c\ d$
shows $b \geq c$
⟨*proof*⟩

18.3 Nested Interval Property

theorem *NIP*:
  fixes $f::nat \Rightarrow real\ set$
  assumes *subset*: $\forall n.\ f\ (Suc\ n) \subseteq f\ n$
  and *closed*: $\forall n.\ \exists a\ b.\ f\ n = closed\text{-}int\ a\ b \wedge a \leq b$
  shows $(\bigcap n.\ f\ n) \neq \{\}$
$\langle proof \rangle$

18.4 Generating the intervals

18.4.1 Existence of non-singleton closed intervals

This lemma asserts that given any non-singleton closed interval (a,b) and any element c, there exists a closed interval that is a subset of (a,b) and that does not contain c and is a non-singleton itself.

lemma *closed-subset-ex*:
  fixes $c::real$
  assumes *alb*: $a < b$
  shows
    $\exists ka\ kb.\ ka < kb \wedge closed\text{-}int\ ka\ kb \subseteq closed\text{-}int\ a\ b \wedge c \notin (closed\text{-}int\ ka\ kb)$
$\langle proof \rangle$

18.5 newInt: Interval generation

Given a function f:$\mathbb{N}\Rightarrow\mathbb{R}$, newInt (Suc n) f returns a closed interval such that *newInt* $(Suc\ n)\ f \subseteq newInt\ n\ f$ and does not contain $f\ (Suc\ n)$. With the base case defined such that $(f\ 0) \notin newInt\ 0\ f$.

18.5.1 Definition

consts $newInt :: nat \Rightarrow (nat \Rightarrow real) \Rightarrow (real\ set)$
primrec
$newInt\ 0\ f = closed\text{-}int\ (f\ 0 + 1)\ (f\ 0 + 2)$
$newInt\ (Suc\ n)\ f =$
  $(SOME\ e.\ (\exists e1\ e2.$
   $e1 < e2 \wedge$
   $e = closed\text{-}int\ e1\ e2 \wedge$
   $e \subseteq (newInt\ n\ f) \wedge$
   $(f\ (Suc\ n)) \notin e)$
)

18.5.2 Properties

We now show that every application of newInt returns an appropriate interval.

lemma *newInt-ex*:
  $\exists a\ b.\ a < b \wedge$

newInt (*Suc* *n*) *f* = *closed-int* *a* *b* ∧
newInt (*Suc* *n*) *f* ⊆ *newInt* *n* *f* ∧
f (*Suc* *n*) ∉ *newInt* (*Suc* *n*) *f*
⟨*proof*⟩

lemma *newInt-subset*:
newInt (*Suc* *n*) *f* ⊆ *newInt* *n* *f*
⟨*proof*⟩

Another fundamental property is that no element in the range of f is in the intersection of all closed intervals generated by newInt.

lemma *newInt-inter*:
∀ *n*. *f* *n* ∉ (⋂ *n*. *newInt* *n* *f*)
⟨*proof*⟩

lemma *newInt-notempty*:
(⋂ *n*. *newInt* *n* *f*) ≠ {}
⟨*proof*⟩

18.6 Final Theorem

theorem *real-non-denum*:
shows ¬ (∃ *f*::*nat*⇒*real*. *surj* *f*)
⟨*proof*⟩

end

19 Natural powers theory

theory *RealPow*
imports *RealDef*
begin

declare *abs-mult-self* [*simp*]

instantiation *real* :: *recpower*
begin

primrec *power-real* **where**
realpow-0: *r* ^ *0* = (*1*::*real*)
| *realpow-Suc*: *r* ^ *Suc* *n* = (*r*::*real*) ∗ *r* ^ *n*

instance ⟨*proof*⟩

end

lemma *two-realpow-ge-one* [*simp*]: $(1::real) \leq 2 \hat{} n$
⟨*proof*⟩

lemma *two-realpow-gt* [*simp*]: $real\ (n::nat) < 2 \hat{} n$
⟨*proof*⟩

lemma *realpow-Suc-le-self*: $[|\ 0 \leq r;\ r \leq (1::real)\ |] ==> r \hat{} Suc\ n \leq r$
⟨*proof*⟩

lemma *realpow-minus-mult* [*rule-format*]:
$0 < n --> (x::real) \hat{} (n - 1) * x = x \hat{} n$
⟨*proof*⟩

lemma *realpow-two-mult-inverse* [*simp*]:
$r \neq 0 ==> r * inverse\ r\ \hat{}Suc\ (Suc\ 0) = inverse\ (r::real)$
⟨*proof*⟩

lemma *realpow-two-minus* [*simp*]: $(-x)\hat{}Suc\ (Suc\ 0) = (x::real)\hat{}Suc\ (Suc\ 0)$
⟨*proof*⟩

lemma *realpow-two-diff*:
$(x::real)\hat{}Suc\ (Suc\ 0) - y\hat{}Suc\ (Suc\ 0) = (x - y) * (x + y)$
⟨*proof*⟩

lemma *realpow-two-disj*:
$((x::real)\hat{}Suc\ (Suc\ 0) = y\hat{}Suc\ (Suc\ 0)) = (x = y\ |\ x = -y)$
⟨*proof*⟩

lemma *realpow-real-of-nat*: $real\ (m::nat) \hat{} n = real\ (m \hat{} n)$
⟨*proof*⟩

lemma *realpow-real-of-nat-two-pos* [*simp*] : $0 < real\ (Suc\ (Suc\ 0) \hat{} n)$
⟨*proof*⟩

lemma *realpow-increasing*:
$[|(0::real) \leq x;\ 0 \leq y;\ x \hat{} Suc\ n \leq y \hat{} Suc\ n|] ==> x \leq y$
⟨*proof*⟩

19.1 Literal Arithmetic Involving Powers, Type *real*

lemma *real-of-int-power*: $real\ (x::int) \hat{} n = real\ (x \hat{} n)$
⟨*proof*⟩
declare *real-of-int-power* [*symmetric*, *simp*]

lemma *power-real-number-of*:
$(number\text{-}of\ v :: real) \hat{} n = real\ ((number\text{-}of\ v :: int) \hat{} n)$
⟨*proof*⟩

declare *power-real-number-of* [*of* - *number-of* *w*, *standard*, *simp*]

19.2 Properties of Squares

lemma *sum-squares-ge-zero*:
fixes $x\ y$ :: $'a$::*ordered-ring-strict*
shows $0 \leq x * x + y * y$
⟨*proof*⟩

lemma *not-sum-squares-lt-zero*:
fixes $x\ y$ :: $'a$::*ordered-ring-strict*
shows $\neg\ x * x + y * y < 0$
⟨*proof*⟩

lemma *sum-nonneg-eq-zero-iff*:
fixes $x\ y$ :: $'a$::*pordered-ab-group-add*
assumes x: $0 \leq x$ **and** y: $0 \leq y$
shows $(x + y = 0) = (x = 0 \wedge y = 0)$
⟨*proof*⟩

lemma *sum-squares-eq-zero-iff*:
fixes $x\ y$ :: $'a$::*ordered-ring-strict*
shows $(x * x + y * y = 0) = (x = 0 \wedge y = 0)$
⟨*proof*⟩

lemma *sum-squares-le-zero-iff*:
fixes $x\ y$ :: $'a$::*ordered-ring-strict*
shows $(x * x + y * y \leq 0) = (x = 0 \wedge y = 0)$
⟨*proof*⟩

lemma *sum-squares-gt-zero-iff*:
fixes $x\ y$ :: $'a$::*ordered-ring-strict*
shows $(0 < x * x + y * y) = (x \neq 0 \vee y \neq 0)$
⟨*proof*⟩

lemma *sum-power2-ge-zero*:
fixes $x\ y$ :: $'a$::{*ordered-idom*,*recpower*}
shows $0 \leq x^2 + y^2$
⟨*proof*⟩

lemma *not-sum-power2-lt-zero*:
fixes $x\ y$ :: $'a$::{*ordered-idom*,*recpower*}
shows $\neg\ x^2 + y^2 < 0$
⟨*proof*⟩

lemma *sum-power2-eq-zero-iff*:
fixes $x\ y$ :: $'a$::{*ordered-idom*,*recpower*}
shows $(x^2 + y^2 = 0) = (x = 0 \wedge y = 0)$
⟨*proof*⟩

lemma *sum-power2-le-zero-iff*:
  fixes $x\ y :: 'a::\{ordered\text{-}idom,recpower\}$
  shows $(x^2 + y^2 \leq 0) = (x = 0 \wedge y = 0)$
$\langle proof \rangle$

lemma *sum-power2-gt-zero-iff*:
  fixes $x\ y :: 'a::\{ordered\text{-}idom,recpower\}$
  shows $(0 < x^2 + y^2) = (x \neq 0 \vee y \neq 0)$
$\langle proof \rangle$

19.3 Squares of Reals

lemma *real-two-squares-add-zero-iff* [*simp*]:
  $(x * x + y * y = 0) = ((x::real) = 0 \wedge y = 0)$
$\langle proof \rangle$

lemma *real-sum-squares-cancel*: $x * x + y * y = 0 ==> x = (0::real)$
$\langle proof \rangle$

lemma *real-sum-squares-cancel2*: $x * x + y * y = 0 ==> y = (0::real)$
$\langle proof \rangle$

lemma *real-mult-self-sum-ge-zero*: $(0::real) \leq x*x + y*y$
$\langle proof \rangle$

lemma *real-sum-squares-cancel-a*: $x * x = -(y * y) ==> x = (0::real)\ \&\ y=0$
$\langle proof \rangle$

lemma *real-squared-diff-one-factored*: $x*x - (1::real) = (x + 1)*(x - 1)$
$\langle proof \rangle$

lemma *real-mult-is-one* [*simp*]: $(x*x = (1::real)) = (x = 1 \mid x = - 1)$
$\langle proof \rangle$

lemma *real-sum-squares-not-zero*: $x\ {}^{\sim}= 0 ==> x * x + y * y\ {}^{\sim}= (0::real)$
$\langle proof \rangle$

lemma *real-sum-squares-not-zero2*: $y\ {}^{\sim}= 0 ==> x * x + y * y\ {}^{\sim}= (0::real)$
$\langle proof \rangle$

lemma *realpow-two-sum-zero-iff* [*simp*]:
    $(x\ \hat{}\ 2 + y\ \hat{}\ 2 = (0::real)) = (x = 0\ \&\ y = 0)$
$\langle proof \rangle$

lemma *realpow-two-le-add-order* [*simp*]: $(0::real) \leq u\ \hat{}\ 2 + v\ \hat{}\ 2$
$\langle proof \rangle$

lemma *realpow-two-le-add-order2* [*simp*]: $(0::real) \leq u\ \hat{}\ 2 + v\ \hat{}\ 2 + w\ \hat{}\ 2$

$\langle proof \rangle$

lemma *real-sum-square-gt-zero*: $x \sim= 0 ==> (0::real) < x * x + y * y$
$\langle proof \rangle$

lemma *real-sum-square-gt-zero2*: $y \sim= 0 ==> (0::real) < x * x + y * y$
$\langle proof \rangle$

lemma *real-minus-mult-self-le* [*simp*]: $-(u * u) \leq (x * (x::real))$
$\langle proof \rangle$

lemma *realpow-square-minus-le* [*simp*]: $-(u \ \hat{} \ 2) \leq (x::real) \ \hat{} \ 2$
$\langle proof \rangle$

lemma *real-sq-order*:
  fixes $x::real$
  assumes *xgt0*: $0 \leq x$ **and** *ygt0*: $0 \leq y$ **and** *sq*: $x\hat{}2 \leq y\hat{}2$
  shows $x \leq y$
$\langle proof \rangle$

19.4 Various Other Theorems

lemma *real-le-add-half-cancel*: $(x + y/2 \leq (y::real)) = (x \leq y /2)$
$\langle proof \rangle$

lemma *real-minus-half-eq* [*simp*]: $(x::real) - x/2 = x/2$
$\langle proof \rangle$

lemma *real-mult-inverse-cancel*:
  $[|(0::real) < x; 0 < x1; x1 * y < x * u |]$
  $==> inverse\ x * y < inverse\ x1 * u$
$\langle proof \rangle$

lemma *real-mult-inverse-cancel2*:
  $[|(0::real) < x;0 < x1; x1 * y < x * u |] ==> y * inverse\ x < u * inverse\ x1$
$\langle proof \rangle$

lemma *inverse-real-of-nat-gt-zero* [*simp*]: $0 < inverse\ (real\ (Suc\ n))$
$\langle proof \rangle$

lemma *inverse-real-of-nat-ge-zero* [*simp*]: $0 \leq inverse\ (real\ (Suc\ n))$
$\langle proof \rangle$

lemma *realpow-num-eq-if*: $(m::real) \ \hat{} \ n = (if\ n=0\ then\ 1\ else\ m * m \ \hat{} \ (n - 1))$
$\langle proof \rangle$

end

20 Vector Spaces and Algebras over the Reals

theory *RealVector*
imports *RealPow*
begin

20.1 Locale for additive functions

locale *additive* =
  fixes f :: $'a$::*ab-group-add* ⇒ $'b$::*ab-group-add*
  assumes *add*: $f\ (x + y) = f\ x + f\ y$

lemma (**in** *additive*) *zero*: $f\ 0 = 0$
⟨*proof*⟩

lemma (**in** *additive*) *minus*: $f\ (-\ x) = -\ f\ x$
⟨*proof*⟩

lemma (**in** *additive*) *diff*: $f\ (x - y) = f\ x - f\ y$
⟨*proof*⟩

lemma (**in** *additive*) *setsum*: f (*setsum* g A) = ($\sum x{\in}A.\ f\ (g\ x)$)
⟨*proof*⟩

20.2 Real vector spaces

class *scaleR* = *type* +
  fixes *scaleR* :: *real* ⇒ $'a$ ⇒ $'a$ (**infixr** $*_R$ *75*)
begin

abbreviation
  divideR :: $'a$ ⇒ *real* ⇒ $'a$ (**infixl** $'/_R$ *70*)
where
  $x\ /_R\ r$ == *scaleR* (*inverse* r) x

end

instantiation *real* :: *scaleR*
begin

definition
  real-scaleR-def [*simp*]: *scaleR* $a\ x = a * x$

instance ⟨*proof*⟩

end

class *real-vector* = *scaleR* + *ab-group-add* +
  assumes *scaleR-right-distrib*: *scaleR* $a\ (x + y)$ = *scaleR* $a\ x$ + *scaleR* $a\ y$
  and *scaleR-left-distrib*: *scaleR* $(a + b)\ x$ = *scaleR* $a\ x$ + *scaleR* $b\ x$

and *scaleR-scaleR* [*simp*]: *scaleR a* (*scaleR b x*) = *scaleR* (*a* ∗ *b*) *x*
and *scaleR-one* [*simp*]: *scaleR 1 x* = *x*

class *real-algebra* = *real-vector* + *ring* +
assumes *mult-scaleR-left* [*simp*]: *scaleR a x* ∗ *y* = *scaleR a* (*x* ∗ *y*)
and *mult-scaleR-right* [*simp*]: *x* ∗ *scaleR a y* = *scaleR a* (*x* ∗ *y*)

class *real-algebra-1* = *real-algebra* + *ring-1*

class *real-div-algebra* = *real-algebra-1* + *division-ring*

class *real-field* = *real-div-algebra* + *field*

instance *real* :: *real-field*
⟨*proof*⟩

lemma *scaleR-left-commute*:
fixes *x* :: *'a::real-vector*
shows *scaleR a* (*scaleR b x*) = *scaleR b* (*scaleR a x*)
⟨*proof*⟩

interpretation *scaleR-left*: *additive* [(λ*a*. *scaleR a x*::*'a*::*real-vector*)]
⟨*proof*⟩

interpretation *scaleR-right*: *additive* [(λ*x*. *scaleR a x*::*'a*::*real-vector*)]
⟨*proof*⟩

lemmas *scaleR-zero-left* [*simp*] = *scaleR-left.zero*

lemmas *scaleR-zero-right* [*simp*] = *scaleR-right.zero*

lemmas *scaleR-minus-left* [*simp*] = *scaleR-left.minus*

lemmas *scaleR-minus-right* [*simp*] = *scaleR-right.minus*

lemmas *scaleR-left-diff-distrib* = *scaleR-left.diff*

lemmas *scaleR-right-diff-distrib* = *scaleR-right.diff*

lemma *scaleR-eq-0-iff* [*simp*]:
fixes *x* :: *'a::real-vector*
shows (*scaleR a x* = *0*) = (*a* = *0* ∨ *x* = *0*)
⟨*proof*⟩

lemma *scaleR-left-imp-eq*:
fixes *x y* :: *'a::real-vector*
shows ⟦*a* ≠ *0*; *scaleR a x* = *scaleR a y*⟧ ⟹ *x* = *y*
⟨*proof*⟩

lemma *scaleR-right-imp-eq*:
fixes $x\ y$ :: $'a$::*real-vector*
shows $[\![x \neq 0;\ scaleR\ a\ x = scaleR\ b\ x]\!] \Longrightarrow a = b$
⟨*proof*⟩

lemma *scaleR-cancel-left*:
fixes $x\ y$ :: $'a$::*real-vector*
shows $(scaleR\ a\ x = scaleR\ a\ y) = (x = y \vee a = 0)$
⟨*proof*⟩

lemma *scaleR-cancel-right*:
fixes $x\ y$ :: $'a$::*real-vector*
shows $(scaleR\ a\ x = scaleR\ b\ x) = (a = b \vee x = 0)$
⟨*proof*⟩

lemma *nonzero-inverse-scaleR-distrib*:
fixes x :: $'a$::*real-div-algebra* **shows**
$[\![a \neq 0;\ x \neq 0]\!] \Longrightarrow inverse\ (scaleR\ a\ x) = scaleR\ (inverse\ a)\ (inverse\ x)$
⟨*proof*⟩

lemma *inverse-scaleR-distrib*:
fixes x :: $'a$::{*real-div-algebra*,*division-by-zero*}
shows $inverse\ (scaleR\ a\ x) = scaleR\ (inverse\ a)\ (inverse\ x)$
⟨*proof*⟩

20.3 Embedding of the Reals into any *real-algebra-1*: *of-real*

definition
of-real :: *real* ⇒ $'a$::*real-algebra-1* **where**
of-real $r = scaleR\ r\ 1$

lemma *scaleR-conv-of-real*: $scaleR\ r\ x = \textit{of-real}\ r * x$
⟨*proof*⟩

lemma *of-real-0* [*simp*]: *of-real* $0 = 0$
⟨*proof*⟩

lemma *of-real-1* [*simp*]: *of-real* $1 = 1$
⟨*proof*⟩

lemma *of-real-add* [*simp*]: *of-real* $(x + y) =$ *of-real* $x +$ *of-real* y
⟨*proof*⟩

lemma *of-real-minus* [*simp*]: *of-real* $(-\ x) = -$ *of-real* x
⟨*proof*⟩

lemma *of-real-diff* [*simp*]: *of-real* $(x - y) =$ *of-real* $x -$ *of-real* y
⟨*proof*⟩

lemma *of-real-mult* [*simp*]: *of-real* $(x * y)$ = *of-real* x * *of-real* y
⟨*proof*⟩

lemma *nonzero-of-real-inverse*:
  $x \neq 0 \Longrightarrow$ *of-real* (*inverse* x) =
    inverse (*of-real* x :: $'a$::*real-div-algebra*)
⟨*proof*⟩

lemma *of-real-inverse* [*simp*]:
  of-real (*inverse* x) =
    inverse (*of-real* x :: $'a$::{*real-div-algebra*,*division-by-zero*})
⟨*proof*⟩

lemma *nonzero-of-real-divide*:
  $y \neq 0 \Longrightarrow$ *of-real* $(x \mathbin{/} y)$ =
    (*of-real* x / *of-real* y :: $'a$::*real-field*)
⟨*proof*⟩

lemma *of-real-divide* [*simp*]:
  of-real $(x \mathbin{/} y)$ =
    (*of-real* x / *of-real* y :: $'a$::{*real-field*,*division-by-zero*})
⟨*proof*⟩

lemma *of-real-power* [*simp*]:
  of-real $(x \mathbin{\hat{}} n)$ = (*of-real* x :: $'a$::{*real-algebra-1*,*recpower*}) ˆ n
⟨*proof*⟩

lemma *of-real-eq-iff* [*simp*]: (*of-real* x = *of-real* y) = $(x = y)$
⟨*proof*⟩

lemmas *of-real-eq-0-iff* [*simp*] = *of-real-eq-iff* [*of - 0*, *simplified*]

lemma *of-real-eq-id* [*simp*]: *of-real* = (*id* :: *real* ⇒ *real*)
⟨*proof*⟩

Collapse nested embeddings

lemma *of-real-of-nat-eq* [*simp*]: *of-real* (*of-nat* n) = *of-nat* n
⟨*proof*⟩

lemma *of-real-of-int-eq* [*simp*]: *of-real* (*of-int* z) = *of-int* z
⟨*proof*⟩

lemma *of-real-number-of-eq*:
  of-real (*number-of* w) = (*number-of* w :: $'a$::{*number-ring*,*real-algebra-1*})
⟨*proof*⟩

Every real algebra has characteristic zero

instance *real-algebra-1* < *ring-char-0*
⟨*proof*⟩

20.4 The Set of Real Numbers

definition
Reals :: $'a$::*real-algebra-1 set* **where**
Reals ≡ *range of-real*

notation (*xsymbols*)
Reals (ℝ)

lemma *Reals-of-real* [*simp*]: *of-real r* ∈ *Reals*
⟨*proof*⟩

lemma *Reals-of-int* [*simp*]: *of-int z* ∈ *Reals*
⟨*proof*⟩

lemma *Reals-of-nat* [*simp*]: *of-nat n* ∈ *Reals*
⟨*proof*⟩

lemma *Reals-number-of* [*simp*]:
(*number-of w*::$'a$::{*number-ring*,*real-algebra-1*}) ∈ *Reals*
⟨*proof*⟩

lemma *Reals-0* [*simp*]: *0* ∈ *Reals*
⟨*proof*⟩

lemma *Reals-1* [*simp*]: *1* ∈ *Reals*
⟨*proof*⟩

lemma *Reals-add* [*simp*]: ⟦*a* ∈ *Reals*; *b* ∈ *Reals*⟧ ⟹ *a* + *b* ∈ *Reals*
⟨*proof*⟩

lemma *Reals-minus* [*simp*]: *a* ∈ *Reals* ⟹ − *a* ∈ *Reals*
⟨*proof*⟩

lemma *Reals-diff* [*simp*]: ⟦*a* ∈ *Reals*; *b* ∈ *Reals*⟧ ⟹ *a* − *b* ∈ *Reals*
⟨*proof*⟩

lemma *Reals-mult* [*simp*]: ⟦*a* ∈ *Reals*; *b* ∈ *Reals*⟧ ⟹ *a* ∗ *b* ∈ *Reals*
⟨*proof*⟩

lemma *nonzero-Reals-inverse*:
fixes *a* :: $'a$::*real-div-algebra*
shows ⟦*a* ∈ *Reals*; *a* ≠ *0*⟧ ⟹ *inverse a* ∈ *Reals*
⟨*proof*⟩

lemma *Reals-inverse* [*simp*]:
fixes *a* :: $'a$::{*real-div-algebra*,*division-by-zero*}
shows *a* ∈ *Reals* ⟹ *inverse a* ∈ *Reals*
⟨*proof*⟩

lemma *nonzero-Reals-divide*:
  fixes $a\ b :: 'a::real\text{-}field$
  shows $[\![a \in Reals;\ b \in Reals;\ b \neq 0]\!] \Longrightarrow a\ /\ b \in Reals$
⟨*proof*⟩

lemma *Reals-divide* [*simp*]:
  fixes $a\ b :: 'a::\{real\text{-}field,division\text{-}by\text{-}zero\}$
  shows $[\![a \in Reals;\ b \in Reals]\!] \Longrightarrow a\ /\ b \in Reals$
⟨*proof*⟩

lemma *Reals-power* [*simp*]:
  fixes $a :: 'a::\{real\text{-}algebra\text{-}1,recpower\}$
  shows $a \in Reals \Longrightarrow a\ \hat{}\ n \in Reals$
⟨*proof*⟩

lemma *Reals-cases* [*cases set*: *Reals*]:
  assumes $q \in \mathbb{R}$
  obtains (*of-real*) r **where** $q = of\text{-}real\ r$
  ⟨*proof*⟩

lemma *Reals-induct* [*case-names of-real*, *induct set*: *Reals*]:
  $q \in \mathbb{R} \Longrightarrow (\bigwedge r.\ P\ (of\text{-}real\ r)) \Longrightarrow P\ q$
  ⟨*proof*⟩

20.5 Real normed vector spaces

class $norm = type\ +$
  fixes $norm :: 'a \Rightarrow real$

instantiation $real :: norm$
begin

definition
  real-norm-def [*simp*]: $norm\ r \equiv |r|$

instance ⟨*proof*⟩

end

class $sgn\text{-}div\text{-}norm = scaleR + norm + sgn\ +$
  assumes *sgn-div-norm*: $sgn\ x = x\ /_R\ norm\ x$

class $real\text{-}normed\text{-}vector = real\text{-}vector + sgn\text{-}div\text{-}norm\ +$
  assumes *norm-ge-zero* [*simp*]: $0 \leq norm\ x$
  and *norm-eq-zero* [*simp*]: $norm\ x = 0 \longleftrightarrow x = 0$
  and *norm-triangle-ineq*: $norm\ (x + y) \leq norm\ x + norm\ y$
  and *norm-scaleR*: $norm\ (scaleR\ a\ x) = |a| * norm\ x$

class $real\text{-}normed\text{-}algebra = real\text{-}algebra + real\text{-}normed\text{-}vector\ +$

assumes *norm-mult-ineq*: $norm\ (x * y) \le norm\ x * norm\ y$

class *real-normed-algebra-1* = *real-algebra-1* + *real-normed-algebra* +
assumes *norm-one* [*simp*]: $norm\ 1 = 1$

class *real-normed-div-algebra* = *real-div-algebra* + *real-normed-vector* +
assumes *norm-mult*: $norm\ (x * y) = norm\ x * norm\ y$

class *real-normed-field* = *real-field* + *real-normed-div-algebra*

instance *real-normed-div-algebra* < *real-normed-algebra-1*
⟨*proof*⟩

instance *real* :: *real-normed-field*
⟨*proof*⟩

lemma *norm-zero* [*simp*]: $norm\ (0::'a::real\text{-}normed\text{-}vector) = 0$
⟨*proof*⟩

lemma *zero-less-norm-iff* [*simp*]:
fixes $x :: 'a::real\text{-}normed\text{-}vector$
shows $(0 < norm\ x) = (x \neq 0)$
⟨*proof*⟩

lemma *norm-not-less-zero* [*simp*]:
fixes $x :: 'a::real\text{-}normed\text{-}vector$
shows $\neg\ norm\ x < 0$
⟨*proof*⟩

lemma *norm-le-zero-iff* [*simp*]:
fixes $x :: 'a::real\text{-}normed\text{-}vector$
shows $(norm\ x \le 0) = (x = 0)$
⟨*proof*⟩

lemma *norm-minus-cancel* [*simp*]:
fixes $x :: 'a::real\text{-}normed\text{-}vector$
shows $norm\ (-\ x) = norm\ x$
⟨*proof*⟩

lemma *norm-minus-commute*:
fixes $a\ b :: 'a::real\text{-}normed\text{-}vector$
shows $norm\ (a - b) = norm\ (b - a)$
⟨*proof*⟩

lemma *norm-triangle-ineq2*:
fixes $a\ b :: 'a::real\text{-}normed\text{-}vector$
shows $norm\ a - norm\ b \le norm\ (a - b)$
⟨*proof*⟩

lemma *norm-triangle-ineq3*:
  fixes $a\ b :: 'a::real\text{-}normed\text{-}vector$
  shows $|norm\ a - norm\ b| \leq norm\ (a - b)$
⟨*proof*⟩

lemma *norm-triangle-ineq4*:
  fixes $a\ b :: 'a::real\text{-}normed\text{-}vector$
  shows $norm\ (a - b) \leq norm\ a + norm\ b$
⟨*proof*⟩

lemma *norm-diff-ineq*:
  fixes $a\ b :: 'a::real\text{-}normed\text{-}vector$
  shows $norm\ a - norm\ b \leq norm\ (a + b)$
⟨*proof*⟩

lemma *norm-diff-triangle-ineq*:
  fixes $a\ b\ c\ d :: 'a::real\text{-}normed\text{-}vector$
  shows $norm\ ((a + b) - (c + d)) \leq norm\ (a - c) + norm\ (b - d)$
⟨*proof*⟩

lemma *abs-norm-cancel* [*simp*]:
  fixes $a :: 'a::real\text{-}normed\text{-}vector$
  shows $|norm\ a| = norm\ a$
⟨*proof*⟩

lemma *norm-add-less*:
  fixes $x\ y :: 'a::real\text{-}normed\text{-}vector$
  shows $[\![norm\ x < r;\ norm\ y < s]\!] \Longrightarrow norm\ (x + y) < r + s$
⟨*proof*⟩

lemma *norm-mult-less*:
  fixes $x\ y :: 'a::real\text{-}normed\text{-}algebra$
  shows $[\![norm\ x < r;\ norm\ y < s]\!] \Longrightarrow norm\ (x * y) < r * s$
⟨*proof*⟩

lemma *norm-of-real* [*simp*]:
  $norm\ (of\text{-}real\ r :: 'a::real\text{-}normed\text{-}algebra\text{-}1) = |r|$
⟨*proof*⟩

lemma *norm-number-of* [*simp*]:
  $norm\ (number\text{-}of\ w::'a::\{number\text{-}ring,real\text{-}normed\text{-}algebra\text{-}1\})$
    $= |number\text{-}of\ w|$
⟨*proof*⟩

lemma *norm-of-int* [*simp*]:
  $norm\ (of\text{-}int\ z::'a::real\text{-}normed\text{-}algebra\text{-}1) = |of\text{-}int\ z|$
⟨*proof*⟩

lemma *norm-of-nat* [*simp*]:

norm (*of-nat* *n*::′*a*::*real-normed-algebra-1*) = *of-nat* *n*
⟨*proof*⟩

lemma *nonzero-norm-inverse*:
fixes *a* :: ′*a*::*real-normed-div-algebra*
shows *a* ≠ *0* ⟹ *norm* (*inverse* *a*) = *inverse* (*norm* *a*)
⟨*proof*⟩

lemma *norm-inverse*:
fixes *a* :: ′*a*::{*real-normed-div-algebra*,*division-by-zero*}
shows *norm* (*inverse* *a*) = *inverse* (*norm* *a*)
⟨*proof*⟩

lemma *nonzero-norm-divide*:
fixes *a* *b* :: ′*a*::*real-normed-field*
shows *b* ≠ *0* ⟹ *norm* (*a* / *b*) = *norm* *a* / *norm* *b*
⟨*proof*⟩

lemma *norm-divide*:
fixes *a* *b* :: ′*a*::{*real-normed-field*,*division-by-zero*}
shows *norm* (*a* / *b*) = *norm* *a* / *norm* *b*
⟨*proof*⟩

lemma *norm-power-ineq*:
fixes *x* :: ′*a*::{*real-normed-algebra-1*,*recpower*}
shows *norm* (*x* ˆ *n*) ≤ *norm* *x* ˆ *n*
⟨*proof*⟩

lemma *norm-power*:
fixes *x* :: ′*a*::{*real-normed-div-algebra*,*recpower*}
shows *norm* (*x* ˆ *n*) = *norm* *x* ˆ *n*
⟨*proof*⟩

20.6 Sign function

lemma *norm-sgn*:
norm (*sgn*(*x*::′*a*::*real-normed-vector*)) = (*if* *x* = *0* *then* *0* *else* *1*)
⟨*proof*⟩

lemma *sgn-zero* [*simp*]: *sgn*(*0*::′*a*::*real-normed-vector*) = *0*
⟨*proof*⟩

lemma *sgn-zero-iff*: (*sgn*(*x*::′*a*::*real-normed-vector*) = *0*) = (*x* = *0*)
⟨*proof*⟩

lemma *sgn-minus*: *sgn* (− *x*) = − *sgn*(*x*::′*a*::*real-normed-vector*)
⟨*proof*⟩

lemma *sgn-scaleR*:

sgn (*scaleR* *r* *x*) = *scaleR* (*sgn* *r*) (*sgn*(*x*::*'a*::*real-normed-vector*))
⟨*proof*⟩

lemma *sgn-one* [*simp*]: *sgn* (*1*::*'a*::*real-normed-algebra-1*) = *1*
⟨*proof*⟩

lemma *sgn-of-real*:
sgn (*of-real* *r*::*'a*::*real-normed-algebra-1*) = *of-real* (*sgn* *r*)
⟨*proof*⟩

lemma *sgn-mult*:
fixes *x* *y* :: *'a*::*real-normed-div-algebra*
shows *sgn* (*x* ∗ *y*) = *sgn* *x* ∗ *sgn* *y*
⟨*proof*⟩

lemma *real-sgn-eq*: *sgn* (*x*::*real*) = *x* / |*x*|
⟨*proof*⟩

lemma *real-sgn-pos*: *0* < (*x*::*real*) ⟹ *sgn* *x* = *1*
⟨*proof*⟩

lemma *real-sgn-neg*: (*x*::*real*) < *0* ⟹ *sgn* *x* = −*1*
⟨*proof*⟩

20.7 Bounded Linear and Bilinear Operators

locale *bounded-linear* = *additive* +
constrains *f* :: *'a*::*real-normed-vector* ⇒ *'b*::*real-normed-vector*
assumes *scaleR*: *f* (*scaleR* *r* *x*) = *scaleR* *r* (*f* *x*)
assumes *bounded*: ∃ *K*. ∀ *x*. *norm* (*f* *x*) ≤ *norm* *x* ∗ *K*

lemma (**in** *bounded-linear*) *pos-bounded*:
∃ *K*>*0*. ∀ *x*. *norm* (*f* *x*) ≤ *norm* *x* ∗ *K*
⟨*proof*⟩

lemma (**in** *bounded-linear*) *nonneg-bounded*:
∃ *K*≥*0*. ∀ *x*. *norm* (*f* *x*) ≤ *norm* *x* ∗ *K*
⟨*proof*⟩

locale *bounded-bilinear* =
fixes *prod* :: [*'a*::*real-normed-vector*, *'b*::*real-normed-vector*]
⇒ *'c*::*real-normed-vector*
(**infixl** ∗∗ *70*)
assumes *add-left*: *prod* (*a* + *a'*) *b* = *prod* *a* *b* + *prod* *a'* *b*
assumes *add-right*: *prod* *a* (*b* + *b'*) = *prod* *a* *b* + *prod* *a* *b'*
assumes *scaleR-left*: *prod* (*scaleR* *r* *a*) *b* = *scaleR* *r* (*prod* *a* *b*)
assumes *scaleR-right*: *prod* *a* (*scaleR* *r* *b*) = *scaleR* *r* (*prod* *a* *b*)
assumes *bounded*: ∃ *K*. ∀ *a* *b*. *norm* (*prod* *a* *b*) ≤ *norm* *a* ∗ *norm* *b* ∗ *K*

lemma (**in** *bounded-bilinear*) *pos-bounded*:
$\exists K>0.\ \forall a\ b.\ norm\ (a ** b) \leq norm\ a * norm\ b * K$
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *nonneg-bounded*:
$\exists K\geq 0.\ \forall a\ b.\ norm\ (a ** b) \leq norm\ a * norm\ b * K$
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *additive-right*: *additive* $(\lambda b.\ prod\ a\ b)$
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *additive-left*: *additive* $(\lambda a.\ prod\ a\ b)$
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *zero-left*: $prod\ 0\ b = 0$
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *zero-right*: $prod\ a\ 0 = 0$
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *minus-left*: $prod\ (-\ a)\ b = -\ prod\ a\ b$
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *minus-right*: $prod\ a\ (-\ b) = -\ prod\ a\ b$
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *diff-left*:
$prod\ (a - a')\ b = prod\ a\ b - prod\ a'\ b$
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *diff-right*:
$prod\ a\ (b - b') = prod\ a\ b - prod\ a\ b'$
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *bounded-linear-left*:
bounded-linear $(\lambda a.\ a ** b)$
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *bounded-linear-right*:
bounded-linear $(\lambda b.\ a ** b)$
⟨*proof*⟩

lemma (**in** *bounded-bilinear*) *prod-diff-prod*:
$(x ** y - a ** b) = (x - a) ** (y - b) + (x - a) ** b + a ** (y - b)$
⟨*proof*⟩

interpretation *mult*:
bounded-bilinear $[op * :: 'a \Rightarrow 'a \Rightarrow 'a::real\text{-}normed\text{-}algebra]$
⟨*proof*⟩

interpretation *mult-left*:
bounded-linear [(λx::'*a*::*real-normed-algebra*. $x * y$)]
⟨*proof*⟩

interpretation *mult-right*:
bounded-linear [(λy::'*a*::*real-normed-algebra*. $x * y$)]
⟨*proof*⟩

interpretation *divide*:
bounded-linear [(λx::'*a*::*real-normed-field*. x / y)]
⟨*proof*⟩

interpretation *scaleR*: *bounded-bilinear* [*scaleR*]
⟨*proof*⟩

interpretation *scaleR-left*: *bounded-linear* [λr. *scaleR r x*]
⟨*proof*⟩

interpretation *scaleR-right*: *bounded-linear* [λx. *scaleR r x*]
⟨*proof*⟩

interpretation *of-real*: *bounded-linear* [λr. *of-real r*]
⟨*proof*⟩

end

theory *Real*
imports *ContNotDenum RealVector*
begin
end

21 Floating Point Representation of the Reals

theory *Float*
imports *Real Parity*
uses ~~/*src*/*Tools*/*float.ML* (*float-arith.ML*)
begin

definition
pow2 :: *int* ⇒ *real* **where**
pow2 a = (*if* (*0* <= *a*) *then* (*2*^(*nat a*)) *else* (*inverse* (*2*^(*nat* (−*a*)))))

definition
float :: *int* ∗ *int* ⇒ *real* **where**

$float\ x = real\ (fst\ x) * pow2\ (snd\ x)$

lemma $pow2\text{-}0[simp]$: $pow2\ 0 = 1$
⟨*proof*⟩

lemma $pow2\text{-}1[simp]$: $pow2\ 1 = 2$
⟨*proof*⟩

lemma $pow2\text{-}neg$: $pow2\ x = inverse\ (pow2\ (-x))$
⟨*proof*⟩

lemma $pow2\text{-}add1$: $pow2\ (1 + a) = 2 * (pow2\ a)$
⟨*proof*⟩

lemma $pow2\text{-}add$: $pow2\ (a+b) = (pow2\ a) * (pow2\ b)$
⟨*proof*⟩

lemma $float\ (a,\ e) + float\ (b,\ e) = float\ (a + b,\ e)$
⟨*proof*⟩

definition
$int\text{-}of\text{-}real :: real \Rightarrow int$ **where**
$int\text{-}of\text{-}real\ x = (SOME\ y.\ real\ y = x)$

definition
$real\text{-}is\text{-}int :: real \Rightarrow bool$ **where**
$real\text{-}is\text{-}int\ x = (EX\ (u::int).\ x = real\ u)$

lemma $real\text{-}is\text{-}int\text{-}def2$: $real\text{-}is\text{-}int\ x = (x = real\ (int\text{-}of\text{-}real\ x))$
⟨*proof*⟩

lemma $float\text{-}transfer$: $real\text{-}is\text{-}int\ ((real\ a)*(pow2\ c)) \Longrightarrow float\ (a,\ b) = float\ (int\text{-}of\text{-}real\ ((real\ a)*(pow2\ c)),\ b - c)$
⟨*proof*⟩

lemma $pow2\text{-}int$: $pow2\ (int\ c) = 2\hat{}c$
⟨*proof*⟩

lemma $float\text{-}transfer\text{-}nat$: $float\ (a,\ b) = float\ (a * 2\hat{}c,\ b - int\ c)$
⟨*proof*⟩

lemma $real\text{-}is\text{-}int\text{-}real[simp]$: $real\text{-}is\text{-}int\ (real\ (x::int))$
⟨*proof*⟩

lemma $int\text{-}of\text{-}real\text{-}real[simp]$: $int\text{-}of\text{-}real\ (real\ x) = x$
⟨*proof*⟩

lemma $real\text{-}int\text{-}of\text{-}real[simp]$: $real\text{-}is\text{-}int\ x \Longrightarrow real\ (int\text{-}of\text{-}real\ x) = x$
⟨*proof*⟩

lemma *real-is-int-add-int-of-real*: *real-is-int a* $\Longrightarrow$ *real-is-int b* $\Longrightarrow$ (*int-of-real* $(a+b)$) = (*int-of-real a*) + (*int-of-real b*)
⟨*proof*⟩

lemma *real-is-int-add*[*simp*]: *real-is-int a* $\Longrightarrow$ *real-is-int b* $\Longrightarrow$ *real-is-int* $(a+b)$
⟨*proof*⟩

lemma *int-of-real-sub*: *real-is-int a* $\Longrightarrow$ *real-is-int b* $\Longrightarrow$ (*int-of-real* $(a-b)$) = (*int-of-real a*) $-$ (*int-of-real b*)
⟨*proof*⟩

lemma *real-is-int-sub*[*simp*]: *real-is-int a* $\Longrightarrow$ *real-is-int b* $\Longrightarrow$ *real-is-int* $(a-b)$
⟨*proof*⟩

lemma *real-is-int-rep*: *real-is-int x* $\Longrightarrow$ *?!* (*a*::*int*). *real a* = *x*
⟨*proof*⟩

lemma *int-of-real-mult*:
  assumes *real-is-int a real-is-int b*
  shows (*int-of-real* $(a*b)$) = (*int-of-real a*) $*$ (*int-of-real b*)
⟨*proof*⟩

lemma *real-is-int-mult*[*simp*]: *real-is-int a* $\Longrightarrow$ *real-is-int b* $\Longrightarrow$ *real-is-int* $(a*b)$
⟨*proof*⟩

lemma *real-is-int-0*[*simp*]: *real-is-int* (*0*::*real*)
⟨*proof*⟩

lemma *real-is-int-1*[*simp*]: *real-is-int* (*1*::*real*)
⟨*proof*⟩

lemma *real-is-int-n1*: *real-is-int* ($-$*1*::*real*)
⟨*proof*⟩

lemma *real-is-int-number-of*[*simp*]: *real-is-int* ((*number-of* :: *int* $\Rightarrow$ *real*) *x*)
⟨*proof*⟩

lemma *int-of-real-0*[*simp*]: *int-of-real* (*0*::*real*) = (*0*::*int*)
⟨*proof*⟩

lemma *int-of-real-1*[*simp*]: *int-of-real* (*1*::*real*) = (*1*::*int*)
⟨*proof*⟩

lemma *int-of-real-number-of*[*simp*]: *int-of-real* (*number-of b*) = *number-of b*
⟨*proof*⟩

lemma *float-transfer-even*: *even a* $\Longrightarrow$ *float* (*a*, *b*) = *float* (*a div 2*, *b*+*1*)
  ⟨*proof*⟩

consts
norm-float :: *int*∗*int* ⇒ *int*∗*int*

lemma *int-div-zdiv*: $int\ (a\ div\ b) = (int\ a)\ div\ (int\ b)$
⟨*proof*⟩

lemma *int-mod-zmod*: $int\ (a\ mod\ b) = (int\ a)\ mod\ (int\ b)$
⟨*proof*⟩

lemma *abs-div-2-less*: $a \neq 0 \Longrightarrow a \neq -1 \Longrightarrow abs((a::int)\ div\ 2) < abs\ a$
⟨*proof*⟩

lemma *terminating-norm-float*: $\forall\ a.\ (a::int) \neq 0 \wedge even\ a \longrightarrow a \neq 0 \wedge |a\ div\ 2| < |a|$
⟨*proof*⟩

declare [[*simp-depth-limit* = *2*]]
recdef *norm-float measure* (% (*a*,*b*). *nat* (*abs a*))
norm-float (*a*,*b*) = (*if* (*a* ≠ *0*) & (*even a*) *then norm-float* (*a div 2*, *b*+*1*) *else* (*if a*=*0 then* (*0*,*0*) *else* (*a*,*b*)))
(**hints** *simp*: *even-def terminating-norm-float*)
declare [[*simp-depth-limit* = *100*]]

lemma *norm-float*: *float x* = *float* (*norm-float x*)
⟨*proof*⟩

lemma *float-add-l0*: *float* (*0*, *e*) + *x* = *x*
⟨*proof*⟩

lemma *float-add-r0*: *x* + *float* (*0*, *e*) = *x*
⟨*proof*⟩

lemma *float-add*:
float (*a1*, *e1*) + *float* (*a2*, *e2*) =
(*if e1*<=*e2 then float* (*a1*+*a2*∗*2*^(*nat*(*e2*−*e1*)), *e1*)
else float (*a1*∗*2*^(*nat* (*e1*−*e2*))+*a2*, *e2*))
⟨*proof*⟩

lemma *float-add-assoc1*:
(*x* + *float* (*y1*, *e1*)) + *float* (*y2*, *e2*) = (*float* (*y1*, *e1*) + *float* (*y2*, *e2*)) + *x*
⟨*proof*⟩

lemma *float-add-assoc2*:
(*float* (*y1*, *e1*) + *x*) + *float* (*y2*, *e2*) = (*float* (*y1*, *e1*) + *float* (*y2*, *e2*)) + *x*
⟨*proof*⟩

lemma *float-add-assoc3*:
float (*y1*, *e1*) + (*x* + *float* (*y2*, *e2*)) = (*float* (*y1*, *e1*) + *float* (*y2*, *e2*)) + *x*

$\langle proof \rangle$

lemma *float-add-assoc4*:
$float\ (y1,\ e1) + (float\ (y2,\ e2) + x) = (float\ (y1,\ e1) + float\ (y2,\ e2)) + x$
$\langle proof \rangle$

lemma *float-mult-l0*: $float\ (0,\ e) * x = float\ (0,\ 0)$
$\langle proof \rangle$

lemma *float-mult-r0*: $x * float\ (0,\ e) = float\ (0,\ 0)$
$\langle proof \rangle$

definition
$lbound :: real \Rightarrow real$
where
$lbound\ x = min\ 0\ x$

definition
$ubound :: real \Rightarrow real$
where
$ubound\ x = max\ 0\ x$

lemma *lbound*: $lbound\ x \leq x$
$\langle proof \rangle$

lemma *ubound*: $x \leq ubound\ x$
$\langle proof \rangle$

lemma *float-mult*:
$float\ (a1,\ e1) * float\ (a2,\ e2) =$
$(float\ (a1 * a2,\ e1 + e2))$
$\langle proof \rangle$

lemma *float-minus*:
$-\ (float\ (a,b)) = float\ (-a,\ b)$
$\langle proof \rangle$

lemma *zero-less-pow2*:
$0 < pow2\ x$
$\langle proof \rangle$

lemma *zero-le-float*:
$(0 <= float\ (a,b)) = (0 <= a)$
$\langle proof \rangle$

lemma *float-le-zero*:
$(float\ (a,b) <= 0) = (a <= 0)$
$\langle proof \rangle$

lemma *float-abs*:
abs (float (a,b)) = (if 0 <= a then (float (a,b)) else (float (−a,b)))
⟨*proof*⟩

lemma *float-zero*:
float (0, b) = 0
⟨*proof*⟩

lemma *float-pprt*:
pprt (float (a, b)) = (if 0 <= a then (float (a,b)) else (float (0, b)))
⟨*proof*⟩

lemma *pprt-lbound*: *pprt (lbound x) = float (0, 0)*
⟨*proof*⟩

lemma *nprt-ubound*: *nprt (ubound x) = float (0, 0)*
⟨*proof*⟩

lemma *float-nprt*:
nprt (float (a, b)) = (if 0 <= a then (float (0,b)) else (float (a, b)))
⟨*proof*⟩

lemma *norm-0-1*: *(0::-::number-ring) = Numeral0 & (1::-::number-ring) = Numeral1*
⟨*proof*⟩

lemma *add-left-zero*: *0 + a = (a::'a::comm-monoid-add)*
⟨*proof*⟩

lemma *add-right-zero*: *a + 0 = (a::'a::comm-monoid-add)*
⟨*proof*⟩

lemma *mult-left-one*: *1 * a = (a::'a::semiring-1)*
⟨*proof*⟩

lemma *mult-right-one*: *a * 1 = (a::'a::semiring-1)*
⟨*proof*⟩

lemma *int-pow-0*: *(a::int)^(Numeral0) = 1*
⟨*proof*⟩

lemma *int-pow-1*: *(a::int)^(Numeral1) = a*
⟨*proof*⟩

lemma *zero-eq-Numeral0-nring*: *(0::'a::number-ring) = Numeral0*
⟨*proof*⟩

lemma *one-eq-Numeral1-nring*: *(1::'a::number-ring) = Numeral1*
⟨*proof*⟩

lemma *zero-eq-Numeral0-nat*: $(0::nat) = Numeral0$
⟨*proof*⟩

lemma *one-eq-Numeral1-nat*: $(1::nat) = Numeral1$
⟨*proof*⟩

lemma *zpower-Pls*: $(z::int)\hat{\ }Numeral0 = Numeral1$
⟨*proof*⟩

lemma *zpower-Min*: $(z::int)\hat{\ }((-1)::nat) = Numeral1$
⟨*proof*⟩

lemma *fst-cong*: $a=a' \Longrightarrow fst\ (a,b) = fst\ (a',b)$
⟨*proof*⟩

lemma *snd-cong*: $b=b' \Longrightarrow snd\ (a,b) = snd\ (a,b')$
⟨*proof*⟩

lemma *lift-bool*: $x \Longrightarrow x=True$
⟨*proof*⟩

lemma *nlift-bool*: $\sim x \Longrightarrow x=False$
⟨*proof*⟩

lemma *not-false-eq-true*: $(\sim\ False) = True$ ⟨*proof*⟩

lemma *not-true-eq-false*: $(\sim\ True) = False$ ⟨*proof*⟩

lemmas *binarith* =
normalize-bin-simps
pred-bin-simps succ-bin-simps
add-bin-simps minus-bin-simps mult-bin-simps

lemma *int-eq-number-of-eq*:
$(((number\text{-}of\ v)::int)=(number\text{-}of\ w)) = iszero\ ((number\text{-}of\ (v + uminus\ w))::int)$
⟨*proof*⟩

lemma *int-iszero-number-of-Pls*: $iszero\ (Numeral0::int)$
⟨*proof*⟩

lemma *int-nonzero-number-of-Min*: $\sim(iszero\ ((-1)::int))$
⟨*proof*⟩

lemma *int-iszero-number-of-Bit0*: $iszero\ ((number\text{-}of\ (Int.Bit0\ w))::int) = iszero\ ((number\text{-}of\ w)::int)$
⟨*proof*⟩

lemma *int-iszero-number-of-Bit1*: $\neg\ iszero\ ((number\text{-}of\ (Int.Bit1\ w))::int)$
⟨*proof*⟩

lemma *int-less-number-of-eq-neg*: $(((number\text{-}of\ x)::int) < number\text{-}of\ y) = neg$ $((number\text{-}of\ (x + (uminus\ y)))::int)$
⟨*proof*⟩

lemma *int-not-neg-number-of-Pls*: $\neg\ (neg\ (Numeral0::int))$
⟨*proof*⟩

lemma *int-neg-number-of-Min*: $neg\ (-1::int)$
⟨*proof*⟩

lemma *int-neg-number-of-Bit0*: $neg\ ((number\text{-}of\ (Int.Bit0\ w))::int) = neg\ ((number\text{-}of$ $w)::int)$
⟨*proof*⟩

lemma *int-neg-number-of-Bit1*: $neg\ ((number\text{-}of\ (Int.Bit1\ w))::int) = neg\ ((number\text{-}of$ $w)::int)$
⟨*proof*⟩

lemma *int-le-number-of-eq*: $(((number\text{-}of\ x)::int) \leq number\text{-}of\ y) = (\neg\ neg\ ((number\text{-}of$ $(y + (uminus\ x)))::int))$
⟨*proof*⟩

lemmas *intarithrel* =
int-eq-number-of-eq
lift-bool[*OF int-iszero-number-of-Pls*] *nlift-bool*[*OF int-nonzero-number-of-Min*]
int-iszero-number-of-Bit0
lift-bool[*OF int-iszero-number-of-Bit1*] *int-less-number-of-eq-neg nlift-bool*[*OF int-not-neg-number-of-Pls*]
lift-bool[*OF int-neg-number-of-Min*]
int-neg-number-of-Bit0 int-neg-number-of-Bit1 int-le-number-of-eq

lemma *int-number-of-add-sym*: $((number\text{-}of\ v)::int) + number\text{-}of\ w = number\text{-}of$ $(v + w)$
⟨*proof*⟩

lemma *int-number-of-diff-sym*: $((number\text{-}of\ v)::int) - number\text{-}of\ w = number\text{-}of$ $(v + (uminus\ w))$
⟨*proof*⟩

lemma *int-number-of-mult-sym*: $((number\text{-}of\ v)::int) * number\text{-}of\ w = number\text{-}of$ $(v * w)$
⟨*proof*⟩

lemma *int-number-of-minus-sym*: $-\ ((number\text{-}of\ v)::int) = number\text{-}of\ (uminus\ v)$
⟨*proof*⟩

lemmas *intarith* = *int-number-of-add-sym int-number-of-minus-sym int-number-of-diff-sym int-number-of-mult-sym*

lemmas *natarith* = *add-nat-number-of diff-nat-number-of mult-nat-number-of eq-nat-number-of less-nat-number-of*

lemmas *powerarith* = *nat-number-of zpower-number-of-even*
zpower-number-of-odd[*simplified zero-eq-Numeral0-nring one-eq-Numeral1-nring*]
zpower-Pls zpower-Min

lemmas *floatarith*[*simplified norm-0-1*] = *float-add float-add-l0 float-add-r0 float-mult float-mult-l0 float-mult-r0*
float-minus float-abs zero-le-float float-pprt float-nprt pprt-lbound nprt-ubound

lemmas *arith* = *binarith intarith intarithrel natarith powerarith floatarith not-false-eq-true not-true-eq-false*

⟨*ML*⟩

end

theory *MatrixGeneral*
imports *Main*
begin

types *'a infmatrix* = [*nat*, *nat*] ⇒ *'a*

constdefs
nonzero-positions :: (*'a*::*zero*) *infmatrix* ⇒ (*nat*∗*nat*) *set*
nonzero-positions A == {*pos*. *A* (*fst pos*) (*snd pos*) ~= *0*}

typedef *'a matrix* = {(*f*::((*'a*::*zero*) *infmatrix*)). *finite* (*nonzero-positions f*)}
⟨*proof*⟩

declare *Rep-matrix-inverse*[*simp*]

lemma *finite-nonzero-positions* : *finite* (*nonzero-positions* (*Rep-matrix A*))
⟨*proof*⟩

constdefs
nrows :: (*'a*::*zero*) *matrix* ⇒ *nat*
nrows A == *if nonzero-positions*(*Rep-matrix A*) = {} *then 0 else Suc*(*Max* ((*image fst*) (*nonzero-positions* (*Rep-matrix A*))))
ncols :: (*'a*::*zero*) *matrix* ⇒ *nat*
ncols A == *if nonzero-positions*(*Rep-matrix A*) = {} *then 0 else Suc*(*Max* ((*image snd*) (*nonzero-positions* (*Rep-matrix A*))))

lemma *nrows*:
assumes *hyp*: *nrows A* ≤ *m*

shows $(Rep\text{-}matrix\ A\ m\ n) = 0$ (**is** *?concl*)
⟨*proof*⟩

constdefs
transpose-infmatrix :: *'a infmatrix* ⇒ *'a infmatrix*
transpose-infmatrix A j i == *A i j*
transpose-matrix :: (*'a*::*zero*) *matrix* ⇒ *'a matrix*
transpose-matrix == *Abs-matrix o transpose-infmatrix o Rep-matrix*

declare *transpose-infmatrix-def*[*simp*]

lemma *transpose-infmatrix-twice*[*simp*]: *transpose-infmatrix* (*transpose-infmatrix A*) = *A*
⟨*proof*⟩

lemma *transpose-infmatrix*: *transpose-infmatrix* (% *j i*. *P j i*) = (% *j i*. *P i j*)
⟨*proof*⟩

lemma *transpose-infmatrix-closed*[*simp*]: *Rep-matrix* (*Abs-matrix* (*transpose-infmatrix* (*Rep-matrix x*))) = *transpose-infmatrix* (*Rep-matrix x*)
⟨*proof*⟩

lemma *infmatrixforward*: (*x*::*'a infmatrix*) = *y* ⟹ ∀ *a b*. *x a b* = *y a b* ⟨*proof*⟩

lemma *transpose-infmatrix-inject*: (*transpose-infmatrix A* = *transpose-infmatrix B*) = (*A* = *B*)
⟨*proof*⟩

lemma *transpose-matrix-inject*: (*transpose-matrix A* = *transpose-matrix B*) = (*A* = *B*)
⟨*proof*⟩

lemma *transpose-matrix*[*simp*]: *Rep-matrix*(*transpose-matrix A*) *j i* = *Rep-matrix A i j*
⟨*proof*⟩

lemma *transpose-transpose-id*[*simp*]: *transpose-matrix* (*transpose-matrix A*) = *A*
⟨*proof*⟩

lemma *nrows-transpose*[*simp*]: *nrows* (*transpose-matrix A*) = *ncols A*
⟨*proof*⟩

lemma *ncols-transpose*[*simp*]: *ncols* (*transpose-matrix A*) = *nrows A*
⟨*proof*⟩

lemma *ncols*: *ncols A* <= *n* ⟹ *Rep-matrix A m n* = *0*
⟨*proof*⟩

lemma *ncols-le*: (*ncols A* <= *n*) = (! *j i*. *n* <= *i* ⟶ (*Rep-matrix A j i*) = *0*) (**is**

- = *?st*)
⟨*proof*⟩

lemma *less-ncols*: (*n* < *ncols A*) = (*? j i. n* <= *i* & (*Rep-matrix A j i*) ≠ *0*) (**is** *?concl*)
⟨*proof*⟩

lemma *le-ncols*: (*n* <= *ncols A*) = (∀ *m*. (∀ *j i. m* <= *i* ⟶ (*Rep-matrix A j i*) = *0*) ⟶ *n* <= *m*) (**is** *?concl*)
⟨*proof*⟩

lemma *nrows-le*: (*nrows A* <= *n*) = (! *j i. n* <= *j* ⟶ (*Rep-matrix A j i*) = *0*) (**is** *?s*)
⟨*proof*⟩

lemma *less-nrows*: (*m* < *nrows A*) = (*? j i. m* <= *j* & (*Rep-matrix A j i*) ≠ *0*) (**is** *?concl*)
⟨*proof*⟩

lemma *le-nrows*: (*n* <= *nrows A*) = (∀ *m*. (∀ *j i. m* <= *j* ⟶ (*Rep-matrix A j i*) = *0*) ⟶ *n* <= *m*) (**is** *?concl*)
⟨*proof*⟩

lemma *nrows-notzero*: *Rep-matrix A m n* ≠ *0* ⟹ *m* < *nrows A*
⟨*proof*⟩

lemma *ncols-notzero*: *Rep-matrix A m n* ≠ *0* ⟹ *n* < *ncols A*
⟨*proof*⟩

lemma *finite-natarray1*: *finite* {*x*. *x* < (*n*::*nat*)}
⟨*proof*⟩

lemma *finite-natarray2*: *finite* {*pos*. (*fst pos*) < (*m*::*nat*) & (*snd pos*) < (*n*::*nat*)}
⟨*proof*⟩

lemma *RepAbs-matrix*:
assumes *aem*: *? m*. ! *j i. m* <= *j* ⟶ *x j i* = *0* (**is** *?em*) **and** *aen*:*? n*. ! *j i*. (*n* <= *i* ⟶ *x j i* = *0*) (**is** *?en*)
shows (*Rep-matrix* (*Abs-matrix x*)) = *x*
⟨*proof*⟩

constdefs
apply-infmatrix :: (*'a* ⇒ *'b*) ⇒ *'a infmatrix* ⇒ *'b infmatrix*
apply-infmatrix f == % *A*. (% *j i*. *f* (*A j i*))
apply-matrix :: (*'a* ⇒ *'b*) ⇒ (*'a*::*zero*) *matrix* ⇒ (*'b*::*zero*) *matrix*
apply-matrix f == % *A*. *Abs-matrix* (*apply-infmatrix f* (*Rep-matrix A*))
combine-infmatrix :: (*'a* ⇒ *'b* ⇒ *'c*) ⇒ *'a infmatrix* ⇒ *'b infmatrix* ⇒ *'c infmatrix*
combine-infmatrix f == % *A B*. (% *j i*. *f* (*A j i*) (*B j i*))
combine-matrix :: (*'a* ⇒ *'b* ⇒ *'c*) ⇒ (*'a*::*zero*) *matrix* ⇒ (*'b*::*zero*) *matrix* ⇒

$('c::zero)$ *matrix*
combine-matrix f == % A B. *Abs-matrix* (*combine-infmatrix* f (*Rep-matrix* A) (*Rep-matrix* B))

lemma *expand-apply-infmatrix*[*simp*]: *apply-infmatrix* f A j i = f (A j i)
⟨*proof*⟩

lemma *expand-combine-infmatrix*[*simp*]: *combine-infmatrix* f A B j i = f (A j i) (B j i)
⟨*proof*⟩

constdefs
commutative :: $('a \Rightarrow 'a \Rightarrow 'b) \Rightarrow$ *bool*
commutative f == ! x y. f x y = f y x
associative :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow$ *bool*
associative f == ! x y z. f (f x y) z = f x (f y z)

To reason about associativity and commutativity of operations on matrices, let's take a step back and look at the general situtation: Assume that we have sets A and B with $B \subset A$ and an abstraction $u : A \to B$. This abstraction has to fulfill $u(b) = b$ for all $b \in B$, but is arbitrary otherwise. Each function $f : A \times A \to A$ now induces a function $f' : B \times B \to B$ by $f' = u \circ f$. It is obvious that commutativity of f implies commutativity of f': $f'xy = u(fxy) = u(fyx) = f'yx$.

lemma *combine-infmatrix-commute*:
commutative f $\Longrightarrow$ *commutative* (*combine-infmatrix* f)
⟨*proof*⟩

lemma *combine-matrix-commute*:
commutative f $\Longrightarrow$ *commutative* (*combine-matrix* f)
⟨*proof*⟩

On the contrary, given an associative function f we cannot expect f' to be associative. A counterexample is given by $A = \mathbb{Z}$, $B = \{-1, 0, 1\}$, as f we take addition on $\mathbb{Z}$, which is clearly associative. The abstraction is given by $u(a) = 0$ for $a \notin B$. Then we have

$$f'(f'11) - 1 = u(f(u(f11)) - 1) = u(f(u2) - 1) = u(f0 - 1) = -1,$$

but on the other hand we have

$$f'1(f'1 - 1) = u(f1(u(f1 - 1))) = u(f10) = 1.$$

A way out of this problem is to assume that $f(A \times A) \subset A$ holds, and this is what we are going to do:

lemma *nonzero-positions-combine-infmatrix*[*simp*]: f *0 0* = *0* $\Longrightarrow$ *nonzero-positions* (*combine-infmatrix* f A B) $\subseteq$ (*nonzero-positions* A) $\cup$ (*nonzero-positions* B)
⟨*proof*⟩

lemma *finite-nonzero-positions-Rep*[*simp*]: *finite* (*nonzero-positions* (*Rep-matrix* *A*))
⟨*proof*⟩

lemma *combine-infmatrix-closed* [*simp*]:
f 0 0 = *0* ⟹ *Rep-matrix* (*Abs-matrix* (*combine-infmatrix f* (*Rep-matrix A*) (*Rep-matrix B*))) = *combine-infmatrix f* (*Rep-matrix A*) (*Rep-matrix B*)
⟨*proof*⟩

We need the next two lemmas only later, but it is analog to the above one, so we prove them now:

lemma *nonzero-positions-apply-infmatrix*[*simp*]: *f 0* = *0* ⟹ *nonzero-positions* (*apply-infmatrix f A*) ⊆ *nonzero-positions A*
⟨*proof*⟩

lemma *apply-infmatrix-closed* [*simp*]:
f 0 = *0* ⟹ *Rep-matrix* (*Abs-matrix* (*apply-infmatrix f* (*Rep-matrix A*))) = *apply-infmatrix f* (*Rep-matrix A*)
⟨*proof*⟩

lemma *combine-infmatrix-assoc*[*simp*]: *f 0 0* = *0* ⟹ *associative f* ⟹ *associative* (*combine-infmatrix f*)
⟨*proof*⟩

lemma *comb*: *f* = *g* ⟹ *x* = *y* ⟹ *f x* = *g y*
⟨*proof*⟩

lemma *combine-matrix-assoc*: *f 0 0* = *0* ⟹ *associative f* ⟹ *associative* (*combine-matrix f*)
⟨*proof*⟩

lemma *Rep-apply-matrix*[*simp*]: *f 0* = *0* ⟹ *Rep-matrix* (*apply-matrix f A*) *j i* = *f* (*Rep-matrix A j i*)
⟨*proof*⟩

lemma *Rep-combine-matrix*[*simp*]: *f 0 0* = *0* ⟹ *Rep-matrix* (*combine-matrix f A B*) *j i* = *f* (*Rep-matrix A j i*) (*Rep-matrix B j i*)
⟨*proof*⟩

lemma *combine-nrows-max*: *f 0 0* = *0* ⟹ *nrows* (*combine-matrix f A B*) <= *max* (*nrows A*) (*nrows B*)
⟨*proof*⟩

lemma *combine-ncols-max*: *f 0 0* = *0* ⟹ *ncols* (*combine-matrix f A B*) <= *max* (*ncols A*) (*ncols B*)
⟨*proof*⟩

lemma *combine-nrows*: *f 0 0* = *0* ⟹ *nrows A* <= *q* ⟹ *nrows B* <= *q* ⟹

$nrows(combine\text{-}matrix\ f\ A\ B) <= q$
⟨*proof*⟩

lemma *combine-ncols*: $f\ 0\ 0 = 0 \Longrightarrow ncols\ A <= q \Longrightarrow ncols\ B <= q \Longrightarrow ncols(combine\text{-}matrix\ f\ A\ B) <= q$
⟨*proof*⟩

constdefs
$zero\text{-}r\text{-}neutral :: ('a \Rightarrow 'b::zero \Rightarrow 'a) \Rightarrow bool$
$zero\text{-}r\text{-}neutral\ f == !\ a.\ f\ a\ 0 = a$
$zero\text{-}l\text{-}neutral :: ('a::zero \Rightarrow 'b \Rightarrow 'b) \Rightarrow bool$
$zero\text{-}l\text{-}neutral\ f == !\ a.\ f\ 0\ a = a$
$zero\text{-}closed :: (('a::zero) \Rightarrow ('b::zero) \Rightarrow ('c::zero)) \Rightarrow bool$
$zero\text{-}closed\ f == (!x.\ f\ x\ 0 = 0)\ \&\ (!y.\ f\ 0\ y = 0)$

consts $foldseq :: ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow (nat \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a$
primrec
$foldseq\ f\ s\ 0 = s\ 0$
$foldseq\ f\ s\ (Suc\ n) = f\ (s\ 0)\ (foldseq\ f\ (\%\ k.\ s(Suc\ k))\ n)$

consts $foldseq\text{-}transposed ::\ ('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow (nat \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a$
primrec
$foldseq\text{-}transposed\ f\ s\ 0 = s\ 0$
$foldseq\text{-}transposed\ f\ s\ (Suc\ n) = f\ (foldseq\text{-}transposed\ f\ s\ n)\ (s\ (Suc\ n))$

lemma *foldseq-assoc* : $associative\ f \Longrightarrow foldseq\ f = foldseq\text{-}transposed\ f$
⟨*proof*⟩

lemma *foldseq-distr*: $[\![associative\ f;\ commutative\ f]\!] \Longrightarrow foldseq\ f\ (\%\ k.\ f\ (u\ k)\ (v\ k))\ n = f\ (foldseq\ f\ u\ n)\ (foldseq\ f\ v\ n)$
⟨*proof*⟩

theorem $[\![associative\ f;\ associative\ g;\ \forall\ a\ b\ c\ d.\ g\ (f\ a\ b)\ (f\ c\ d) = f\ (g\ a\ c)\ (g\ b\ d);\ ?\ x\ y.\ (f\ x) \neq (f\ y);\ ?\ x\ y.\ (g\ x) \neq (g\ y);\ f\ x\ x = x;\ g\ x\ x = x]\!] \Longrightarrow f{=}g\ |\ (!\ y.\ f\ y\ x = y)\ |\ (!\ y.\ g\ y\ x = y)$
⟨*proof*⟩

lemma *foldseq-zero*:
assumes fz: $f\ 0\ 0 = 0$ **and** sz: $!\ i.\ i <= n \longrightarrow s\ i = 0$
shows $foldseq\ f\ s\ n = 0$
⟨*proof*⟩

lemma *foldseq-significant-positions*:
assumes p: $!\ i.\ i <= N \longrightarrow S\ i = T\ i$
shows $foldseq\ f\ S\ N = foldseq\ f\ T\ N$ (**is** *?concl*)
⟨*proof*⟩

lemma *foldseq-tail*: $M <= N \Longrightarrow foldseq\ f\ S\ N = foldseq\ f\ (\%\ k.\ (if\ k < M\ then$

$(S\ k)$ *else* $(foldseq\ f\ (\%\ k.\ S(k{+}M))\ (N{-}M))))\ M$ (**is** $?p \Longrightarrow ?concl$)
⟨*proof*⟩

lemma *foldseq-zerotail*:
assumes
fz: $f\ 0\ 0 = 0$
and sz: $!\ i.\ \ n <= i \longrightarrow s\ i = 0$
and nm: $n <= m$
shows
$foldseq\ f\ s\ n = foldseq\ f\ s\ m$
⟨*proof*⟩

lemma *foldseq-zerotail2*:
assumes $!\ x.\ f\ x\ 0 = x$
and $!\ i.\ n < i \longrightarrow s\ i = 0$
and nm: $n <= m$
shows
$foldseq\ f\ s\ n = foldseq\ f\ s\ m$ (**is** $?concl$)
⟨*proof*⟩

lemma *foldseq-zerostart*:
$!\ x.\ f\ 0\ (f\ 0\ x) = f\ 0\ x \Longrightarrow\ !\ i.\ i <= n \longrightarrow s\ i = 0 \Longrightarrow foldseq\ f\ s\ (Suc\ n) = f\ 0\ (s\ (Suc\ n))$
⟨*proof*⟩

lemma *foldseq-zerostart2*:
$!\ x.\ f\ 0\ x = x \Longrightarrow\ !\ i.\ i < n \longrightarrow s\ i = 0 \Longrightarrow foldseq\ f\ s\ n = s\ n$
⟨*proof*⟩

lemma *foldseq-almostzero*:
assumes $f0x$:$!\ x.\ f\ 0\ x = x$ **and** $fx0$: $!\ x.\ f\ x\ 0 = x$ **and** $s0$:$!\ i.\ i \neq j \longrightarrow s\ i = 0$
shows $foldseq\ f\ s\ n = (if\ (j <= n)\ then\ (s\ j)\ else\ 0)$ (**is** $?concl$)
⟨*proof*⟩

lemma *foldseq-distr-unary*:
assumes $!!\ a\ b.\ g\ (f\ a\ b) = f\ (g\ a)\ (g\ b)$
shows $g(foldseq\ f\ s\ n) = foldseq\ f\ (\%\ x.\ g(s\ x))\ n$ (**is** $?concl$)
⟨*proof*⟩

constdefs
$mult\text{-}matrix\text{-}n :: nat \Rightarrow (('a::zero) \Rightarrow ('b::zero) \Rightarrow ('c::zero)) \Rightarrow ('c \Rightarrow 'c \Rightarrow 'c) \Rightarrow 'a\ matrix \Rightarrow 'b\ matrix \Rightarrow 'c\ matrix$
$mult\text{-}matrix\text{-}n\ n\ fmul\ fadd\ A\ B == Abs\text{-}matrix(\%\ j\ i.\ foldseq\ fadd\ (\%\ k.\ fmul\ (Rep\text{-}matrix\ A\ j\ k)\ (Rep\text{-}matrix\ B\ k\ i))\ n)$
$mult\text{-}matrix :: (('a::zero) \Rightarrow ('b::zero) \Rightarrow ('c::zero)) \Rightarrow ('c \Rightarrow 'c \Rightarrow 'c) \Rightarrow 'a\ matrix \Rightarrow 'b\ matrix \Rightarrow 'c\ matrix$
$mult\text{-}matrix\ fmul\ fadd\ A\ B == mult\text{-}matrix\text{-}n\ (max\ (ncols\ A)\ (nrows\ B))\ fmul\ fadd\ A\ B$

lemma *mult-matrix-n*:
assumes *prems*: *ncols A* $\leq$ *n* (**is** *?An*) *nrows B* $\leq$ *n* (**is** *?Bn*) *fadd 0 0* = *0 fmul 0 0* = *0*
shows *c*:*mult-matrix fmul fadd A B* = *mult-matrix-n n fmul fadd A B* (**is** *?concl*)
⟨*proof*⟩

lemma *mult-matrix-nm*:
assumes *prems*: *ncols A* <= *n nrows B* <= *n ncols A* <= *m nrows B* <= *m fadd 0 0* = *0 fmul 0 0* = *0*
shows *mult-matrix-n n fmul fadd A B* = *mult-matrix-n m fmul fadd A B*
⟨*proof*⟩

constdefs
r-distributive :: ('*a* ⇒ '*b* ⇒ '*b*) ⇒ ('*b* ⇒ '*b* ⇒ '*b*) ⇒ *bool*
r-distributive fmul fadd == ! *a u v*. *fmul a* (*fadd u v*) = *fadd* (*fmul a u*) (*fmul a v*)
l-distributive :: ('*a* ⇒ '*b* ⇒ '*a*) ⇒ ('*a* ⇒ '*a* ⇒ '*a*) ⇒ *bool*
l-distributive fmul fadd == ! *a u v*. *fmul* (*fadd u v*) *a* = *fadd* (*fmul u a*) (*fmul v a*)
distributive :: ('*a* ⇒ '*a* ⇒ '*a*) ⇒ ('*a* ⇒ '*a* ⇒ '*a*) ⇒ *bool*
distributive fmul fadd == *l-distributive fmul fadd* & *r-distributive fmul fadd*

lemma *max1*: !! *a x y*. (*a*::*nat*) <= *x* ⟹ *a* <= *max x y* ⟨*proof*⟩
lemma *max2*: !! *b x y*. (*b*::*nat*) <= *y* ⟹ *b* <= *max x y* ⟨*proof*⟩

lemma *r-distributive-matrix*:
assumes *prems*:
r-distributive fmul fadd
associative fadd
commutative fadd
fadd 0 0 = *0*
! *a*. *fmul a 0* = *0*
! *a*. *fmul 0 a* = *0*
shows *r-distributive* (*mult-matrix fmul fadd*) (*combine-matrix fadd*) (**is** *?concl*)
⟨*proof*⟩

lemma *l-distributive-matrix*:
assumes *prems*:
l-distributive fmul fadd
associative fadd
commutative fadd
fadd 0 0 = *0*
! *a*. *fmul a 0* = *0*
! *a*. *fmul 0 a* = *0*
shows *l-distributive* (*mult-matrix fmul fadd*) (*combine-matrix fadd*) (**is** *?concl*)
⟨*proof*⟩

instantiation *matrix* :: (*zero*) *zero*
begin

definition
zero-matrix-def: (*0*::(*'a*::*zero*) *matrix*) == *Abs-matrix*(% *j i*. *0*)

instance ⟨*proof*⟩

end

lemma *Rep-zero-matrix-def*[*simp*]: *Rep-matrix 0 j i* = *0*
⟨*proof*⟩

lemma *zero-matrix-def-nrows*[*simp*]: *nrows 0* = *0*
⟨*proof*⟩

lemma *zero-matrix-def-ncols*[*simp*]: *ncols 0* = *0*
⟨*proof*⟩

lemma *combine-matrix-zero-l-neutral*: *zero-l-neutral f* ⟹ *zero-l-neutral* (*combine-matrix f*)
⟨*proof*⟩

lemma *combine-matrix-zero-r-neutral*: *zero-r-neutral f* ⟹ *zero-r-neutral* (*combine-matrix f*)
⟨*proof*⟩

lemma *mult-matrix-zero-closed*: ⟦*fadd 0 0* = *0*; *zero-closed fmul*⟧ ⟹ *zero-closed* (*mult-matrix fmul fadd*)
⟨*proof*⟩

lemma *mult-matrix-n-zero-right*[*simp*]: ⟦*fadd 0 0* = *0*; !*a*. *fmul a 0* = *0*⟧ ⟹ *mult-matrix-n n fmul fadd A 0* = *0*
⟨*proof*⟩

lemma *mult-matrix-n-zero-left*[*simp*]: ⟦*fadd 0 0* = *0*; !*a*. *fmul 0 a* = *0*⟧ ⟹ *mult-matrix-n n fmul fadd 0 A* = *0*
⟨*proof*⟩

lemma *mult-matrix-zero-left*[*simp*]: ⟦*fadd 0 0* = *0*; !*a*. *fmul 0 a* = *0*⟧ ⟹ *mult-matrix fmul fadd 0 A* = *0*
⟨*proof*⟩

lemma *mult-matrix-zero-right*[*simp*]: ⟦*fadd 0 0* = *0*; !*a*. *fmul a 0* = *0*⟧ ⟹ *mult-matrix fmul fadd A 0* = *0*
⟨*proof*⟩

lemma *apply-matrix-zero*[*simp*]: *f 0* = *0* ⟹ *apply-matrix f 0* = *0*
⟨*proof*⟩

lemma *combine-matrix-zero*: *f 0 0* = *0* ⟹ *combine-matrix f 0 0* = *0*

⟨*proof*⟩

lemma *transpose-matrix-zero*[*simp*]: *transpose-matrix 0 = 0*
⟨*proof*⟩

lemma *apply-zero-matrix-def*[*simp*]: *apply-matrix* (% *x*. *0*) *A* = *0*
⟨*proof*⟩

constdefs
singleton-matrix :: *nat* ⇒ *nat* ⇒ ('*a*::*zero*) ⇒ '*a matrix*
singleton-matrix j i a == *Abs-matrix*(% *m n*. *if j* = *m* & *i* = *n then a else 0*)
move-matrix :: ('*a*::*zero*) *matrix* ⇒ *int* ⇒ *int* ⇒ '*a matrix*
move-matrix A y x == *Abs-matrix*(% *j i*. *if* (*neg* ((*int j*)−*y*)) | (*neg* ((*int i*)−*x*)) *then 0 else Rep-matrix A* (*nat* ((*int j*)−*y*)) (*nat* ((*int i*)−*x*)))
take-rows :: ('*a*::*zero*) *matrix* ⇒ *nat* ⇒ '*a matrix*
take-rows A r == *Abs-matrix*(% *j i*. *if* (*j* < *r*) *then* (*Rep-matrix A j i*) *else 0*)
take-columns :: ('*a*::*zero*) *matrix* ⇒ *nat* ⇒ '*a matrix*
take-columns A c == *Abs-matrix*(% *j i*. *if* (*i* < *c*) *then* (*Rep-matrix A j i*) *else 0*)

constdefs
column-of-matrix :: ('*a*::*zero*) *matrix* ⇒ *nat* ⇒ '*a matrix*
column-of-matrix A n == *take-columns* (*move-matrix A 0* (− *int n*)) *1*
row-of-matrix :: ('*a*::*zero*) *matrix* ⇒ *nat* ⇒ '*a matrix*
row-of-matrix A m == *take-rows* (*move-matrix A* (− *int m*) *0*) *1*

lemma *Rep-singleton-matrix*[*simp*]: *Rep-matrix* (*singleton-matrix j i e*) *m n* = (*if j* = *m* & *i* = *n then e else 0*)
⟨*proof*⟩

lemma *apply-singleton-matrix*[*simp*]: *f 0* = *0* ⟹ *apply-matrix f* (*singleton-matrix j i x*) = (*singleton-matrix j i* (*f x*))
⟨*proof*⟩

lemma *singleton-matrix-zero*[*simp*]: *singleton-matrix j i 0* = *0*
⟨*proof*⟩

lemma *nrows-singleton*[*simp*]: *nrows*(*singleton-matrix j i e*) = (*if e* = *0 then 0 else Suc j*)
⟨*proof*⟩

lemma *ncols-singleton*[*simp*]: *ncols*(*singleton-matrix j i e*) = (*if e* = *0 then 0 else Suc i*)
⟨*proof*⟩

lemma *combine-singleton*: *f 0 0* = *0* ⟹ *combine-matrix f* (*singleton-matrix j i a*) (*singleton-matrix j i b*) = *singleton-matrix j i* (*f a b*)
⟨*proof*⟩

lemma *transpose-singleton*[*simp*]: *transpose-matrix* (*singleton-matrix j i a*) = *singleton-matrix i j a*
⟨*proof*⟩

lemma *Rep-move-matrix*[*simp*]:
Rep-matrix (*move-matrix A y x*) *j i* =
(*if* (*neg* ((*int j*)−*y*)) | (*neg* ((*int i*)−*x*)) *then 0 else Rep-matrix A* (*nat*((*int j*)−*y*)) (*nat*((*int i*)−*x*)))
⟨*proof*⟩

lemma *move-matrix-0-0*[*simp*]: *move-matrix A 0 0* = *A*
⟨*proof*⟩

lemma *move-matrix-ortho*: *move-matrix A j i* = *move-matrix* (*move-matrix A j 0*) *0 i*
⟨*proof*⟩

lemma *transpose-move-matrix*[*simp*]:
transpose-matrix (*move-matrix A x y*) = *move-matrix* (*transpose-matrix A*) *y x*
⟨*proof*⟩

lemma *move-matrix-singleton*[*simp*]: *move-matrix* (*singleton-matrix u v x*) *j i* =
(*if* (*j* + *int u* < *0*) | (*i* + *int v* < *0*) *then 0 else* (*singleton-matrix* (*nat* (*j* + *int u*)) (*nat* (*i* + *int v*)) *x*))
⟨*proof*⟩

lemma *Rep-take-columns*[*simp*]:
Rep-matrix (*take-columns A c*) *j i* =
(*if i* < *c then* (*Rep-matrix A j i*) *else 0*)
⟨*proof*⟩

lemma *Rep-take-rows*[*simp*]:
Rep-matrix (*take-rows A r*) *j i* =
(*if j* < *r then* (*Rep-matrix A j i*) *else 0*)
⟨*proof*⟩

lemma *Rep-column-of-matrix*[*simp*]:
Rep-matrix (*column-of-matrix A c*) *j i* = (*if i* = *0 then* (*Rep-matrix A j c*) *else 0*)
⟨*proof*⟩

lemma *Rep-row-of-matrix*[*simp*]:
Rep-matrix (*row-of-matrix A r*) *j i* = (*if j* = *0 then* (*Rep-matrix A r i*) *else 0*)
⟨*proof*⟩

lemma *column-of-matrix*: *ncols A* <= *n* ⟹ *column-of-matrix A n* = *0*
⟨*proof*⟩

lemma *row-of-matrix*: *nrows A* <= *n* ⟹ *row-of-matrix A n* = *0*

⟨*proof*⟩

lemma *mult-matrix-singleton-right*[*simp*]:
assumes *prems*:
! *x*. *fmul x 0* = *0*
! *x*. *fmul 0 x* = *0*
! *x*. *fadd 0 x* = *x*
! *x*. *fadd x 0* = *x*
shows (*mult-matrix fmul fadd A* (*singleton-matrix j i e*)) = *apply-matrix* (% *x*. *fmul x e*) (*move-matrix* (*column-of-matrix A j*) *0* (*int i*))
⟨*proof*⟩

lemma *mult-matrix-ext*:
assumes
eprem:
? *e*. (! *a b*. *a* ≠ *b* ⟶ *fmul a e* ≠ *fmul b e*)
and *fprems*:
! *a*. *fmul 0 a* = *0*
! *a*. *fmul a 0* = *0*
! *a*. *fadd a 0* = *a*
! *a*. *fadd 0 a* = *a*
and *contraprems*:
mult-matrix fmul fadd A = *mult-matrix fmul fadd B*
shows
A = *B*
⟨*proof*⟩

constdefs
foldmatrix :: (′*a* ⇒ ′*a* ⇒ ′*a*) ⇒ (′*a* ⇒ ′*a* ⇒ ′*a*) ⇒ (′*a infmatrix*) ⇒ *nat* ⇒ *nat* ⇒ ′*a*
foldmatrix f g A m n == *foldseq-transposed g* (% *j*. *foldseq f* (*A j*) *n*) *m*
foldmatrix-transposed :: (′*a* ⇒ ′*a* ⇒ ′*a*) ⇒ (′*a* ⇒ ′*a* ⇒ ′*a*) ⇒ (′*a infmatrix*) ⇒ *nat* ⇒ *nat* ⇒ ′*a*
foldmatrix-transposed f g A m n == *foldseq g* (% *j*. *foldseq-transposed f* (*A j*) *n*) *m*

lemma *foldmatrix-transpose*:
assumes
! *a b c d*. *g*(*f a b*) (*f c d*) = *f* (*g a c*) (*g b d*)
shows
foldmatrix f g A m n = *foldmatrix-transposed g f* (*transpose-infmatrix A*) *n m* (**is** *?concl*)
⟨*proof*⟩

lemma *foldseq-foldseq*:
assumes
associative f
associative g
! *a b c d*. *g*(*f a b*) (*f c d*) = *f* (*g a c*) (*g b d*)

shows
foldseq g (% *j*. *foldseq f* (*A j*) *n*) *m* = *foldseq f* (% *j*. *foldseq g* ((*transpose-infmatrix* *A*) *j*) *m*) *n*
⟨*proof*⟩

lemma *mult-n-nrows*:
assumes
! *a*. *fmul 0 a* = *0*
! *a*. *fmul a 0* = *0*
fadd 0 0 = *0*
shows *nrows* (*mult-matrix-n n fmul fadd A B*) ≤ *nrows A*
⟨*proof*⟩

lemma *mult-n-ncols*:
assumes
! *a*. *fmul 0 a* = *0*
! *a*. *fmul a 0* = *0*
fadd 0 0 = *0*
shows *ncols* (*mult-matrix-n n fmul fadd A B*) ≤ *ncols B*
⟨*proof*⟩

lemma *mult-nrows*:
assumes
! *a*. *fmul 0 a* = *0*
! *a*. *fmul a 0* = *0*
fadd 0 0 = *0*
shows *nrows* (*mult-matrix fmul fadd A B*) ≤ *nrows A*
⟨*proof*⟩

lemma *mult-ncols*:
assumes
! *a*. *fmul 0 a* = *0*
! *a*. *fmul a 0* = *0*
fadd 0 0 = *0*
shows *ncols* (*mult-matrix fmul fadd A B*) ≤ *ncols B*
⟨*proof*⟩

lemma *nrows-move-matrix-le*: *nrows* (*move-matrix A j i*) <= *nat*((*int* (*nrows A*)) + *j*)
⟨*proof*⟩

lemma *ncols-move-matrix-le*: *ncols* (*move-matrix A j i*) <= *nat*((*int* (*ncols A*)) + *i*)
⟨*proof*⟩

lemma *mult-matrix-assoc*:
assumes *prems*:
! *a*. *fmul1 0 a* = *0*
! *a*. *fmul1 a 0* = *0*

! *a*. *fmul2 0 a* = *0*
! *a*. *fmul2 a 0* = *0*
fadd1 0 0 = *0*
fadd2 0 0 = *0*
! *a b c d*. *fadd2* (*fadd1 a b*) (*fadd1 c d*) = *fadd1* (*fadd2 a c*) (*fadd2 b d*)
associative fadd1
associative fadd2
! *a b c*. *fmul2* (*fmul1 a b*) *c* = *fmul1 a* (*fmul2 b c*)
! *a b c*. *fmul2* (*fadd1 a b*) *c* = *fadd1* (*fmul2 a c*) (*fmul2 b c*)
! *a b c*. *fmul1 c* (*fadd2 a b*) = *fadd2* (*fmul1 c a*) (*fmul1 c b*)
shows *mult-matrix fmul2 fadd2* (*mult-matrix fmul1 fadd1 A B*) *C* = *mult-matrix fmul1 fadd1 A* (*mult-matrix fmul2 fadd2 B C*) (**is** *?concl*)
⟨*proof*⟩

lemma
assumes *prems*:
! *a*. *fmul1 0 a* = *0*
! *a*. *fmul1 a 0* = *0*
! *a*. *fmul2 0 a* = *0*
! *a*. *fmul2 a 0* = *0*
fadd1 0 0 = *0*
fadd2 0 0 = *0*
! *a b c d*. *fadd2* (*fadd1 a b*) (*fadd1 c d*) = *fadd1* (*fadd2 a c*) (*fadd2 b d*)
associative fadd1
associative fadd2
! *a b c*. *fmul2* (*fmul1 a b*) *c* = *fmul1 a* (*fmul2 b c*)
! *a b c*. *fmul2* (*fadd1 a b*) *c* = *fadd1* (*fmul2 a c*) (*fmul2 b c*)
! *a b c*. *fmul1 c* (*fadd2 a b*) = *fadd2* (*fmul1 c a*) (*fmul1 c b*)
shows
(*mult-matrix fmul1 fadd1 A*) *o* (*mult-matrix fmul2 fadd2 B*) = *mult-matrix fmul2 fadd2* (*mult-matrix fmul1 fadd1 A B*)
⟨*proof*⟩

lemma *mult-matrix-assoc-simple*:
assumes *prems*:
! *a*. *fmul 0 a* = *0*
! *a*. *fmul a 0* = *0*
fadd 0 0 = *0*
associative fadd
commutative fadd
associative fmul
distributive fmul fadd
shows *mult-matrix fmul fadd* (*mult-matrix fmul fadd A B*) *C* = *mult-matrix fmul fadd A* (*mult-matrix fmul fadd B C*) (**is** *?concl*)
⟨*proof*⟩

lemma *transpose-apply-matrix*: *f 0* = *0* ⟹ *transpose-matrix* (*apply-matrix f A*) = *apply-matrix f* (*transpose-matrix A*)
⟨*proof*⟩

lemma *transpose-combine-matrix*: $f\ 0\ 0 = 0 \Longrightarrow$ *transpose-matrix* (*combine-matrix* $f\ A\ B$) = *combine-matrix* f (*transpose-matrix* A) (*transpose-matrix* B)
⟨*proof*⟩

lemma *Rep-mult-matrix*:
assumes
! a. *fmul* $0\ a = 0$
! a. *fmul* $a\ 0 = 0$
fadd $0\ 0 = 0$
shows
Rep-matrix(*mult-matrix fmul fadd* $A\ B$) $j\ i$ =
foldseq fadd (% k. *fmul* (*Rep-matrix* $A\ j\ k$) (*Rep-matrix* $B\ k\ i$)) (*max* (*ncols* A) (*nrows* B))
⟨*proof*⟩

lemma *transpose-mult-matrix*:
assumes
! a. *fmul* $0\ a = 0$
! a. *fmul* $a\ 0 = 0$
fadd $0\ 0 = 0$
! $x\ y$. *fmul* $y\ x$ = *fmul* $x\ y$
shows
transpose-matrix (*mult-matrix fmul fadd* $A\ B$) = *mult-matrix fmul fadd* (*transpose-matrix* B) (*transpose-matrix* A)
⟨*proof*⟩

lemma *column-transpose-matrix*: *column-of-matrix* (*transpose-matrix* A) n = *transpose-matrix* (*row-of-matrix* $A\ n$)
⟨*proof*⟩

lemma *take-columns-transpose-matrix*: *take-columns* (*transpose-matrix* A) n = *transpose-matrix* (*take-rows* $A\ n$)
⟨*proof*⟩

instantiation *matrix* :: ({*ord*, *zero*}) *ord*
begin

definition
le-matrix-def: $A \le B \longleftrightarrow (\forall j\ i.$ *Rep-matrix* $A\ j\ i \le$ *Rep-matrix* $B\ j\ i$)

definition
less-def: $A < (B::'a\ matrix) \longleftrightarrow A \le B \land A \neq B$

instance ⟨*proof*⟩

end

instance *matrix* :: ({*order*, *zero*}) *order*

⟨*proof*⟩

lemma *le-apply-matrix*:
assumes
$f\ 0 = 0$
$!\ x\ y.\ x <= y \longrightarrow f\ x <= f\ y$
$(a::('a::\{ord,\ zero\})\ matrix) <= b$
shows
$apply\text{-}matrix\ f\ a <= apply\text{-}matrix\ f\ b$
⟨*proof*⟩

lemma *le-combine-matrix*:
assumes
$f\ 0\ 0 = 0$
$!\ a\ b\ c\ d.\ a <= b\ \&\ c <= d \longrightarrow f\ a\ c <= f\ b\ d$
$A <= B$
$C <= D$
shows
$combine\text{-}matrix\ f\ A\ C <= combine\text{-}matrix\ f\ B\ D$
⟨*proof*⟩

lemma *le-left-combine-matrix*:
assumes
$f\ 0\ 0 = 0$
$!\ a\ b\ c.\ a <= b \longrightarrow f\ c\ a <= f\ c\ b$
$A <= B$
shows
$combine\text{-}matrix\ f\ C\ A <= combine\text{-}matrix\ f\ C\ B$
⟨*proof*⟩

lemma *le-right-combine-matrix*:
assumes
$f\ 0\ 0 = 0$
$!\ a\ b\ c.\ a <= b \longrightarrow f\ a\ c <= f\ b\ c$
$A <= B$
shows
$combine\text{-}matrix\ f\ A\ C <= combine\text{-}matrix\ f\ B\ C$
⟨*proof*⟩

lemma *le-transpose-matrix*: $(A <= B) = (transpose\text{-}matrix\ A <= transpose\text{-}matrix\ B)$
⟨*proof*⟩

lemma *le-foldseq*:
assumes
$!\ a\ b\ c\ d\ .\ a <= b\ \&\ c <= d \longrightarrow f\ a\ c <= f\ b\ d$
$!\ i.\ i <= n \longrightarrow s\ i <= t\ i$
shows
$foldseq\ f\ s\ n <= foldseq\ f\ t\ n$

⟨*proof*⟩

lemma *le-left-mult*:
assumes
! *a b c d. a <= b & c <= d* ⟶ *fadd a c <= fadd b d*
! *c a b. 0 <= c & a <= b* ⟶ *fmul c a <= fmul c b*
! *a. fmul 0 a = 0*
! *a. fmul a 0 = 0*
fadd 0 0 = 0
0 <= C
A <= B
shows
mult-matrix fmul fadd C A <= mult-matrix fmul fadd C B
⟨*proof*⟩

lemma *le-right-mult*:
assumes
! *a b c d. a <= b & c <= d* ⟶ *fadd a c <= fadd b d*
! *c a b. 0 <= c & a <= b* ⟶ *fmul a c <= fmul b c*
! *a. fmul 0 a = 0*
! *a. fmul a 0 = 0*
fadd 0 0 = 0
0 <= C
A <= B
shows
mult-matrix fmul fadd A C <= mult-matrix fmul fadd B C
⟨*proof*⟩

lemma *spec2*: ! *j i. P j i* ⟹ *P j i* ⟨*proof*⟩
lemma *neg-imp*: (¬ *Q* ⟶ ¬ *P*) ⟹ *P* ⟶ *Q* ⟨*proof*⟩

lemma *singleton-matrix-le*[*simp*]: (*singleton-matrix j i a <= singleton-matrix j i b*) = (*a <= (b::-::order)*)
⟨*proof*⟩

lemma *singleton-le-zero*[*simp*]: (*singleton-matrix j i x <= 0*) = (*x <= (0::'a::{order,zero})*)
⟨*proof*⟩

lemma *singleton-ge-zero*[*simp*]: (*0 <= singleton-matrix j i x*) = ((*0::'a::{order,zero}*) *<= x*)
⟨*proof*⟩

lemma *move-matrix-le-zero*[*simp*]: *0 <= j* ⟹ *0 <= i* ⟹ (*move-matrix A j i <= 0*) = (*A <= (0::('a::{order,zero}) matrix)*)
⟨*proof*⟩

lemma *move-matrix-zero-le*[*simp*]: *0 <= j* ⟹ *0 <= i* ⟹ (*0 <= move-matrix A j i*) = ((*0::('a::{order,zero}) matrix*) *<= A*)
⟨*proof*⟩

lemma *move-matrix-le-move-matrix-iff* [*simp*]: *0* <= *j* ⟹ *0* <= *i* ⟹ (*move-matrix* *A j i* <= *move-matrix B j i*) = (*A* <= (*B*::(*'a*::{*order*,*zero*}) *matrix*))
⟨*proof*⟩

end

theory *Matrix*
imports *MatrixGeneral*
begin

instantiation *matrix* :: ({*zero*, *lattice*}) *lattice*
begin

definition
inf = *combine-matrix inf*

definition
sup = *combine-matrix sup*

instance
⟨*proof*⟩

end

instantiation *matrix* :: ({*plus*, *zero*}) *plus*
begin

definition
plus-matrix-def: *A* + *B* = *combine-matrix* (*op* +) *A B*

instance ⟨*proof*⟩

end

instantiation *matrix* :: ({*uminus*, *zero*}) *uminus*
begin

definition
minus-matrix-def: − *A* = *apply-matrix uminus A*

instance ⟨*proof*⟩

end

instantiation *matrix* :: ({*minus*, *zero*}) *minus*
begin

definition
diff-matrix-def: $A - B = combine\text{-}matrix\ (op\ -)\ A\ B$

instance ⟨*proof*⟩

end

instantiation *matrix* :: ({*plus*, *times*, *zero*}) *times*
begin

definition
times-matrix-def: $A * B = mult\text{-}matrix\ (op\ *)\ (op\ +)\ A\ B$

instance ⟨*proof*⟩

end

instantiation *matrix* :: (*lordered-ab-group-add*) *abs*
begin

definition
abs-matrix-def: $abs\ (A :: 'a\ matrix) = sup\ A\ (-\ A)$

instance ⟨*proof*⟩

end

instance *matrix* :: (*lordered-ab-group-add*) *lordered-ab-group-add-meet*
⟨*proof*⟩

instance *matrix* :: (*lordered-ring*) *lordered-ring*
⟨*proof*⟩

lemma *Rep-matrix-add*[*simp*]:
Rep-matrix ((*a*::('*a*::*lordered-ab-group-add*)*matrix*)+*b*) *j i* = (*Rep-matrix a j i*) + (*Rep-matrix b j i*)
⟨*proof*⟩

lemma *Rep-matrix-mult*: *Rep-matrix* ((*a*::('*a*::*lordered-ring*) *matrix*) ∗ *b*) *j i* =
foldseq (*op* +) (% *k*. (*Rep-matrix a j k*) ∗ (*Rep-matrix b k i*)) (*max* (*ncols a*) (*nrows b*))
⟨*proof*⟩

lemma *apply-matrix-add*: ! *x y*. *f* (*x*+*y*) = (*f x*) + (*f y*) ⟹ *f 0* = (*0*::'*a*) ⟹ *apply-matrix f* ((*a*::('*a*::*lordered-ab-group-add*) *matrix*) + *b*) = (*apply-matrix f a*) + (*apply-matrix f b*)
⟨*proof*⟩

lemma *singleton-matrix-add*: *singleton-matrix j i* ((*a*::-::*lordered-ab-group-add*)+*b*) = (*singleton-matrix j i a*) + (*singleton-matrix j i b*)
⟨*proof*⟩

lemma *nrows-mult*: *nrows* ((*A*::(′*a*::*lordered-ring*) *matrix*) ∗ *B*) <= *nrows A*
⟨*proof*⟩

lemma *ncols-mult*: *ncols* ((*A*::(′*a*::*lordered-ring*) *matrix*) ∗ *B*) <= *ncols B*
⟨*proof*⟩

definition
one-matrix :: *nat* ⇒ (′*a*::{*zero*,*one*}) *matrix* **where**
one-matrix n = *Abs-matrix* (% *j i*. *if j* = *i* & *j* < *n then 1 else 0*)

lemma *Rep-one-matrix*[*simp*]: *Rep-matrix* (*one-matrix n*) *j i* = (*if* (*j* = *i* & *j* < *n*) *then 1 else 0*)
⟨*proof*⟩

lemma *nrows-one-matrix*[*simp*]: *nrows* ((*one-matrix n*) :: (′*a*::*zero-neq-one*)*matrix*) = *n* (**is** *?r* = -)
⟨*proof*⟩

lemma *ncols-one-matrix*[*simp*]: *ncols* ((*one-matrix n*) :: (′*a*::*zero-neq-one*)*matrix*) = *n* (**is** *?r* = -)
⟨*proof*⟩

lemma *one-matrix-mult-right*[*simp*]: *ncols A* <= *n* ⟹ (*A*::(′*a*::{*lordered-ring*,*ring-1*}) *matrix*) ∗ (*one-matrix n*) = *A*
⟨*proof*⟩

lemma *one-matrix-mult-left*[*simp*]: *nrows A* <= *n* ⟹ (*one-matrix n*) ∗ *A* = (*A*::(′*a*::{*lordered-ring*, *ring-1*}) *matrix*)
⟨*proof*⟩

lemma *transpose-matrix-mult*: *transpose-matrix* ((*A*::(′*a*::{*lordered-ring*,*comm-ring*}) *matrix*)∗*B*) = (*transpose-matrix B*) ∗ (*transpose-matrix A*)
⟨*proof*⟩

lemma *transpose-matrix-add*: *transpose-matrix* ((*A*::(′*a*::*lordered-ab-group-add*) *matrix*)+*B*) = *transpose-matrix A* + *transpose-matrix B*
⟨*proof*⟩

lemma *transpose-matrix-diff*: *transpose-matrix* ((*A*::(′*a*::*lordered-ab-group-add*) *matrix*)−*B*) = *transpose-matrix A* − *transpose-matrix B*
⟨*proof*⟩

lemma *transpose-matrix-minus*: *transpose-matrix* (−(*A*::(′*a*::*lordered-ring*) *matrix*)) = − *transpose-matrix* (*A*::(′*a*::*lordered-ring*) *matrix*)
⟨*proof*⟩

constdefs
right-inverse-matrix :: $('a::\{lordered\text{-}ring, ring\text{-}1\})$ *matrix* $\Rightarrow$ *'a matrix* $\Rightarrow$ *bool*
right-inverse-matrix A X == $(A * X = one\text{-}matrix\ (max\ (nrows\ A)\ (ncols\ X)))$ $\wedge$ *nrows* $X \leq$ *ncols* A
left-inverse-matrix :: $('a::\{lordered\text{-}ring, ring\text{-}1\})$ *matrix* $\Rightarrow$ *'a matrix* $\Rightarrow$ *bool*
left-inverse-matrix A X == $(X * A = one\text{-}matrix\ (max(nrows\ X)\ (ncols\ A)))$ $\wedge$ *ncols* $X \leq$ *nrows* A
inverse-matrix :: $('a::\{lordered\text{-}ring, ring\text{-}1\})$ *matrix* $\Rightarrow$ *'a matrix* $\Rightarrow$ *bool*
inverse-matrix A X == $(right\text{-}inverse\text{-}matrix\ A\ X) \wedge (left\text{-}inverse\text{-}matrix\ A\ X)$

lemma *right-inverse-matrix-dim*: *right-inverse-matrix* A $X \Longrightarrow$ *nrows* $A =$ *ncols* X
⟨*proof*⟩

lemma *left-inverse-matrix-dim*: *left-inverse-matrix* A $Y \Longrightarrow$ *ncols* $A =$ *nrows* Y
⟨*proof*⟩

lemma *left-right-inverse-matrix-unique*:
assumes *left-inverse-matrix* A Y *right-inverse-matrix* A X
shows $X = Y$
⟨*proof*⟩

lemma *inverse-matrix-inject*: $[\![$ *inverse-matrix* A X; *inverse-matrix* A Y $]\!] \Longrightarrow X = Y$
⟨*proof*⟩

lemma *one-matrix-inverse*: *inverse-matrix* (*one-matrix* n) (*one-matrix* n)
⟨*proof*⟩

lemma *zero-imp-mult-zero*: $(a::'a::ring) = 0 \mid b = 0 \Longrightarrow a * b = 0$
⟨*proof*⟩

lemma *Rep-matrix-zero-imp-mult-zero*:
$!\ j\ i\ k.\ (Rep\text{-}matrix\ A\ j\ k = 0) \mid (Rep\text{-}matrix\ B\ k\ i) = 0 \Longrightarrow A * B = (0::('a::lordered\text{-}ring)\ matrix)$
⟨*proof*⟩

lemma *add-nrows*: *nrows* $(A::('a::comm\text{-}monoid\text{-}add)\ matrix) <= u \Longrightarrow$ *nrows* $B <= u \Longrightarrow$ *nrows* $(A + B) <= u$
⟨*proof*⟩

lemma *move-matrix-row-mult*: *move-matrix* $((A::('a::lordered\text{-}ring)\ matrix) * B)$ j 0 = (*move-matrix* A j 0) $* B$
⟨*proof*⟩

lemma *move-matrix-col-mult*: *move-matrix* $((A::('a::lordered\text{-}ring)\ matrix) * B)$ 0 i = $A *$ (*move-matrix* B 0 i)
⟨*proof*⟩

lemma *move-matrix-add*: ((*move-matrix* (*A* + *B*) *j i*)::((′*a*::*lordered-ab-group-add*) *matrix*)) = (*move-matrix A j i*) + (*move-matrix B j i*)
⟨*proof*⟩

lemma *move-matrix-mult*: *move-matrix* ((*A*::(′*a*::*lordered-ring*) *matrix*)∗*B*) *j i* = (*move-matrix A j 0*) ∗ (*move-matrix B 0 i*)
⟨*proof*⟩

constdefs
  scalar-mult :: (′*a*::*lordered-ring*) ⇒ ′*a matrix* ⇒ ′*a matrix*
  scalar-mult a m == *apply-matrix* (*op* ∗ *a*) *m*

lemma *scalar-mult-zero*[*simp*]: *scalar-mult y 0* = *0*
⟨*proof*⟩

lemma *scalar-mult-add*: *scalar-mult y* (*a*+*b*) = (*scalar-mult y a*) + (*scalar-mult y b*)
⟨*proof*⟩

lemma *Rep-scalar-mult*[*simp*]: *Rep-matrix* (*scalar-mult y a*) *j i* = *y* ∗ (*Rep-matrix a j i*)
⟨*proof*⟩

lemma *scalar-mult-singleton*[*simp*]: *scalar-mult y* (*singleton-matrix j i x*) = *singleton-matrix j i* (*y* ∗ *x*)
⟨*proof*⟩

lemma *Rep-minus*[*simp*]: *Rep-matrix* (−(*A*::-::*lordered-ab-group-add*)) *x y* = − (*Rep-matrix A x y*)
⟨*proof*⟩

lemma *Rep-abs*[*simp*]: *Rep-matrix* (*abs* (*A*::-::*lordered-ring*)) *x y* = *abs* (*Rep-matrix A x y*)
⟨*proof*⟩

end

theory *LP*
imports *Main*
begin

lemma *linprog-dual-estimate*:
  assumes
  A ∗ *x* ≤ (*b*::′*a*::*lordered-ring*)
  0 ≤ *y*
  abs (*A* − *A*′) ≤ δ*A*

$b \le b'$
$abs\ (c - c') \le \delta c$
$abs\ x \le r$
shows
$c * x \le y * b' + (y * \delta A + abs\ (y * A' - c') + \delta c) * r$
⟨*proof*⟩

lemma *le-ge-imp-abs-diff-1*:
assumes
$A1 <= (A::'a::lordered\text{-}ring)$
$A <= A2$
shows $abs\ (A-A1) <= A2-A1$
⟨*proof*⟩

lemma *mult-le-prts*:
assumes
$a1 <= (a::'a::lordered\text{-}ring)$
$a <= a2$
$b1 <= b$
$b <= b2$
shows
$a * b <= pprt\ a2 * pprt\ b2 + pprt\ a1 * nprt\ b2 + nprt\ a2 * pprt\ b1 + nprt\ a1 * nprt\ b1$
⟨*proof*⟩

lemma *mult-le-dual-prts*:
assumes
$A * x \le (b::'a::lordered\text{-}ring)$
$0 \le y$
$A1 \le A$
$A \le A2$
$c1 \le c$
$c \le c2$
$r1 \le x$
$x \le r2$
shows
$c * x \le y * b + (let\ s1 = c1 - y * A2;\ s2 = c2 - y * A1\ in\ pprt\ s2 * pprt\ r2 + pprt\ s1 * nprt\ r2 + nprt\ s2 * pprt\ r1 + nprt\ s1 * nprt\ r1)$
(**is** - <= - + *?C*)
⟨*proof*⟩

end

theory *SparseMatrix* **imports** *Matrix LP* **begin**

types
$'a\ spvec = (nat * 'a)\ list$
$'a\ spmat = ('a\ spvec)\ spvec$

consts
sparse-row-vector :: $('a::lordered\text{-}ring)$ *spvec* $\Rightarrow$ *'a matrix*
sparse-row-matrix :: $('a::lordered\text{-}ring)$ *spmat* $\Rightarrow$ *'a matrix*

defs
sparse-row-vector-def : *sparse-row-vector arr* == *foldl* (% *m x*. *m* + (*singleton-matrix 0* (*fst x*) (*snd x*))) *0 arr*
sparse-row-matrix-def : *sparse-row-matrix arr* == *foldl* (% *m r*. *m* + (*move-matrix* (*sparse-row-vector* (*snd r*)) (*int* (*fst r*)) *0*)) *0 arr*

lemma *sparse-row-vector-empty*[*simp*]: *sparse-row-vector* [] = *0*
⟨*proof*⟩

lemma *sparse-row-matrix-empty*[*simp*]: *sparse-row-matrix* [] = *0*
⟨*proof*⟩

lemma *foldl-distrstart*[*rule-format*]: ! *a x y*. (*f* (*g x y*) *a* = *g x* (*f y a*)) $\Longrightarrow$! *x y*. (*foldl f* (*g x y*) *l* = *g x* (*foldl f y l*))
⟨*proof*⟩

lemma *sparse-row-vector-cons*[*simp*]: *sparse-row-vector* (*a*#*arr*) = (*singleton-matrix 0* (*fst a*) (*snd a*)) + (*sparse-row-vector arr*)
⟨*proof*⟩

lemma *sparse-row-vector-append*[*simp*]: *sparse-row-vector* (*a* @ *b*) = (*sparse-row-vector a*) + (*sparse-row-vector b*)
⟨*proof*⟩

lemma *nrows-spvec*[*simp*]: *nrows* (*sparse-row-vector x*) <= (*Suc 0*)
⟨*proof*⟩

lemma *sparse-row-matrix-cons*: *sparse-row-matrix* (*a*#*arr*) = ((*move-matrix* (*sparse-row-vector* (*snd a*)) (*int* (*fst a*)) *0*)) + *sparse-row-matrix arr*
⟨*proof*⟩

lemma *sparse-row-matrix-append*: *sparse-row-matrix* (*arr*@*brr*) = (*sparse-row-matrix arr*) + (*sparse-row-matrix brr*)
⟨*proof*⟩

consts
sorted-spvec :: *'a spvec* $\Rightarrow$ *bool*
sorted-spmat :: *'a spmat* $\Rightarrow$ *bool*

primrec
sorted-spmat [] = *True*
sorted-spmat (*a*#*as*) = ((*sorted-spvec* (*snd a*)) & (*sorted-spmat as*))

primrec

sorted-spvec [] = *True*
sorted-spvec-step: *sorted-spvec* (*a*#*as*) = (*case as of* [] ⇒ *True* | *b*#*bs* ⇒ ((*fst a* < *fst b*) & (*sorted-spvec as*)))

declare *sorted-spvec.simps* [*simp del*]

lemma *sorted-spvec-empty*[*simp*]: *sorted-spvec* [] = *True*
⟨*proof*⟩

lemma *sorted-spvec-cons1*: *sorted-spvec* (*a*#*as*) ⟹ *sorted-spvec as*
⟨*proof*⟩

lemma *sorted-spvec-cons2*: *sorted-spvec* (*a*#*b*#*t*) ⟹ *sorted-spvec* (*a*#*t*)
⟨*proof*⟩

lemma *sorted-spvec-cons3*: *sorted-spvec*(*a*#*b*#*t*) ⟹ *fst a* < *fst b*
⟨*proof*⟩

lemma *sorted-sparse-row-vector-zero*[*rule-format*]: *m* <= *n* ⟶ *sorted-spvec* ((*n*,*a*)#*arr*) ⟶ *Rep-matrix* (*sparse-row-vector arr*) *j m* = *0*
⟨*proof*⟩

lemma *sorted-sparse-row-matrix-zero*[*rule-format*]: *m* <= *n* ⟶ *sorted-spvec* ((*n*,*a*)#*arr*) ⟶ *Rep-matrix* (*sparse-row-matrix arr*) *m j* = *0*
⟨*proof*⟩

consts
abs-spvec :: ('*a*::*lordered-ring*) *spvec* ⇒ '*a spvec*
minus-spvec :: ('*a*::*lordered-ring*) *spvec* ⇒ '*a spvec*
smult-spvec :: ('*a*::*lordered-ring*) ⇒ '*a spvec* ⇒ '*a spvec*
addmult-spvec :: ('*a*::*lordered-ring*) ∗ '*a spvec* ∗ '*a spvec* ⇒ '*a spvec*

primrec
minus-spvec [] = []
minus-spvec (*a*#*as*) = (*fst a*, −(*snd a*))#(*minus-spvec as*)

primrec
abs-spvec [] = []
abs-spvec (*a*#*as*) = (*fst a*, *abs* (*snd a*))#(*abs-spvec as*)

lemma *sparse-row-vector-minus*:
sparse-row-vector (*minus-spvec v*) = − (*sparse-row-vector v*)
⟨*proof*⟩

lemma *sparse-row-vector-abs*:
sorted-spvec v ⟹ *sparse-row-vector* (*abs-spvec v*) = *abs* (*sparse-row-vector v*)
⟨*proof*⟩

lemma *sorted-spvec-minus-spvec*:

sorted-spvec v $\Longrightarrow$ *sorted-spvec* (*minus-spvec v*)
⟨*proof*⟩

lemma *sorted-spvec-abs-spvec*:
sorted-spvec v $\Longrightarrow$ *sorted-spvec* (*abs-spvec v*)
⟨*proof*⟩

defs
smult-spvec-def: *smult-spvec y arr* == *map* (% *a*. (*fst a*, *y* $*$ *snd a*)) *arr*

lemma *smult-spvec-empty*[*simp*]: *smult-spvec y* [] = []
⟨*proof*⟩

lemma *smult-spvec-cons*: *smult-spvec y* (*a*#*arr*) = (*fst a*, *y* $*$ (*snd a*)) # (*smult-spvec y arr*)
⟨*proof*⟩

recdef *addmult-spvec measure* (% (*y*, *a*, *b*). *length a* + (*length b*))
addmult-spvec (*y*, *arr*, []) = *arr*
addmult-spvec (*y*, [], *brr*) = *smult-spvec y brr*
addmult-spvec (*y*, *a*#*arr*, *b*#*brr*) = (
if (*fst a*) < (*fst b*) *then* (*a*#(*addmult-spvec* (*y*, *arr*, *b*#*brr*)))
else (*if* (*fst b* < *fst a*) *then* ((*fst b*, *y* $*$ (*snd b*))#(*addmult-spvec* (*y*, *a*#*arr*, *brr*)))
else ((*fst a*, (*snd a*)+ *y*$*$(*snd b*))#(*addmult-spvec* (*y*, *arr*,*brr*)))))

lemma *addmult-spvec-empty1*[*simp*]: *addmult-spvec* (*y*, [], *a*) = *smult-spvec y a*
⟨*proof*⟩

lemma *addmult-spvec-empty2*[*simp*]: *addmult-spvec* (*y*, *a*, []) = *a*
⟨*proof*⟩

lemma *sparse-row-vector-map*: (! *x y*. *f* (*x*+*y*) = (*f x*) + (*f y*)) $\Longrightarrow$ (*f*::'*a*⇒('*a*::*lordered-ring*)) *0* = *0* $\Longrightarrow$
sparse-row-vector (*map* (% *x*. (*fst x*, *f* (*snd x*))) *a*) = *apply-matrix f* (*sparse-row-vector a*)
⟨*proof*⟩

lemma *sparse-row-vector-smult*: *sparse-row-vector* (*smult-spvec y a*) = *scalar-mult y* (*sparse-row-vector a*)
⟨*proof*⟩

lemma *sparse-row-vector-addmult-spvec*: *sparse-row-vector* (*addmult-spvec* (*y*::'*a*::*lordered-ring*, *a*, *b*)) =
(*sparse-row-vector a*) + (*scalar-mult y* (*sparse-row-vector b*))
⟨*proof*⟩

lemma *sorted-smult-spvec*[*rule-format*]: *sorted-spvec a* $\Longrightarrow$ *sorted-spvec* (*smult-spvec y a*)

⟨*proof*⟩

lemma *sorted-spvec-addmult-spvec-helper*: ⟦*sorted-spvec* (*addmult-spvec* (*y*, (*a*, *b*) # *arr*, *brr*)); *aa* < *a*; *sorted-spvec* ((*a*, *b*) # *arr*);
sorted-spvec ((*aa*, *ba*) # *brr*)⟧ ⟹ *sorted-spvec* ((*aa*, *y* ∗ *ba*) # *addmult-spvec* (*y*, (*a*, *b*) # *arr*, *brr*))
⟨*proof*⟩

lemma *sorted-spvec-addmult-spvec-helper2*:
⟦*sorted-spvec* (*addmult-spvec* (*y*, *arr*, (*aa*, *ba*) # *brr*)); *a* < *aa*; *sorted-spvec* ((*a*, *b*) # *arr*); *sorted-spvec* ((*aa*, *ba*) # *brr*)⟧
⟹ *sorted-spvec* ((*a*, *b*) # *addmult-spvec* (*y*, *arr*, (*aa*, *ba*) # *brr*))
⟨*proof*⟩

lemma *sorted-spvec-addmult-spvec-helper3*[*rule-format*]:
sorted-spvec (*addmult-spvec* (*y*, *arr*, *brr*)) ⟶ *sorted-spvec* ((*aa*, *b*) # *arr*) ⟶ *sorted-spvec* ((*aa*, *ba*) # *brr*)
⟶ *sorted-spvec* ((*aa*, *b* + *y* ∗ *ba*) # (*addmult-spvec* (*y*, *arr*, *brr*)))
⟨*proof*⟩

lemma *sorted-addmult-spvec*[*rule-format*]: *sorted-spvec* *a* ⟶ *sorted-spvec* *b* ⟶ *sorted-spvec* (*addmult-spvec* (*y*, *a*, *b*))
⟨*proof*⟩

consts
mult-spvec-spmat :: (′*a*::*lordered-ring*) *spvec* ∗ ′*a spvec* ∗ ′*a spmat* ⇒ ′*a spvec*

recdef *mult-spvec-spmat measure* (% (*c*, *arr*, *brr*). (*length arr*) + (*length brr*))
mult-spvec-spmat (*c*, [], *brr*) = *c*
mult-spvec-spmat (*c*, *arr*, []) = *c*
mult-spvec-spmat (*c*, *a*#*arr*, *b*#*brr*) = (
if ((*fst a*) < (*fst b*)) *then* (*mult-spvec-spmat* (*c*, *arr*, *b*#*brr*))
else (*if* ((*fst b*) < (*fst a*)) *then* (*mult-spvec-spmat* (*c*, *a*#*arr*, *brr*))
else (*mult-spvec-spmat* (*addmult-spvec* (*snd a*, *c*, *snd b*), *arr*, *brr*))))

lemma *sparse-row-mult-spvec-spmat*[*rule-format*]: *sorted-spvec* (*a*::(′*a*::*lordered-ring*) *spvec*) ⟶ *sorted-spvec B* ⟶
sparse-row-vector (*mult-spvec-spmat* (*c*, *a*, *B*)) = (*sparse-row-vector c*) + (*sparse-row-vector a*) ∗ (*sparse-row-matrix B*)
⟨*proof*⟩

lemma *sorted-mult-spvec-spmat*[*rule-format*]:
sorted-spvec (*c*::(′*a*::*lordered-ring*) *spvec*) ⟶ *sorted-spmat B* ⟶ *sorted-spvec* (*mult-spvec-spmat* (*c*, *a*, *B*))
⟨*proof*⟩

consts
mult-spmat :: (′*a*::*lordered-ring*) *spmat* ⇒ ′*a spmat* ⇒ ′*a spmat*

primrec
mult-spmat [] A = []
mult-spmat (a#as) A = (*fst* a, *mult-spvec-spmat* ([], *snd* a, A))#(*mult-spmat* as A)

lemma *sparse-row-mult-spmat*[*rule-format*]:
sorted-spmat A ⟶ *sorted-spvec* B ⟶ *sparse-row-matrix* (*mult-spmat* A B) = (*sparse-row-matrix* A) ∗ (*sparse-row-matrix* B)
⟨*proof*⟩

lemma *sorted-spvec-mult-spmat*[*rule-format*]:
sorted-spvec (A::('a::*lordered-ring*) *spmat*) ⟶ *sorted-spvec* (*mult-spmat* A B)
⟨*proof*⟩

lemma *sorted-spmat-mult-spmat*[*rule-format*]:
sorted-spmat (B::('a::*lordered-ring*) *spmat*) ⟶ *sorted-spmat* (*mult-spmat* A B)
⟨*proof*⟩

consts
add-spvec :: ('a::*lordered-ab-group-add*) *spvec* ∗ 'a *spvec* ⇒ 'a *spvec*
add-spmat :: ('a::*lordered-ab-group-add*) *spmat* ∗ 'a *spmat* ⇒ 'a *spmat*

recdef *add-spvec measure* (% (a, b). *length* a + (*length* b))
add-spvec (*arr*, []) = *arr*
add-spvec ([], *brr*) = *brr*
add-spvec (a#*arr*, b#*brr*) = (
if (*fst* a) < (*fst* b) *then* (a#(*add-spvec* (*arr*, b#*brr*)))
else (*if* (*fst* b < *fst* a) *then* (b#(*add-spvec* (a#*arr*, *brr*)))
else ((*fst* a, (*snd* a)+(*snd* b))#(*add-spvec* (*arr*,*brr*)))))

lemma *add-spvec-empty1*[*simp*]: *add-spvec* ([], a) = a
⟨*proof*⟩

lemma *add-spvec-empty2*[*simp*]: *add-spvec* (a, []) = a
⟨*proof*⟩

lemma *sparse-row-vector-add*: *sparse-row-vector* (*add-spvec* (a,b)) = (*sparse-row-vector* a) + (*sparse-row-vector* b)
⟨*proof*⟩

recdef *add-spmat measure* (% (A,B). (*length* A)+(*length* B))
add-spmat ([], *bs*) = *bs*
add-spmat (*as*, []) = *as*
add-spmat (a#*as*, b#*bs*) = (
if fst a < *fst* b *then*
(a#(*add-spmat* (*as*, b#*bs*)))
else (*if fst* b < *fst* a *then*
(b#(*add-spmat* (a#*as*, *bs*)))
else

((*fst a*, *add-spvec* (*snd a*, *snd b*))#(*add-spmat* (*as*, *bs*)))))

lemma *sparse-row-add-spmat*: *sparse-row-matrix* (*add-spmat* (*A*, *B*)) = (*sparse-row-matrix A*) + (*sparse-row-matrix B*)
⟨*proof*⟩

lemma *sorted-add-spvec-helper1*[*rule-format*]: *add-spvec* ((*a*,*b*)#*arr*, *brr*) = (*ab*, *bb*) # *list* ⟶ (*ab* = *a* | (*brr* ≠ [] & *ab* = *fst* (*hd brr*)))
⟨*proof*⟩

lemma *sorted-add-spmat-helper1*[*rule-format*]: *add-spmat* ((*a*,*b*)#*arr*, *brr*) = (*ab*, *bb*) # *list* ⟶ (*ab* = *a* | (*brr* ≠ [] & *ab* = *fst* (*hd brr*)))
⟨*proof*⟩

lemma *sorted-add-spvec-helper*[*rule-format*]: *add-spvec* (*arr*, *brr*) = (*ab*, *bb*) # *list* ⟶ ((*arr* ≠ [] & *ab* = *fst* (*hd arr*)) | (*brr* ≠ [] & *ab* = *fst* (*hd brr*)))
⟨*proof*⟩

lemma *sorted-add-spmat-helper*[*rule-format*]: *add-spmat* (*arr*, *brr*) = (*ab*, *bb*) # *list* ⟶ ((*arr* ≠ [] & *ab* = *fst* (*hd arr*)) | (*brr* ≠ [] & *ab* = *fst* (*hd brr*)))
⟨*proof*⟩

lemma *add-spvec-commute*: *add-spvec* (*a*, *b*) = *add-spvec* (*b*, *a*)
⟨*proof*⟩

lemma *add-spmat-commute*: *add-spmat* (*a*, *b*) = *add-spmat* (*b*, *a*)
⟨*proof*⟩

lemma *sorted-add-spvec-helper2*: *add-spvec* ((*a*,*b*)#*arr*, *brr*) = (*ab*, *bb*) # *list* ⟹ *aa* < *a* ⟹ *sorted-spvec* ((*aa*, *ba*) # *brr*) ⟹ *aa* < *ab*
⟨*proof*⟩

lemma *sorted-add-spmat-helper2*: *add-spmat* ((*a*,*b*)#*arr*, *brr*) = (*ab*, *bb*) # *list* ⟹ *aa* < *a* ⟹ *sorted-spvec* ((*aa*, *ba*) # *brr*) ⟹ *aa* < *ab*
⟨*proof*⟩

lemma *sorted-spvec-add-spvec*[*rule-format*]: *sorted-spvec a* ⟶ *sorted-spvec b* ⟶ *sorted-spvec* (*add-spvec* (*a*, *b*))
⟨*proof*⟩

lemma *sorted-spvec-add-spmat*[*rule-format*]: *sorted-spvec A* ⟶ *sorted-spvec B* ⟶ *sorted-spvec* (*add-spmat* (*A*, *B*))
⟨*proof*⟩

lemma *sorted-spmat-add-spmat*[*rule-format*]: *sorted-spmat A* ⟶ *sorted-spmat B* ⟶ *sorted-spmat* (*add-spmat* (*A*, *B*))
⟨*proof*⟩

consts

le-spvec :: (*'a*::*lordered-ab-group-add*) *spvec* ∗ *'a spvec* ⇒ *bool*
le-spmat :: (*'a*::*lordered-ab-group-add*) *spmat* ∗ *'a spmat* ⇒ *bool*

recdef *le-spvec measure* (% (*a*,*b*). (*length a*) + (*length b*))
le-spvec ([], []) = *True*
le-spvec (*a*#*as*, []) = ((*snd a* <= *0*) & (*le-spvec* (*as*, [])))
le-spvec ([], *b*#*bs*) = ((*0* <= *snd b*) & (*le-spvec* ([], *bs*)))
le-spvec (*a*#*as*, *b*#*bs*) = (
if (*fst a* < *fst b*) *then*
((*snd a* <= *0*) & (*le-spvec* (*as*, *b*#*bs*)))
else (*if* (*fst b* < *fst a*) *then*
((*0* <= *snd b*) & (*le-spvec* (*a*#*as*, *bs*)))
else
((*snd a* <= *snd b*) & (*le-spvec* (*as*, *bs*)))))

recdef *le-spmat measure* (% (*a*,*b*). (*length a*) + (*length b*))
le-spmat ([], []) = *True*
le-spmat (*a*#*as*, []) = (*le-spvec* (*snd a*, []) & (*le-spmat* (*as*, [])))
le-spmat ([], *b*#*bs*) = (*le-spvec* ([], *snd b*) & (*le-spmat* ([], *bs*)))
le-spmat (*a*#*as*, *b*#*bs*) = (
if fst a < *fst b then*
(*le-spvec*(*snd a*,[]) & *le-spmat*(*as*, *b*#*bs*))
else (*if* (*fst b* < *fst a*) *then*
(*le-spvec*([], *snd b*) & *le-spmat*(*a*#*as*, *bs*))
else
(*le-spvec*(*snd a*, *snd b*) & *le-spmat* (*as*, *bs*))))

constdefs
disj-matrices :: (*'a*::*zero*) *matrix* ⇒ *'a matrix* ⇒ *bool*
disj-matrices A B == (! *j i*. (*Rep-matrix A j i* ≠ *0*) ⟶ (*Rep-matrix B j i* = *0*)) & (! *j i*. (*Rep-matrix B j i* ≠ *0*) ⟶ (*Rep-matrix A j i* = *0*))

declare [[*simp-depth-limit* = *6*]]

lemma *disj-matrices-contr1*: *disj-matrices A B* ⟹ *Rep-matrix A j i* ≠ *0* ⟹ *Rep-matrix B j i* = *0*
⟨*proof*⟩

lemma *disj-matrices-contr2*: *disj-matrices A B* ⟹ *Rep-matrix B j i* ≠ *0* ⟹ *Rep-matrix A j i* = *0*
⟨*proof*⟩

lemma *disj-matrices-add*: *disj-matrices A B* ⟹ *disj-matrices C D* ⟹ *disj-matrices A D* ⟹ *disj-matrices B C* ⟹
(*A* + *B* <= *C* + *D*) = (*A* <= *C* & *B* <= (*D*::(*'a*::*lordered-ab-group-add*) *matrix*))
⟨*proof*⟩

lemma *disj-matrices-zero1*[*simp*]: *disj-matrices 0 B*
⟨*proof*⟩

lemma *disj-matrices-zero2*[*simp*]: *disj-matrices A 0*
⟨*proof*⟩

lemma *disj-matrices-commute*: *disj-matrices A B = disj-matrices B A*
⟨*proof*⟩

lemma *disj-matrices-add-le-zero*: *disj-matrices A B* $\Longrightarrow$
(*A + B <= 0*) = (*A <= 0* & (*B*::('*a*::*lordered-ab-group-add*) *matrix*) *<= 0*)
⟨*proof*⟩

lemma *disj-matrices-add-zero-le*: *disj-matrices A B* $\Longrightarrow$
(*0 <= A + B*) = (*0 <= A* & *0 <=* (*B*::('*a*::*lordered-ab-group-add*) *matrix*))
⟨*proof*⟩

lemma *disj-matrices-add-x-le*: *disj-matrices A B* $\Longrightarrow$ *disj-matrices B C* $\Longrightarrow$
(*A <= B + C*) = (*A <= C* & *0 <=* (*B*::('*a*::*lordered-ab-group-add*) *matrix*))
⟨*proof*⟩

lemma *disj-matrices-add-le-x*: *disj-matrices A B* $\Longrightarrow$ *disj-matrices B C* $\Longrightarrow$
(*B + A <= C*) = (*A <= C* & (*B*::('*a*::*lordered-ab-group-add*) *matrix*) *<= 0*)
⟨*proof*⟩

lemma *disj-sparse-row-singleton*: *i <= j* $\Longrightarrow$ *sorted-spvec*((*j*,*y*)#*v*) $\Longrightarrow$ *disj-matrices* (*sparse-row-vector v*) (*singleton-matrix 0 i x*)
⟨*proof*⟩

lemma *disj-matrices-x-add*: *disj-matrices A B* $\Longrightarrow$ *disj-matrices A C* $\Longrightarrow$ *disj-matrices* (*A*::('*a*::*lordered-ab-group-add*) *matrix*) (*B+C*)
⟨*proof*⟩

lemma *disj-matrices-add-x*: *disj-matrices A B* $\Longrightarrow$ *disj-matrices A C* $\Longrightarrow$ *disj-matrices* (*B+C*) (*A*::('*a*::*lordered-ab-group-add*) *matrix*)
⟨*proof*⟩

lemma *disj-singleton-matrices*[*simp*]: *disj-matrices* (*singleton-matrix j i x*) (*singleton-matrix u v y*) = (*j* $\neq$ *u* | *i* $\neq$ *v* | *x = 0* | *y = 0*)
⟨*proof*⟩

lemma *disj-move-sparse-vec-mat*[*simplified disj-matrices-commute*]:
j <= a $\Longrightarrow$ *sorted-spvec*((*a*,*c*)#*as*) $\Longrightarrow$ *disj-matrices* (*move-matrix* (*sparse-row-vector b*) (*int j*) *i*) (*sparse-row-matrix as*)
⟨*proof*⟩

lemma *disj-move-sparse-row-vector-twice*:
j $\neq$ *u* $\Longrightarrow$ *disj-matrices* (*move-matrix* (*sparse-row-vector a*) *j i*) (*move-matrix* (*sparse-row-vector b*) *u v*)

⟨*proof*⟩

lemma *le-spvec-iff-sparse-row-le*[*rule-format*]: (*sorted-spvec a*) $\longrightarrow$ (*sorted-spvec b*) $\longrightarrow$ (*le-spvec* (*a*,*b*)) = (*sparse-row-vector a* <= *sparse-row-vector b*)
⟨*proof*⟩

lemma *le-spvec-empty2-sparse-row*[*rule-format*]: (*sorted-spvec b*) $\longrightarrow$ (*le-spvec* (*b*,[]) = (*sparse-row-vector b* <= *0*))
⟨*proof*⟩

lemma *le-spvec-empty1-sparse-row*[*rule-format*]: (*sorted-spvec b*) $\longrightarrow$ (*le-spvec* ([],*b*) = (*0* <= *sparse-row-vector b*))
⟨*proof*⟩

lemma *le-spmat-iff-sparse-row-le*[*rule-format*]: (*sorted-spvec A*) $\longrightarrow$ (*sorted-spmat A*) $\longrightarrow$ (*sorted-spvec B*) $\longrightarrow$ (*sorted-spmat B*) $\longrightarrow$
le-spmat(*A*, *B*) = (*sparse-row-matrix A* <= *sparse-row-matrix B*)
⟨*proof*⟩

declare [[*simp-depth-limit* = *999*]]

consts
abs-spmat :: (*'a*::*lordered-ring*) *spmat* ⇒ *'a spmat*
minus-spmat :: (*'a*::*lordered-ring*) *spmat* ⇒ *'a spmat*

primrec
abs-spmat [] = []
abs-spmat (*a*#*as*) = (*fst a*, *abs-spvec* (*snd a*))#(*abs-spmat as*)

primrec
minus-spmat [] = []
minus-spmat (*a*#*as*) = (*fst a*, *minus-spvec* (*snd a*))#(*minus-spmat as*)

lemma *sparse-row-matrix-minus*:
sparse-row-matrix (*minus-spmat A*) = − (*sparse-row-matrix A*)
⟨*proof*⟩

lemma *Rep-sparse-row-vector-zero*: $x \neq 0 \Longrightarrow$ *Rep-matrix* (*sparse-row-vector v*) *x y* = *0*
⟨*proof*⟩

lemma *sparse-row-matrix-abs*:
sorted-spvec A $\Longrightarrow$ *sorted-spmat A* $\Longrightarrow$ *sparse-row-matrix* (*abs-spmat A*) = *abs* (*sparse-row-matrix A*)
⟨*proof*⟩

lemma *sorted-spvec-minus-spmat*: *sorted-spvec A* $\Longrightarrow$ *sorted-spvec* (*minus-spmat A*)
⟨*proof*⟩

lemma *sorted-spvec-abs-spmat*: *sorted-spvec A* $\Longrightarrow$ *sorted-spvec* (*abs-spmat A*)
⟨*proof*⟩

lemma *sorted-spmat-minus-spmat*: *sorted-spmat A* $\Longrightarrow$ *sorted-spmat* (*minus-spmat A*)
⟨*proof*⟩

lemma *sorted-spmat-abs-spmat*: *sorted-spmat A* $\Longrightarrow$ *sorted-spmat* (*abs-spmat A*)
⟨*proof*⟩

constdefs
diff-spmat :: (*'a::lordered-ring*) *spmat* $\Rightarrow$ *'a spmat* $\Rightarrow$ *'a spmat*
diff-spmat A B == *add-spmat* (*A*, *minus-spmat B*)

lemma *sorted-spmat-diff-spmat*: *sorted-spmat A* $\Longrightarrow$ *sorted-spmat B* $\Longrightarrow$ *sorted-spmat* (*diff-spmat A B*)
⟨*proof*⟩

lemma *sorted-spvec-diff-spmat*: *sorted-spvec A* $\Longrightarrow$ *sorted-spvec B* $\Longrightarrow$ *sorted-spvec* (*diff-spmat A B*)
⟨*proof*⟩

lemma *sparse-row-diff-spmat*: *sparse-row-matrix* (*diff-spmat A B*) = (*sparse-row-matrix A*) − (*sparse-row-matrix B*)
⟨*proof*⟩

constdefs
sorted-sparse-matrix :: *'a spmat* $\Rightarrow$ *bool*
sorted-sparse-matrix A == (*sorted-spvec A*) & (*sorted-spmat A*)

lemma *sorted-sparse-matrix-imp-spvec*: *sorted-sparse-matrix A* $\Longrightarrow$ *sorted-spvec A*
⟨*proof*⟩

lemma *sorted-sparse-matrix-imp-spmat*: *sorted-sparse-matrix A* $\Longrightarrow$ *sorted-spmat A*
⟨*proof*⟩

lemmas *sorted-sp-simps* =
sorted-spvec.simps
sorted-spmat.simps
sorted-sparse-matrix-def

lemma *bool1*: (¬ *True*) = *False* ⟨*proof*⟩
lemma *bool2*: (¬ *False*) = *True* ⟨*proof*⟩
lemma *bool3*: ((*P*::*bool*) ∧ *True*) = *P* ⟨*proof*⟩
lemma *bool4*: (*True* ∧ (*P*::*bool*)) = *P* ⟨*proof*⟩
lemma *bool5*: ((*P*::*bool*) ∧ *False*) = *False* ⟨*proof*⟩
lemma *bool6*: (*False* ∧ (*P*::*bool*)) = *False* ⟨*proof*⟩

lemma *bool7*: $((P::bool) \vee True) = True$ $\langle proof \rangle$
lemma *bool8*: $(True \vee (P::bool)) = True$ $\langle proof \rangle$
lemma *bool9*: $((P::bool) \vee False) = P$ $\langle proof \rangle$
lemma *bool10*: $(False \vee (P::bool)) = P$ $\langle proof \rangle$
lemmas *boolarith* = *bool1 bool2 bool3 bool4 bool5 bool6 bool7 bool8 bool9 bool10*

lemma *if-case-eq*: (*if b then x else y*) = (*case b of True => x | False => y*)
$\langle proof \rangle$

consts
  pprt-spvec :: ('*a*::{*lordered-ab-group-add*}) *spvec* ⇒ '*a spvec*
  nprt-spvec :: ('*a*::{*lordered-ab-group-add*}) *spvec* ⇒ '*a spvec*
  pprt-spmat :: ('*a*::{*lordered-ab-group-add*}) *spmat* ⇒ '*a spmat*
  nprt-spmat :: ('*a*::{*lordered-ab-group-add*}) *spmat* ⇒ '*a spmat*

primrec
  pprt-spvec [] = []
  pprt-spvec (*a*#*as*) = (*fst a*, *pprt* (*snd a*)) # (*pprt-spvec as*)

primrec
  nprt-spvec [] = []
  nprt-spvec (*a*#*as*) = (*fst a*, *nprt* (*snd a*)) # (*nprt-spvec as*)

primrec
  pprt-spmat [] = []
  pprt-spmat (*a*#*as*) = (*fst a*, *pprt-spvec* (*snd a*))#(*pprt-spmat as*)

primrec
  nprt-spmat [] = []
  nprt-spmat (*a*#*as*) = (*fst a*, *nprt-spvec* (*snd a*))#(*nprt-spmat as*)

lemma *pprt-add*: *disj-matrices A* (*B*::(-::*lordered-ring*) *matrix*) ⟹ *pprt* (*A*+*B*)
= *pprt A* + *pprt B*
  $\langle proof \rangle$

lemma *nprt-add*: *disj-matrices A* (*B*::(-::*lordered-ring*) *matrix*) ⟹ *nprt* (*A*+*B*)
= *nprt A* + *nprt B*
  $\langle proof \rangle$

lemma *pprt-singleton*[*simp*]: *pprt* (*singleton-matrix j i* (*x*::-::*lordered-ring*)) = *singleton-matrix j i* (*pprt x*)
  $\langle proof \rangle$

lemma *nprt-singleton*[*simp*]: *nprt* (*singleton-matrix j i* (*x*::-::*lordered-ring*)) = *singleton-matrix j i* (*nprt x*)
  $\langle proof \rangle$

lemma *less-imp-le*: $a < b \Longrightarrow a <= (b::-::order)$ $\langle proof \rangle$

lemma *sparse-row-vector-pprt*: *sorted-spvec* $v \Longrightarrow$ *sparse-row-vector* (*pprt-spvec* v) = *pprt* (*sparse-row-vector* v)
$\langle proof \rangle$

lemma *sparse-row-vector-nprt*: *sorted-spvec* $v \Longrightarrow$ *sparse-row-vector* (*nprt-spvec* v) = *nprt* (*sparse-row-vector* v)
$\langle proof \rangle$

lemma *pprt-move-matrix*: *pprt* (*move-matrix* (A::($'a$::*lordered-ring*) *matrix*) j i) = *move-matrix* (*pprt* A) j i
$\langle proof \rangle$

lemma *nprt-move-matrix*: *nprt* (*move-matrix* (A::($'a$::*lordered-ring*) *matrix*) j i) = *move-matrix* (*nprt* A) j i
$\langle proof \rangle$

lemma *sparse-row-matrix-pprt*: *sorted-spvec* $m \Longrightarrow$ *sorted-spmat* $m \Longrightarrow$ *sparse-row-matrix* (*pprt-spmat* m) = *pprt* (*sparse-row-matrix* m)
$\langle proof \rangle$

lemma *sparse-row-matrix-nprt*: *sorted-spvec* $m \Longrightarrow$ *sorted-spmat* $m \Longrightarrow$ *sparse-row-matrix* (*nprt-spmat* m) = *nprt* (*sparse-row-matrix* m)
$\langle proof \rangle$

lemma *sorted-pprt-spvec*: *sorted-spvec* $v \Longrightarrow$ *sorted-spvec* (*pprt-spvec* v)
$\langle proof \rangle$

lemma *sorted-nprt-spvec*: *sorted-spvec* $v \Longrightarrow$ *sorted-spvec* (*nprt-spvec* v)
$\langle proof \rangle$

lemma *sorted-spvec-pprt-spmat*: *sorted-spvec* $m \Longrightarrow$ *sorted-spvec* (*pprt-spmat* m)
$\langle proof \rangle$

lemma *sorted-spvec-nprt-spmat*: *sorted-spvec* $m \Longrightarrow$ *sorted-spvec* (*nprt-spmat* m)
$\langle proof \rangle$

lemma *sorted-spmat-pprt-spmat*: *sorted-spmat* $m \Longrightarrow$ *sorted-spmat* (*pprt-spmat* m)
$\langle proof \rangle$

lemma *sorted-spmat-nprt-spmat*: *sorted-spmat* $m \Longrightarrow$ *sorted-spmat* (*nprt-spmat* m)
$\langle proof \rangle$

constdefs

mult-est-spmat :: (*'a*::*lordered-ring*) *spmat* ⇒ *'a spmat* ⇒ *'a spmat* ⇒ *'a spmat* ⇒ *'a spmat*
mult-est-spmat r1 r2 s1 s2 ==
add-spmat (*mult-spmat* (*pprt-spmat s2*) (*pprt-spmat r2*), *add-spmat* (*mult-spmat* (*pprt-spmat s1*) (*nprt-spmat r2*),
add-spmat (*mult-spmat* (*nprt-spmat s2*) (*pprt-spmat r1*), *mult-spmat* (*nprt-spmat s1*) (*nprt-spmat r1*))))

lemmas *sparse-row-matrix-op-simps* =
sorted-sparse-matrix-imp-spmat sorted-sparse-matrix-imp-spvec
sparse-row-add-spmat sorted-spvec-add-spmat sorted-spmat-add-spmat
sparse-row-diff-spmat sorted-spvec-diff-spmat sorted-spmat-diff-spmat
sparse-row-matrix-minus sorted-spvec-minus-spmat sorted-spmat-minus-spmat
sparse-row-mult-spmat sorted-spvec-mult-spmat sorted-spmat-mult-spmat
sparse-row-matrix-abs sorted-spvec-abs-spmat sorted-spmat-abs-spmat
le-spmat-iff-sparse-row-le
sparse-row-matrix-pprt sorted-spvec-pprt-spmat sorted-spmat-pprt-spmat
sparse-row-matrix-nprt sorted-spvec-nprt-spmat sorted-spmat-nprt-spmat

lemma *zero-eq-Numeral0*: (*0*::-::*number-ring*) = *Numeral0* ⟨*proof*⟩

lemmas *sparse-row-matrix-arith-simps*[*simplified zero-eq-Numeral0*] =
mult-spmat.simps mult-spvec-spmat.simps
addmult-spvec.simps
smult-spvec-empty smult-spvec-cons
add-spmat.simps add-spvec.simps
minus-spmat.simps minus-spvec.simps
abs-spmat.simps abs-spvec.simps
diff-spmat-def
le-spmat.simps le-spvec.simps
pprt-spmat.simps pprt-spvec.simps
nprt-spmat.simps nprt-spvec.simps
mult-est-spmat-def

lemma *spm-mult-le-dual-prts*:
assumes
sorted-sparse-matrix A1
sorted-sparse-matrix A2
sorted-sparse-matrix c1
sorted-sparse-matrix c2
sorted-sparse-matrix y
sorted-sparse-matrix r1
sorted-sparse-matrix r2
sorted-spvec b
le-spmat ([], *y*)
sparse-row-matrix A1 ≤ *A*

$A \le$ *sparse-row-matrix A2*
sparse-row-matrix c1 $\le c$
$c \le$ *sparse-row-matrix c2*
sparse-row-matrix r1 $\le x$
$x \le$ *sparse-row-matrix r2*
$A * x \le$ *sparse-row-matrix* (*b*::(*'a*::*lordered-ring*) *spmat*)
shows
$c * x \le$ *sparse-row-matrix* (*add-spmat* (*mult-spmat y b*,
(*let s1* = *diff-spmat c1* (*mult-spmat y A2*); *s2* = *diff-spmat c2* (*mult-spmat y A1*) *in*
add-spmat (*mult-spmat* (*pprt-spmat s2*) (*pprt-spmat r2*), *add-spmat* (*mult-spmat* (*pprt-spmat s1*) (*nprt-spmat r2*),
add-spmat (*mult-spmat* (*nprt-spmat s2*) (*pprt-spmat r1*), *mult-spmat* (*nprt-spmat s1*) (*nprt-spmat r1*)))))))
⟨*proof*⟩

lemma *spm-mult-le-dual-prts-no-let*:
assumes
sorted-sparse-matrix A1
sorted-sparse-matrix A2
sorted-sparse-matrix c1
sorted-sparse-matrix c2
sorted-sparse-matrix y
sorted-sparse-matrix r1
sorted-sparse-matrix r2
sorted-spvec b
le-spmat ([], *y*)
sparse-row-matrix A1 $\le A$
$A \le$ *sparse-row-matrix A2*
sparse-row-matrix c1 $\le c$
$c \le$ *sparse-row-matrix c2*
sparse-row-matrix r1 $\le x$
$x \le$ *sparse-row-matrix r2*
$A * x \le$ *sparse-row-matrix* (*b*::(*'a*::*lordered-ring*) *spmat*)
shows
$c * x \le$ *sparse-row-matrix* (*add-spmat* (*mult-spmat y b*,
mult-est-spmat r1 r2 (*diff-spmat c1* (*mult-spmat y A2*)) (*diff-spmat c2* (*mult-spmat y A1*))))
⟨*proof*⟩

end

theory *FloatSparseMatrix* **imports** *Float SparseMatrix* **begin**

end

theory *Compute-Oracle* **imports** *Pure*
uses *am.ML am-compiler.ML am-interpreter.ML am-ghc.ML am-sml.ML report.ML compute.ML linker.ML*
begin

$\langle ML \rangle$

end
theory *ComputeHOL*
imports *Main* ~~/*src*/*Tools*/*Compute-Oracle*/*Compute-Oracle*
begin

lemma *Trueprop-eq-eq*: $Trueprop\ X == (X == True)$ $\langle proof \rangle$
lemma *meta-eq-trivial*: $x == y \Longrightarrow x == y$ $\langle proof \rangle$
lemma *meta-eq-imp-eq*: $x == y \Longrightarrow x = y$ $\langle proof \rangle$
lemma *eq-trivial*: $x = y \Longrightarrow x = y$ $\langle proof \rangle$
lemma *bool-to-true*: $x :: bool \Longrightarrow x == True$ $\langle proof \rangle$
lemma *transmeta-1*: $x = y \Longrightarrow y == z \Longrightarrow x = z$ $\langle proof \rangle$
lemma *transmeta-2*: $x == y \Longrightarrow y = z \Longrightarrow x = z$ $\langle proof \rangle$
lemma *transmeta-3*: $x == y \Longrightarrow y == z \Longrightarrow x = z$ $\langle proof \rangle$

lemma *If-True*: $If\ True = (\lambda\ x\ y.\ x)$ $\langle proof \rangle$
lemma *If-False*: $If\ False = (\lambda\ x\ y.\ y)$ $\langle proof \rangle$

lemmas *compute-if* = *If-True If-False*

lemma *bool1*: $(\neg\ True) = False$ $\langle proof \rangle$
lemma *bool2*: $(\neg\ False) = True$ $\langle proof \rangle$
lemma *bool3*: $(P \wedge True) = P$ $\langle proof \rangle$
lemma *bool4*: $(True \wedge P) = P$ $\langle proof \rangle$
lemma *bool5*: $(P \wedge False) = False$ $\langle proof \rangle$
lemma *bool6*: $(False \wedge P) = False$ $\langle proof \rangle$
lemma *bool7*: $(P \vee True) = True$ $\langle proof \rangle$
lemma *bool8*: $(True \vee P) = True$ $\langle proof \rangle$
lemma *bool9*: $(P \vee False) = P$ $\langle proof \rangle$
lemma *bool10*: $(False \vee P) = P$ $\langle proof \rangle$
lemma *bool11*: $(True \longrightarrow P) = P$ $\langle proof \rangle$
lemma *bool12*: $(P \longrightarrow True) = True$ $\langle proof \rangle$
lemma *bool13*: $(True \longrightarrow P) = P$ $\langle proof \rangle$
lemma *bool14*: $(P \longrightarrow False) = (\neg\ P)$ $\langle proof \rangle$
lemma *bool15*: $(False \longrightarrow P) = True$ $\langle proof \rangle$
lemma *bool16*: $(False = False) = True$ $\langle proof \rangle$
lemma *bool17*: $(True = True) = True$ $\langle proof \rangle$

lemma *bool18*: (*False* = *True*) = *False* ⟨*proof*⟩
lemma *bool19*: (*True* = *False*) = *False* ⟨*proof*⟩

lemmas *compute-bool* = *bool1 bool2 bool3 bool4 bool5 bool6 bool7 bool8 bool9 bool10 bool11 bool12 bool13 bool14 bool15 bool16 bool17 bool18 bool19*

lemma *compute-fst*: *fst* (*x*,*y*) = *x* ⟨*proof*⟩
lemma *compute-snd*: *snd* (*x*,*y*) = *y* ⟨*proof*⟩
lemma *compute-pair-eq*: ((*a*, *b*) = (*c*, *d*)) = (*a* = *c* ∧ *b* = *d*) ⟨*proof*⟩

lemma *prod-case-simp*: *prod-case f* (*x*,*y*) = *f x y* ⟨*proof*⟩

lemmas *compute-pair* = *compute-fst compute-snd compute-pair-eq prod-case-simp*

lemma *compute-the*: *the* (*Some x*) = *x* ⟨*proof*⟩
lemma *compute-None-Some-eq*: (*None* = *Some x*) = *False* ⟨*proof*⟩
lemma *compute-Some-None-eq*: (*Some x* = *None*) = *False* ⟨*proof*⟩
lemma *compute-None-None-eq*: (*None* = *None*) = *True* ⟨*proof*⟩
lemma *compute-Some-Some-eq*: (*Some x* = *Some y*) = (*x* = *y*) ⟨*proof*⟩

definition
option-case-compute :: *'b option* ⇒ *'a* ⇒ (*'b* ⇒ *'a*) ⇒ *'a*
where
option-case-compute opt a f = *option-case a f opt*

lemma *option-case-compute*: *option-case* = (λ *a f opt*. *option-case-compute opt a f*)
⟨*proof*⟩

lemma *option-case-compute-None*: *option-case-compute None* = (λ *a f*. *a*)
⟨*proof*⟩

lemma *option-case-compute-Some*: *option-case-compute* (*Some x*) = (λ *a f*. *f x*)
⟨*proof*⟩

lemmas *compute-option-case* = *option-case-compute option-case-compute-None option-case-compute-Some*

lemmas *compute-option* = *compute-the compute-None-Some-eq compute-Some-None-eq compute-None-None-eq compute-Some-Some-eq compute-option-case*

lemma *length-cons*:*length* (*x#xs*) = *1* + (*length xs*)
⟨*proof*⟩

lemma *length-nil*: *length* [] = *0*
⟨*proof*⟩

lemmas *compute-list-length* = *length-nil length-cons*

definition
list-case-compute :: ′*b list* ⇒ ′*a* ⇒ (′*b* ⇒ ′*b list* ⇒ ′*a*) ⇒ ′*a*
where
list-case-compute l a f = *list-case a f l*

lemma *list-case-compute*: *list-case* = (λ (*a*::′*a*) *f* (*l*::′*b list*). *list-case-compute l a f*)
⟨*proof*⟩

lemma *list-case-compute-empty*: *list-case-compute* ([]::′*b list*) = (λ (*a*::′*a*) *f*. *a*)
⟨*proof*⟩

lemma *list-case-compute-cons*: *list-case-compute* (*u*#*v*) = (λ (*a*::′*a*) *f*. (*f* (*u*::′*b*) *v*))
⟨*proof*⟩

lemmas *compute-list-case* = *list-case-compute list-case-compute-empty list-case-compute-cons*

lemma *compute-list-nth*: ((*x*#*xs*) ! *n*) = (*if n* = *0 then x else* (*xs* ! (*n* − *1*)))
⟨*proof*⟩

lemmas *compute-list* = *compute-list-case compute-list-length compute-list-nth*

lemmas *compute-let* = *Let-def*

lemmas *compute-hol* = *compute-if compute-bool compute-pair compute-option compute-list compute-let*

⟨*ML*⟩

end

theory *ComputeNumeral*
imports *ComputeHOL* ~~/*src*/*HOL*/*Real*/*Float*
begin

lemmas *bitnorm* = *normalize-bin-simps*

lemma *neg1*: *neg Int.Pls* = *False* ⟨*proof*⟩
lemma *neg2*: *neg Int.Min* = *True* ⟨*proof*⟩
lemma *neg3*: *neg* (*Int.Bit0 x*) = *neg x* ⟨*proof*⟩
lemma *neg4*: *neg* (*Int.Bit1 x*) = *neg x* ⟨*proof*⟩
lemmas *bitneg* = *neg1 neg2 neg3 neg4*

lemma *iszero1*: *iszero Int.Pls* = *True* ⟨*proof*⟩
lemma *iszero2*: *iszero Int.Min* = *False* ⟨*proof*⟩
lemma *iszero3*: *iszero* (*Int.Bit0 x*) = *iszero x* ⟨*proof*⟩
lemma *iszero4*: *iszero* (*Int.Bit1 x*) = *False* ⟨*proof*⟩
lemmas *bitiszero* = *iszero1 iszero2 iszero3 iszero4*

constdefs
  lezero x == ($x \leq 0$)
lemma *lezero1*: *lezero Int.Pls* = *True* ⟨*proof*⟩
lemma *lezero2*: *lezero Int.Min* = *True* ⟨*proof*⟩
lemma *lezero3*: *lezero* (*Int.Bit0 x*) = *lezero x* ⟨*proof*⟩
lemma *lezero4*: *lezero* (*Int.Bit1 x*) = *neg x* ⟨*proof*⟩
lemmas *bitlezero* = *lezero1 lezero2 lezero3 lezero4*

lemma *biteq1*: (*Int.Pls* = *Int.Pls*) = *True* ⟨*proof*⟩
lemma *biteq2*: (*Int.Min* = *Int.Min*) = *True* ⟨*proof*⟩
lemma *biteq3*: (*Int.Pls* = *Int.Min*) = *False* ⟨*proof*⟩
lemma *biteq4*: (*Int.Min* = *Int.Pls*) = *False* ⟨*proof*⟩
lemma *biteq5*: (*Int.Bit0 x* = *Int.Bit0 y*) = (*x* = *y*) ⟨*proof*⟩
lemma *biteq6*: (*Int.Bit1 x* = *Int.Bit1 y*) = (*x* = *y*) ⟨*proof*⟩
lemma *biteq7*: (*Int.Bit0 x* = *Int.Bit1 y*) = *False* ⟨*proof*⟩
lemma *biteq8*: (*Int.Bit1 x* = *Int.Bit0 y*) = *False* ⟨*proof*⟩
lemma *biteq9*: (*Int.Pls* = *Int.Bit0 x*) = (*Int.Pls* = *x*) ⟨*proof*⟩
lemma *biteq10*: (*Int.Pls* = *Int.Bit1 x*) = *False* ⟨*proof*⟩
lemma *biteq11*: (*Int.Min* = *Int.Bit0 x*) = *False* ⟨*proof*⟩
lemma *biteq12*: (*Int.Min* = *Int.Bit1 x*) = (*Int.Min* = *x*) ⟨*proof*⟩
lemma *biteq13*: (*Int.Bit0 x* = *Int.Pls*) = (*x* = *Int.Pls*) ⟨*proof*⟩
lemma *biteq14*: (*Int.Bit1 x* = *Int.Pls*) = *False* ⟨*proof*⟩
lemma *biteq15*: (*Int.Bit0 x* = *Int.Min*) = *False* ⟨*proof*⟩
lemma *biteq16*: (*Int.Bit1 x* = *Int.Min*) = (*x* = *Int.Min*) ⟨*proof*⟩

lemmas *biteq* = *biteq1 biteq2 biteq3 biteq4 biteq5 biteq6 biteq7 biteq8 biteq9 biteq10 biteq11 biteq12 biteq13 biteq14 biteq15 biteq16*

lemma *bitless1*: $(Int.Pls < Int.Min) = False$ ⟨*proof*⟩
lemma *bitless2*: $(Int.Pls < Int.Pls) = False$ ⟨*proof*⟩
lemma *bitless3*: $(Int.Min < Int.Pls) = True$ ⟨*proof*⟩
lemma *bitless4*: $(Int.Min < Int.Min) = False$ ⟨*proof*⟩
lemma *bitless5*: $(Int.Bit0\ x < Int.Bit0\ y) = (x < y)$ ⟨*proof*⟩
lemma *bitless6*: $(Int.Bit1\ x < Int.Bit1\ y) = (x < y)$ ⟨*proof*⟩
lemma *bitless7*: $(Int.Bit0\ x < Int.Bit1\ y) = (x \leq y)$ ⟨*proof*⟩
lemma *bitless8*: $(Int.Bit1\ x < Int.Bit0\ y) = (x < y)$ ⟨*proof*⟩
lemma *bitless9*: $(Int.Pls < Int.Bit0\ x) = (Int.Pls < x)$ ⟨*proof*⟩
lemma *bitless10*: $(Int.Pls < Int.Bit1\ x) = (Int.Pls \leq x)$ ⟨*proof*⟩
lemma *bitless11*: $(Int.Min < Int.Bit0\ x) = (Int.Pls \leq x)$ ⟨*proof*⟩
lemma *bitless12*: $(Int.Min < Int.Bit1\ x) = (Int.Min < x)$ ⟨*proof*⟩
lemma *bitless13*: $(Int.Bit0\ x < Int.Pls) = (x < Int.Pls)$ ⟨*proof*⟩
lemma *bitless14*: $(Int.Bit1\ x < Int.Pls) = (x < Int.Pls)$ ⟨*proof*⟩
lemma *bitless15*: $(Int.Bit0\ x < Int.Min) = (x < Int.Pls)$ ⟨*proof*⟩
lemma *bitless16*: $(Int.Bit1\ x < Int.Min) = (x < Int.Min)$ ⟨*proof*⟩
lemmas *bitless* = *bitless1 bitless2 bitless3 bitless4 bitless5 bitless6 bitless7 bitless8 bitless9 bitless10 bitless11 bitless12 bitless13 bitless14 bitless15 bitless16*

lemma *bitle1*: $(Int.Pls \leq Int.Min) = False$ ⟨*proof*⟩
lemma *bitle2*: $(Int.Pls \leq Int.Pls) = True$ ⟨*proof*⟩
lemma *bitle3*: $(Int.Min \leq Int.Pls) = True$ ⟨*proof*⟩
lemma *bitle4*: $(Int.Min \leq Int.Min) = True$ ⟨*proof*⟩
lemma *bitle5*: $(Int.Bit0\ x \leq Int.Bit0\ y) = (x \leq y)$ ⟨*proof*⟩
lemma *bitle6*: $(Int.Bit1\ x \leq Int.Bit1\ y) = (x \leq y)$ ⟨*proof*⟩
lemma *bitle7*: $(Int.Bit0\ x \leq Int.Bit1\ y) = (x \leq y)$ ⟨*proof*⟩
lemma *bitle8*: $(Int.Bit1\ x \leq Int.Bit0\ y) = (x < y)$ ⟨*proof*⟩
lemma *bitle9*: $(Int.Pls \leq Int.Bit0\ x) = (Int.Pls \leq x)$ ⟨*proof*⟩
lemma *bitle10*: $(Int.Pls \leq Int.Bit1\ x) = (Int.Pls \leq x)$ ⟨*proof*⟩
lemma *bitle11*: $(Int.Min \leq Int.Bit0\ x) = (Int.Pls \leq x)$ ⟨*proof*⟩
lemma *bitle12*: $(Int.Min \leq Int.Bit1\ x) = (Int.Min \leq x)$ ⟨*proof*⟩
lemma *bitle13*: $(Int.Bit0\ x \leq Int.Pls) = (x \leq Int.Pls)$ ⟨*proof*⟩
lemma *bitle14*: $(Int.Bit1\ x \leq Int.Pls) = (x < Int.Pls)$ ⟨*proof*⟩
lemma *bitle15*: $(Int.Bit0\ x \leq Int.Min) = (x < Int.Pls)$ ⟨*proof*⟩
lemma *bitle16*: $(Int.Bit1\ x \leq Int.Min) = (x \leq Int.Min)$ ⟨*proof*⟩
lemmas *bitle* = *bitle1 bitle2 bitle3 bitle4 bitle5 bitle6 bitle7 bitle8 bitle9 bitle10 bitle11 bitle12 bitle13 bitle14 bitle15 bitle16*

lemmas *bitsucc* = *succ-bin-simps*

lemmas *bitpred* = *pred-bin-simps*

lemmas *bituminus* = *minus-bin-simps*

lemmas *bitadd* = *add-bin-simps*

lemma *mult-Pls-right*: $x * Int.Pls = Int.Pls$ ⟨*proof*⟩
lemma *mult-Min-right*: $x * Int.Min = - x$ ⟨*proof*⟩
lemma *multb0x*: $(Int.Bit0\ x) * y = Int.Bit0\ (x * y)$ ⟨*proof*⟩
lemma *multxb0*: $x * (Int.Bit0\ y) = Int.Bit0\ (x * y)$ ⟨*proof*⟩
lemma *multb1*: $(Int.Bit1\ x) * (Int.Bit1\ y) = Int.Bit1\ (Int.Bit0\ (x * y) + x + y)$
⟨*proof*⟩
lemmas *bitmul* = *mult-Pls mult-Min mult-Pls-right mult-Min-right multb0x multxb0 multb1*

lemmas *bitarith* = *bitnorm bitiszero bitneg bitlezero biteq bitless bitle bitsucc bitpred bituminus bitadd bitmul*

constdefs
nat-norm-number-of (*x*::*nat*) == *x*

lemma *nat-norm-number-of*: *nat-norm-number-of* (*number-of w*) = (*if lezero w then 0 else number-of w*)
⟨*proof*⟩

lemma *natnorm0*: (*0*::*nat*) = *number-of* (*Int.Pls*) ⟨*proof*⟩
lemma *natnorm1*: (*1* :: *nat*) = *number-of* (*Int.Bit1 Int.Pls*) ⟨*proof*⟩
lemmas *natnorm* = *natnorm0 natnorm1 nat-norm-number-of*

lemma *natsuc*: *Suc* (*number-of x*) = (*if neg x then 1 else number-of* (*Int.succ x*))
⟨*proof*⟩

lemma *natadd*: *number-of x* + ((*number-of y*)::*nat*) = (*if neg x then* (*number-of y*) *else* (*if neg y then number-of x else* (*number-of* (*x* + *y*))))
⟨*proof*⟩

lemma *natsub*: (*number-of x*) − ((*number-of y*)::*nat*) =
(*if neg x then 0 else* (*if neg y then number-of x else* (*nat-norm-number-of* (*number-of* (*x* + (− *y*))))))
⟨*proof*⟩

lemma *natmul*: (*number-of x*) ∗ ((*number-of y*)::*nat*) =
(*if neg x then 0 else* (*if neg y then 0 else number-of* (*x* ∗ *y*)))

⟨*proof*⟩

lemma *nateq*: $(((number\text{-}of\ x)::nat) = (number\text{-}of\ y)) = ((lezero\ x \wedge lezero\ y) \vee (x = y))$
⟨*proof*⟩

lemma *natless*: $(((number\text{-}of\ x)::nat) < (number\text{-}of\ y)) = ((x < y) \wedge (\neg\ (lezero\ y)))$
⟨*proof*⟩

lemma *natle*: $(((number\text{-}of\ x)::nat) \leq (number\text{-}of\ y)) = (y < x \longrightarrow lezero\ x)$
⟨*proof*⟩

fun *natfac* :: *nat* ⇒ *nat*
where
$natfac\ n = (if\ n = 0\ then\ 1\ else\ n * (natfac\ (n - 1)))$

lemmas *compute-natarith* = *bitarith natnorm natsuc natadd natsub natmul nateq natless natle natfac.simps*

lemma *number-eq*: $(((number\text{-}of\ x)::'a::\{number\text{-}ring,\ ordered\text{-}idom\}) = (number\text{-}of\ y)) = (x = y)$
⟨*proof*⟩

lemma *number-le*: $(((number\text{-}of\ x)::'a::\{number\text{-}ring,\ ordered\text{-}idom\}) \leq (number\text{-}of\ y)) = (x \leq y)$
⟨*proof*⟩

lemma *number-less*: $(((number\text{-}of\ x)::'a::\{number\text{-}ring,\ ordered\text{-}idom\}) < (number\text{-}of\ y)) = (x < y)$
⟨*proof*⟩

lemma *number-diff*: $((number\text{-}of\ x)::'a::\{number\text{-}ring,\ ordered\text{-}idom\}) - number\text{-}of\ y = number\text{-}of\ (x + (-\ y))$
⟨*proof*⟩

lemmas *number-norm* = *number-of-Pls*[*symmetric*] *numeral-1-eq-1*[*symmetric*]

lemmas *compute-numberarith* = *number-of-minus*[*symmetric*] *number-of-add*[*symmetric*] *number-diff number-of-mult*[*symmetric*] *number-norm number-eq number-le number-less*

lemma *compute-real-of-nat-number-of*: $real\ ((number\text{-}of\ v)::nat) = (if\ neg\ v\ then\ 0\ else\ number\text{-}of\ v)$
⟨*proof*⟩

lemma *compute-nat-of-int-number-of*: $nat\ ((number\text{-}of\ v)::int) = (number\text{-}of\ v)$
⟨*proof*⟩

lemmas *compute-num-conversions* = *compute-real-of-nat-number-of compute-nat-of-int-number-of* *real-number-of*

lemmas *zpowerarith* = *zpower-number-of-even*
zpower-number-of-odd[*simplified zero-eq-Numeral0-nring one-eq-Numeral1-nring*]
zpower-Pls zpower-Min

lemma *adjust*: *adjust b* (*q*, *r*) = (*if 0* ≤ *r* − *b then* (*2* ∗ *q* + *1*, *r* − *b*) *else* (*2* ∗ *q*, *r*))
⟨*proof*⟩

lemma *negateSnd*: *negateSnd* (*q*, *r*) = (*q*, −*r*)
⟨*proof*⟩

lemma *divAlg*: *divAlg* (*a*, *b*) = (*if 0*≤*a then*
if 0≤*b then posDivAlg a b*
else if a=*0 then* (*0*, *0*)
else negateSnd (*negDivAlg* (−*a*) (−*b*))
else
if 0<*b then negDivAlg a b*
else negateSnd (*posDivAlg* (−*a*) (−*b*)))
⟨*proof*⟩

lemmas *compute-div-mod* = *div-def mod-def divAlg adjust negateSnd posDivAlg.simps negDivAlg.simps*

lemma *even-Pls*: *even* (*Int.Pls*) = *True*
⟨*proof*⟩

lemma *even-Min*: *even* (*Int.Min*) = *False*
⟨*proof*⟩

lemma *even-B0*: *even* (*Int.Bit0 x*) = *True*
⟨*proof*⟩

lemma *even-B1*: *even* (*Int.Bit1 x*) = *False*
⟨*proof*⟩

lemma *even-number-of*: *even* ((*number-of w*)::*int*) = *even w*
⟨*proof*⟩

lemmas *compute-even* = *even-Pls even-Min even-B0 even-B1 even-number-of*

lemmas *compute-numeral* = *compute-if compute-let compute-pair compute-bool* *compute-natarith compute-numberarith max-def min-def compute-num-conversions zpowerarith compute-div-mod compute-even*

end

theory *Cplex*
imports *FloatSparseMatrix ~~/src/HOL/Tools/ComputeNumeral*
uses *Cplex-tools.ML CplexMatrixConverter.ML FloatSparseMatrixBuilder.ML fspmlp.ML*
begin

end

theory *MatrixLP*
imports *Cplex*
uses *matrixlp.ML*
begin
end

theory *Acc-tools* **imports** *Main*
begin

22 Definition of some results about the accesible part of a relation.

term *acc*
thm *accp.intros*

lemma *wf-imp-subset-accP*[*rule-format*]: $wf\ \{(y,x)\ .\ Q\ y \wedge Q\ x \wedge r\ y\ x\} \Longrightarrow (\forall\ y\ x.\ Q\ x \longrightarrow r\ y\ x \longrightarrow Q\ y) \Longrightarrow Q\ x \longrightarrow accp\ r\ x$
⟨*proof*⟩

lemma *subset-accP-imp-wf*:
assumes *Q-subset*: $\forall\ x.\ Q\ x \longrightarrow accp\ r\ x$
shows $wf\ (\{(a,b).\ Q\ b \wedge r\ a\ b\})$
⟨*proof*⟩

lemma *downchain-contra-imp-subset-accP*:
assumes *downchain*: $\bigwedge f.\ (\bigwedge i.\ Q\ (f\ i)) \Longrightarrow (\bigwedge i.\ r\ (f\ (Suc\ i))\ (f\ i)) \Longrightarrow False$

and *downclosed*: $\bigwedge$ *y x*. $[\![Q\ x;\ r\ y\ x]\!] \Longrightarrow Q\ y$
shows $Q\ x \Longrightarrow accp\ r\ x$
⟨*proof*⟩

lemma *accP-subset-induct*:
assumes *Q-subset*: $\forall\ x.\ Q\ x \longrightarrow accp\ r\ x$
assumes *Q-downward*: $\forall\ x\ y.\ Q\ x \longrightarrow r\ y\ x \longrightarrow Q\ y$
assumes *Q-a*: $Q\ a$
assumes *Q-induct*: $!!\ x.\ Q\ x \Longrightarrow \forall\ y.\ r\ y\ x \longrightarrow P\ y \Longrightarrow P\ x$
shows $P\ a$
⟨*proof*⟩

end
theory *Orbit*
imports *Main*
begin

23 Definition of orbits of functions and termination conditions.

lemma *funpow-1*: $(f::'a{\Rightarrow}'a)\ \hat{}(1::nat) = f$
⟨*proof*⟩

lemma *funpow-2*: $f\hat{}2 = f\ o\ f$
⟨*proof*⟩

lemma *funpow-zip*: $(f\hat{}n)\ (f\ x) = (f\hat{}(n+1))\ x$
⟨*proof*⟩

lemma *funpow-mult*: $(f::'a \Rightarrow\ 'a)\ \hat{}(m*n) = (f\hat{}m)\ \hat{}n$
⟨*proof*⟩

lemma *funpow-swap*: $(f\hat{}n)((f\hat{}m)\ x) = (f\hat{}m)((f\hat{}n)\ x)$
⟨*proof*⟩

lemma *nat-remainder-div*: $0 < (n::nat) \Longrightarrow \exists\ q\ r.\ r < n \wedge m = q * n + r$
⟨*proof*⟩

lemma *cyclic-fun-range*: **assumes** *n*: $0 < n$ **and** *cycle*: $(f\hat{}n)\ v = v$ **shows** $\exists\ r.$ $r < n \wedge (f\hat{}m)\ v = (f\hat{}r)\ v$
⟨*proof*⟩

23.1 Definition of the orbit of a function over a given point.

constdefs
$Orbit :: ('a \Rightarrow\ 'a) \Rightarrow\ 'a \Rightarrow\ 'a\ set$
$Orbit\ f\ d \equiv \{\ (f\hat{}n)\ d\ |\ n.\ True\}$

lemma *Orbit-refl*[*simp*]: $x \in Orbit\ f\ x$
⟨*proof*⟩

lemma *Orbit-next*[*dest*]: $y \in Orbit\ f\ (f\ x) \Longrightarrow y \in Orbit\ f\ x$
⟨*proof*⟩

lemma *Orbit-reduce*:
shows $Orbit\ f\ x = \{\ (f\hat{\ }m)(x) \mid m.\ \forall\ i \leq m.\ i > 0 \longrightarrow (f\hat{\ }i)\ x \neq x\ \}$ (**is** $?L = ?R$)
⟨*proof*⟩

lemma *Orbit-unfold1*: $Orbit\ f\ x = \{x\} \cup Orbit\ f\ (f\ x)$
⟨*proof*⟩

23.2 Definition of the section of a function over a given point.

constdefs
$Section\ continue\ (f::'a \Rightarrow 'a)\ x0 \equiv \{\ (f\hat{\ }n)\ x0 \mid n.\ (\forall\ m.\ m \leq n \longrightarrow continue\ ((f\hat{\ }m)(x0)))\ \}$

lemma *Section-x-x*: $\neg\ (continue\ x) \Longrightarrow Section\ continue\ f\ x = \{\}$
⟨*proof*⟩

lemma *Section-unfold*: $continue\ i \Longrightarrow Section\ continue\ f\ i = insert\ i\ (Section\ continue\ f\ (f\ i))$
⟨*proof*⟩

lemma *Section-rightopen*[*dest*]: $y \in\ Section\ continue\ f\ x \Longrightarrow continue\ y$
⟨*proof*⟩

lemma *Section-is-Orbit*:
assumes *x0-elem-S*: $x0 \in Section\ continue\ f\ (f\ x0)$ (**is** $x0 \in ?S$)
shows $?S = Orbit\ f\ x0$
⟨*proof*⟩

thm *Section-is-Orbit*

thm *Section-def*

term $continue\ y = (\forall\ m.\ y = (f\hat{\ }m)\ x \longrightarrow (\forall\ n.\ n > 0 \wedge n \leq m \longrightarrow (f\hat{\ }m)\ x \neq x))$

lemma *Section-is-Orbit'*: $insert\ x\ (Section\ (\lambda y.\ y \neq x)\ f\ (f\ x)) = Orbit\ f\ x$
⟨*proof*⟩

23.3 Definition of a termination condition in terms of orbits.

fun $terminates :: ('a \Rightarrow bool) \times ('a \Rightarrow 'a) \times 'a \Rightarrow bool$
where

terminates (*continue*, *f*, *x*) = ($\exists$ *y*. (*y* $\in$ *Orbit f x*) $\wedge$ $\neg$ (*continue y*))

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

lemmas *terminates-simp* = *terminates.simps*

lemma *finite-Section*:
assumes *terminates*: *terminates* (*continue*, *f*, *x*)
shows *finite* (*Section continue f x*)
⟨*proof*⟩

lemma *terminates-imp-notin-Section*:
assumes *terminates*: *terminates* (*continue*, *f*, *x*)
shows *x* $\notin$ *Section continue f* (*f x*)
⟨*proof*⟩

lemma *orbit-stepback*: *i* $\neq$ *x* $\Longrightarrow$ (*x* $\in$ *Orbit f* (*f i*)) = (*x* $\in$ *Orbit f i*)
⟨*proof*⟩

lemma *terminates-rec*: *terminates* (*continue*, *f*, *x*) = (*if continue x then terminates* (*continue*, *f*, *f x*) *else True*)
⟨*proof*⟩

lemma *Suc-card-Section-eq*:
assumes *x0-neq-x1*: *continue x0*
and *terminates*: *terminates* (*continue*, *f*, *f x0*)
and *finite-A*: *finite A*
and *x0-elem-A*: *x0* $\in$ *A*
shows *Suc* (*card* (*Section continue f* (*f x0*) $\cap$ *A*)) = *card* (*Section continue f x0* $\cap$ *A*)
⟨*proof*⟩

end
theory *Cycle* **imports** *Main*
begin

constdefs
closed :: *'a set* $\Rightarrow$ (*'a* $\Rightarrow$ *'a*) $\Rightarrow$ *bool*
closed A f $\equiv$ $\forall$ *x* $\in$ *A*. *f x* $\in$ *A*

constdefs
cyclic :: *'a set* $\Rightarrow$ (*'a* $\Rightarrow$ *'a*) $\Rightarrow$ *bool*
cyclic A f $\equiv$ $\forall$ *d* $\in$ *A*. $\exists$ *n*. *n* > *0* $\wedge$ (*f^n*) *d* = *d*

constdefs
cyclic-equiv :: *'a set* $\Rightarrow$ (*'a* $\Rightarrow$ *'a*) $\Rightarrow$ (*'a* $\times$ *'a*) *set*
cyclic-equiv A f $\equiv$ { (*a*,*b*) . *a* $\in$ *A* $\wedge$ *b* $\in$ *A* $\wedge$ (*? n*. (*f^n*) *a* = *b*) }

lemma *refl-cyclic-equiv*: *refl A* (*cyclic-equiv A f*)

⟨*proof*⟩

lemma *archimedian-law*: $(m::nat) > 0 \Longrightarrow ?\ q.\ n < q{*}m$
⟨*proof*⟩

lemma *cyclic-wrap*:
assumes *c*: *cyclic A f*
assumes *x*: $x \in A$
shows $?\ n'.\ (f\hat{\ }n')\ ((f\hat{\ }n)\ x) = x$
⟨*proof*⟩

lemma *sym-cyclic-equiv*: *cyclic A f* $\Longrightarrow$ *sym* (*cyclic-equiv A f*)
⟨*proof*⟩

lemma *trans-cyclic-equiv*: *trans* (*cyclic-equiv A f*)
⟨*proof*⟩

lemma *cyclic A f* $\Longrightarrow$ *equiv A* (*cyclic-equiv A f*)
⟨*proof*⟩

constdefs
cyclic-on A f $\equiv$ *closed A f* $\wedge$ *cyclic A f*

constdefs
representing A f R $\equiv$ *closed A f* $\wedge$ *cyclic A f* $\wedge$ $R \subseteq A \wedge (\forall\ x\ y.\ (x \in R \wedge y \in R \wedge (\exists\ n.\ (f\hat{\ }n)\ (x) = y)) \longrightarrow x = y) \wedge A = \{\ (f\hat{\ }n)\ (x) \mid n\ x.\ x \in R\ \}$

end
theory *While* **imports** *Acc-tools Orbit Cycle*
begin

24 Definition of *while* loops as tail recursive functions.

function (*tailrec*) *While* :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$
where
While continue f s = (*if continue s then While continue f* (*f s*) *else s*)
⟨*proof*⟩

We delete the definition from the simplifier to avoid infinite loops.

declare *While.simps*[*simp del*]
lemmas *While-simp* = *While.simps*

An alternative definition for termination sets.

fun *terminates-slice* :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow ('a \Rightarrow bool) \times ('a \Rightarrow 'a) \times 'a \Rightarrow bool$
where

terminates-slice continue0 f0 (*continue*, *f*, *x*) = (*continue* = *continue0* ∧ *f* = *f0* ∧ *terminates* (*continue*, *f*, *x*))

lemma *lfp-const*: *lfp* (λ *p*. *P*) = *P*
⟨*proof*⟩

Definition of the relation condition of *While* loops.

lemmas *While-rel-def'* = *While-rel-def*[*simplified lfp-const*]

lemma *While-rel-continue*: *While-rel q* (*continue*, *f*, *s*) ⟹ *continue s*
⟨*proof*⟩

lemma assumes *n-g-0*: *0* < (*n*::*nat*) **shows** ∃ *m*. *n* = *Suc m*
⟨*proof*⟩

lemma *helper*: **shows** ⋀*n*::*nat*. *0* < *n* ==> ∃ *m*. *n* = *Suc m* ⟨*proof*⟩

lemma *terminates-downward*: *terminates x* ⟹ *While-rel y x* ⟹ *terminates y*
⟨*proof*⟩

lemma *terminates-slice-downward*: *terminates-slice continue f x* ⟹ *While-rel y x* ⟹ *terminates-slice continue f y*
⟨*proof*⟩

lemma *terminates-subset-dom*[*rule-format*]: *terminates* (*continue*, *f*, *s*) ⟶ *While-dom* (*continue*, *f*, *s*)
⟨*proof*⟩

lemma *terminates-slice-subset-dom*: *terminates-slice continue f x* ⟹ *While-dom x*
⟨*proof*⟩

Some additional induction rules for the previous *While* definition.

lemma *While-pinduct*:
assumes *terminates*: *terminates* (*continue*, *f*, *s*)
and *I*: !! *s*. ⟦ *terminates* (*continue*, *f*, *s*); *continue s* ⟹ *P* (*f s*)⟧ ⟹ *P s*
shows *P s*
⟨*proof*⟩

lemma *While-pinduct-weak*:
assumes *terminates*: *terminates* (*continue*, *f*, *s*)
and *I*: !! *s*. ⟦ *continue s* ⟹ *P* (*f s*)⟧ ⟹ *P s*
shows *P s*
⟨*proof*⟩

lemma *While-hoare-total*:
assumes *wf-R*: *wf R*
and *R-down*: !! *x*. *P x* ⟹ *continue x* ⟹ (*f x*, *x*) ∈ *R*

and *P-cont*: !! x. $P\ x \Longrightarrow$ *continue* $x \Longrightarrow P\ (f\ x)$
and *P-not-cont*: !! x. $P\ x \Longrightarrow \neg$ (*continue* x) $\Longrightarrow Q\ x$
and *P-start*: $P\ s$
shows Q (*While continue f s*)
⟨*proof*⟩

25 Definition of *For* loops.

constdefs
For′ :: (′$a \Rightarrow$ *bool*) $\Rightarrow$ (′$a \Rightarrow$ ′a) $\Rightarrow$ (′$a \Rightarrow$ ′$b \Rightarrow$ ′b) $\Rightarrow$ ′$a \Rightarrow$ ′$b \Rightarrow$ (′$b \times$ ′a)
For′ continue f Ac x ac $\equiv$ *While* (λ (*ac*, *x*). *continue x*) (λ (*ac*, *x*). (*Ac x ac*, *f x*)) (*ac*, *x*)
For continue f Ac x ac $\equiv$ *fst* (*For′ continue f Ac x ac*)

term *For*

lemma *For′-simp*: *For′ continue f Ac x ac* = (*if continue x then For′ continue f Ac* (*f x*) (*Ac x ac*) *else* (*ac*, *x*))
⟨*proof*⟩
thm *While-simp*
⟨*proof*⟩

lemma *For-simp*: *For continue f Ac x ac* = (*if continue x then For continue f Ac* (*f x*) (*Ac x ac*) *else ac*)
⟨*proof*⟩

lemma *split-power-of-for-state*: *? ac′*. (((λ(*ac*, *x*). ((*Ac* :: ′$a \Rightarrow$ ′$b \Rightarrow$ ′b) *x ac*, (*f* :: ′$a \Rightarrow$ ′a) *x*)) ^ *n*) (*a*, *b*) = (*ac′*, (*f*^*n*) *b*))
⟨*proof*⟩

lemma *swap-Ex*: ($\exists$ *a b*. *P a b*) = ($\exists$ *b a*. *P a b*) ⟨*proof*⟩

lemma *terminates-For*: *terminates* (λ(*ac*, *x*). *continue x*, λ(*ac*, *x*). (*Ac x ac*, *f x*), *s*) = *terminates* (*continue*, *f*, *snd s*)
⟨*proof*⟩

Additional induction rules for *For* loops.

lemma *For-pinduct*:
assumes *terminates*: *terminates* (*continue*, *f*, *i*)
and *I*: $\bigwedge$*i s*. ⟦*terminates* (*continue*, *f*, *i*); *continue i* $\Longrightarrow$ *P* (*f i*) (*Ac* (*i*::′*a*) (*s*::′*b*::*type*))⟧ $\Longrightarrow$ *P i s*
shows *P i s*
⟨*proof*⟩

lemma *For-pinduct-weak*:
assumes *terminates*: *terminates* (*continue*, *f*, *i*)
and *I*: $\bigwedge$*i s*. ⟦*continue i* $\Longrightarrow$ *P* (*f i*) (*Ac* (*i*::′*a*) (*s*::′*b*::*type*))⟧ $\Longrightarrow$ *P i s*
shows *P i s*
⟨*proof*⟩

constdefs
  find f x $\equiv$ *While* $(\lambda\ x.\ f\ x \neq x)\ f\ x$
  step1 f $\equiv$ $\{(y,x).\ y = f\ x \wedge y \neq x\}$

thm *find-def*[*symmetric*]
thm *While-simp*

lemma *find-simp*: *find f x* = (*if* $f\ x \neq x$ *then find f* $(f\ x)$ *else* x)
  ⟨*proof*⟩

lemma *find-wf-step1-terminates*: *wf* (*step1 f*) $\Longrightarrow$ *terminates* $(\lambda\ x.\ f\ x \neq x,\ f,\ i)$
  ⟨*proof*⟩

lemmas *find-pinduct* = *While-pinduct-weak*[*of* $(\lambda\ x.\ f\ x \neq x)\ f\ x$]

lemma *find*:
  assumes *terminates*: *terminates* $(\lambda\ x.\ f\ x \neq x,\ f,\ x)$
  shows f(*find f x*) = *find f x*
⟨*proof*⟩

constdefs
  section continue f x0 $\equiv$ *For continue f insert x0*
  card-section continue f x0 $\equiv$ *For continue f* $(\lambda\ x\ y.\ y + (1{::}nat))$ *x0*

lemmas *section-pinduct*=*For-pinduct*[**where** *Ac*=*insert*]
lemmas *section-pinduct-weak*=*For-pinduct-weak*[**where** *Ac*=*insert*]
lemmas *card-section-pinduct-weak*=*For-pinduct-weak*[**where** *Ac*=$\lambda\ x\ y.\ y + (1{::}nat)$]

lemma *section-simp*: *section continue f x A* = (*if continue x then section continue f* $(f\ x)$ (*insert x A*) *else A*)
  ⟨*proof*⟩

lemma *card-section-simp*:*card-section continue f x A* = (*if continue x then card-section continue f* $(f\ x)$ $(A + 1)$ *else A*)
  ⟨*proof*⟩

lemma *section-is-Section*:
  assumes *terminates*: *terminates* (*continue*, *f*, *x0*)
  shows *section continue f x0 A* = $A \cup$ (*Section continue f x0*)
⟨*proof*⟩

constdefs
  orbit f x $\equiv$ *section* $(\lambda\ y.\ y \neq x)\ f\ (f\ x)\ \{x\}$
  card-orbit f x $\equiv$ *card-section* $(\lambda\ y.\ y \neq x)\ f\ (f\ x)\ 1$

lemma *terminates-implies*: *terminates* (*cond*, *f*, *x*) $\Longrightarrow \exists\ n.\ \neg$ (*cond* $((f\hat{}n)\ x)$)

⟨*proof*⟩

lemma *orbit-is-Orbit*:
  assumes *terminates*: *terminates* $(\lambda\ y.\ y \neq x,\ f,\ f\ x)$
  shows *orbit* $f\ x = $ *Orbit* $f\ x$
  ⟨*proof*⟩

lemma *card-section-add*:
  assumes *terminates*: *terminates* (*continue*, f, *x0*)
  shows *card-section continue f x0* $(a + b) = a +$ (*card-section continue f x0* $(b::nat)$)
⟨*proof*⟩

lemma *card-section-suc*:
  assumes *terminates*: *terminates* (*continue*, f, *x0*)
  shows *card-section continue f x0* (*Suc a*) = *Suc* (*card-section continue f x0 a*)
  ⟨*proof*⟩

lemma *terminates-imp*: *terminates* (*continue*, f, i) $\Longrightarrow$ *continue* i $\Longrightarrow$ *terminates* (*continue*, f, $f\ i$)
  ⟨*proof*⟩

lemma *Suc-first*: *Suc* $(a + b) =$ *Suc* $a + b$ ⟨*proof*⟩

lemma *card-section-eq*[*rule-format*]:
  terminates (*continue*, f, *x0*) $\Longrightarrow$ *finite* $A \longrightarrow$ *card-section continue f x0* (*card A*) = *card* (*section continue f x0 A*) + *card* (*Section continue f x0* $\cap$ *A*)
  ⟨*proof*⟩

lemma
  assumes *terminates*: *terminates* (*continue*, f, x)
  shows *card-section continue f x 0* = *card* (*Section continue f x*)
  ⟨*proof*⟩

constdefs
  orbit-terminates $f\ x\ x' \equiv$ *terminates* $(\lambda\ y.\ y \neq x,\ f,\ f\ x')$

lemma *card-orbit-is-card-Orbit*:
  assumes *terminates*: *orbit-terminates f x x*
  shows *card-orbit* $f\ x =$ *card* (*Orbit* $f\ x$)
  ⟨*proof*⟩

lemma *orbit-terminates-rec*: *orbit-terminates* $f\ x\ x' =$ (*if* $f\ x' \neq x$ *then* *orbit-terminates* $f\ x\ (f\ x')$ *else True*)
  ⟨*proof*⟩

constdefs
  fold-section-def: *fold-section continue h f g z a* == *For continue h* (λ *z a*. *f* (*g* *z*) *a*) *z a*

lemma *finite-Orbit*:
  assumes *terminates*: *orbit-terminates f a a*
  shows *finite* (*Orbit f a*)
⟨*proof*⟩

lemma
  fold-section-simp:
    fold-section continue h f g x ac =
    (*if continue x*
     then fold-section continue h f g (*h x*) (*f* (*g x*) *ac*) *else ac*)
  ⟨*proof*⟩

The following locale is simply a rewriting, or an interpretation, of *ab-semigroup-mult*, and is only used to use *f* as a a binary operation instead of *op* ∗

locale *ACf* =
  fixes *f* :: *'a* => *'a* => *'a*   (**infixl** · *70*)
  assumes *commute*: $x \cdot y = y \cdot x$
    and *assoc*: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$
begin

lemma *left-commute*: $x \cdot (y \cdot z) = y \cdot (x \cdot z)$
⟨*proof*⟩

lemmas *AC* = *assoc commute left-commute*

end

interpretation *ACf* ⊆ *ab-semigroup-mult f* (**infixl** · *70*)
⟨*proof*⟩

lemma (**in** *ACf*) *fold-Section-eq*:
  assumes *terminates*: *terminates* (*continue*, *h*, *z*)
  shows *fold f g a* (*Section continue h z*) = *fold-section continue h f g z a*
⟨*proof*⟩

lemma (**in** *ACf*) *fold-Orbit-eq*:
  assumes *terminates*: *orbit-terminates h z z*
  shows *fold f g a* (*Orbit h z*) = *fold-section* (λ *y*. $y \neq z$) *h f g* (*h z*) (*f* (*g z*) *a*)
⟨*proof*⟩

lemma *While-postcondition*:
  assumes *terminates*: *terminates* (*continue*, *f*, *x*)
  shows ¬ (*continue* (*While continue f x*))
  ⟨*proof*⟩

end

theory *BPL-classes-2008*
  imports
  Basic-Perturbation-Lemma-local-nilpot
  While
begin

26 Additional type classes

In this section we introduce some additional type classes to those provided by the Isabelle standard distribution

For instance, we need a class *diff-group-add* that can be defined from the *ab-group-add* type class from the Isabelle library:

class *diff-group-add* = *ab-group-add* +
  fixes *diff* :: *'a* => *'a* (*d* - [*81*] *80*)
  assumes *diff-hom*: *d* (*x* + *y*) = (*d x*) + (*d y*)
  and *diff-nilpot*: *diff* ∘ *diff* = (λ*x*. *0*)

lemma (**in** *diff-group-add*) [*simp*]: *d* (*d x*) = *0*
  ⟨*proof*⟩

According to the previous syntax definitions, *diff-group-add-class.diff* is to be used with the parameter over which it is applied, and *diff-group-add-class.diff* remains to be used as a function

We can indeed prove instances of the specified type classes. An instance of a type class makes the type class sound.

instantiation *int* :: *diff-group-add*
begin

definition *diff-int-def*: *diff* ≡ (λ*x*. *0*::*int*)

instance
⟨*proof*⟩

end

A limitation of type classes can be observed in the following definition; using the *op* + symbol for *fun* is not possible. In a type class definition, symbols only refer to the type class being defined.

The following type class definition is not valid; the + operation can only be used for the type class being defined

The following type class represents a differential group and a perturbation over it.

class *diff-group-add-pert* = *diff-group-add* +
  fixes *pert* :: $'a \Rightarrow 'a$ (δ - [*81*] *80*)
  assumes *pert-hom-ab*: δ (a + b) = δ a + δ b
  and *pert-preserv-diff-group-add*:
  diff-group-add (*op* −) ($\lambda x.$ − x) *0* (*op* +) ($\lambda x.$ d x + δ x)

instantiation *int* :: *diff-group-add-pert*
begin

definition *pert-int-def*: *pert* ≡ ($\lambda x.$ *0*::*int*)

instance ⟨*proof*⟩

end

We now prove some facts about generic functions. With appropriate restrictions over the type classes over which they are defined, functions can be proved to be also instances of some type classes.

instantiation *fun* :: (*ab-semigroup-add*, *ab-semigroup-add*) *ab-semigroup-add*
begin

definition *plus-fun-def*: f + g == (%x. f x + g x)

instance ⟨*proof*⟩

end

instantiation *fun* :: (*comm-monoid-add*, *comm-monoid-add*) *comm-monoid-add*
begin

definition *zero-fun-def*: *0* == ($\lambda x.$ *0*)

instance ⟨*proof*⟩

end

The Isabelle release 2008 already contains the definition of the difference of functions and also the unary minus

instantiation *fun* :: (*ab-group-add*, *ab-group-add*) *ab-group-add*
begin

instance ⟨*proof*⟩

end

The following type class specifies a differential group with a perturbation and also a homotopy operator.

The previous fact about the *fun* datatype contructor allows us now to use *op* − to define α in a more readable way

class *diff-group-add-pert-hom* = *diff-group-add-pert* +
 fixes *hom-oper*:: $'a$ => $'a$ (*h* - [*81*] *80*)
 assumes *h-hom-ab*: $h\ (a + b) = h\ a + h\ b$
 and *h-nilpot*: $(\lambda x.\ h\ x) \circ (\lambda x.\ h\ x) = (\lambda x.\ 0)$
begin

definition α :: $'a$ => $'a$
 where $\alpha = (\lambda x.\ -\ (pert\ (hom\text{-}oper\ x)))$

end

instantiation *int* :: *diff-group-add-pert-hom*
begin

definition *hom-oper-int-def*: *hom-oper* ≡ ($\lambda x.$ *0*::*int*)

instance ⟨*proof*⟩

end

lemma [*code*]: **shows** $\alpha = (-\ ((\lambda\ x.\ \delta\ x) \circ (\lambda x.\ h\ x)))$
 ⟨*proof*⟩

27 Local nilpotency

We add now the notion of *local-bounded-func*in a purely *existential* way; from the existential definition we will later define the function providing this local bound for every x.

The reason to introduce now this notion is that α is the function verifying the local nilpotency condition

context *ab-group-add*
begin

definition *local-bounded-func* :: ($'a$ => $'a$) => *bool*
 where *local-bounded-func f* = (∀ *x*. ∃ *n*. (*f*ˆ*n*) *x* = *0*)

Here is a relevant difference with the previous proof of the BPL; there, the local bound was defined as the Least natural number n satisfying the property (α ˆ n) x = (*0*::$'a$). Now, in our attempt to make this definition computable, or executable, we define it as an iterating structure (a *For* loop), where the boolean condition in the loop is expressed as $\lambda y.\ y \neq (0{::}'a)$

Later we will try to apply the code generator over these definitions

definition *local-bound-gen* :: ('a => 'a) => 'a => nat => nat
  where *local-bound-gen f x n == For (λ y. y ≠ 0) f (λ y n. n + (1::nat)) x n*

definition *local-bound*:: ('a => 'a) => 'a => nat
  where *local-bound f x = local-bound-gen f x 0*

end

We now define the simplification rule for *local-bound-gen*:

lemmas *local-bound-gen-simp =*
  For-simp[of (λ y. y ≠ (0::'a::ab-group-add)) - λ y n. n+(1::nat),
  simplified local-bound-gen-def[symmetric]]

Two simple "calculations" with *local-bound*:

lemma *local-bound f 0 = 0*
  ⟨*proof*⟩

lemma *x ≠ 0 ⟹ local-bound (λ x. 0) x = 1*
  ⟨*proof*⟩

Now, we connect the neccesary termination of For with our termination condition, *local-bounded-func*.

Then, under the *local-bounded-func* premise the loop will be terminating.

lemma *local-bounded-func-impl-terminates-loop*:
  local-bounded-func f = (∀ x. terminates (λ y. y ≠ 0, f, x))
  ⟨*proof*⟩

lemma *LEAST-local-bound-0*:
  (LEAST n::nat. (f ^ n) (0::'a::ab-group-add) = (0::'a)) = 0
  ⟨*proof*⟩

lemma *local-bound-gen-correct*:
  terminates (λ y. y ≠ (0::'a::ab-group-add), f, x)
  ⟹ local-bound-gen f x m = m + (LEAST n::nat. (f^n) x = 0)
  ⟨*proof*⟩

The following lemma exactly represents the difference between our old definitions, with which we proved the BPL, and the new ones, from which we are trying to generate code; under the termination premise, both *LEAST n. (f ^ n) x = (0::'b)*, the old definition of local nilpotency, and *local-bound f x*, the loop computing the lower bound, are equivalent

Whereas *LEAST n.* $(f\ \hat{}\ n)\ x = (0::'b)$ does not have a computable interpretation, *local-bound f x* does have it, and code can be generated from it.

lemma *local-bound-correct*:
  terminates $(\lambda\ y.\ y \neq (0::'a::ab\text{-}group\text{-}add), f, x)$
  $\Longrightarrow$ *local-bound f x* $=$ (*LEAST* $n::nat.\ (f\hat{}n)\ x = 0$)
  ⟨*proof*⟩

lemma *local-bounded-func-impl-local-bound-is-Least*:
  assumes *lbf-f*:*local-bounded-func f*
  shows *local-bound f x* $=$ (*LEAST* $n::nat.\ (f\hat{}n)\ x = 0$)
  ⟨*proof*⟩

Both *local-bound* and *terminates* are executable.

This is another possible definition of our iterative process as a tail recursion, instead of using *While*, suggested by Alexander Krauss; code generation is also possible from this definition.

A good motivation to use the *While* operator instead of this one, is that some additional induction principles have been provided for the *For* and *While* operators.

function (*tailrec*) *local-bound'* :: $('a::zero \Rightarrow 'a) \Rightarrow 'a \Rightarrow nat \Rightarrow nat$
  where
  local-bound' f x n
  $=$ (*if* $(f\hat{}n)\ x = 0$ *then n else local-bound' f x* (*Suc n*))
  ⟨*proof*⟩

lemma [*code func*]:
  local-bound' f $(x::'a::zero)$ *n*
  $=$ (*if* $(f\hat{}n)\ x = 0$ *then n else local-bound' f x* (*Suc n*))
  ⟨*proof*⟩

export-code *local-bound'*
  in *SML* **file** *local-bound.ML*

lemma [*code func*]:
  While continue f s
  $=$ (*if continue s then While continue f* (*f s*) *else s*)
  ⟨*proof*⟩

export-code *While*
  in *SML* **file** *Loop.ML*

export-code *local-bound*
  in *SML* **file** *local-bound2.ML*

28 Finite sums

The following definition of *fin-sum* will replace the definitions for sums of series used in the formal proof of the BPL.

That definitions were based on the fold operator over sets, from which direct code generation cannot be obtained.

The finite sum of a series is defined as a primitive recursive function over the natural numbers.

This definition will have to be proved later equivalent, in our setting, to the sums appearing in the BPL proof.

primrec *fin-sum* :: (('a::ab-group-add) => 'a) => nat => ('a => 'a)
  where
  fin-sum f 0 =id
  | *fin-sum f (Suc n) = f^(Suc n) + (fin-sum f n)*

The following definition of *diff-group-add-pert-hom-bound-exist* contains the local nilpotency condition. It is based on an existential statement.

For all x belonging to our differential group, we state the existence of a natural number n which is a bound for α

We then prove that it is equivalent to the previously given definition of *locale-bounded-func*.

Finally, we link this fact to the previous results about computation of the bounds as a *For* operator.

class *diff-group-add-pert-hom-bound-exist* =
  diff-group-add-pert-hom +
  assumes *local-nilp-cond*: $\forall x.\exists n::nat.\ (\alpha\hat{}n)\ x = 0$

lemma *diff-group-add-pert-hom-bound-exist-impl-local-bounded-func-alpha*:
  assumes *diff-group-add-pert-hom-bound-exist*:
  diff-group-add-pert-hom-bound-exist
  op $-$ $(\lambda x.\ -\ x)$ *0 op* $+$ $(\lambda x.\ d\ x)$ $(\lambda x.\ \delta\ x)$ $(\lambda x.\ h\ x)$
  shows *local-bounded-func* (α::'a::*diff-group-add-pert-hom-bound-exist* => 'a)
  ⟨*proof*⟩

lemma *diff-group-add-pert-hom-bound-exist-impl-local-bound-is-Least*:
  assumes *diff*:
  diff-group-add-pert-hom-bound-exist
  op $-$ $(\lambda x.\ -\ x)$ *0 op* $+$ $(\lambda x.\ d\ x)$ $(\lambda x.\ \delta\ x)$ $(\lambda x.\ h\ x)$
  shows *local-bound* α (x::'a::*diff-group-add-pert-hom-bound-exist*)
  = (*LEAST* n::nat. $(\alpha\hat{}n)\ x = 0$)
  ⟨*proof*⟩

Apparently, $'a$ does not belong to the appropriate type class.

It does not seem either a good option to use long qualifiers with the locale name

Instead, we have to use the following explicit restriction of the type parameter

The following definitions will have to be later compared with the ones Φ, ...

Additionally, code generation from them must be possible.

definition Φ ::
  $('a::\mathit{diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist} => 'a)$
  where $\Phi = (\lambda x.\ \mathit{fin\text{-}sum}\ \alpha\ (\mathit{local\text{-}bound}\ \alpha\ x)\ x)$

definition β :: $('a::\mathit{diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist} => 'a)$
  where $\beta = (-\ ((\lambda x.\ h\ x) \circ (\lambda\ x.\ \delta\ x)))$

definition Ψ :: $('a::\mathit{diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist} \Rightarrow 'a)$
  where $\Psi = (\lambda x.\ \mathit{fin\text{-}sum}\ \beta\ (\mathit{local\text{-}bound}\ \beta\ x)\ x)$

The following definitions are also to be compared with the ones appearing in the output of the $\mathit{BPL}\ ?D\ ?R\ ?h\ ?C\ ?f\ ?g\ ?\delta\ ?\mathit{bound\text{-}phi} \Longrightarrow \mathit{reduction}$ $(\mathit{lemma\text{-}2\text{-}2\text{-}15.D'}\ ?D\ ?R\ ?\delta)$ $(\!|\mathit{carrier} = \mathit{carrier}\ ?C,\ \mathit{mult} = \mathit{op} \otimes_{?C},\ \mathit{one} = \mathbf{1}_{?C},\ \mathit{diff\text{-}group.diff} = \lambda x.\ \mathit{if}\ x \in \mathit{carrier}\ ?C\ \mathit{then}\ \mathit{diff\text{-}group.diff}\ ?C\ x \otimes_{?C} (?f \circ ?\delta \circ \mathit{local\text{-}nilpotent\text{-}alpha}.\Psi\ ?D\ ?R\ ?h\ ?\delta \circ ?g)\ x\ \mathit{else}\ \mathbf{1}_{?C}|\!)$ $(?f \circ \mathit{local\text{-}nilpotent\text{-}alpha}.\Phi\ ?D\ ?R\ ?h\ ?\delta\ ?\mathit{bound\text{-}phi})$ $(\mathit{local\text{-}nilpotent\text{-}alpha}.\Psi\ ?D\ ?R\ ?h\ ?\delta \circ ?g)$ $(\mathit{lemma\text{-}2\text{-}2\text{-}15.h'}\ ?D\ ?R\ ?h\ ?\delta\ ?\mathit{bound\text{-}phi})$

definition dC' :: $('a::\mathit{diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist} => 'b::\mathit{diff\text{-}group\text{-}add})$
  $=> ('b => 'a) => ('b => 'b)$
  where $dC'\ f\ g = \mathit{diff} + (f \circ (\lambda x.\ \delta\ x) \circ \Psi \circ g)$

definition f' :: $('a::\mathit{diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist} => 'b::\mathit{diff\text{-}group\text{-}add})$
  $=> ('a => 'b)$
  where $f'\ f = f \circ \Phi$

definition g' :: $('b::\mathit{diff\text{-}group\text{-}add} => 'a::\mathit{diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist})$
  $=> ('b => 'a)$
  where $g'\text{-}\mathit{def}$: $g'\ g == \Psi \circ g$

definition h' :: $('a::\mathit{diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist} \Rightarrow 'a)$
  where $h' == (\lambda x.\ h\ x) \circ \Phi$

export-code $\Phi\ \Psi\ f'\ g'\ h'\ dC'$ **in** *SML* **file** *output-reduction.ML*

Some facts about the product of types:

instantiation $*$ :: $(\mathit{ab\text{-}semigroup\text{-}add},\ \mathit{ab\text{-}semigroup\text{-}add})\ \mathit{ab\text{-}semigroup\text{-}add}$

begin

definition *mult-plus-def*:
  $x + y \equiv (let\ (x1,\ x2) = x;\ (y1,\ y2) = y\ in\ (x1 + y1,\ x2 + y2))$

instance ⟨*proof*⟩

end

definition $x5$:: $(int \times int)$
  where $x5 = ((3::int), (5::int)) + (5,\ 7)$

definition $x6$ :: $(int \times int) \times (int \times int)$
  where $x6 = (((3::int), (5::int)), ((3::int), (5::int))) + ((5,\ 7), (5,\ 7))$

export-code *x5 x6*
  in *SML* **file** *x6.ML*

instantiation $*$ :: (*comm-monoid-add*, *comm-monoid-add*) *comm-monoid-add*
begin

definition *mult-zero-def*: $0 \equiv (0,\ 0)$

instance ⟨*proof*⟩

end

29 Equivalence of both approaches

29.1 Algebraic structures

In the following section we prove that the results already proved using the definitions provided by the Algebra Isabelle Library (leading to the *reduction* D' ⦇*carrier* = *carrier* C, *mult* = *op* $\otimes_C$, *one* = $\mathbf{1}_C$, *diff-group.diff* = λx. *if* $x \in$ *carrier* C *then diff-group.diff* $C\ x \otimes_C$ ($f \circ \delta \circ$ *D-R-C-f-g-h-δ-α-bound-phi*.Ψ $\circ\ g$) x *else* $\mathbf{1}_C$⦈ ($f \circ$ *D-R-C-f-g-h-δ-α-bound-phi*.Φ) (*D-R-C-f-g-h-δ-α-bound-phi*.Ψ $\circ\ g$) *D-R-h-C-f-g-δ-α-bound-phi*.h') also hold for the new definitions, where algebraic structures are implemented by means of type classes.

This does not mean that the proof of the BPL can be developed only by using type classes; the degree of expressivity needed in its proofs should be quite hard to achieve using type classes; specially, the parts where restrictions of domains of functions have to be used.

We only pretend to prove that the new definitions, to some extent simplified (see for instance the new proposed series in relation to the previous implementation of series), are equivalent to the old ones, and thus, also satisfy the BPL.

Functions (or functors) translating type classes into algebraic structures implemented as recods are used; these translations are the natural ones

definition *monoid-functor* :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow ('a) \Rightarrow 'a$ *monoid*
 where *monoid-functor A B* == (| *carrier* = *UNIV*, *mult* = *A*, *one* = *B*|)

lemma *monoid-add-impl-monoid*:
 assumes *mon-add*: *monoid-add zero' plus'*
 shows *monoid* (| *carrier* = *UNIV*, *mult* = *plus'*, *one* = *zero'*|)
 ⟨*proof*⟩

lemma *monoid-functor-preserv*:
 assumes *monoid-add*:*monoid-add zero' plus'*
 shows *monoid* (*monoid-functor plus' zero'*)
 ⟨*proof*⟩

lemma *comm-monoid-add-impl-monoid-add*:
 assumes *comm-monoid-add*: *comm-monoid-add zero' plus'*
 shows *monoid-add zero' plus'*
 ⟨*proof*⟩

lemma *comm-monoid-add-impl-monoid*:
 assumes *c-m*: *comm-monoid-add zero' plus'*
 shows *monoid* (| *carrier* = *UNIV*, *mult* = *plus'*, *one* = *zero'*|)
 ⟨*proof*⟩

lemma *ab-group-add-impl-comm-monoid-add*:
 assumes *ab-gr-add*: *ab-group-add uminus' minus' zero' plus'*
 shows *comm-monoid-add zero' plus'*
 ⟨*proof*⟩

lemma *ab-group-class-impl-group*:
 assumes *ab-gr-class*: *ab-group-add uminus' minus' zero' plus'*
 shows *group* (| *carrier* = *UNIV*, *mult* = *plus'*, *one* = *zero'*|)
⟨*proof*⟩

lemma *monoid-functor-preserv-group*:
 assumes *ab-gr*: *ab-group-add uminus' minus' zero' plus'*
 shows *group* (*monoid-functor plus' zero'*)
 ⟨*proof*⟩

lemma *ab-group-add-impl-comm-group*:
 assumes *ab-gr-add*: *ab-group-add uminus' minus' zero' plus'*
 shows *comm-group* (| *carrier* = *UNIV*, *mult* = *plus'*, *one* = *zero'*|)
⟨*proof*⟩

lemma *monoid-functor-preserv-ab-group*:
 assumes *ab-gr-add*: *ab-group-add uminus' minus' zero' plus'*
 shows *comm-group* (*monoid-functor plus' zero'*)
 ⟨*proof*⟩

lemma *diff-group-add-impl-comm-group*:
assumes *diff-gr-add*: *diff-group-add uminus′ minus′ zero′ plus′ diff′*
shows *comm-group* (|*carrier* = *UNIV*, *mult* = *plus′*, *one* = *zero′*|)
⟨*proof*⟩

lemma *diff-group-add-impl-diff-group*:
assumes *diff-gr-add*: *diff-group-add uminus′ minus′ zero′ prod′ diff′*
shows *diff-group* (|*carrier* = *UNIV*, *mult* = *prod′*, *one* = *zero′*, *diff-group.diff* = *diff′* |)
⟨*proof*⟩

definition *diff-group-functor* ::
(′*a* => ′*a*) => (′*a* => ′*a* => ′*a*) => (′*a*)
=> (′*a* => ′*a* => ′*a*) => (′*a* => ′*a*) => ′*a diff-group*
where *diff-group-functor uminus′ minus′ zero′ prod′ diff′* =
(| *carrier* = *UNIV*, *mult* = *prod′*, *one* = *zero′*, *diff-group.diff* = *diff′*|)

lemma *diff-group-functor-preserves*:
assumes *diff-gr-add*: *diff-group-add minus′ uminus′ zero′ prod′ diff′*
shows *diff-group* (*diff-group-functor uminus′ minus′ zero′ prod′ diff′*)
⟨*proof*⟩

After the previous equivalences between algebraic structures, now we prove the equivalence between the old definitions about homomorphisms and the new ones:

29.2 Homomorphisms and endomorphisms.

definition *homo-ab* :: (′*a*::*comm-monoid-add* => ′*b*::*comm-monoid-add*) => *bool*

where *homo-ab f* = (*ALL a b*. *f* (*a* + *b*) = *f a* + *f b*)

lemma *homo-ab-apply*:
assumes *h-f*: *homo-ab f*
shows *f* (*a* + *b*) = *f a* + *f b*
⟨*proof*⟩

lemma *homo-ab-preserves-hom-completion*:
assumes *homo-ab-f*: *homo-ab f*
shows *f* ∈ *hom-completion* (*monoid-functor* (*op* +) *0*) (*monoid-functor* (*op* +) *0*)
⟨*proof*⟩

lemma *plus-fun-apply*:
(*f* + *g*) (*x*::′*a*::*ab-semigroup-add*) = *f x* + *g x*
⟨*proof*⟩

lemma *homo-ab-plus-closed*:

assumes *comm-monoid-add-A*: *comm-monoid-add* $(0::'a::comm\text{-}monoid\text{-}add)$ *op* $+$
and *comm-monoid-add-B*: *comm-monoid-add* $(0::'b::comm\text{-}monoid\text{-}add)$ *op* $+$
and x: *homo-ab* $(x::'a::comm\text{-}monoid\text{-}add => 'b::comm\text{-}monoid\text{-}add)$
and y: *homo-ab* y
shows *homo-ab* $(x + y)$
⟨*proof*⟩

lemma *end-comm-monoid-add-closed*:
assumes *comm-monoid-add*: *comm-monoid-add* $(0::'a::comm\text{-}monoid\text{-}add)$ *op* $+$
and x: *homo-ab* $(x::'a::comm\text{-}monoid\text{-}add => 'a)$
and y: *homo-ab* y
shows *homo-ab* $(x + y)$
⟨*proof*⟩

lemma *comm-monoid-add-impl-homo-abelian-monoid*:
assumes *comm-monoid-add*: *comm-monoid-add* $(0::'a::comm\text{-}monoid\text{-}add)$ *op* $+$
shows *abelian-monoid* $(\!|$*carrier* $= \{f::'a::comm\text{-}monoid\text{-}add => 'a.\ homo\text{-}ab\ f\}$,

mult $=$ *op* $\circ$,
one $=$ *id*,
zero $= 0$,
add $=$ *op* $+|\!)$
⟨*proof*⟩

lemma *ab-group-add-impl-uminus-fun-closed*:
assumes *ab-group-add*: *ab-group-add op* $-$ $(\lambda x.\ - x)$ $(0::'a::ab\text{-}group\text{-}add)$ *op* $+$

and f: *homo-ab* $(f::'a::ab\text{-}group\text{-}add => 'a)$
shows *homo-ab* $(- f)$
⟨*proof*⟩

lemma *ab-group-add-impl-homo-abelian-group-axioms*:
assumes *ab-group-add*: *ab-group-add op* $-$ $(\lambda x.\ - x)$ $(0::'a::ab\text{-}group\text{-}add)$ *op* $+$
shows *abelian-group-axioms* $(\!|$*carrier* $= \{f::'a::ab\text{-}group\text{-}add => 'a.\ homo\text{-}ab\ f\}$,

mult $=$ *op* $\circ$,
one $=$ *id*,
zero $= 0$,
add $=$ *op* $+|\!)$
⟨*proof*⟩

The previous lemma, *ab-group-add op* $-$ *uminus* $(0::?'a)$ *op* $+ \Longrightarrow$ *abelian-group-axioms* $(\!|$*carrier* $= \{f.\ homo\text{-}ab\ f\}$, *mult* $=$ *op* $\circ$, *one* $=$ *id*, *zero* $= 0$, *add* $=$ *op* $+|\!)$, proves the elements of *homo-ab* to be an abelian monoid under suitable operations.

In order to show that composition gives place to a monoid, the underlying structure needs not to be even a monoid

lemma *homo-monoid*:
  shows *monoid* (|*carrier* = {*f*. *homo-ab f*},
  monoid.mult = *op* ∘,
  one = *id*,
  zero = *0*,
  add = *op* +|)
  (**is** *monoid ?HOMO-AB*)
⟨*proof*⟩

A couple of lemmas completing the proof of the set *homo-ab* being a ring, with suitable operations

lemma *ab-group-add-impl-homo-ring-axioms*:
  assumes *ab-group-add*: *ab-group-add op* − (λ*x*. − *x*) (*0*::'*a*::*ab-group-add*) *op* +
  shows *ring-axioms* (|*carrier* = {*f*. *homo-ab f*},
  mult = *op* ∘,
  one = *id*,
  zero = (*0*::'*a*::*ab-group-add* => '*a*),
  add = *op* +|)
⟨*proof*⟩

lemma *ab-group-add-impl-homo-ring*:
  assumes *ab-group-add*: *ab-group-add op* − (λ*x*. − *x*) (*0*::'*a*::*ab-group-add*) *op* +
  shows *ring* (|*carrier* = {*f*::'*a*::*ab-group-add* => '*a*. *homo-ab f*},
  mult = *op* ∘,
  one = *id*,
  zero = (*0*::'*a* => '*a*),
  add = *op* +|)
  ⟨*proof*⟩

The following definition includes the notion of differential homomorphism, a homomorphism that additionally commutes with the corresponding differentials.

definition *homo-diff* :: ('*a*::*diff-group-add* => '*b*::*diff-group-add*) => *bool*
  where *homo-diff f* = ((*ALL a b*. *f* (*a* + *b*) = *f a* + *f b*) ∧ *f* ∘ *diff* = *diff* ∘ *f*)

lemma *homo-diff-preserves-hom-diff*:
  assumes *homo-diff-f*: *homo-diff f*
  shows *f* ∈ *hom-diff* (*diff-group-functor* (λ*x*. − *x*) *op* − *0* (*op* +) *diff*)
  (*diff-group-functor* (λ*x*. − *x*) *op* − *0* (*op* +) *diff*)
  ⟨*proof*⟩

29.3 Definition of constants.

The following definition of reduction is to be understand as follows: a pair of homomorphisms (*f*, *g*) will be a reduction iff: the underlying algebraic structures (given as classes) are, respectively, a differential group class with a perturbation and a homology operator (i.e, class *diff-group-add-pert-hom-bound-exists*) satisfying the local nilpotency condition, and a differential group class (*diff-group-add*), and the homomorphisms *f*, *g* and *h* satisfy the properties required by the usual reduction definition.

In the definition it can be noted the convenience of using overloading symbols provided by the class mechanism.

definition
reduction-class-ext ::
($'a$::*diff-group-add-pert-hom-bound-exist* => $'b$::*diff-group-add*)
=> ($'b$ => $'a$) => *bool*
where *reduction-class-ext f g* =
((*diff-group-add-pert-hom-bound-exist op* − (λx::$'a$. − x) *0* (*op* +) *diff pert hom-oper*)
∧ (*diff-group-add op* − (λx::$'b$. − x) *0* (*op* +) *diff*)
∧ (*homo-diff f*) ∧ (*homo-diff g*)
∧ ($f \circ g$ = *id*)
∧ (($g \circ f$) + (*diff* ∘ *hom-oper*) + (*hom-oper* ∘ *diff*) = *id*)
∧ (f ∘ *hom-oper* = (*0*::$'a$ => $'b$))
∧ (*hom-oper* ∘ g = (*0*::$'b$ => $'a$)))

The previous definition contains all the ingredients required to apply the old proof of the BPL.

lemma *reduction-class-ext-impl-diff-group-add-pert-hom-bound-exist*:
assumes *r-c-e*: *reduction-class-ext* (f::$'a$::*diff-group-add-pert-hom-bound-exist* => $'b$::*diff-group-add*) g
shows (*diff-group-add-pert-hom-bound-exist op* −
(λx::$'a$::*diff-group-add-pert-hom-bound-exist*. − x) *0* (*op* +) *diff pert hom-oper*)
⟨*proof*⟩

lemma *reduction-class-ext-impl-diff-group-add*:
assumes *r-c-e*: *reduction-class-ext* (f::$'a$::*diff-group-add-pert-hom-bound-exist* => $'b$::*diff-group-add*) g
shows (*diff-group-add op* − (λx::$'b$::*diff-group-add*. − x) *0* (*op* +) *diff*)
⟨*proof*⟩

lemma *reduction-class-ext-impl-homo-diff-f*:
assumes *r-c-e*: *reduction-class-ext* (f::$'a$::*diff-group-add-pert-hom-bound-exist* => $'b$::*diff-group-add*) g
shows *homo-diff f*
⟨*proof*⟩

lemma *reduction-class-ext-impl-homo-diff-g*:
assumes *r-c-e*: *reduction-class-ext* (f::$'a$::*diff-group-add-pert-hom-bound-exist* => $'b$::*diff-group-add*) g
shows *homo-diff g*
⟨*proof*⟩

lemma *reduction-class-ext-impl-fg-id*:
assumes *r-c-e*: *reduction-class-ext* (f::$'a$::*diff-group-add-pert-hom-bound-exist*

$=>$ $'b::diff\text{-}group\text{-}add$) g
shows $f \circ g = id$
⟨*proof*⟩

lemma *reduction-class-ext-impl-gf-dh-hd-id*:
assumes *r-c-e*: *reduction-class-ext* ($f::'a::diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist$
$=>$ $'b::diff\text{-}group\text{-}add$) g
shows $(g \circ f) + (diff \circ hom\text{-}oper) + (hom\text{-}oper \circ diff) = id$
⟨*proof*⟩

lemma *reduction-class-ext-impl-fh-0*:
assumes *r-c-e*: *reduction-class-ext* ($f::'a::diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist$
$=>$ $'b::diff\text{-}group\text{-}add$) g
shows $f \circ hom\text{-}oper = 0$
⟨*proof*⟩

lemma *reduction-class-ext-impl-hg-0*:
assumes *r-c-e*: *reduction-class-ext* ($f::'a::diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist$
$=>$ $'b::diff\text{-}group\text{-}add$) g
shows $hom\text{-}oper \circ g = 0$
⟨*proof*⟩

The following lemma will be useful later, when we verify the premises of the BPL

lemma *hdh-eq-h*:
assumes *r-c-e*: *reduction-class-ext f* ($g::'b::diff\text{-}group\text{-}add$
$=>$ $'a::diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist$)
shows ($hom\text{-}oper::'a::diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist$ $=>$ $'a$)
$\circ$ $diff \circ hom\text{-}oper = hom\text{-}oper$
⟨*proof*⟩

lemma *diff-group-add-pert-hom-bound-exist-impl-diff-group-add*:
assumes *d-g*: (*diff-group-add-pert-hom-bound-exist op* $-$
($\lambda x::'a::diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist.\ - x$) *0* (*op* $+$) *diff pert hom-oper*)
shows *diff-group-add op* $-$
($\lambda x::'a::diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist.\ - x$) *0* (*op* $+$) *diff*
⟨*proof*⟩

The new definition of *reduction-class-ext* preserves the previous definition of reduction in the old approach, *reduction*

lemma *reduction-class-ext-preserves-reduction*:
assumes *r-c-e*: *reduction-class-ext f g*
shows *reduction* (*diff-group-functor* ($\lambda x::'a::diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist.$
$- x$)
(*op* $-$) *0* (*op* $+$) *diff*)
(*diff-group-functor* ($\lambda x::'b::diff\text{-}group\text{-}add.\ - x$) (*op* $-$) *0* (*op* $+$) *diff*)
f g hom-oper
(**is** *reduction ?D ?C f g hom-oper*)
⟨*proof*⟩

The new definition of perturbation, included in the definition of *diff-group-add-pert*, also preserves the old definition of perturbation, *analytic-part-local.pert*

lemma *diff-group-add-pert-hom-bound-exist-preserves-pert*:
assumes *diff-group-add-pert-hom-bound-exist*:
diff-group-add-pert-hom-bound-exist $(op\ -)$
$(\lambda x::'a::diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist.\ -\ x)\ 0\ (op\ +)\ diff\ pert\ hom\text{-}oper$
shows $pert \in analytic\text{-}part\text{-}local.pert$
$(diff\text{-}group\text{-}functor\ (\lambda x::'a::diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist.\ -\ x)\ (op\ -)\ 0$
$(op\ +)\ diff)$
(**is** $pert \in analytic\text{-}part\text{-}local.pert\ ?D$)
$\langle proof \rangle$

From the premises stated in *diff-group-add-pert-hom-bound-exist*, α is nilpotent

lemma *α-locally-nilpotent*:
assumes *diff-group-add-pert-hom-bound-exist*:
diff-group-add-pert-hom-bound-exist $(op\ -)$
$(\lambda x::'a::diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist.\ -\ x)\ 0\ (op\ +)\ diff\ pert\ hom\text{-}oper$
shows $(\alpha\ \hat{}\ (local\text{-}bound\ \alpha\ x))\ (x::'a::diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist) = 0$
$\langle proof \rangle$

The algebraic structure given by the endomorphisms of a *diff-group-add* with suitable operations is a ring

lemma (**in** *group-add*) **shows** $op\ - = (\lambda x\ y.\ x + (-\ y))$
$\langle proof \rangle$

lemma (**in** *diff-group-add*) *hom-completion-ring*:
shows $ring\ (\!|carrier = hom\text{-}completion$
$(diff\text{-}group\text{-}functor\ (\lambda x::'a.\ -\ x)\ (op\ -)\ 0\ (op\ +)\ diff)$
$(diff\text{-}group\text{-}functor\ (\lambda x::'a.\ -\ x)\ (op\ -)\ 0\ (op\ +)\ diff),$
$mult = op\ \circ,$
$one = \lambda x::'a.\ if\ x \in carrier\ (diff\text{-}group\text{-}functor\ (\lambda x::'a.\ -\ x)\ (op\ -)\ 0\ (op\ +)$
$diff)\ then\ id\ x$
$else\ one\ (diff\text{-}group\text{-}functor\ (\lambda x::'a.\ -\ x)\ (op\ -)\ 0\ (op\ +)\ diff),$
$zero = \lambda x::'a.\ if\ x \in carrier\ (diff\text{-}group\text{-}functor\ (\lambda x::'a.\ -\ x)\ (op\ -)\ 0\ (op\ +)$
$diff)$
$then\ one\ (diff\text{-}group\text{-}functor\ (\lambda x::'a.\ -\ x)\ (op\ -)\ 0\ (op\ +)\ diff)$
$else\ one\ (diff\text{-}group\text{-}functor\ (\lambda x::'a.\ -\ x)\ (op\ -)\ 0\ (op\ +)\ diff),$
$add = \lambda(f::'a \Rightarrow 'a)\ (g::'a \Rightarrow 'a)\ x::'a.$
$if\ x \in carrier\ (diff\text{-}group\text{-}functor\ (\lambda x::'a.\ -\ x)\ (op\ -)\ 0\ (op\ +)\ diff)\ then$
$mult\ (diff\text{-}group\text{-}functor\ (\lambda x::'a.\ -\ x)\ (op\ -)\ 0\ (op\ +)\ diff)\ (f\ x)\ (g\ x)$
$else\ one\ (diff\text{-}group\text{-}functor\ (\lambda x::'a.\ -\ x)\ (op\ -)\ 0\ (op\ +)\ diff)|\!)$
$\langle proof \rangle$

lemma *homo-ab-is-hom-completion*:
assumes *homo-ab-f*: *homo-ab f*
and *diff-group-add*: *diff-group-add* $(op\ -)$
$(\lambda x::'a::diff\text{-}group\text{-}add.\ -\ x)\ 0\ (op\ +)\ diff$

shows $f \in$ *hom-completion*
(*diff-group-functor* $(\lambda x::'a::\mathit{diff\text{-}group\text{-}add}.\ - x)$ $(op\ -)$ 0 $(op\ +)$ *diff*)
(*diff-group-functor* $(\lambda x::'a.\ - x)$ $(op\ -)$ 0 $(op\ +)$ *diff*)
⟨*proof*⟩

lemma *hom-completion-is-homo-ab*:
assumes *f-hom-compl*: $f \in$ *hom-completion*
(*diff-group-functor* $(\lambda x::'a::\mathit{diff\text{-}group\text{-}add}.\ - x)$ $(op\ -)$ 0 $(op\ +)$ *diff*)
(*diff-group-functor* $(\lambda x::'a.\ - x)$ $(op\ -)$ 0 $(op\ +)$ *diff*)
and *diff-group-add*: *diff-group-add* $(op\ -)$ $(\lambda x::'a::\mathit{diff\text{-}group\text{-}add}.\ - x)$ 0 $(op\ +)$ *diff*
shows *homo-ab* f
⟨*proof*⟩

lemma *hom-completion-equiv-homo-ab*:
assumes *diff-group-add*: *diff-group-add* $(op\ -)$ $(\lambda x::'a::\mathit{diff\text{-}group\text{-}add}.\ - x)$ 0 $(op\ +)$ *diff*
shows *homo-ab* $f \longleftrightarrow f \in$ *hom-completion*
(*diff-group-functor* $(\lambda x::'a::\mathit{diff\text{-}group\text{-}add}.\ - x)$ $(op\ -)$ 0 $(op\ +)$ *diff*)
(*diff-group-functor* $(\lambda x::'a.\ - x)$ $(op\ -)$ 0 $(op\ +)$ *diff*)
⟨*proof*⟩

Equivalence between the definition of power in the Isabelle Algebra Library, *nat-pow-def*, over the ring of endomorphisms, and the definition of power for functions, definition *fun-pow*

definition *ring-hom-compl* :: $('a\ \mathit{diff\text{-}group}) => ('a => 'a)\ \mathit{ring}$
where *ring-hom-compl* D == (|*carrier* = *hom-completion* D D,
mult = *op* $\circ$,
one = $\lambda x::'a.$ *if* $x \in$ *carrier* D *then id* x *else one* D,
zero = $\lambda x::'a.$ *if* $x \in$ *carrier* D *then one* D *else one* D,
add = $\lambda(f::'a \Rightarrow 'a)\ (g::'a \Rightarrow 'a)\ x::'a.$ *if* $x \in$ *carrier* D *then mult* D $(f\ x)$ $(g\ x)$ *else one* D|)

lemma *ring-nat-pow-equiv-funpow*:
assumes *diff-group-add*: *diff-group-add* $(op\ -)$ $(\lambda x::'a::\mathit{diff\text{-}group\text{-}add}.\ - x)$ 0 $(op\ +)$ *diff*
and *f-hom-completion*: $f \in$ *hom-completion*
(*diff-group-functor* $(\lambda x::'a::\mathit{diff\text{-}group\text{-}add}.\ - x)$ $(op\ -)$ 0 $(op\ +)$ *diff*)
(*diff-group-functor* $(\lambda x::'a.\ - x)$ $(op\ -)$ 0 $(op\ +)$ *diff*)
shows $f\ (\hat{\ })_{\mathit{ring\text{-}hom\text{-}compl}\ (\mathit{diff\text{-}group\text{-}functor}\ (\lambda x::'a::\mathit{diff\text{-}group\text{-}add}.\ - x)\ (op\ -)\ 0\ (op\ +)\ \mathit{diff})}$ $(n::nat) = f\hat{\ }n$
(**is** $f\ (\hat{\ })_{?R}\ n = f\hat{\ }n$)
⟨*proof*⟩

Equivalence between the *uminus* definition in the ring of endomorphisms, and the $-\ ?A = (\lambda x.\ -\ ?A\ x)$

lemma *minus-ring-homo-equal-uminus-fun*:

assumes *diff-group-add*: *diff-group-add* $(op\ -)$ $(\lambda x{::}'a{::}diff\text{-}group\text{-}add.\ -\ x)$ *0* $(op\ +)$ *diff*
and *homo-ab-f*: *homo-ab f*
shows $\ominus_{ring\text{-}hom\text{-}compl}$ (*diff-group-functor* $(\lambda x{::}'a{::}diff\text{-}group\text{-}add.\ -\ x)$ $(op\ -)$ *0* $(op\ +)$ *diff*) $(f{::}'a{::}diff\text{-}group\text{-}add => 'a) = -\ f$
(**is** $\ominus_{?R}\ f = -\ f$)
⟨*proof*⟩

lemma *minus-ring-hom-completion-equal-uminus-fun*:
assumes *diff-group-add*: *diff-group-add* $(op\ -)$ $(\lambda x{::}'a{::}diff\text{-}group\text{-}add.\ -\ x)$ *0* $(op\ +)$ *diff*
and *f-hom-completion*: $f \in$ *hom-completion*
(*diff-group-functor* $(\lambda x{::}'a{::}diff\text{-}group\text{-}add.\ -\ x)$ $(op\ -)$ *0* $(op\ +)$ *diff*)
(*diff-group-functor* $(\lambda x{::}'a.\ -\ x)$ $(op\ -)$ *0* $(op\ +)$ *diff*)
shows $\ominus_{ring\text{-}hom\text{-}compl}$ (*diff-group-functor* $(\lambda x{::}'a{::}diff\text{-}group\text{-}add.\ -\ x)$ $(op\ -)$ *0* $(op\ +)$ *diff*) $f = -\ f$
(**is** $\ominus_{?R}\ f = -\ f$)
⟨*proof*⟩

lemma α*-in-hom-completion*:
assumes *diff-group-add-pert-hom*:
diff-group-add-pert-hom $(op\ -)$ $(\lambda x{::}'a{::}diff\text{-}group\text{-}add\text{-}pert\text{-}hom.\ -\ x)$
0 op + diff pert hom-oper
shows $\alpha \in$ *hom-completion*
(*diff-group-functor* $(\lambda x{::}'a{::}diff\text{-}group\text{-}add\text{-}pert\text{-}hom.\ -\ x)$ *op − 0 op + diff*)
(*diff-group-functor* $(\lambda x{::}'a.\ -\ x)$ *op − 0 op + diff*)
(**is** $\alpha \in$ *hom-completion ?D ?D*)
⟨*proof*⟩

lemma β*-in-hom-completion*:
assumes *diff-group-add-pert-hom*:
diff-group-add-pert-hom op − $(\lambda x{::}'a{::}diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist.\ -\ x)$
0 op + diff pert hom-oper
shows $\beta \in$ *hom-completion*
(*diff-group-functor* $(\lambda x{::}'a{::}diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist.\ -\ x)$ *op − 0 op + diff*)
(*diff-group-functor* $(\lambda x{::}'a.\ -\ x)$ *op − 0 op + diff*)
(**is** $\beta \in$ *hom-completion ?D ?D*)
⟨*proof*⟩

Our previous deinition of *reduction-class-ext* satisfies the definition of the locale *local-nilpotent-term*

lemma *reduction-class-ext-preserves-local-nilpotent-term*:
assumes *reduction-class-ext-f-g*:
reduction-class-ext $(f{::}'a{::}diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist => 'b{::}diff\text{-}group\text{-}add)$ g
shows *local-nilpotent-term*
(*diff-group-functor* $(\lambda x{::}'a{::}diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist.\ -\ x)$ *op − 0 op + diff*)

$(\textit{ring-hom-compl}\ (\textit{diff-group-functor}\ (\lambda x::'a.\ -\ x)\ \textit{op}\ -\ \textit{0}\ \textit{op}\ +\ \textit{diff}))$
$\alpha\ (\textit{local-bound}\ \alpha)$
$(\mathbf{is}\ \textit{local-nilpotent-term}\ ?D\ ?R\ \alpha\ (\textit{local-bound}\ \alpha))$
$\langle \textit{proof} \rangle$

The following lemma states that the *reduction-class-ext* definition together with *local-bound-exists* satisifies the premises of the $\textit{BPL}\ ?D\ ?R\ ?h\ ?C\ ?f\ ?g\ ?\delta\ ?\textit{bound-phi} \Longrightarrow \textit{reduction}\ (\textit{lemma-2-2-15}.D'\ ?D\ ?R\ ?\delta)\ (\!|\textit{carrier} = \textit{carrier}\ ?C,\ \textit{mult} = \textit{op} \otimes_{?C},\ \textit{one} = \mathbf{1}_{?C},\ \textit{diff-group.diff} = \lambda x.\ \textit{if}\ x \in \textit{carrier}\ ?C\ \textit{then}\ \textit{diff-group.diff}\ ?C\ x \otimes_{?C} (?f \circ ?\delta \circ \textit{local-nilpotent-alpha}.\Psi\ ?D\ ?R\ ?h\ ?\delta \circ ?g)\ x\ \textit{else}\ \mathbf{1}_{?C}|\!)\ (?f \circ \textit{local-nilpotent-alpha}.\Phi\ ?D\ ?R\ ?h\ ?\delta\ ?\textit{bound-phi})\ (\textit{local-nilpotent-alpha}.\Psi\ ?D\ ?R\ ?h\ ?\delta \circ ?g)\ (\textit{lemma-2-2-15}.h'\ ?D\ ?R\ ?h\ ?\delta\ ?\textit{bound-phi})$.

In addition to this result, we also have to prove later that the definitions given in this file for f', g', Φ, are equivalent to the ones given inside of the local BPL

lemma *reduction-class-ext-preserves-BPL*:
assumes *r-c-e*:
$\textit{reduction-class-ext}\ (f::'a::\textit{diff-group-add-pert-hom-bound-exist} => 'b::\textit{diff-group-add})$
g
shows *BPL*
$(\textit{diff-group-functor}\ (\lambda x::'a::\textit{diff-group-add-pert-hom-bound-exist}.\ -\ x)\ \textit{op}\ -\ \textit{0}\ \textit{op}$
$+\ \textit{diff})$
$(\textit{ring-hom-compl}\ (\textit{diff-group-functor}\ (\lambda x::'a.\ -\ x)\ \textit{op}\ -\ \textit{0}\ \textit{op}\ +\ \textit{diff}))$
hom-oper
$(\textit{diff-group-functor}\ (\lambda x::'b::\textit{diff-group-add}.\ -\ x)\ \textit{op}\ -\ \textit{0}\ \textit{op}\ +\ \textit{diff})$
f g pert
$(\textit{local-bound}\ \alpha)$
$(\mathbf{is}\ \textit{BPL}\ ?D\ ?R\ \textit{hom-oper}\ ?C\ f\ g\ \textit{pert}\ (\textit{local-bound}\ \alpha))$
$\langle \textit{proof} \rangle$

The definition of *reduction-class-ext* satisfies the definition of the locale *lemma-2-2-15*.

lemma *reduction-class-ext-preserves-lemma-2-2-15*:
assumes *r-c-e*:
$\textit{reduction-class-ext}\ (f::'a::\textit{diff-group-add-pert-hom-bound-exist}$
$=> 'b::\textit{diff-group-add})\ g$
shows *lemma-2-2-15*
$(\textit{diff-group-functor}\ (\lambda x::'a::\textit{diff-group-add-pert-hom-bound-exist}.\ -\ x)$
$\textit{op}\ -\ \textit{0}\ \textit{op}\ +\ \textit{diff})$
$(\textit{ring-hom-compl}\ (\textit{diff-group-functor}\ (\lambda x::'a.\ -\ x)\ \textit{op}\ -\ \textit{0}\ \textit{op}\ +\ \textit{diff}))$
$\textit{hom-oper}\ (\textit{diff-group-functor}\ (\lambda x::'b::\textit{diff-group-add}.\ -\ x)\ \textit{op}\ -\ \textit{0}\ \textit{op}\ +\ \textit{diff})$
f g pert
$(\textit{local-bound}\ \alpha)$
$\langle \textit{proof} \rangle$

The definition of *reduction-class-ext* satisfies the definiton of the locale *local-nilpotent-alpha*

lemma *reduction-class-ext-preserves-local-nilpotent-alpha*:
  assumes *r-c-e*:
  reduction-class-ext (f::$'a$::*diff-group-add-pert-hom-bound-exist*
  => $'b$::*diff-group-add*) g
  shows *local-nilpotent-alpha*
  (*diff-group-functor* (λx::$'a$::*diff-group-add-pert-hom-bound-exist*. $-$ x)
  op $-$ *0* *op* $+$ *diff*)
  (*ring-hom-compl* (*diff-group-functor* (λx::$'a$. $-$ x) *op* $-$ *0* *op* $+$ *diff*))
  (*diff-group-functor* (λx::$'b$::*diff-group-add*. $-$ x) *op* $-$ *0* *op* $+$ *diff*)
  f g *hom-oper pert*
  (*local-bound* α)
  $\langle$*proof*$\rangle$

The definition λx. *fin-sum* α (*local-bound* α x) x in *reduction-class-ext* is equivalent to the previous definition of *power-series* in locale *local-nilpotent-term*.

lemma *reduction-class-ext-preserves-power-series*:
  assumes *r-c-e*:
  reduction-class-ext (f::$'a$::*diff-group-add-pert-hom-bound-exist*
  => $'b$::*diff-group-add*) g
  shows *local-nilpotent-term.power-series*
  (*diff-group-functor* (λx::$'a$::*diff-group-add-pert-hom-bound-exist*. $-$ x) *op* $-$ *0* *op*
$+$ *diff*)
  (*ring-hom-compl* (*diff-group-functor* (λx::$'a$. $-$ x) *op* $-$ *0* *op* $+$ *diff*))
  α
  (*local-bound* α) = (λx::$'a$. *fin-sum* α (*local-bound* α x) x)
  (**is** *local-nilpotent-term.power-series* *?D* *?R* α (*local-bound* α)
    = (λx. *fin-sum* α (*local-bound* α x) x))
$\langle$*proof*$\rangle$

The definition of $\Phi = (\lambda x.$ *fin-sum* α (*local-bound* α x) $x)$ is equivalent to the previous definition of *D-R-C-f-g-h-δ-α-bound-phi*.Φ in *locale-nilpotent-alpha*

lemma *reduction-class-ext-preserves-*Φ:
  assumes *r-c-e*:
  reduction-class-ext (f::$'a$::*diff-group-add-pert-hom-bound-exist*
  => $'b$::*diff-group-add*) g
  shows *local-nilpotent-alpha*.Φ
  (*diff-group-functor* (λx::$'a$::*diff-group-add-pert-hom-bound-exist*. $-$ x)
  op $-$ *0* *op* $+$ *diff*)
  (*ring-hom-compl* (*diff-group-functor* (λx::$'a$. $-$ x) *op* $-$ *0* *op* $+$ *diff*))
  hom-oper
  pert
  (*local-bound* α) = Φ
  (**is** *local-nilpotent-alpha*.Φ *?D* *?R* *hom-oper pert* (*local-bound* α) = Φ)
  $\langle$*proof*$\rangle$

Now, as a corollary, we prove that the previous definition of the output *D-R-h-C-f-g-δ-α-bound-phi*.f' of the *BPL*, is equivalent to the definition $f \circ \Phi$.

corollary *reduction-class-ext-preserves-output-f*:
assumes *r-c-e*: *reduction-class-ext* $(f::'a::diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist$
$=> 'b::diff\text{-}group\text{-}add)$ g
shows $f \circ$
local-nilpotent-alpha.Φ
(*diff-group-functor* $(\lambda x::'a::diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist. - x)$ *op* $-$ *0 op*
$+$ *diff*)
(*ring-hom-compl* (*diff-group-functor* $(\lambda x::'a. - x)$ *op* $-$ *0 op* $+$ *diff*))
hom-oper
pert
(*local-bound* α)
$= f \circ \Phi$
⟨*proof*⟩

Now, as a corollary, we prove that the previous definition of the output h' of the *BPL*, is equivalent to the definition $h' \equiv$ *hom-oper* $\circ$ Φ.

corollary *reduction-class-ext-preserves-output-h*:
assumes *r-c-e*:
reduction-class-ext $(f::'a::diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist$
$=> 'b::diff\text{-}group\text{-}add)$ g
shows *lemma-2-2-15*.h'
(*diff-group-functor* $(\lambda x::'a::diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist. - x)$ *op* $-$ *0 op*
$+$ *diff*)
(*ring-hom-compl* (*diff-group-functor* $(\lambda x::'a. - x)$ *op* $-$ *0 op* $+$ *diff*))
hom-oper
pert
(*local-bound* α)
$= h'$
⟨*proof*⟩

The definition of *reduction-class-ext* satisfies the definition of the locale *alpha-beta*

lemma *reduction-class-ext-preserves-alpha-beta*:
assumes *r-c-e*:
reduction-class-ext $(f::'a::diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist$
$=> 'b::diff\text{-}group\text{-}add)$ g
shows *alpha-beta* (*diff-group-functor* $(\lambda x::'a. - x)$ *op* $-$ *0 op* $+$ *diff*)
(*ring-hom-compl*
(*diff-group-functor* $(\lambda x::'a::diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist. - x)$ *op* $-$ *0 op*
$+$ *diff*))
(*diff-group-functor* $(\lambda x::'b::diff\text{-}group\text{-}add. - x)$ *op* $-$ *0 op* $+$ *diff*)
f
g $(\lambda x.\ h\ x)$ $(\lambda x.\ \delta\ x)$
⟨*proof*⟩

The new definition of the power series over $\beta = -$ (*hom-oper* $\circ$ *diff-group-add-pert-class.pert*) is equivalent to the definition of the power series over β in the previous version.

lemma *reduction-class-ext-preserves-beta-bound*:

assumes *r-c-e*:
reduction-class-ext $(f::'a::$*diff-group-add-pert-hom-bound-exist*
$=> 'b::$*diff-group-add*$)$ *g*
shows *local-bounded-func* $(\beta::'a::$*diff-group-add-pert-hom-bound-exist* $=> 'a)$
⟨*proof*⟩

lemma *reduction-class-ext-preserves-power-series-*β:
assumes *r-c-e*:
reduction-class-ext $(f::'a::$*diff-group-add-pert-hom-bound-exist*
$=> 'b::$*diff-group-add*$)$ *g*
shows *local-nilpotent-term.power-series*
(*diff-group-functor* $(\lambda x::'a::$*diff-group-add-pert-hom-bound-exist*$. - x)$ *op* $-$ *0 op*
$+$ *diff*)
(*ring-hom-compl* (*diff-group-functor* $(\lambda x::'a. - x)$ *op* $-$ *0 op* $+$ *diff*))
β (*local-bound* β)
$= (\lambda x::'a.$ *fin-sum* β (*local-bound* β *x*) *x*)
(**is** *local-nilpotent-term.power-series ?D ?R* β (*local-bound* β)
$= (\lambda x::'a.$ *fin-sum* β (*local-bound* β *x*) *x*))
⟨*proof*⟩

As well as the equivalence between both definitions of the power series, also the definitions of the bounds are equivalent.

lemma *reduction-class-ext-preserves-bound-psi*:
assumes *r-c-e*:
reduction-class-ext $(f::'a::$*diff-group-add-pert-hom-bound-exist*
$=> 'b::$*diff-group-add*$)$ *g*
shows *local-nilpotent-alpha.bound-psi*
(*diff-group-functor* $(\lambda x::'a::$*diff-group-add-pert-hom-bound-exist*$. - x)$ *op* $-$ *0 op*
$+$ *diff*)
(*ring-hom-compl* (*diff-group-functor* $(\lambda x::'a. - x)$ *op* $-$ *0 op* $+$ *diff*))
hom-oper pert
$=$ (*local-bound* β)
(**is** *local-nilpotent-alpha.bound-psi ?D ?R* $(\lambda x.\ h\ x)$ $(\lambda x.\ \delta\ x)$ $=$ (*local-bound* β))
⟨*proof*⟩

From the equivalence betwen the power series and the equality of the bounds, it follows the equivalence between the old and the new definition of *D-R-C-f-g-h-*δ*-*α*-bound-phi*$.\Psi$

lemma *reduction-class-ext-preserves-*Ψ:
assumes *r-c-e*:
reduction-class-ext $(f::'a::$*diff-group-add-pert-hom-bound-exist*
$=> 'b::$*diff-group-add*$)$ *g*
shows *local-nilpotent-alpha*$.\Psi$
(*diff-group-functor* $(\lambda x::'a::$*diff-group-add-pert-hom-bound-exist*$. - x)$ *op* $-$ *0 op*
$+$ *diff*)
(*ring-hom-compl* (*diff-group-functor* $(\lambda x::'a. - x)$ *op* $-$ *0 op* $+$ *diff*))
hom-oper pert
$= \Psi$
(**is** *local-nilpotent-alpha*$.\Psi$ *?D ?R* $(\lambda x.\ h\ x)$ $(\lambda x.\ \delta\ x) = \Psi)$
⟨*proof*⟩

Now, as a corollary, we prove the equivalence between the previous definition of the output g of the BPL, and the one in this new approach

corollary *reduction-class-ext-preserves-output-g*:
assumes *r-c-e*:
reduction-class-ext $(f::'a::$*diff-group-add-pert-hom-bound-exist*
$=> 'b::$*diff-group-add*$)$ g
shows *local-nilpotent-alpha*.Ψ
$($*diff-group-functor* $(\lambda x::'a::$*diff-group-add-pert-hom-bound-exist*$.\ -\ x)$ *op* $-$ *0* *op* $+$ *diff*$)$
$($*ring-hom-compl* $($*diff-group-functor* $(\lambda x::'a.\ -\ x)$ *op* $-$ *0* *op* $+$ *diff*$))$
hom-oper pert $\circ$ g
$= \Psi \circ g$
$\langle$*proof*$\rangle$

It also follows the equality of the previous definition of $d\ C'$ and the new definition, dC' *?f ?g* $=$ *diff-group-add-class*.*diff* $+$ (*?f* $\circ$ *diff-group-add-pert-class*.*pert* $\circ$ Ψ $\circ$ *?g*)

corollary *reduction-class-ext-preserves-output-dC*:
assumes *r-c-e*:
reduction-class-ext $(f::'a::$*diff-group-add-pert-hom-bound-exist*
$=> 'b::$*diff-group-add*$)$ g
shows $(\lambda x::'b.$
if $x \in$ *carrier* $($*diff-group-functor* $(\lambda x::'b::$*diff-group-add*$.\ -\ x)$ *op* $-$ *0* *op* $+$ *diff*$)$

then mult $($*diff-group-functor* $(\lambda x::'b::$*diff-group-add*$.\ -\ x)$ *op* $-$ *0* *op* $+$ *diff*$)$
$($*diff-group.diff* $($*diff-group-functor* $(\lambda x::'b::$*diff-group-add*$.\ -\ x)$ *op* $-$ *0* *op* $+$ *diff*$)$
$x)$
$((f \circ$ *pert* $\circ$
$($*local-nilpotent-alpha*.Ψ
$($*diff-group-functor* $(\lambda x::'a::$*diff-group-add-pert-hom-bound-exist*$.\ -\ x)$ *op* $-$ *0* *op* $+$ *diff*$)$
$($*ring-hom-compl* $($*diff-group-functor* $(\lambda x::'a.\ -\ x)$ *op* $-$ *0* *op* $+$ *diff*$))$
hom-oper pert$)$ $\circ$ $g)$ $x)$ *else one* $($*diff-group-functor* $(\lambda x::'b.\ -\ x)$ *op* $-$ *0* *op* $+$ *diff*$))$
$= dC'\ f\ g$
$\langle$*proof*$\rangle$

Now, from he previous equivalences, we are ready to give the proof of the *reduction* D' (|*carrier* $=$ *carrier* C, *mult* $=$ *op* $\otimes_C$, *one* $= \mathbf{1}_C$, *diff-group*.*diff* $=$ $\lambda x.$ *if* $x \in$ *carrier* C *then diff-group*.*diff* $C\ x \otimes_C (f \circ \delta \circ$ *D-R-C-f-g-h-*δ*-*α*-bound-phi*.Ψ $\circ\ g)\ x$ *else* $\mathbf{1}_C$|) $(f \circ$ *D-R-C-f-g-h-*δ*-*α*-bound-phi*.$\Phi)$ (*D-R-C-f-g-h-*δ*-*α*-bound-phi*.Ψ $\circ\ g)$ *D-R-h-C-f-g-*δ*-*α*-bound-phi*.h' with the new introduced definitions in terms of classes:

lemma assumes *reduction-class-ext-f-g*:
reduction-class-ext $(f::'a::$*diff-group-add-pert-hom-bound-exist*
$=> 'b::$*diff-group-add*$)$ g
shows *reduction*
$($*lemma-2-2-15*.D'

($diff\text{-}group\text{-}functor$ ($\lambda x::'a::diff\text{-}group\text{-}add\text{-}pert\text{-}hom\text{-}bound\text{-}exist.\ -\ x$) $op\ -\ 0\ op\ +\ diff$)
($ring\text{-}hom\text{-}compl$
($diff\text{-}group\text{-}functor$ ($\lambda x::'a.\ -\ x$) $op\ -\ 0\ op\ +\ diff$)) $pert$)
(|$carrier = carrier$ ($diff\text{-}group\text{-}functor$ ($\lambda x::'b::diff\text{-}group\text{-}add.\ -\ x$) $op\ -\ 0\ op\ +\ diff$),
$mult = mult$ ($diff\text{-}group\text{-}functor$ ($\lambda x::'b.\ -\ x$) $op\ -\ 0\ op\ +\ diff$),
$one = one$ ($diff\text{-}group\text{-}functor$ ($\lambda x::'b.\ -\ x$) $op\ -\ 0\ op\ +\ diff$),
$diff\text{-}group.diff = dC'\ f\ g$|)
($f'\ f$)
($g'\ g$)
(h')
⟨*proof*⟩

end

30 Monolithic strings (message strings) for code generation

theory *Code-Message*
imports *List*
begin

30.1 Datatype of messages

datatype *message-string* = *STR string*

lemmas [*code func del*] = *message-string.recs message-string.cases*

lemma [*code func*]: *size* (*s*::*message-string*) = *0*
⟨*proof*⟩

lemma [*code func*]: *message-string-size* (*s*::*message-string*) = *0*
⟨*proof*⟩

30.2 ML interface

⟨*ML*⟩

30.3 Code serialization

code-type *message-string*
(*SML string*)
(*OCaml string*)
(*Haskell String*)

⟨*ML*⟩

code-reserved *SML string*

code-reserved *OCaml string*

code-instance *message-string* :: *eq*
  (*Haskell* −)

code-const *op* = :: *message-string* ⇒ *message-string* ⇒ *bool*
  (*SML* !((- : *string*) = -))
  (*OCaml* !((- : *string*) = -))
  (*Haskell* **infixl** *4* ==)

end

31 Type of indices

theory *Code-Index*
imports *ATP-Linkup*
begin

Indices are isomorphic to HOL *nat* but mapped to target-language builtin integers

31.1 Datatype of indices

typedef *index* = *UNIV* :: *nat set*
  morphisms *nat-of-index index-of-nat* ⟨*proof*⟩

lemma *index-of-nat-nat-of-index* [*simp*]:
  index-of-nat (*nat-of-index k*) = *k*
  ⟨*proof*⟩

lemma *nat-of-index-index-of-nat* [*simp*]:
  nat-of-index (*index-of-nat n*) = *n*
  ⟨*proof*⟩

lemma *index*:
  $(\bigwedge n{::}index.\ PROP\ P\ n) \equiv (\bigwedge n{::}nat.\ PROP\ P\ (index\text{-}of\text{-}nat\ n))$
⟨*proof*⟩

lemma *index-case*:
  assumes $\bigwedge n.\ k = index\text{-}of\text{-}nat\ n \Longrightarrow P$
  shows *P*
  ⟨*proof*⟩

lemma *index-induct-raw*:
  assumes $\bigwedge n.\ P\ (index\text{-}of\text{-}nat\ n)$
  shows *P k*
⟨*proof*⟩

lemma *nat-of-index-inject* [*simp*]:
  nat-of-index k = *nat-of-index* l $\longleftrightarrow$ $k = l$
  ⟨*proof*⟩

lemma *index-of-nat-inject* [*simp*]:
  index-of-nat n = *index-of-nat* m $\longleftrightarrow$ $n = m$
  ⟨*proof*⟩

instantiation *index* :: *zero*
begin

definition [*simp*, *code func del*]:
  0 = *index-of-nat 0*

instance ⟨*proof*⟩

end

definition [*simp*]:
  Suc-index k = *index-of-nat* (*Suc* (*nat-of-index* k))

lemma *index-induct*: P *0* $\Longrightarrow$ ($\bigwedge k$. $P\ k \Longrightarrow P$ (*Suc-index* k)) $\Longrightarrow P\ k$
⟨*proof*⟩

lemma *Suc-not-Zero-index*: *Suc-index* $k \neq$ *0*
  ⟨*proof*⟩

lemma *Zero-not-Suc-index*: *0* $\neq$ *Suc-index* k
  ⟨*proof*⟩

lemma *Suc-Suc-index-eq*: *Suc-index* k = *Suc-index* l $\longleftrightarrow$ $k = l$
  ⟨*proof*⟩

rep-datatype *index*
  distinct *Suc-not-Zero-index Zero-not-Suc-index*
  inject *Suc-Suc-index-eq*
  induction *index-induct*

lemmas [*code func del*] = *index.recs index.cases*

declare *index-case* [*case-names nat*, *cases type*: *index*]
declare *index-induct* [*case-names nat*, *induct type*: *index*]

lemma [*code func*]:
  index-size = *nat-of-index*
⟨*proof*⟩

lemma [*code func*]:
  size = *nat-of-index*

⟨*proof*⟩

lemma [*code func*]:
$k = l \longleftrightarrow$ *nat-of-index* k = *nat-of-index* l
⟨*proof*⟩

31.2 Indices as datatype of ints

instantiation *index* :: *number*
begin

definition
number-of = *index-of-nat o nat*

instance ⟨*proof*⟩

end

lemma *nat-of-index-number* [*simp*]:
nat-of-index (*number-of* k) = *number-of* k
⟨*proof*⟩

code-datatype *number-of* :: *int* ⇒ *index*

31.3 Basic arithmetic

instantiation *index* :: {*minus*, *ordered-semidom*, *Divides.div*, *linorder*}
begin

lemma *zero-index-code* [*code inline*, *code func*]:
(*0*::*index*) = *Numeral0*
⟨*proof*⟩
lemma [*code post*]: *Numeral0* = (*0*::*index*)
⟨*proof*⟩

definition [*simp*, *code func del*]:
(*1*::*index*) = *index-of-nat 1*

lemma *one-index-code* [*code inline*, *code func*]:
(*1*::*index*) = *Numeral1*
⟨*proof*⟩
lemma [*code post*]: *Numeral1* = (*1*::*index*)
⟨*proof*⟩

definition [*simp*, *code func del*]:
$n + m$ = *index-of-nat* (*nat-of-index* n + *nat-of-index* m)

lemma *plus-index-code* [*code func*]:
index-of-nat n + *index-of-nat* m = *index-of-nat* $(n + m)$
⟨*proof*⟩

definition [*simp*, *code func del*]:
$n - m = \textit{index-of-nat}\ (\textit{nat-of-index}\ n - \textit{nat-of-index}\ m)$

definition [*simp*, *code func del*]:
$n * m = \textit{index-of-nat}\ (\textit{nat-of-index}\ n * \textit{nat-of-index}\ m)$

lemma *times-index-code* [*code func*]:
$\textit{index-of-nat}\ n * \textit{index-of-nat}\ m = \textit{index-of-nat}\ (n * m)$
⟨*proof*⟩

definition [*simp*, *code func del*]:
$n\ \textit{div}\ m = \textit{index-of-nat}\ (\textit{nat-of-index}\ n\ \textit{div}\ \textit{nat-of-index}\ m)$

definition [*simp*, *code func del*]:
$n\ \textit{mod}\ m = \textit{index-of-nat}\ (\textit{nat-of-index}\ n\ \textit{mod}\ \textit{nat-of-index}\ m)$

lemma *div-index-code* [*code func*]:
$\textit{index-of-nat}\ n\ \textit{div}\ \textit{index-of-nat}\ m = \textit{index-of-nat}\ (n\ \textit{div}\ m)$
⟨*proof*⟩

lemma *mod-index-code* [*code func*]:
$\textit{index-of-nat}\ n\ \textit{mod}\ \textit{index-of-nat}\ m = \textit{index-of-nat}\ (n\ \textit{mod}\ m)$
⟨*proof*⟩

definition [*simp*, *code func del*]:
$n \leq m \longleftrightarrow \textit{nat-of-index}\ n \leq \textit{nat-of-index}\ m$

definition [*simp*, *code func del*]:
$n < m \longleftrightarrow \textit{nat-of-index}\ n < \textit{nat-of-index}\ m$

lemma *less-eq-index-code* [*code func*]:
$\textit{index-of-nat}\ n \leq \textit{index-of-nat}\ m \longleftrightarrow n \leq m$
⟨*proof*⟩

lemma *less-index-code* [*code func*]:
$\textit{index-of-nat}\ n < \textit{index-of-nat}\ m \longleftrightarrow n < m$
⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *Suc-index-minus-one*: $\textit{Suc-index}\ n - \mathit{1} = n$ ⟨*proof*⟩

lemma *index-of-nat-code* [*code*]:
$\textit{index-of-nat} = \textit{of-nat}$
⟨*proof*⟩

lemma *index-not-eq-zero*: $i \neq$ *index-of-nat* $0 \longleftrightarrow i \geq 1$
⟨*proof*⟩

definition
nat-of-index-aux :: *index* ⇒ *nat* ⇒ *nat*
where
nat-of-index-aux i n = *nat-of-index i* + *n*

lemma *nat-of-index-aux-code* [*code*]:
nat-of-index-aux i n = (*if i* = *0 then n else nat-of-index-aux* (*i* − *1*) (*Suc n*))
⟨*proof*⟩

lemma *nat-of-index-code* [*code*]:
nat-of-index i = *nat-of-index-aux i 0*
⟨*proof*⟩

31.4 ML interface

⟨*ML*⟩

31.5 Specialized *op* −, *op div* and *op mod* operations

definition
minus-index-aux :: *index* ⇒ *index* ⇒ *index*
where
[*code func del*]: *minus-index-aux* = *op* −

lemma [*code func*]: *op* − = *minus-index-aux*
⟨*proof*⟩

definition
div-mod-index :: *index* ⇒ *index* ⇒ *index* × *index*
where
[*code func del*]: *div-mod-index n m* = (*n div m*, *n mod m*)

lemma [*code func*]:
div-mod-index n m = (*if m* = *0 then* (*0*, *n*) *else* (*n div m*, *n mod m*))
⟨*proof*⟩

lemma [*code func*]:
n div m = *fst* (*div-mod-index n m*)
⟨*proof*⟩

lemma [*code func*]:
n mod m = *snd* (*div-mod-index n m*)
⟨*proof*⟩

31.6 Code serialization

Implementation of indices by bounded integers

code-type *index*
  (*SML int*)
  (*OCaml int*)
  (*Haskell Int*)

code-instance *index* :: *eq*
  (*Haskell* −)

⟨*ML*⟩

code-reserved *SML Int int*
code-reserved *OCaml Pervasives int*

code-const *op* + :: *index* ⇒ *index* ⇒ *index*
  (*SML Int*.+/ ((-),/ (-)))
  (*OCaml Pervasives*.(+))
  (*Haskell* **infixl** *6* +)

code-const *minus-index-aux* :: *index* ⇒ *index* ⇒ *index*
  (*SML Int*.*max*/ (-/ −/ -,/ *0* : *int*))
  (*OCaml Pervasives*.*max*/ (-/ −/ -)/ (*0* : *int*))
  (*Haskell max*/ (-/ −/ -)/ (*0* :: *Int*))

code-const *op* ∗ :: *index* ⇒ *index* ⇒ *index*
  (*SML Int*.∗/ ((-),/ (-)))
  (*OCaml Pervasives*.(∗))
  (*Haskell* **infixl** *7* ∗)

code-const *div-mod-index*
  (*SML* (*fn n* => *fn m* =>/ (*n div m*, *n mod m*)))
  (*OCaml* (*fun n* −> *fun m* −>/ (*n* ′/ *m*, *n mod m*)))
  (*Haskell divMod*)

code-const *op* = :: *index* ⇒ *index* ⇒ *bool*
  (*SML* !((- : *Int*.*int*) = -))
  (*OCaml* !((- : *int*) = -))
  (*Haskell* **infixl** *4* ==)

code-const *op* ≤ :: *index* ⇒ *index* ⇒ *bool*
  (*SML Int*.<=/ ((-),/ (-)))
  (*OCaml* !((- : *int*) <= -))
  (*Haskell* **infix** *4* <=)

code-const *op* < :: *index* ⇒ *index* ⇒ *bool*
  (*SML Int*.</ ((-),/ (-)))
  (*OCaml* !((- : *int*) < -))
  (*Haskell* **infix** *4* <)

end

32 Reflecting Pure types into HOL

theory *RType*
imports *Main Code-Message Code-Index*
begin

datatype *rtype* = *RType message-string rtype list*

class *rtype* =
  fixes *rtype* :: $'a$::{} *itself* ⇒ *rtype*
begin

definition
  rtype-of :: $'a$ ⇒ *rtype*
where
  [*simp*]: *rtype-of x* = *rtype TYPE*($'a$)

end

⟨*ML*⟩

lemma [*code func*]:
  RType tyco1 tys1 = *RType tyco2 tys2* ⟷ *tyco1* = *tyco2*
    ∧ *list-all2* (*op* =) *tys1 tys2*
  ⟨*proof*⟩

code-type *rtype*
  (*SML Term.typ*)

code-const *RType*
  (*SML Term.Type/* (-, -))

code-reserved *SML Term*

hide (**open**) *const rtype RType*

end

33 A simple term evaluation mechanism

theory *Eval*
imports
  RType
  Code-Index
begin

33.1 Term representation

33.1.1 Terms and class *term-of*

datatype *term* = *dummy-term*

definition
Const :: *message-string* ⇒ *rtype* ⇒ *term*
where
Const - - = *dummy-term*

definition
App :: *term* ⇒ *term* ⇒ *term*
where
App - - = *dummy-term*

code-datatype *Const App*

class *term-of* = *rtype* +
fixes *term-of* :: *'a* ⇒ *term*

lemma *term-of-anything*: *term-of x* ≡ *t*
⟨*proof*⟩

⟨*ML*⟩

33.1.2 *term-of* instances

⟨*ML*⟩

33.1.3 Code generator setup

lemmas [*code func del*] = *term.recs term.cases term.size*
lemma [*code func*, *code func del*]: (*t1*::*term*) = *t2* ⟷ *t1* = *t2* ⟨*proof*⟩

lemma [*code func*, *code func del*]: (*term-of* :: *rtype* ⇒ *term*) = *term-of* ⟨*proof*⟩
lemma [*code func*, *code func del*]: (*term-of* :: *term* ⇒ *term*) = *term-of* ⟨*proof*⟩
lemma [*code func*, *code func del*]: (*term-of* :: *index* ⇒ *term*) = *term-of* ⟨*proof*⟩
lemma [*code func*, *code func del*]: (*term-of* :: *message-string* ⇒ *term*) = *term-of*
⟨*proof*⟩

code-type *term*
(*SML Term.term*)

code-const *Const* **and** *App*
(*SML Term.Const*/ (-, -) **and** *Term.$*/ (-, -))

code-const *term-of* :: *index* ⇒ *term*
(*SML HOLogic.mk'-number*/ *HOLogic.indexT*)

code-const *term-of* :: *message-string* ⇒ *term*
(*SML Message′-String.mk*)

33.1.4 Syntax

⟨*ML*⟩

notation (**output**)
rterm-of («-»)

locale (**open**) *rterm-syntax* =
fixes *rterm-of-syntax* :: ′*a* ⇒ ′*b* («-»)

⟨*ML*⟩

hide *const dummy-term*
hide (**open**) *const Const App*
hide (**open**) *const term-of*

33.2 Evaluation setup

⟨*ML*⟩

end

34 Pretty integer literals for code generation

theory *Code-Integer*
imports *ATP-Linkup*
begin

HOL numeral expressions are mapped to integer literals in target languages, using predefined target language operations for abstract integer operations.

code-type *int*
(*SML IntInf.int*)
(*OCaml Big′-int.big′-int*)
(*Haskell Integer*)

code-instance *int* :: *eq*
(*Haskell* −)

⟨*ML*⟩

code-const *Int.Pls* **and** *Int.Min* **and** *Int.Bit0* **and** *Int.Bit1*
(*SML raise/ Fail/ Pls*
and *raise/ Fail/ Min*
and !((-);/ *raise/ Fail/ Bit0*)
and !((-);/ *raise/ Fail/ Bit1*))

(*OCaml failwith/ Pls*
  and *failwith/ Min*
  and !((-);/ *failwith/ Bit0*)
  and !((-);/ *failwith/ Bit1*))
(*Haskell error/ Pls*
  and *error/ Min*
  and *error/ Bit0*
  and *error/ Bit1*)

code-const *Int.pred*
  (*SML IntInf*.− ((-), *1*))
  (*OCaml Big′-int.pred′-big′-int*)
  (*Haskell* !(-/ −/ *1*))

code-const *Int.succ*
  (*SML IntInf*.+ ((-), *1*))
  (*OCaml Big′-int.succ′-big′-int*)
  (*Haskell* !(-/ +/ *1*))

code-const *op* + :: *int* ⇒ *int* ⇒ *int*
  (*SML IntInf*.+ ((-), (-)))
  (*OCaml Big′-int.add′-big′-int*)
  (*Haskell* **infixl** *6* +)

code-const *uminus* :: *int* ⇒ *int*
  (*SML IntInf*.~)
  (*OCaml Big′-int.minus′-big′-int*)
  (*Haskell negate*)

code-const *op* − :: *int* ⇒ *int* ⇒ *int*
  (*SML IntInf*.− ((-), (-)))
  (*OCaml Big′-int.sub′-big′-int*)
  (*Haskell* **infixl** *6* −)

code-const *op* ∗ :: *int* ⇒ *int* ⇒ *int*
  (*SML IntInf*.∗ ((-), (-)))
  (*OCaml Big′-int.mult′-big′-int*)
  (*Haskell* **infixl** *7* ∗)

code-const *op* = :: *int* ⇒ *int* ⇒ *bool*
  (*SML* !((- : *IntInf.int*) = -))
  (*OCaml Big′-int.eq′-big′-int*)
  (*Haskell* **infixl** *4* ==)

code-const *op* ≤ :: *int* ⇒ *int* ⇒ *bool*
  (*SML IntInf*.<= ((-), (-)))
  (*OCaml Big′-int.le′-big′-int*)
  (*Haskell* **infix** *4* <=)

code-const *op* < :: *int* ⇒ *int* ⇒ *bool*
(*SML IntInf*.< ((-), (-)))
(*OCaml Big'-int.lt'-big'-int*)
(*Haskell* **infix** *4* <)

code-reserved *SML IntInf*
code-reserved *OCaml Big-int*

end

35 Implementation of natural numbers by target-language integers

theory *Efficient-Nat*
imports *Code-Integer Code-Index*
begin

When generating code for functions on natural numbers, the canonical representation using *0* and *Suc* is unsuitable for computations involving large numbers. The efficiency of the generated code can be improved drastically by implementing natural numbers by target-language integers. To do this, just include this theory.

35.1 Basic arithmetic

Most standard arithmetic functions on natural numbers are implemented using their counterparts on the integers:

code-datatype *number-nat-inst.number-of-nat*

lemma *zero-nat-code* [*code*, *code unfold*]:
0 = (*Numeral0* :: *nat*)
⟨*proof*⟩
lemmas [*code post*] = *zero-nat-code* [*symmetric*]

lemma *one-nat-code* [*code*, *code unfold*]:
1 = (*Numeral1* :: *nat*)
⟨*proof*⟩
lemmas [*code post*] = *one-nat-code* [*symmetric*]

lemma *Suc-code* [*code*]:
Suc n = *n* + *1*
⟨*proof*⟩

lemma *plus-nat-code* [*code*]:
n + *m* = *nat* (*of-nat n* + *of-nat m*)
⟨*proof*⟩

lemma *minus-nat-code* [*code*]:
$n - m = nat\ (\textit{of-nat}\ n - \textit{of-nat}\ m)$
⟨*proof*⟩

lemma *times-nat-code* [*code*]:
$n * m = nat\ (\textit{of-nat}\ n * \textit{of-nat}\ m)$
⟨*proof*⟩

Specialized *op div* and *op mod* operations.

definition
$\textit{divmod-aux} :: nat \Rightarrow nat \Rightarrow nat \times nat$
where
[*code func del*]: $\textit{divmod-aux} = divmod$

lemma [*code func*]:
$divmod\ n\ m = (\textit{if}\ m = 0\ \textit{then}\ (0,\ n)\ \textit{else}\ \textit{divmod-aux}\ n\ m)$
⟨*proof*⟩

lemma *divmod-aux-code* [*code*]:
$\textit{divmod-aux}\ n\ m = (nat\ (\textit{of-nat}\ n\ div\ \textit{of-nat}\ m),\ nat\ (\textit{of-nat}\ n\ mod\ \textit{of-nat}\ m))$
⟨*proof*⟩

lemma *eq-nat-code* [*code*]:
$n = m \longleftrightarrow (\textit{of-nat}\ n :: int) = \textit{of-nat}\ m$
⟨*proof*⟩

lemma *less-eq-nat-code* [*code*]:
$n \leq m \longleftrightarrow (\textit{of-nat}\ n :: int) \leq \textit{of-nat}\ m$
⟨*proof*⟩

lemma *less-nat-code* [*code*]:
$n < m \longleftrightarrow (\textit{of-nat}\ n :: int) < \textit{of-nat}\ m$
⟨*proof*⟩

35.2 Case analysis

Case analysis on natural numbers is rephrased using a conditional expression:

lemma [*code func*, *code unfold*]:
$\textit{nat-case} = (\lambda f\ g\ n.\ \textit{if}\ n = 0\ \textit{then}\ f\ \textit{else}\ g\ (n - 1))$
⟨*proof*⟩

35.3 Preprocessors

In contrast to *Suc n*, the term $n + 1$ is no longer a constructor term. Therefore, all occurrences of this term in a position where a pattern is expected (i.e. on the left-hand side of a recursion equation or in the arguments of an inductive relation in an introduction rule) must be eliminated. This can be accomplished by applying the following transformation rules:

lemma *Suc-if-eq*: $(\bigwedge n.\ f\ (Suc\ n) = h\ n) \Longrightarrow f\ 0 = g \Longrightarrow$
  $f\ n = (if\ n = 0\ then\ g\ else\ h\ (n - 1))$
  ⟨*proof*⟩

lemma *Suc-clause*: $(\bigwedge n.\ P\ n\ (Suc\ n)) \Longrightarrow n \neq 0 \Longrightarrow P\ (n - 1)\ n$
  ⟨*proof*⟩

The rules above are built into a preprocessor that is plugged into the code generator. Since the preprocessor for introduction rules does not know anything about modes, some of the modes that worked for the canonical representation of natural numbers may no longer work.

⟨*ML*⟩

35.4 Target language setup

For ML, we map *nat* to target language integers, where we assert that values are always non-negative.

code-type *nat*
  (*SML int*)
  (*OCaml Big′-int.big′-int*)

types-code
  nat (*int*)
attach (*term-of*) ⟪
val term-of-nat = HOLogic.mk-number HOLogic.natT;
⟫
attach (*test*) ⟪
fun gen-nat i =
  let val n = random-range 0 i
  in (n, fn () => term-of-nat n) end;
⟫

For Haskell we define our own *nat* type. The reason is that we have to distinguish type class instances for *nat* and *int*.

code-include *Haskell Nat* ⟪
newtype Nat = Nat Integer deriving (Show, Eq);

instance Num Nat where {
  fromInteger k = Nat (if k >= 0 then k else 0);
  Nat n + Nat m = Nat (n + m);
  Nat n − Nat m = fromInteger (n − m);
  Nat n ∗ Nat m = Nat (n ∗ m);
  abs n = n;
  signum - = 1;
  negate n = error negate Nat;

};

instance Ord Nat where {
  Nat n <= *Nat m* = *n* <= *m*;
  Nat n < *Nat m* = *n* < *m*;
};

instance Real Nat where {
  toRational (*Nat n*) = *toRational n*;
};

instance Enum Nat where {
  toEnum k = *fromInteger* (*toEnum k*);
  fromEnum (*Nat n*) = *fromEnum n*;
};

instance Integral Nat where {
  toInteger (*Nat n*) = *n*;
  divMod n m = *quotRem n m*;
  quotRem (*Nat n*) (*Nat m*) = (*Nat k*, *Nat l*) *where* (*k*, *l*) = *quotRem n m*;
};
⟫

code-reserved *Haskell Nat*

code-type *nat*
  (*Haskell Nat*)

code-instance *nat* :: *eq*
  (*Haskell* −)

Natural numerals.

lemma [*code inline*, *symmetric*, *code post*]:
  nat (*number-of i*) = *number-nat-inst.number-of-nat i*
  — this interacts as desired with *number-of ?v* = *nat* (*number-of ?v*)
  ⟨*proof*⟩

⟨*ML*⟩

Since natural numbers are implemented using integers in ML, the coercion function *of-nat* of type *nat* ⇒ *int* is simply implemented by the identity function. For the *nat* function for converting an integer to a natural number, we give a specific implementation using an ML function that returns its input value, provided that it is non-negative, and otherwise returns *0*.

definition
  int :: *nat* ⇒ *int*
where
  [*code func del*]: *int* = *of-nat*

lemma *int-code′* [*code func*]:
  int (*number-of l*) = (*if neg* (*number-of l* :: *int*) *then 0 else number-of l*)
  ⟨*proof*⟩

lemma *nat-code′* [*code func*]:
  nat (*number-of l*) = (*if neg* (*number-of l* :: *int*) *then 0 else number-of l*)
  ⟨*proof*⟩

lemma *of-nat-int* [*code unfold*]:
  of-nat = *int* ⟨*proof*⟩
declare *of-nat-int* [*symmetric*, *code post*]

code-const *int*
  (*SML* -)
  (*OCaml* -)

consts-code
  int ((-))
  nat (⟨**module**⟩*nat*)
attach ⟪
fun nat i = *if i* < *0 then 0 else i*;
⟫

code-const *nat*
  (*SML IntInf.max/* (*/0*,*/* -))
  (*OCaml Big′-int.max′-big′-int/ Big′-int.zero′-big′-int*)

For Haskell, things are slightly different again.

code-const *int* **and** *nat*
  (*Haskell toInteger* **and** *fromInteger*)

Conversion from and to indices.

code-const *index-of-nat*
  (*SML IntInf.toInt*)
  (*OCaml Big′-int.int′-of′-big′-int*)
  (*Haskell toEnum*)

code-const *nat-of-index*
  (*SML IntInf.fromInt*)
  (*OCaml Big′-int.big′-int′-of′-int*)
  (*Haskell fromEnum*)

Using target language arithmetic operations whenever appropriate

code-const *op* + :: *nat* ⇒ *nat* ⇒ *nat*
  (*SML IntInf.*+ ((-), (-)))
  (*OCaml Big′-int.add′-big′-int*)
  (*Haskell* **infixl** *6* +)

code-const *op* ∗ :: *nat* ⇒ *nat* ⇒ *nat*

(*SML IntInf*.∗ ((-), (-)))
(*OCaml Big'-int.mult'-big'-int*)
(*Haskell* **infixl** *7* ∗)

code-const *divmod-aux*
(*SML IntInf*.*divMod*/ ((-),/ (-)))
(*OCaml Big'-int.quomod'-big'-int*)
(*Haskell divMod*)

code-const *op* = :: *nat* ⇒ *nat* ⇒ *bool*
(*SML* !((- : *IntInf*.*int*) = -))
(*OCaml Big'-int.eq'-big'-int*)
(*Haskell* **infixl** *4* ==)

code-const *op* ≤ :: *nat* ⇒ *nat* ⇒ *bool*
(*SML IntInf*.<= ((-), (-)))
(*OCaml Big'-int.le'-big'-int*)
(*Haskell* **infix** *4* <=)

code-const *op* < :: *nat* ⇒ *nat* ⇒ *bool*
(*SML IntInf*.< ((-), (-)))
(*OCaml Big'-int.lt'-big'-int*)
(*Haskell* **infix** *4* <)

consts-code
0 (*0*)
Suc ((- +/ *1*))
op + :: *nat* ⇒ *nat* ⇒ *nat* ((- +/ -))
op ∗ :: *nat* ⇒ *nat* ⇒ *nat* ((- ∗/ -))
op ≤ :: *nat* ⇒ *nat* ⇒ *bool* ((- <=/ -))
op < :: *nat* ⇒ *nat* ⇒ *bool* ((- </ -))

Module names

code-modulename *SML*
Nat Integer
Divides Integer
Efficient-Nat Integer

code-modulename *OCaml*
Nat Integer
Divides Integer
Efficient-Nat Integer

code-modulename *Haskell*
Nat Integer
Divides Integer
Efficient-Nat Integer

hide *const int*

end

theory *example-Bicomplex*
imports
~~/src/HOL/Real/Float
Matrix/SparseMatrix
Matrix/cplex/MatrixLP
BPL-classes-2008
~~/src/HOL/Library/Eval
~~/src/HOL/Library/Efficient-Nat
begin

36 An example of the BPL based on bicomplexes.

We fit a concrete example of the BPL into the code previously generated

The example is as follows: we have a "big" differential group D which will be represented as matrices of any dimension of integers, with usual addition of matrices (two matrices of different dimension can be added obtaining as result a matrix with the greatest number of rows and the greatest number of columns between the two matrices being added), and where the null matrix is any matrix containing exclusively zeros.

The minus operation is the result of applying minus to every component of a matrix. Such structure can be proved to be an abelian group.

Then, the differential d_D is defined over the matrices by considering various particular cases (by means of an *if* structure). The homology operator h and the perturbation δ_D are also defined over matrices.

The nilpotency condition is locally satisfied for the number of columns of a matrix plus one.

The small differential group C is null.

Then, f and g are also null.

lemma $- (- x) = (x::'a::lordered\text{-}ring)$
⟨*proof*⟩

lemma assumes *x-n-0*: $x \neq 0$
shows $(- x) \neq (0::'a::lordered\text{-}ring)$
⟨*proof*⟩

lemma assumes *x-n-0*: $(- x) \neq 0$
shows $x \neq (0::'a::lordered\text{-}ring)$

⟨*proof*⟩

lemma *nonzero-positions-minus-f*:
fixes *f*:: *'a::lordered-ring matrix => 'b::lordered-ring matrix*
shows *nonzero-positions* (*Rep-matrix* (*f A*))
= *nonzero-positions* (*Rep-matrix* ((− *f*) *A*))
⟨*proof*⟩

36.1 Definition of the differential.

definition *diff-infmatrix* :: *'a::{zero} infmatrix => 'a infmatrix*
where *diff-infmatrix-def* :
diff-infmatrix A = (%*i*::*nat*. %*j*::*nat*. *if* ((*i* + *j*) *mod 2* = *1*) *then 0*
else (*A i* (*j* + *1*)))

definition *diff-inv* :: (*nat* × *nat*) => (*nat* × *nat*)
where *diff-inv-def* : *diff-inv pos* = ((*fst pos*, (*snd pos*) + *1*))

lemma *diff-infmatrix-finite*[*simp*]:
shows *finite* (*nonzero-positions* (*diff-infmatrix* (*Rep-matrix x*)))
⟨*proof*⟩

lemma *diff-infmatrix-matrix*:
shows (*diff-infmatrix* (*Rep-matrix* (*x*::*'a*::*zero matrix*))) ∈ *matrix*
⟨*proof*⟩

lemma *diff-infmatrix-closed*[*simp*]:
Rep-matrix (*Abs-matrix* (*diff-infmatrix* (*Rep-matrix x*)))
= *diff-infmatrix* (*Rep-matrix x*)
⟨*proof*⟩

lemma *diff-infmatrix-is-matrix*: **assumes** *as*: *A* ∈ *matrix*
shows *diff-infmatrix A* ∈ *matrix*
⟨*proof*⟩

lemma *diff-infmatrix-twice* [*simp*]:
shows *diff-infmatrix* (*diff-infmatrix A*) = (*Rep-matrix 0*)
⟨*proof*⟩

The name *diff-matrix* is already used to express the difference between two matrices

definition *diff-matrix1* :: *'a::lordered-ring matrix => 'a matrix*
where *diff-matrix1-def*: *diff-matrix1* = *Abs-matrix* ∘ *diff-infmatrix* ∘ *Rep-matrix*

lemma *combine-infmatrix-matrix*:
shows *f 0 0* = *0* ⟹ *combine-infmatrix f* (*Rep-matrix A*) (*Rep-matrix B*) ∈ *matrix*
⟨*proof*⟩

lemma *combine-infmatrix-diff-infmatrix-matrix*[*simp*]:
f 0 0 = 0 $\Longrightarrow$ *combine-infmatrix f*
(*diff-infmatrix* (*Rep-matrix A*)) (*diff-infmatrix* (*Rep-matrix B*)) $\in$ *matrix*
⟨*proof*⟩

lemma *diff-infmatrix-combine-infmatrix-matrix*[*simp*]:
f 0 0 = 0 $\Longrightarrow$ *diff-infmatrix* (*combine-infmatrix f* (*Rep-matrix A*)
(*Rep-matrix B*)) $\in$ *matrix*
⟨*proof*⟩

36.2 Matrices as an instance of differential group.

lemma *Abs-matrix-inject2*:
assumes *x*: *x* $\in$ *matrix* **and** *y*: *y* $\in$ *matrix* **and** *eq*: *x* = *y*
shows *Abs-matrix x* = *Abs-matrix y*
⟨*proof*⟩

lemma *diff-matrix1-hom*:
shows *diff-matrix1* (*A* + *B*) = *diff-matrix1 A* + *diff-matrix1 B*
⟨*proof*⟩

lemma *diff-matrix1-twice*:
shows *diff-matrix1* (*diff-matrix1 A*) = (*0*::(*'a*::*lordered-ring matrix*))
⟨*proof*⟩

instantiation *matrix* :: (*lordered-ring*) *diff-group-add*
begin

definition *diff-matrix-def*: *diff A* == *diff-matrix1 A*

instance
⟨*proof*⟩

end

36.3 Definition of the perturbation δ_D.

definition *pert-infmatrix* :: *'a*::{*zero*} *infmatrix* => *'a infmatrix*
where *pert-infmatrix-def* : *pert-infmatrix A* =
(%*i*::*nat*. %*j*::*nat*. *if* ((*i* + *j*) *mod 2* = *1*) *then 0*
else (*A* (*i* + *1*) *j*))

definition *pert-inv* :: (*nat* × *nat*) => (*nat* × *nat*)
where *pert-inv-def* : *pert-inv pos* = ((*fst pos*) + *1*, *snd pos*)

lemma *pert-infmatrix-finite*[*simp*]:
shows *finite* (*nonzero-positions* (*pert-infmatrix* (*Rep-matrix x*)))
⟨*proof*⟩

lemma *pert-infmatrix-matrix*:

shows (*pert-infmatrix* (*Rep-matrix* (*x*::'*a*::*zero matrix*))) ∈ *matrix*
⟨*proof*⟩

lemma *pert-infmatrix-closed*[*simp*]:
Rep-matrix (*Abs-matrix* (*pert-infmatrix* (*Rep-matrix x*)))
= *pert-infmatrix* (*Rep-matrix x*)
⟨*proof*⟩

lemma *pert-infmatrix-is-matrix*: **assumes** *as*: *A* ∈ *matrix*
shows *pert-infmatrix A* ∈ *matrix*
⟨*proof*⟩

lemma *pert-infmatrix-twice* [*simp*]:
shows *pert-infmatrix* (*pert-infmatrix A*) = (*Rep-matrix 0*)
⟨*proof*⟩

We will use the name *pert-matrix1* to define the perturbation in this context, leaving thus the name *pert-matrix* to be used inside of the instance *matrix*::*diff-group-add-pert*.

definition *pert-matrix1* :: '*a*::*lordered-ring matrix* => '*a matrix*
where *pert-matrix1-def*: *pert-matrix1* == *Abs-matrix* ∘ *pert-infmatrix* ∘ *Rep-matrix*

lemma *combine-infmatrix-pert-infmatrix-matrix*[*simp*]:
f 0 0 = *0* ⟹ *combine-infmatrix f* (*pert-infmatrix* (*Rep-matrix A*))
(*pert-infmatrix* (*Rep-matrix B*)) ∈ *matrix*
⟨*proof*⟩

lemma *pert-infmatrix-combine-infmatrix-matrix*[*simp*]:
f 0 0 = *0* ⟹ *pert-infmatrix* (*combine-infmatrix f* (*Rep-matrix A*)
(*Rep-matrix B*)) ∈ *matrix*
⟨*proof*⟩

lemma *combine-diff-pert-matrix*[*simp*]:
f 0 0 = *0* ⟹ (*combine-infmatrix f* (*diff-infmatrix*
(*combine-infmatrix f* (*Rep-matrix A*) (*Rep-matrix B*)))
(*pert-infmatrix* (*combine-infmatrix f* (*Rep-matrix A*) (*Rep-matrix B*)))) ∈ *matrix*
⟨*proof*⟩

lemma *combine-diff-pert-matrix-1* [*simp*]:
f 0 0 = *0* ⟹ (*combine-infmatrix f* (*diff-infmatrix* (*Rep-matrix A*))
(*pert-infmatrix* (*Rep-matrix A*))) ∈ *matrix*
⟨*proof*⟩

lemma *combine-diff-pert-matrix-2* [*simp*]:
f 0 0 = *0* ⟹ (*combine-infmatrix f*
(*combine-infmatrix f* (*diff-infmatrix* (*Rep-matrix A*))
(*pert-infmatrix* (*Rep-matrix A*)))
(*combine-infmatrix f* (*diff-infmatrix* (*Rep-matrix B*)) (*pert-infmatrix* (*Rep-matrix B*)))) ∈ *matrix*

⟨*proof*⟩

lemma *combine-diff-pert-matrix-3* [*simp*]:
f 0 0 = *0* ⟹ (*diff-infmatrix* (*combine-infmatrix f* (*diff-infmatrix* (*Rep-matrix* *A*))
(*pert-infmatrix* (*Rep-matrix A*)))) ∈ *matrix*
⟨*proof*⟩

lemma *combine-diff-pert-matrix-4* [*simp*]:
f 0 0 = *0* ⟹ (*pert-infmatrix* (*combine-infmatrix f*
(*diff-infmatrix* (*Rep-matrix A*))
(*pert-infmatrix* (*Rep-matrix A*)))) ∈ *matrix*
⟨*proof*⟩

lemma *combine-diff-pert-matrix-5* [*simp*]:
f 0 0 = *0* ⟹ (*combine-infmatrix f* (*diff-infmatrix* (*combine-infmatrix f*
(*diff-infmatrix* (*Rep-matrix A*))
(*pert-infmatrix* (*Rep-matrix A*))))
(*pert-infmatrix* (*combine-infmatrix*
f (*diff-infmatrix* (*Rep-matrix A*)) (*pert-infmatrix* (*Rep-matrix A*))))) ∈ *matrix*
⟨*proof*⟩

36.4 Matrices as an instance of differential group with a perturbation.

lemma *pert-matrix1-hom*: **shows** *pert-matrix1* (*A* + *B*) = *pert-matrix1 A* + *pert-matrix1 B*
⟨*proof*⟩

lemma *pert-matrix1-twice*:
shows *pert-matrix1* (*pert-matrix1 A*) = (*0*::(*'a*::*lordered-ring matrix*))
⟨*proof*⟩

lemma *diff-matrix1-pert-matrix1-is-zero*: *diff-matrix1* (*pert-matrix1 A*) = *0*
⟨*proof*⟩

lemma *pert-matrix1-diff-matrix1-is-zero*: *pert-matrix1* (*diff-matrix1 A*) = *0*
⟨*proof*⟩

instantiation *matrix* :: (*lordered-ring*) *diff-group-add-pert*
begin

definition *pert-matrix-def*: *pert A* == *pert-matrix1 A*

instance
⟨*proof*⟩

end

36.5 Definition of the homotopy operator h.

definition *hom-oper-infmatrix* :: *'a*::{*zero*} *infmatrix* => *'a infmatrix*
where *hom-oper-infmatrix-def* : *hom-oper-infmatrix A* =
(%*i*::*nat*. %*j*::*nat*. *if* (*j* = *0*) *then 0 else* (*if* ((*i* + *j*) *mod* (*2*::*nat*) = *0*) *then* (*0*::*'a*)
else (*A i* (*j* − *1*))))

definition *hom-oper-inv* :: (*nat* × *nat*) => (*nat* × *nat*)
where *hom-oper-inv-def* : *hom-oper-inv pos* = (*fst pos*, (*snd pos*) − *1*)

lemma *hom-oper-infmatrix-finite*[*simp*]:
shows *finite* (*nonzero-positions* (*hom-oper-infmatrix* (*Rep-matrix x*)))
⟨*proof*⟩

lemma *hom-oper-infmatrix-matrix*:
shows (*hom-oper-infmatrix* (*Rep-matrix* (*x*::*'a*::*zero matrix*))) ∈ *matrix*
⟨*proof*⟩

lemma *hom-oper-infmatrix-closed*[*simp*]:
Rep-matrix (*Abs-matrix* (*hom-oper-infmatrix* (*Rep-matrix x*))) = *hom-oper-infmatrix* (*Rep-matrix x*)
⟨*proof*⟩

lemma *hom-oper-infmatrix-is-matrix*: **assumes** *as*: *A* ∈ *matrix*
shows *hom-oper-infmatrix A* ∈ *matrix*
⟨*proof*⟩

definition *hom-oper-matrix1* :: *'a*::*lordered-ring matrix* => *'a matrix*
where *hom-oper-matrix1-def*:
hom-oper-matrix1 = *Abs-matrix* ∘ *hom-oper-infmatrix* ∘ *Rep-matrix*

lemma *hom-oper-infmatrix-twice* [*simp*]:
hom-oper-infmatrix (*hom-oper-infmatrix A*) = (*Rep-matrix 0*)
⟨*proof*⟩

lemma *combine-infmatrix-hom-oper-infmatrix-matrix*[*simp*]:
f 0 0 = *0* ⟹ *combine-infmatrix f* (*hom-oper-infmatrix* (*Rep-matrix A*))
(*hom-oper-infmatrix* (*Rep-matrix B*)) ∈ *matrix*
⟨*proof*⟩

lemma *hom-oper-infmatrix-combine-infmatrix-matrix*[*simp*]:
f 0 0 = *0* ⟹ *hom-oper-infmatrix* (*combine-infmatrix f* (*Rep-matrix A*) (*Rep-matrix B*)) ∈ *matrix*
⟨*proof*⟩

36.6 Matrices as an instance of differential group with a homotopy operator.

Matrices with the given operations for differential, perturbation and homotopy operator are an instance of the type class representing differential groups with a perturbation and a homotpy operator:

lemma *hom-oper-matrix1-hom*:
hom-oper-matrix1 $(A + B)$ = *hom-oper-matrix1* A + *hom-oper-matrix1* B
⟨*proof*⟩

lemma *hom-oper-matrix1-twice*:
shows *hom-oper-matrix1* (*hom-oper-matrix1* A) = (0::('a::*lordered-ring matrix*))
⟨*proof*⟩

instantiation *matrix* :: (*lordered-ring*) *diff-group-add-pert-hom*
begin

definition *hom-oper-matrix-def*: *hom-oper* A == *hom-oper-matrix1* A

instance
⟨*proof*⟩

end

36.7 Local nilpotency condition of the previous definitions.

lemma *hom-oper-closed*:
shows *Rep-matrix* ∘ *Abs-matrix* ∘ *hom-oper-infmatrix* ∘ *Rep-matrix*
= *hom-oper-infmatrix* ∘ *Rep-matrix*
⟨*proof*⟩

lemma *pert-infmatrix-hom-oper-infmatrix-finite*[*simp*]:
shows *finite* (*nonzero-positions* (*pert-infmatrix* (*hom-oper-infmatrix* (*Rep-matrix* x))))
⟨*proof*⟩

lemma *pert-infmatrix-hom-oper-infmatrix-matrix*:
shows (*pert-infmatrix* (*hom-oper-infmatrix* (*Rep-matrix* (x::'a::*zero matrix*))))
∈ *matrix*
⟨*proof*⟩

lemma *pert-infmatrix-hom-oper-infmatrix-closed*[*simp*]:
shows *Rep-matrix* (*Abs-matrix* (*pert-infmatrix* (*hom-oper-infmatrix* (*Rep-matrix* A)))) =
pert-infmatrix (*hom-oper-infmatrix* (*Rep-matrix* A))
⟨*proof*⟩

lemma *hom-oper-infmatrix-pert-infmatrix-finite*[*simp*]:
shows *finite* (*nonzero-positions* (*hom-oper-infmatrix* (*pert-infmatrix* (*Rep-matrix*

x))))
⟨*proof*⟩

lemma *hom-oper-infmatrix-pert-infmatrix-matrix*:
shows (*hom-oper-infmatrix* (*pert-infmatrix* (*Rep-matrix* (*x*::′*a*::*zero matrix*))))
∈ *matrix*
⟨*proof*⟩

lemma *hom-oper-infmatrix-pert-infmatrix-closed*[*simp*]:
shows *Rep-matrix* (*Abs-matrix* (*hom-oper-infmatrix* (*pert-infmatrix* (*Rep-matrix*
A)))) =
hom-oper-infmatrix (*pert-infmatrix* (*Rep-matrix A*))
⟨*proof*⟩

lemma *f-1-n*: **shows** $f\ ((f\hat{}n)\ a) = (f\hat{}(n + 1))\ a$
⟨*proof*⟩

This is the crucial lemma to prove the local nilpotency condition; the result states that there is no infinite strictly descending chain of natural numbers. Applied to matrices, it states that there is no infinite descending path across a matrix of finite dimension, and thus, being α = *diff-group-add-pert-class.pert* ∘ *hom-oper* strictly decreasing with repect to the dimension of matrices, the local nilpotency condition holds.

lemma *infinite-descending-chain-false*:
fixes *g* :: *nat* => *nat*
assumes *n-inf-decr*: $\bigwedge n.\ g\ (Suc\ n) < g\ n$ **shows** *False*
⟨*proof*⟩

Making use of the previous result, if the number of rows is strictly decreasing, finally matrix *0* is reached.

lemma *P-strict-decr*:
fixes *f* :: ′*a*::*lordered-ring matrix* => ′*a matrix*
and *P* :: ′*a matrix* => *nat*
assumes *P-decr*: $\bigwedge A.\ A \neq 0 \Longrightarrow (P\ (f\ A) < P\ A)$ **and** *P-0*: $P\ 0 = 0$
shows $\exists\, n{::}nat.\ P\ ((f\hat{}n)\ A) = 0$
⟨*proof*⟩

lemma *nrows-strict-decr*:
fixes *f* :: ′*a*::*lordered-ring matrix* => ′*a matrix*
assumes *nrows-decr*: $\bigwedge A.\ A \neq 0 \Longrightarrow (nrows\ (f\ A) < nrows\ A)$
shows $\exists\, n{::}nat.\ nrows\ (\ (f\hat{}n)\ A) = 0$
⟨*proof*⟩

lemma *ncols-strict-decr*:
fixes *f* :: ′*a*::*lordered-ring matrix* => ′*a matrix*
assumes *ncols-decr*: $\bigwedge A.\ A \neq 0 \Longrightarrow (ncols\ (f\ A) < ncols\ A)$
shows $\exists\, n{::}nat.\ ncols\ (\ (f\hat{}n)\ A) = 0$
⟨*proof*⟩

lemma *nonzero-empty-set*:
  shows *nonzero-positions* (*Rep-matrix* (0::$'a$::*lordered-ring matrix*)) = {}
  ⟨*proof*⟩

lemma *Rep-matrix-inject2*:
  assumes *R-A-B*: *Rep-matrix A* = *Rep-matrix B*
  shows $A = B$
  ⟨*proof*⟩

lemma *nonzero-imp-zero-matrix*:
  assumes *nz-p*: *nonzero-positions* (*Rep-matrix* (*A*::$'a$::*lordered-ring matrix*)) =
{}
  shows $A = 0$
⟨*proof*⟩

lemma *nrows-zero-impl-zero*:
  assumes *nrows-0*: *nrows* (*A*::$'a$::*lordered-ring matrix*) = (0::*nat*)
  shows $A = 0$
⟨*proof*⟩

lemma *ncols-zero-impl-zero*:
  assumes *ncols-0*: *ncols* (*A*::$'a$::*lordered-ring matrix*) = (0::*nat*)
  shows $A = 0$
⟨*proof*⟩

lemma *not-zero-impl-exist*:
  assumes *A-not-null*: (*A*::$'a$::*lordered-ring matrix*) $\neq 0$
  shows $\exists$ *m n*::*nat*. *Rep-matrix A m n* $\neq 0$
⟨*proof*⟩

corollary *not-zero-impl-nonzero-pos*:
  assumes *A-not-null*: (*A*::$'a$::*lordered-ring matrix*) $\neq 0$
  shows *nonzero-positions* (*Rep-matrix A*) $\neq$ {}
  ⟨*proof*⟩

corollary *not-zero-impl-nrows-g-0*:
  assumes *A-not-null*: (*A*::$'a$::*lordered-ring matrix*) $\neq 0$
  shows *nrows A* > 0
  ⟨*proof*⟩

corollary *not-zero-impl-ncols-g-0*:
  assumes *A-not-null*: (*A*::$'a$::*lordered-ring matrix*) $\neq 0$
  shows *ncols A* > 0
  ⟨*proof*⟩

lemma *hom-oper-nonzero-positions*:
  assumes *a-b-1*: (a, $b + 1$) $\in$ *nonzero-positions* (*Rep-matrix A*)
  and *b-n-0*: (0::*nat*) $< b$

and *ab-even*: $(a + b)$ *mod 2* $\neq$ $(0::nat)$
shows $(a,\ b + 2) \in$ *nonzero-positions* (*hom-oper-infmatrix* (*Rep-matrix A*))
⟨*proof*⟩

lemma *pert-nonzero-positions*:
assumes *a-b-1*: $(a + 1,\ b) \in$ *nonzero-positions* (*Rep-matrix A*)
and *ab-pair*: $(a + b)$ *mod 2* $=$ $(0::nat)$
shows $(a,\ b) \in$ *nonzero-positions* (*pert-infmatrix* (*Rep-matrix A*))
⟨*proof*⟩

lemma *pert-nonzero-incr*:
assumes *ab-nonzero*: $(a,\ b) \in$ *nonzero-positions* (*pert-infmatrix* (*Rep-matrix A*))

shows $(a + 1,\ b) \in$ *nonzero-positions* (*Rep-matrix A*)
⟨*proof*⟩

lemma *pert-hom-oper-nonzero-ncols-g-1*:
assumes *ab-nonzero*: $(a,\ b) \in$ *nonzero-positions*
(*pert-infmatrix* (*hom-oper-infmatrix* (*Rep-matrix A*)))
shows $1 \leq b$
⟨*proof*⟩

lemma *pert-hom-oper-nonzero-incr-rows*:
assumes *ab-nonzero*: $(a,\ b) \in$ *nonzero-positions*
(*pert-infmatrix* (*hom-oper-infmatrix* (*Rep-matrix A*)))
shows $(a + 1,\ b - 1) \in$ *nonzero-positions* (*Rep-matrix A*)
⟨*proof*⟩

lemma *hom-oper-pert-nonzero-incr-rows*:
assumes *ab-nonzero*: $(a,\ b) \in$ *nonzero-positions*
(*hom-oper-infmatrix* (*pert-infmatrix* (*Rep-matrix A*)))
shows $(a + 1,\ b - 1) \in$ *nonzero-positions* (*Rep-matrix A*)
⟨*proof*⟩

lemma *Max-A-B*:
assumes *exist*: $\exists\, m > Max\ B.\ m \in A$
and *fin-A*: *finite A* **and** *fin-B*: *finite B***shows** $Max\ B < Max\ A$
⟨*proof*⟩

lemma *nrows-pert-decre*:
assumes *A-not-null*: $(A::'a::lordered\text{-}ring\ matrix) \neq 0$
shows *nrows* ((− (*pert-matrix1*)) *A*) < *nrows A*
(**is** *nrows* ((− *?f*) *A*) < *nrows A*)
⟨*proof*⟩

lemma *nrows-pert-hom-oper-decre*:
assumes *A-not-null*: $(A::'a::lordered\text{-}ring\ matrix) \neq 0$
shows *nrows* ((− (*pert-matrix1* ∘ *hom-oper-matrix1*)) *A*) < *nrows A*
(**is** *nrows* ((− *?f*) *A*) < *nrows A*)

⟨*proof*⟩

corollary *nrows-alpha-decre*:
assumes *A-not-null*: $(A::'a::lordered\text{-}ring\ matrix) \neq 0$
shows $nrows\ (\alpha\ A) < nrows\ A$
⟨*proof*⟩

36.8 Matrices as an instance of differential group satisfying the local nilpotency condition.

With the previous results, we can finally prove that matrices are an instance of differential group with a perturbation and a homotpy operator that additionally satisfies the local nilpotency condition.

instantiation *matrix* :: (*lordered-ring*) *diff-group-add-pert-hom-bound-exist*
begin

lemma *α-matrix-definition*: $\alpha = -\ (pert\text{-}matrix1 \circ hom\text{-}oper\text{-}matrix1)$
⟨*proof*⟩

instance
⟨*proof*⟩

end

Unfortunately, the previous type *matrix* cannot be used with the code generation facility. In order to get code generation, it is not enoguh to prove the instances, but also the underlying types need to have an executable counterpart.

Thus, we used new definitions based on *sparse matrices*, which are based on lists and can be used with the code generation facility.

Unfortunately, sparse matrices cannot be proved to be an instance of the previous type classes, since the results required are not equations or definitional theorems, but more elaborated ones.

36.9 Definition of the operations over sparse matrices.

fun *transpose-spmat* :: ('*a*::*lordered-ring*) *spmat* => '*a spmat*
where
transpose-spmat-Em: *transpose-spmat* [] = []
| *transpose-spmat-C-cons*: *transpose-spmat* ((*i*,*v*)#*vs*) =
(*let m* = *map* (λ (*j*, *x*). (*j*, [(*i*, *x*)])) *v*;
M = *transpose-spmat vs*
in add-spmat (*m*, *M*))

Unfortunately, the following definition of *differ-spmat* does not preserve the sortedness of the input matrix, as it is shown in the lemmas below

```
fun differ-spmat :: ('a::lordered-ring) spmat => 'a spmat
  where
  differ-spmat-Em: differ-spmat [] = []
  | differ-spmat-C-cons: differ-spmat ((i,v)#vs) =
  (let
  m = map (λ (j, x). (if (j = 0) then (j, 0)
                      else (if ((i + j) mod 2 = 0)
                            then (j − 1, (0::'a))
                                 else (j − 1, x)))) v;
  M = differ-spmat vs
  in add-spmat ([(i, m)], M))
```

Let's see what happens with *differ-spmat* which has degree -1 and should not preserve the order of the colums

In the first lemma it canbe seen that the predicate *sorted-spvec* holds for *differ-spmat*. This means that it keeps th sortedness of rows. This will be later proved.

Unfortunately, *differ-spmat* doesnot preserve *sorted-spmat*, which is the predicate checking that the elements in each column are sorted.

A lemma proving that *differ-spmat* is distributive with respect to the appending of lists.

```
lemma differ-spmat-dist: differ-spmat ((a, b) # A) = add-spmat (
         ([(a, (map (λ (j, x). if (j = 0) then (j, (0::'a::lordered-ring))
                              else (if ((a + j) mod 2 = 0) then (j − 1, (0::'a))
                                    else (j − 1, x))) b))]), differ-spmat A)
  ⟨proof⟩
```

The following definition *differ-spmat'* is equal to the one of *differ-spmat* after the application of function *sparse-row-matrix*. This means that both functions produce similar results over terms of *matrix* type.

We used *differ-spmat'* for some induction methods since it produces always elements of type *nat* × (*nat* × *int*) *list* which somehow simplify induction proofs

Later we will introduce a third operation *differ-smat-sort*, which is also equivalent to these two in producing terms of *matrix* type and which additionally preserves sortedness of vectors.

```
fun differ-spmat' :: ('a::lordered-ring) spmat => 'a spmat
  where
  differ-spmat'-Em: differ-spmat' [] = []
  | differ-spmat'-C-cons: differ-spmat' ((i,v)#vs) =
  (let
  m = map (λ (j, x). if (j = 0) then (i, [(j, (0::'a))])
                     else (if ((i + j) mod 2 = 0) then (i, [(j − 1, (0::'a))])
```

else ((i, [(j − 1, x)])))) v;
M = differ-spmat′ vs
in add-spmat (m, M))

lemma *differ-spmat′-dist*: *differ-spmat′ ((a, b) # A) = add-spmat (*
map (λ (j, x). if (j = 0) then (a, [(j, (0::′a::lordered-ring))])
else (if ((a + j) mod 2 = 0) then (a, [(j − 1, (0::′a))])
else (a, [(j − 1, x)]))) b, differ-spmat′ A)
⟨*proof*⟩

A similar situation arises with *pert-spmat* and *pert-spmat′*; one is better in preserving sortedness and the other one is better for induction proofs.

The following definition reiles on the fact that the result of evaluating *0 − 1* over the natural numbers is equal to *0*

fun *pert-spmat* :: (*′a::lordered-ring*) *spmat* => *′a spmat*
where
pert-spmat-Em: *pert-spmat* [] = []
| *pert-spmat-C-cons*: *pert-spmat* ((*i*,*v*)#*vs*) =
(*let*
m = map (λ (j, x). if (i = 0) then (j, (0::′a)) else
if ((i + j) mod 2 = 0) then (j, (0::′a))
else (j, x)) v;
M = pert-spmat vs
in add-spmat ([(i − 1, m)], M))

lemma *pert-spmat-dist*: *pert-spmat ((a, b) # A) = add-spmat (*
([(a − 1, map (λ (j, x). if (a = 0) then (j, (0::′a::lordered-ring)) else
if ((a + j) mod 2 = 0) then (j, (0::′a))
else (j, x)) b)]) , pert-spmat A)
⟨*proof*⟩

fun *pert-spmat′* :: (*′a::lordered-ring*) *spmat* => *′a spmat*
where
pert-spmat′-Em: *pert-spmat′* [] = []
| *pert-spmat′-C-cons*: *pert-spmat′* ((*i*,*v*)#*vs*) =
(*let*
m = map (λ (j, x). if (i = 0) then (i − 1, [(j, (0::′a))]) else
if ((i + j) mod 2 = 0) then (i − 1, [(j, (0::′a))])
else (i − 1, [(j, x)])) v;
M = pert-spmat′ vs
in add-spmat (m, M))

lemma *pert-spmat′-dist*: *pert-spmat′ ((a, b) # A) = add-spmat (*
(map (λ (j, x). if (a = 0) then (a − 1, [(j, (0::′a::lordered-ring))]) else
if ((a + j) mod 2 = 0) then (a − 1, [(j, (0::′a))])

else $(a - 1, [(j, x)]))$ $b)$, *pert-spmat'* $A)$
⟨*proof*⟩

In a similar way we provide two definitions for the *hom-oper-spmat* operation

fun *hom-oper-spmat* :: (*'a*::*lordered-ring*) *spmat* => *'a spmat*
where
hom-oper-spmat-Em: *hom-oper-spmat* [] = []
| *hom-oper-spmat-C-cons*: *hom-oper-spmat* ((*i*,*v*)#*vs*) =
(*let*
m = *map* (λ (*j*, *x*). *if* ((*i* + *j*) *mod* (*2*::*nat*) = *1*) *then* (*j* + *1*, (*0*::*'a*))
else (*j* + *1*, *x*)) *v*;
M = *hom-oper-spmat vs*
in add-spmat ((*i*, *m*) # [], *M*))

fun *hom-oper-spmat'* :: (*'a*::*lordered-ring*) *spmat* => *'a spmat*
where
hom-oper-spmat'-Em: *hom-oper-spmat'* [] = []
| *hom-oper-spmat'-C-cons*: *hom-oper-spmat'* ((*i*,*v*)#*vs*) =
(*let*
m = *map* (λ (*j*, *x*). *if* ((*i* + *j*) *mod* (*2*::*nat*) = *1*) *then* (*i*, [(*j* + *1*, (*0*::*'a*))])
else (*i*, [(*j* + *1*, *x*)])) *v*;
M = *hom-oper-spmat' vs*
in add-spmat (*m*, *M*))

lemma *hom-oper-spmat-dist*: *hom-oper-spmat* ((*a*, *b*) # *A*) = *add-spmat* (
([(*a*, *map* (λ(*j*, *x*). *if* (*a* + *j*) *mod 2* = *1 then* (*j* + *1*, *0*::*'a*::*lordered-ring*)
else (*j* + *1*, *x*)) *b*)],
hom-oper-spmat A))
⟨*proof*⟩

lemma *hom-oper-spmat'-dist*: *hom-oper-spmat'* ((*a*, *b*) # *A*) = *add-spmat* (
map (λ (*j*, *x*). *if* ((*a* + *j*) *mod 2* = *1*) *then* (*a*, [(*j* + *1*, (*0*::*'a*::*lordered-ring*))])
else (*a*, [(*j* + *1*, *x*)])) *b*, *hom-oper-spmat' A*)
⟨*proof*⟩

36.10 Some auxiliary lemmas

lemma *map-impl-existence*:
assumes *map-f-l*: *map f l* = *a* # *l'*
shows ∃ *a'*. *a' mem l* ∧ *f a'* = *a*
⟨*proof*⟩

lemma *move-matrix-zero*: **shows** *move-matrix 0 i j* = *0*
⟨*proof*⟩

lemma *sparse-row-matrix-cons2*:
sparse-row-matrix [(*i*, (*a*::(*nat*×*'a*::*lordered-ring*)) # *v*)]
= *singleton-matrix i* (*fst a*) (*snd a*) + *sparse-row-matrix* [(*i*,*v*)]
⟨*proof*⟩

lemma *sorted-spvec-trans*:
assumes *srtd*: *sorted-spvec* $(a\#b)$ **and** *c-in-b*: *c mem b*
shows *fst a* < *fst c*
⟨*proof*⟩

36.11 Properties of the operation *hom-oper-spmat*

The operation *hom-oper-spmat* preserves sortedness of vectors and matrices.

It is also equivalent to the operation *hom-matrix1* that we introduced previously, under the operation *sparse-row-matrix*.

lemma *add-spmat-el*[*simp*]: *add-spmat* (*a*, []) = *a*
⟨*proof*⟩

thm *foldl-absorb0*

The following lemma is almost a copy of *?x* + *foldl op* + (*0*::*?'a*) *?zs* = *foldl op* + *?x ?zs* except that it considers different types in the arguments

lemma *foldl-absorb0'*:
fixes *l* :: *nat* × (*nat* × *'a*::*lordered-ring*) *list* => *'a matrix*
shows $\bigwedge M$. *foldl* (λ(*m*::*'a*::*lordered-ring matrix*) *x*::*nat* × (*nat* × *'a*) *list*.
m + (*l x*)) *M arr* =
M + *foldl* (λ*m x*. *m* + (*l x*)) (*0*::*'a matrix*) *arr*
⟨*proof*⟩

lemma *sparse-row-matrix-hom-oper-spvec*:
shows *sparse-row-matrix* (*hom-oper-spmat* ((*i*, *v*)#[]))
= *sparse-row-matrix* (*hom-oper-spmat'* ((*i*, *v*)# []))
⟨*proof*⟩

The following operation proves that the two operations *hom-oper-spmat* and *hom-oper-spmat'* are equivalent under the application of *sparse-row-matrix*.

lemma *sparse-row-matrix-hom-oper-spmat*:
sparse-row-matrix (*hom-oper-spmat A*)
= *sparse-row-matrix* (*hom-oper-spmat' A*)
⟨*proof*⟩

The following lemma proves that *hom-oper-spmat* preserves the sortedness of matrices w.r.t *hom-oper-spvec*.

lemma *sorted-spvec-hom-oper*:
assumes *srtd-A*: *sorted-spvec A*
shows *sorted-spvec* (*hom-oper-spmat A*)
⟨*proof*⟩

lemma *sorted-spvec-singleton*[*simp*]: *sorted-spvec* [(*i*, *v*)]
⟨*proof*⟩

lemma *sorted-spvec-hom-oper-vec*: **assumes** *ss-v*: *sorted-spvec v*
shows *sorted-spvec* (*map* (λ(*j*, *x*). *if* (*i* + *j*) *mod 2* = *Suc 0*
then (*j* + *1*, *0*::'*a*::*lordered-ring*) *else* (*j* + *1*, *x*)) *v*)
(**is** *sorted-spvec* (*map ?f v*))
⟨*proof*⟩

lemma *sorted-spmat-hom-oper*:
assumes *sorted-spmat-A*: *sorted-spmat A*
shows *sorted-spmat* (*hom-oper-spmat A*)
⟨*proof*⟩

36.12 Properties of the function *differ-spmat*

lemma *sparse-row-matrix-differ-spvec*:
sparse-row-matrix (*differ-spmat* ((*i*, *v*)#[]))
= *sparse-row-matrix* (*differ-spmat'* ((*i*, *v*)# []))
⟨*proof*⟩

lemma *sparse-row-matrix-differ-spmat*:
sparse-row-matrix (*differ-spmat A*)
= *sparse-row-matrix* (*differ-spmat' A*)
⟨*proof*⟩

This lemma might be omitted. *differ-spmat* preserves the sortedness of the rows but not the one of vectors, so this is why we will later introduce *differ-spmat-sort*

lemma *sorted-spvec-differ*:
assumes *srtd-A*: *sorted-spvec A*
shows *sorted-spvec* (*differ-spmat A*)
⟨*proof*⟩

36.13 Some ideas to keep sorted vectors inside a matrix

We will define two functions to keep vectors sorted. The first one, *insert-sorted* inserts a pair in a list in its right position.

From the definition it can be seen that it does not pay attention to whether the list was originally sorted or not.

primrec *insert-sorted*:: (*nat* × '*a*::*lordered-ring*) => '*a spvec* => '*a spvec*
where
insert-sorted-nil: *insert-sorted a* [] = [*a*]
| *insert-sorted-cons*: *insert-sorted a* (*b* # *c*) =
(*if* (*fst a* < *fst b*) *then* (*a* # *b* # *c*) *else*
(*if fst a* = *fst b then* (*fst a*, *snd a* + *snd b*) # *c*
else b # (*insert-sorted a c*)))

lemma [*simp*]: *sorted-spvec* [*a*]
⟨*proof*⟩

The following lemma proves that the *insert-sorted* preserves sortedness of vectors.

lemma *insert-preserves-sorted*:
  assumes *sorted-spvec-v*: *sorted-spvec v*
  shows *sorted-spvec* (*insert-sorted a v*)
  ⟨*proof*⟩

The following operation, *sort-spvec*, takes the first element of a list and introduces it in its corresponding position, by means of the *insert-sort* operation.

Applying this process recursively, we get to sort the vector.

primrec *sort-spvec* :: *'a::lordered-ring spvec => 'a spvec*
  where
  sort-spvec-nil: *sort-spvec* [] = []
  | *sort-spvec-cons*: *sort-spvec* (*a* # *b*) = (*insert-sorted a* (*sort-spvec b*))

lemma *sort-is-sorted*: **shows** *sorted-spvec* (*sort-spvec v*)
⟨*proof*⟩

The operation of sorting is compatible with the addition of sparse vectors

lemma *insert-pair-add-pair*:
  shows *insert-sorted* (*a*, *b*) *v* = *add-spvec* ([(*a*, *b*)], *v*)
⟨*proof*⟩

The following operation is similar to *differ-spmat* except that it applies *sort-spvec* to every vector.

fun *differ-spmat-sort* :: (*'a::lordered-ring*) *spmat => 'a spmat*
  where
  differ-spmat-sort-Em: *differ-spmat-sort* [] = []
  | *differ-spmat-sort-C-cons*: *differ-spmat-sort* ((*i*,*v*)#*vs*) =
  (*let*
  m = *sort-spvec* (*map* (λ (*j*, *x*). *if j* = *0 then* (*j*, *0*)
                          else if (*i* + *j*) *mod 2* = *0 then* (*j* − *1*, *0*) *else* (*j* − *1*, *x*))
v);
  M = *differ-spmat-sort vs*
  in add-spmat ([(*i*, *m*)], *M*))

lemma *differ-spmat-sort-dist*:
  shows *differ-spmat-sort* ((*i*, *v*) # *vs*) = *add-spmat* (
        ([(*i*, *sort-spvec* (
              map (λ (*j*, *x*::*'a*::*lordered-ring*). *if j* = *0 then* (*j*, *0*)
                else if (*i* + *j*) *mod 2* = *0 then* (*j* − *1*, *0*)
                  else (*j* − *1*, *x*)) *v*))]), *differ-spmat-sort vs*)
  ⟨*proof*⟩

lemma *sorted-spvec-differ-spmat-sort*:
  assumes *srtd-A*: *sorted-spvec A*
  shows *sorted-spvec* (*differ-spmat-sort A*)

⟨*proof*⟩

As it may be seen in the following lemma, as far as we now have the operation *sort-spvec* sorting every vector, the lemma does not require the premise of A being previously sorted.

lemma *sorted-spmat-differ-spmat-sort*:
shows *sorted-spmat* (*differ-spmat-sort* A)
⟨*proof*⟩

primrec *sort-spvec-spmat* :: *'a::lordered-ring spmat* => *'a spmat*
where
sort-spvec-spmat-nil: *sort-spvec-spmat* [] = []
| *sort-spvec-spmat-cons*: *sort-spvec-spmat* (*a* # *as*)
= (*fst a*, *sort-spvec* (*snd a*)) # (*sort-spvec-spmat as*)

lemma *sort-spvec-spmat-impl-sorted-spmat*:
shows *sorted-spmat* (*sort-spvec-spmat* A)
⟨*proof*⟩

The following lemma shows that even in the cases where *differ-spmat* produces non sorted matrices, they are equal to the ones obtained with *differ-spmat-sort* under the application *sparse-row-matrix*

We illustrate this fact with an example with *foo*.

lemma *sparse-row-vector-not-sorted*:
shows *sparse-row-vector* (*a* # *b* # []) = *sparse-row-vector* (*b* # *a* # [])
⟨*proof*⟩

lemma *sparse-row-vector-dist*:
shows *sparse-row-vector* ([(i, j + k)]) = *sparse-row-vector* ((i, j) # [(i, k)])
⟨*proof*⟩

lemma *insert-sorted-preserves-sparse-row-vector*:
shows *sparse-row-vector* (*a* # *v*) = *sparse-row-vector* (*insert-sorted a v*)
⟨*proof*⟩

lemma *sort-spvec-eq-sparse-row-vector*:
shows *sparse-row-vector v* = *sparse-row-vector* (*sort-spvec v*)
⟨*proof*⟩

lemma *sort-spvec-eq-sparse-row-matrix*:
shows *sparse-row-matrix* [(i, v)]
= *sparse-row-matrix* [(i, *sort-spvec v*)]
⟨*proof*⟩

lemma *sparse-row-matrix-differ-spmat-differ-spmat-sort*:
shows *sparse-row-matrix* (*differ-spmat* A)
= *sparse-row-matrix* (*differ-spmat-sort* A)
⟨*proof*⟩

36.14 Properties of the perturbation operator

lemma *sparse-row-matrix-pert-spvec*:
  shows *sparse-row-matrix* (*pert-spmat* ((*i*, *v*)#[]))
  = *sparse-row-matrix* (*pert-spmat'* ((*i*, *v*)# []))
⟨*proof*⟩

lemma *sparse-row-matrix-pert-spmat*:
  shows *sparse-row-matrix* (*pert-spmat A*) = *sparse-row-matrix* (*pert-spmat' A*)
⟨*proof*⟩

The following oepration is equivalent to *pert-spmat* except that it sorts every vector.

fun *pert-spmat-sort* :: (*'a*::*lordered-ring*) *spmat* => *'a spmat*
  where
  pert-spmat-sort-Em: *pert-spmat-sort* [] = []
  | *pert-spmat-sort-C-cons*: *pert-spmat-sort* ((*i*,*v*)#*vs*) =
  (*let*
  m = *sort-spvec* (*map* (λ (*j*, *x*). *if i* = *0 then* (*j*, *0*) *else*
    if (*i* + *j*) *mod 2* = *0 then* (*j*, *0*) *else* (*j*, *x*)) *v*);
  M = *pert-spmat-sort vs*
  in add-spmat ([(*i* − *1*, *m*)], *M*))

lemma *pert-spmat-sort-dist*: *pert-spmat-sort* ((*i*, *v*) # *vs*) = *add-spmat* (
  ([(*i* − *1*, *sort-spvec* (*map* (λ (*j*, *x*). *if i* = *0 then* (*j*, *0*) *else*
    if (*i* + *j*) *mod 2* = *0 then* (*j*, *0*) *else* (*j*, *x*)) *v*))]), *pert-spmat-sort vs*)
  ⟨*proof*⟩

Once again, the premise of *A* being sorted can be avoided in the following lemma since *pert-spmat-sort* will do the job

lemma *sorted-spmat-pert-spmat-sort*:
  shows *sorted-spmat* (*pert-spmat-sort A*)
⟨*proof*⟩

The following lemma proves that both *pert-spmat* and *pert-spmat-sort* produce the same result under the application of *sparse-row-matrix*, or, in other words, as terms of *matrix* type.

lemma *sparse-row-matrix-pert-spmat-eq-pert-spmat-sort*:
  shows *sparse-row-matrix* (*pert-spmat A*) = *sparse-row-matrix* (*pert-spmat-sort A*)
⟨*proof*⟩

lemma *sorted-spvec-pert-spmat-sort*:
  assumes *srtd-A*: *sorted-spvec A*
  shows *sorted-spvec* (*pert-spmat-sort A*)
  ⟨*proof*⟩

The followig lemma may be omitted since we do not care about the properties of *pert-spmat*, but of the ones from *pert-spmat-sort*

lemma *sorted-spvec-pert*:
assumes *srtd-A*: *sorted-spvec A*
shows *sorted-spvec* (*pert-spmat A*)
⟨*proof*⟩

37 Equivalence between operations over matrices and sparse matrices.

lemma *diff-matrix1-zero-eq-zero*: *diff-matrix1 0 = 0*
⟨*proof*⟩

lemma *foldl-zero-matrix*:
shows *foldl* (λ(*m*::'*a matrix*) *x*::*nat* × '*a*. *m*) (*0*::'*a*::*lordered-ring matrix*) *v* = *0*
⟨*proof*⟩

lemma *Rep-Abs-zero-func*:
shows *Rep-matrix* (*Abs-matrix* (λ*m n*::*nat*. *0*::'*a*::*lordered-ring*)) *p q*
= (*0*::'*a*)
⟨*proof*⟩

lemma *is-matrix-i-j*:
shows *Rep-matrix* (*Abs-matrix* (λ(*m*::*nat*) *n*::*nat*. *if i* = *m* ∧ *j* = *n then v else 0*)) =
(λ*m n*. *if i* = *m* ∧ *j* = *n then v else* (*0*::'*b*::*lordered-ring*))
⟨*proof*⟩

lemma *diff-matrix1-singleton*:
shows *diff-matrix1* (*singleton-matrix i j v*) =
Abs-matrix (λ*k l*. *if* (*k* + *l*) *mod 2* = *1 then 0*::'*a*::*lordered-ring*
else if i = *k* ∧ *j* = *l* + *1 then v else 0*)
⟨*proof*⟩

lemma *comb-matrix*:
assumes *f-eq*: *f* = *g* **and** *matrix-eq*: (*A*::'*a matrix*) = *B*
and *x-eq*: (*x*::*int*) = *x'* **and** *y-eq*: (*y*::*int*) = *y'*
shows *f A x y* = *g B x' y'*
⟨*proof*⟩

lemma *diff-matrix1-execute-sparse*:
shows *diff-matrix1* (*sparse-row-matrix A*)
= *sparse-row-matrix* (*differ-spmat' A*)
⟨*proof*⟩

lemma *hom-oper-matrix1-zero-eq-zero*: *hom-oper-matrix1 0 = 0*
⟨*proof*⟩

lemma *hom-oper-matrix1-singleton*:
shows *hom-oper-matrix1* (*singleton-matrix k l v*) =
Abs-matrix ($\lambda i\ j$. *if* ($j = 0$) *then* 0 *else*
if (($i + j$) *mod* $2 = 0$) *then* 0 *else*
if ($k = i \wedge l = j - (1{::}nat)$) *then* v
else ($0{::}'b{::}lordered\text{-}ring$))
⟨*proof*⟩

lemma *hom-oper-matrix1-execute-sparse*:
shows *hom-oper-matrix1* (*sparse-row-matrix A*)
= *sparse-row-matrix* (*hom-oper-spmat′ A*)
⟨*proof*⟩

lemma *hom-oper-matrix1-execute-sparse-simp*:
shows *hom-oper-matrix1* (*sparse-row-matrix A*)
= *sparse-row-matrix* (*hom-oper-spmat A*)
⟨*proof*⟩

lemma *pert-matrix1-zero-eq-zero*: *pert-matrix1* $0 = 0$
⟨*proof*⟩

lemma *pert-matrix1-singleton*:
shows *pert-matrix1* (*singleton-matrix k l v*) =
Abs-matrix ($\lambda i\ j$. *if* (($i + j$) *mod* $2 = 1$) *then* 0 *else*
if ($k = i + (1{::}nat) \wedge l = j$) *then* v
else ($0{::}'b{::}lordered\text{-}ring$))
⟨*proof*⟩

lemma *pert-matrix1-execute-sparse*:
shows *pert-matrix1* (*sparse-row-matrix A*)
= *sparse-row-matrix* (*pert-spmat′ A*)
⟨*proof*⟩

lemma *pert-matrix1-execute-sparse-simp*:
shows *pert-matrix1* (*sparse-row-matrix A*)
= *sparse-row-matrix* (*pert-spmat-sort A*)
⟨*proof*⟩

37.1 Zero as sparse matrix

In order to get the bound of a function we need a notion of a zero matrix, up to which the bound is computed.

In sparse matrices there is no single zero matrix, but any matrix containing only zeros will be such.

We provide a function that computes the number of non-zero elements of a vector.

If the number of non-zero positions of a vector is zero, this should be the null vector.

primrec *non-zero-spvec* :: *'a::zero spvec => nat*
  where *non-zero-spvec* [] = *0*
  | *non-zero-spvec* (*a#b*) = (*if snd a* ≠ *0 then Suc* (*non-zero-spvec b*)
  else non-zero-spvec b)

lemma *non-zero-spvec-ge-0*: (*0::nat*) ≤ *non-zero-spvec a* ⟨*proof*⟩

We have proven the following lemma in terms of sets since automatic methods apply much better than with Lists and functions *op mem* or *ListMem*

lemma *non-zero-spvec-g-0*:
  assumes *non-zero-a*: *non-zero-spvec* (*a::'a::zero spvec*) ≠ *0*
  shows ∃ *pair*. *pair* ∈ *set a* ∧ *snd pair* ≠ (*0::'a::zero*)
⟨*proof*⟩

Based on the previous function for vectors, a function for matrices is defined.

This function will be later used in the computation of the bound of nilpotency of a matrix.

primrec *non-zero-spmat*:: *'a::zero spmat => nat*
  where *non-zero-spmat* [] = *0*
  | *non-zero-spmat* (*a#b*) = *non-zero-spvec* (*snd a*) + *non-zero-spmat b*

lemma *non-zero-spmat-g-0*:
  assumes *non-zero-a*: *non-zero-spmat* (*a::'a::zero spmat*) ≠ *0*
  shows ∃ *list*. *list* ∈ *set a* ∧ *non-zero-spvec* (*snd list*) ≠ *0*
⟨*proof*⟩

lemma *non-zero-spvec-impl-sp-rw-vector*:
  assumes *non-zero-spvec*: *non-zero-spvec A* = *0*
  shows *sparse-row-vector A* = (*0::'a::lordered-ring matrix*)
  ⟨*proof*⟩

lemma *Rep-matrix-inject-exp*:
  shows (*A* = *B*) = (∀ *j i*. *Rep-matrix A j i* = *Rep-matrix B j i*)
  ⟨*proof*⟩

lemma *move-matrix-zero-le2*:
  assumes *j*: *0* ≤ *j* **and** *i*: *0* ≤ *i*
  shows (*0* = *move-matrix A j i*) = ((*0::('a::{order,zero}) matrix*) = *A*)
  ⟨*proof*⟩

lemma *Abs-matrix-inject3*:
  assumes *x*: *x* ∈ *matrix*
  and *y*: *y* ∈ *matrix*
  and *eq*: *Abs-matrix x* = *Abs-matrix y*
  shows *x* = *y*
  ⟨*proof*⟩

lemma *Abs-matrix-ne*:
  assumes *x*: $x \in$ *matrix*
  and *y*: $y \in$ *matrix*
  and *eq*: $x \neq y$
  shows *Abs-matrix* $x \neq$ *Abs-matrix* y
⟨*proof*⟩

lemma *singl-mat-zero*:
  assumes *singl-mat*: *singleton-matrix* $i\ j\ x = 0$
  shows $x = (0::'a::zero)$
⟨*proof*⟩

lemma *zero-impl-sing-mat-zero*:
  assumes *x-zero*: $x = (0::'a::zero)$
  shows *singleton-matrix* $i\ j\ x = 0$
  ⟨*proof*⟩

lemma *singl-mat-not-zero*:
  assumes *singl-mat*: *singleton-matrix* $i\ j\ x \neq 0$
  shows $x \neq (0::'a::zero)$
⟨*proof*⟩

The following lemma is the first one where we had to add the sortedness property of vectors.

The following lemmas will allow us to relate our definition of the zero matrix for the type *'a spmat*, i.e., *non-zero-spmat*, with the one of the zero matrix for type *'a matrix*.

lemma *sparse-row-vector-zero-impl-non-zero-spvec*:
  assumes *sp-r*: *sparse-row-vector* $A = 0$
  and *srtd*: *sorted-spvec* A
  shows *non-zero-spvec* $A = 0$
  ⟨*proof*⟩

lemma *sorted-sparse-matrix-cons2*:
  assumes *srtd*: *sorted-sparse-matrix* $(a \# b \# A)$
  shows *sorted-sparse-matrix* $(a \# A)$
  ⟨*proof*⟩

lemma *disj-matrices-A-minus-A*:
  assumes *A-not-zero*: $(A::'a::lordered\text{-}ring\ matrix) \neq 0$
  shows $\neg$ (*disj-matrices* $A\ (-\ A)$)
⟨*proof*⟩

lemma *sorted-sparse-matrix-cons*:
  assumes *srtd*: *sorted-sparse-matrix* $(a \# A)$
  shows *sorted-sparse-matrix* A
  ⟨*proof*⟩

The following lemma proves the equivalence between the zero matrix and our definition of zero as a sparse matrix. The lemma requires the sparse matrix to be sorted.

lemma *sparse-row-matrix-zero-eq-non-zero-spmat-zero*:
  assumes *sorted-sp-mat-A*: *sorted-sparse-matrix A*
  shows (*sparse-row-matrix A* = (*0*::*'a*::*lordered-ring matrix*))
  = (*non-zero-spmat A* = (*0*::*nat*))
  ⟨*proof*⟩

The computation of the bound is now made by means of a For loop, where the terminating condition is to get to a null matrix.

The process is similar to the one applied in the constructive version in the BPL, but using sparse matrices instead of matrices.

definition *local-bound-gen-spmat* ::
  ((*'a*::*lordered-ring*) *spmat* => *'a spmat*) => (*'a spmat*) => *nat* => *nat*
  where *local-bound-gen-spmat f A n*
  = *For* (λ *y*. *non-zero-spmat y* $\neq$ *0*) *f* (λ *y n*. *n* + (*1*::*nat*)) *A n*

definition *local-bound-spmat* ::
  ((*'a*::*lordered-ring*) *spmat* => *'a spmat*) => (*'a spmat*) => *nat*
  where *local-bound-spmat f A* $\equiv$ *local-bound-gen-spmat f A 0*

We also need a definition of the summation of the power series of a given matrix up to a given bound.

primrec *fin-sum-spmat*
  :: ((*'a*::*lordered-ring*) *spmat* => *'a spmat*) => (*'a spmat*) => *nat* => *'a spmat*
  where *fin-sum-spmat f A 0* = *A*
  | *fin-sum-spmat f A* (*Suc n*) = *add-spmat* ((*f*^(*Suc n*)) *A*, *fin-sum-spmat f A n*)

With the previous definitions we are ready to give the definition of α and Φ for sparse matrices.

definition α-*spmat* :: (*'a*::*lordered-ring*) *spmat* => *'a spmat*
  where α-*spmat A* = (*minus-spmat* (*pert-spmat-sort* (*hom-oper-spmat A*)))

The following definition is the one to be later compared to the one of Φ

definition Φ-*spmat* :: (*'a*::*lordered-ring*) *spmat* => *'a spmat*
  where Φ-*spmat A* = *fin-sum-spmat* α-*spmat A* (*local-bound-spmat* α-*spmat A*)

definition *example-zero* :: *int spmat*
  where *example-zero* == [(*3*::*nat*, [((*3*::*nat*), (*5*::*int*))])]

definition *example-one* :: *int spmat*
  where*example-one* ==
   (*0*::*nat*, (*0*::*nat*, *5*::*int*) # (*2*, − *9*) # (*6*, *8*) # [])
  # (*1*::*nat*, (*1*::*nat*, *4*::*int*) # (*2*, *0*) # (*8*, *8*) # [])
  # (*2*::*nat*, (*2*::*nat*, *6*::*int*) # (*3*, *7*) # (*9*, *23*) # [])

$(16::nat, (13::nat, -8::int)$ # $(19, 12)$ # [])
$(29::nat, (3::nat, \quad 5::int)$ # $(4, \quad 6)$ # $(7, 8)$ # [])
[]

definition *example-one′* :: *int spmat*
where *example-one′* == $(0::nat, (0::nat, 0::int)$ # $(2, 0)$ # $(6, 0)$ # [])
$(1::nat, (1::nat, 0::int)$ # $(2, 0)$ # $(8, 0)$ # [])
$(2::nat, (2::nat, 0::int)$ # $(3, 0)$ # $(9, 0)$ # [])
$(29::nat, (3::nat, 0::int)$ # $(4, 0)$ # $(7, 0)$ # [])
[]

lemma *alpha-execute-sparse*:
shows α (*sparse-row-matrix* (*A*::′*a*::*lordered-ring spmat*))
= *sparse-row-matrix* (α-*spmat A*)
⟨*proof*⟩

lemma *fun-matrix-add*:
shows ((*f*::′*a matrix* => ′*a matrix*) + *g*) (*A*::′*a*::*lordered-ring matrix*) = *f A* + *g A*
⟨*proof*⟩

lemma *nth-power-execute-sparse*:
fixes *f* :: ′*a matrix* => ′*a*::*lordered-ring matrix*
fixes *f-spmat* :: ′*a spmat* => ′*a spmat*
assumes *f-exec*: $\bigwedge A$. *f* (*sparse-row-matrix A*)
= *sparse-row-matrix* (*f-spmat A*)
shows (*f* ^ *n*) (*sparse-row-matrix A*)
= *sparse-row-matrix* ((*f-spmat*^*n*) *A*)
⟨*proof*⟩

corollary *alpha-nth-power-execute-sparse*:
shows (α ^ *n*) (*sparse-row-matrix A*)
= *sparse-row-matrix* ((α-*spmat*^*n*) *A*)
⟨*proof*⟩

lemma *fin-sum-alpha-up-to-n-execute-sparse*:
shows *fin-sum* α *n* (*sparse-row-matrix A*)
= *sparse-row-matrix* (*fin-sum-spmat* α-*spmat A n*)
⟨*proof*⟩

lemma *fin-sum-up-to-n-execute-sparse′*:
assumes *m-eq-n*: *m* = *n*
shows *fin-sum* α *m* (*sparse-row-matrix A*)
= *sparse-row-matrix* (*fin-sum-spmat* α-*spmat A n*)
⟨*proof*⟩

lemma *sorted-sparse-matrix-add-spmat*:
assumes *srtd-A*: *sorted-sparse-matrix A*
and *srtd-B*: *sorted-sparse-matrix B*

shows *sorted-sparse-matrix* (*add-spmat* (*A*, *B*))
⟨*proof*⟩

As we have proved, the *hom-oper-spmat* operation preserves order

lemma *sorted-spvec* (α*-spmat example-one*)
⟨*proof*⟩

lemma *sorted-spvec-alpha-spmat*:
assumes *srtd*: *sorted-spvec A*
shows *sorted-spvec* (α*-spmat A*)
⟨*proof*⟩

lemma *sorted-spvec-alpha-spmat-nth*:
assumes *srtd*: *sorted-spvec A*
shows *sorted-spvec* ((α*-spmat*ˆ*n*) *A*)
⟨*proof*⟩

lemma *sorted-spmat-alpha-spmat*:
shows *sorted-spmat* (α*-spmat A*)
⟨*proof*⟩

lemma *sorted-spmat-alpha-spmat-nth*:
assumes *srtd-A*: *sorted-spmat A*
shows *sorted-spmat* ((α*-spmat*ˆ*n*) *A*)
⟨*proof*⟩

lemma *sorted-sparse-matrix-alpha*:
assumes *ssm*: *sorted-sparse-matrix A*
shows *sorted-sparse-matrix* (α*-spmat A*)
⟨*proof*⟩

lemma *sorted-sparse-matrix-alpha-nth*:
assumes *ssm*: *sorted-sparse-matrix A*
shows *sorted-sparse-matrix* ((α*-spmat*ˆ*n*) *A*)
⟨*proof*⟩

37.2 Equivalence between the local nilpotency bound for sparse matrices and matrices.

lemma *local-bounded-func-*α*-matrix*:
shows *local-bounded-func* (α::'*a*::*lordered-ring matrix* => '*a matrix*)
⟨*proof*⟩

corollary α*-matrix-terminates*:
shows *terminates* (($\lambda y.\ y \neq 0$),
α::'*a*::*lordered-ring matrix* => '*a matrix*,
sparse-row-matrix A)
⟨*proof*⟩

lemma *local-bound-spmat*:
shows *local-bound* α (*sparse-row-matrix* *A*)
= (*LEAST* *n*. *sparse-row-matrix* (
(α-*spmat* ^ *n*) (*A*::'*a*::*lordered-ring spmat*)) = (*0*::'*a matrix*))
⟨*proof*⟩

lemma *local-bound-spmat-termin*:
shows *local-bound* α (*sparse-row-matrix* *A*) =
(*LEAST* *n*. *sparse-row-matrix* (
(α-*spmat* ^ *n*) (*A*::'*a*::*lordered-ring spmat*)) = (*0*::'*a matrix*))
⟨*proof*⟩

lemma *least-eq*:
assumes *eq*: ∀ *n*::*nat*. *f n* = *g n*
shows (*LEAST* *n*. *f n*) = (*LEAST* *n*. *g n*)
⟨*proof*⟩

lemma *LEAST-execute-sparse*:
assumes *ssm*: *sorted-sparse-matrix A*
shows (*LEAST* *n*. *sparse-row-matrix*
((α-*spmat* ^ *n*) (*A*::'*a*::*lordered-ring spmat*)) = (*0*::'*a matrix*)) =
(*LEAST* *n*. *non-zero-spmat*
((α-*spmat* ^ *n*) (*A*::'*a*::*lordered-ring spmat*)) = (*0*::*nat*))
⟨*proof*⟩

lemma *local-bound-alpha-spmat*:
assumes *ssm*: *sorted-sparse-matrix A*
shows *local-bound* α (*sparse-row-matrix* *A*)
= (*LEAST* *n*. *non-zero-spmat* (
(α-*spmat* ^ *n*) (*A*::'*a*::*lordered-ring spmat*)) = (*0*::*nat*))
⟨*proof*⟩

Make the simplification rule for *local-bound-gen*:

lemmas *local-bound-gen-spmat-simp* =
For-simp[*of* (λ *y*. *non-zero-spmat y* ≠ (*0*::*nat*)) - λ *y n*. *n*+(*1*::*nat*),
simplified local-bound-gen-spmat-def[*symmetric*]]

Now, we connect the neccesary termination of For with our termination condition, *local-bounded-func*.

Then, under the *local-bounded-func* premise the loop will be terminating.

lemma *local-bounded-func-alpha-impl-terminates-loop*:
local-bounded-func (α::'*a matrix* => '*a matrix*)
= (∀ *x*. *terminates* (λ *y*. *y* ≠ *0*, α, (*x*::'*a*::*lordered-ring matrix*)))
⟨*proof*⟩

The following lemma transfers the condition of termination from matrices to sparse matrices.

lemma *local-bounded-func-alpha-impl-terminates-loop-spmat*:

assumes *ssm*: *sorted-sparse-matrix* (A::$'a$::*lordered-ring spmat*)
shows *terminates* (λy. $y \neq 0$, α, *sparse-row-matrix* A)
= *terminates* (λy. *non-zero-spmat* $y \neq$ (0::*nat*), α-*spmat*, A)
⟨*proof*⟩

lemma *local-bounded-func-impl-terminates-spmat-loop*:
assumes *ssm*: *sorted-sparse-matrix* (A::$'a$::*lordered-ring spmat*)
shows *terminates* (λ y. *non-zero-spmat* $y \neq$ (0::*nat*), α-*spmat*, A)
⟨*proof*⟩

lemma *LEAST-local-bound-non-zero-spmat*:
assumes *is-zero*: *non-zero-spmat* $A = 0$
shows (*LEAST* n::*nat*. *non-zero-spmat* ((α-*spmat* ^ n) A) = (0::*nat*)) = 0
⟨*proof*⟩

lemma *local-bound-gen-spmat-correct*:
terminates (λ y. *non-zero-spmat* $y \neq$ (0::*nat*), α-*spmat*, A)
$\Longrightarrow$ *local-bound-gen-spmat* α-*spmat* A m
= m + (*LEAST* n::*nat*. *non-zero-spmat* ((α-*spmat*^n) A) = 0)
⟨*proof*⟩

The following lemma exactly represents the difference between our old definitions, with which we proved the BPL, and the new ones, from which we are trying to generate code; under the termination premise, both *LEAST* n. (f ^ n) x = (0::$'b$), the old definition of local nilpotency, and *local-bound* f x, the loop computing the lower bound, are equivalent

Whereas *LEAST* n. (f ^ n) x = (0::$'b$) does not have a computable interpretation, *local-bound* f x does have it, and code can be generated from it.

lemma *local-bound-spmat-correct*:
terminates (λ y. *non-zero-spmat* $y \neq$ (0::*nat*), α-*spmat*, A) $\Longrightarrow$
local-bound-spmat α-*spmat* A = (*LEAST* n::*nat*. *non-zero-spmat* ((α-*spmat*^n) A) = 0)
⟨*proof*⟩

lemma *local-bounded-func-impl-local-bound-spmat-is-Least*:
assumes *ssm*: *sorted-sparse-matrix* (A::$'a$::*lordered-ring spmat*)
shows *local-bound-spmat* α-*spmat* A
= (*LEAST* n::*nat*. *non-zero-spmat* ((α-*spmat*^n) A) = 0)
⟨*proof*⟩

lemma *local-bound-eq-local-bound-spmat*:
assumes *ssm*: *sorted-sparse-matrix* (A::$'a$::*lordered-ring spmat*)
shows *local-bound-spmat* α-*spmat* A
= *local-bound* α (*sparse-row-matrix* A)
⟨*proof*⟩

lemma *phi-execute-sparse*:

assumes *ssm*: *sorted-sparse-matrix* (A::$'a$::*lordered-ring spmat*)
shows *sparse-row-matrix* (Φ-*spmat* A)
= (Φ::$'a$::*lordered-ring matrix* => $'a$ *matrix*) (*sparse-row-matrix* A)
⟨*proof*⟩

37.3 Some examples of code generation.

It must be noted that these examples execute the definitions over sparse matrices, and are carried out by means of Haftmann',s code generation tool.

Consequently, they are not computed directly over the definitions on $'a$ *matrix*, which were the ones proved to be an instance of the BPL.

Later we will use the HCL to carry out computations with the definitions over the type $'a$ *matrix*.

The value operator is rather slow even for easy examples

value *example-zero*
value α-*spmat example-zero*

definition *foos*:: *int spmat*
where *foos* == *transpose-spmat* [(*3*::*nat*, [((*4*::*nat*), (*5*::*int*))])]

definition *foos1*:: *int spmat*
where *foos1* == (α-*spmat*ˆ(*1*::*nat*)) *example-zero*

definition *foos2*:: *int spmat*
where*foos2* == (α-*spmat*ˆ(*2*::*nat*)) *example-zero*

definition *foos3*:: *int spmat*
where *foos3* == (α-*spmat*ˆ(*3*::*nat*)) *example-zero*

definition *foos4*:: *int spmat*
where *foos4* == (α-*spmat*ˆ(*4*::*nat*)) *example-zero*

definition *foos5*:: *int spmat*
where *foos5* == (α-*spmat*ˆ(*5*::*nat*)) *example-zero*

definition *foos6*:: *int spmat*
where *foos6* == (α-*spmat*ˆ(*6*::*nat*)) *example-zero*

definition *foos7*:: *int spmat*
where *foos7* == (α-*spmat*ˆ(*7*::*nat*)) *example-zero*

definition *foos-local-bound* :: *nat*
where *foos-local-bound* == *local-bound-spmat* α-*spmat example-zero*

definition *foos-Phi*:: *int spmat*
where *foos-Phi* == Φ-*spmat example-zero*

definition *goos*:: *int spmat*
where *goos* == *transpose-spmat example-one*

definition *goos1*:: *int spmat*
where *goos1* == (α*-spmat*^(*1*::*nat*)) *example-one*

definition *goos2*:: *int spmat*
where *goos2* == (α*-spmat*^(*2*::*nat*)) *example-one*

definition *goos3*:: *int spmat*
where *goos3* == (α*-spmat*^(*3*::*nat*)) *example-one*

definition *goos4*:: *int spmat*
where *goos4* == (α*-spmat*^(*4*::*nat*)) *example-one*

definition *goos5*:: *int spmat*
where *goos5* == (α*-spmat*^(*5*::*nat*)) *example-one*

definition *goos6*:: *int spmat*
where *goos6* == (α*-spmat*^(*6*::*nat*)) *example-one*

definition *goos7*:: *int spmat*
where *goos7* == (α*-spmat*^(*7*::*nat*)) *example-one*

definition *goos30*:: *int spmat*
where *goos30* == (α*-spmat*^(*local-bound-spmat* α*-spmat example-one*)) *example-one*

We can also compare a matrix to the zero matrix

definition *goos30-eq-0*:: *bool*
where *goos30-eq-0* = (*non-zero-spmat* ((α*-spmat*^(*local-bound-spmat*
α*-spmat example-one*)) *example-one*) = (*0*::*nat*))

definition *goos-local-bound*::*nat*
where *goos-local-bound* = *local-bound-spmat* α*-spmat example-one*

definition *goos-Phi*:: *int spmat*
where *goos-Phi* == Φ*-spmat example-one*

definition *blaa* :: *int spmat*
where *blaa* == *hom-oper-spmat example-one*

definition *blaa-square*:: *int spmat*
where *blaa-square* == (*hom-oper-spmat*^*2*) *example-one*

definition *blaa-three*:: *int spmat*
where *blaa-three* == (*hom-oper-spmat*^*3*) *example-one*

definition *blaas*:: *int spmat*

where *blaas* == *pert-spmat example-one*

definition *blaas-square*:: *int spmat*
where *blaas-square* == (*pert-spmat*^*2*) *example-one*

The following are just examples of the code generated from the Isabelle definitions over sparse matrices in order to check that they are compatible with the code generation facility.

export-code *blaa blaa-square blaa-three blaas blaas-square*
goos goos1 goos2 goos3 goos4 goos5 goos6 goos7 goos30
goos30-eq-0
goos-local-bound goos-Phi
foos foos1 foos2 foos3 foos4 foos5 foos6 foos7
foos-local-bound foos-Phi
in *SML* **module-name** *example-Bicomplex*
file *example-Bicomplex.ML*

⟨*ML*⟩

37.4 Some lemmas to get code generation in Obua's way

In this Section we will pay attention to get to execute the definitions over which we have proved the instance before, for instance, Φ.

Before executing them, we have to provide the HXL with the lemmas that link such definitions with the definitions over sparse matrices, such as α-*spmat*.

Some preliminar results are also required to get executions in the HCL.

lemma *neg-int*: *neg* ((*number-of x*)::*int*) = *neg x*
⟨*proof*⟩

lemma *split-apply*: *split f* (*a*, *b*) = *f a b*
⟨*proof*⟩

definition *add-3*:: *nat* => *nat*
where *add-3 x* = *x* + (*3*::*nat*)

lemma *funpow-rec*: *f*^*n* = (*if n* = (*0*::*nat*) *then id else f o* (*f*^(*n* − *1*)))
⟨*proof*⟩

lemma *funpow-rec-calc*:
f^(*number-of w*)
= (*if* (*number-of w* = (*0*::*nat*)) *then id else f o* (*f*^((*number-of w*) − *1*)))
⟨*proof*⟩

lemma *comp-calc*: (*f o g*) *x* = (*f* (*g x*))
⟨*proof*⟩

lemma *id-calc*: *id* $x = x$
⟨*proof*⟩

lemmas *compute-iterative* =
add-3-def
funpow-rec-calc
comp-calc
id-calc
compute-hol
compute-numeral
NatBin.mod-nat-number-of
NatBin.div-nat-number-of
NatBin.nat-number-of

⟨*ML*⟩

lemma *fin-sum-spmat-rec*:
fin-sum-spmat f A n =
(*if n* = *0 then A else add-spmat* (((*f*ˆ*n*) *A*), *fin-sum-spmat f A* (*n* − *1*)))
⟨*proof*⟩

lemma *fin-sum-spmat-rec-calc*:
fin-sum-spmat f A (*number-of n*) =
(*if* (*number-of n* = (*0*::*nat*)) *then A*
else add-spmat (((*f*ˆ(*number-of n*)) *A*),
fin-sum-spmat f A ((*number-of n*) − *1*)))
⟨*proof*⟩

lemmas *compute-phi-sparse* =

— The set of lemmas we use from this theory
— The lemma computing Φ
sym [*OF phi-execute-sparse*]
Φ*-spmat-def*

— The lemmas computing the bound
sym [*OF local-bound-eq-local-bound-spmat*]
local-bound-spmat-def
local-bound-gen-spmat-def
For-simp
non-zero-spmat.simps
non-zero-spvec.simps

— The lemmas dealing with the termination predicate
terminates-rec

— The lemmas computing the finite sum of sparse matrices
fin-sum-spmat-rec
fin-sum-spmat-rec-calc

— The lemma computing addition of matrices
add-spmat.simps

— The lemmas computing function powers
funpow-rec-calc comp-calc
id-calc
alpha-nth-power-execute-sparse

— The lemmas computing alpha
alpha-execute-sparse
α-spmat-def

— The lemmas computing the differential
diff-matrix1-execute-sparse
sym [OF sparse-row-matrix-differ-spmat]
sparse-row-matrix-differ-spmat-differ-spmat-sort
differ-spmat-sort.simps

— The lemmas computing the homotopy operator
hom-oper-matrix1-execute-sparse-simp
hom-oper-spmat.simps

— The lemmas computing the perturbation operator
pert-matrix1-execute-sparse-simp
pert-spmat-sort.simps

— Lemma for the minus operation over matrices
minus-spmat.simps

— Some aditional library lemmas also required
sort-spvec.simps
insert-sorted.simps
compute-hol
compute-numeral
NatBin.mod-nat-number-of
NatBin.div-nat-number-of
NatBin.nat-number-of
neg-int
sorted-sp-simps
sparse-row-matrix-arith-simps
map.simps
split-apply

— And of course, some matrices:
example-one-def
example-one'-def
example-zero-def

The BARRAS mode seems to be a more flexible way for code generation, where, for instance, lambda abstractions are allowed in the definitions.

Apparently is also slower.

⟨*ML*⟩

Different ways to compute the local bound:

⟨*ML*⟩

Different ways to compute the result of applying Φ to the matrix *example-one*.

⟨*ML*⟩

Some other experiments making use of the SML mode of the HCL, instead of the BARRAS mode.

⟨*ML*⟩

37.5 Code generation formt the series Ψ

The following results will permit us to apply code generation also from the series Ψ.

They are similar to the ones already proved for α and Φ.

The following definition is the one to be compared with β.

definition *β-spmat* :: (*'a::lordered-ring*) *spmat* => *'a spmat*
  where *β-spmat A* = (*minus-spmat* (*hom-oper-spmat* (*pert-spmat-sort A*)))

The following definition is the one to be later compared with the one of Ψ.

definition *Ψ-spmat* :: (*'a::lordered-ring*) *spmat* => *'a spmat*
  where *Ψ-spmat A* == *fin-sum-spmat β-spmat A* (*local-bound-spmat β-spmat A*)

lemma *beta-equiv*: $\beta = (\lambda A. - (h\ (\delta\ A)))$
  ⟨*proof*⟩

lemma *sorted-spvec-beta-spmat*:
  assumes *srtd*: *sorted-spvec A*
  shows *sorted-spvec* (*β-spmat A*)
  ⟨*proof*⟩

lemma *sorted-spvec-beta-spmat-nth*:
  assumes *srtd*: *sorted-spvec A*
  shows *sorted-spvec* ((*β-spmat^n*) *A*)
⟨*proof*⟩

lemma *sorted-spmat-beta-spmat*:
  shows *sorted-spmat* (*β-spmat A*)
  ⟨*proof*⟩

lemma *sorted-spmat-beta-spmat-nth*:
  assumes *srtd-A*: *sorted-spmat A*

shows *sorted-spmat* ((β*-spmat*^*n*) *A*)
⟨*proof*⟩

lemma *sorted-sparse-matrix-beta*:
assumes *ssm*: *sorted-sparse-matrix A*
shows *sorted-sparse-matrix* (β*-spmat A*)
⟨*proof*⟩

lemma *sorted-sparse-matrix-beta-nth*:
assumes *ssm*: *sorted-sparse-matrix A*
shows *sorted-sparse-matrix* ((β*-spmat*^*n*) *A*)
⟨*proof*⟩

lemma *beta-execute-sparse*:
β (*sparse-row-matrix A*) = *sparse-row-matrix* (β*-spmat A*)
⟨*proof*⟩

corollary *beta-nth-power-execute-sparse*:
shows (β ^ *n*) (*sparse-row-matrix A*)
= *sparse-row-matrix* ((β*-spmat*^*n*) *A*)
⟨*proof*⟩

lemma *fin-sum-beta-up-to-n-execute-sparse*:
shows *fin-sum* β *n* (*sparse-row-matrix A*)
= *sparse-row-matrix* (*fin-sum-spmat* β*-spmat A n*)
⟨*proof*⟩

lemma *local-bound-spmat-beta*:
assumes *bounded-*β: *local-bounded-func* (β::'*a*::*lordered-ring matrix* => '*a matrix*)
shows *local-bound* β (*sparse-row-matrix* (*A*::'*a*::*lordered-ring spmat*))
= (*LEAST n*. *sparse-row-matrix* ((β*-spmat* ^ *n*) *A*) = *0*)
⟨*proof*⟩

lemma *LEAST-execute-sparse-beta*:
assumes *ssm*: *sorted-sparse-matrix A*
shows (*LEAST n*. *sparse-row-matrix* ((β*-spmat* ^ *n*)
(*A*::'*a*::*lordered-ring spmat*))
= (*0*::'*a matrix*)) =
(*LEAST n*. *non-zero-spmat* ((β*-spmat* ^ *n*) *A*) = (*0*::*nat*))
⟨*proof*⟩

lemma *local-bound-beta-spmat*:
assumes *bounded-*β: *local-bounded-func*
(β::'*a*::*lordered-ring matrix* => '*a matrix*)
and *ssm*: *sorted-sparse-matrix A*
shows *local-bound* β (*sparse-row-matrix A*)
= (*LEAST n*. *non-zero-spmat* ((β*-spmat* ^ *n*)
(*A*::'*a*::*lordered-ring spmat*)) = (*0*::*nat*))

$\langle proof \rangle$

We make the simplification rule for *local-bound-gen*:

Now, we connect the neccesary termination of For with our termination condition, *local-bounded-func*.

Then, under the *local-bounded-func* premise the loop will be terminating.

lemma *local-bounded-func-beta-impl-terminates-loop*:
local-bounded-func (β::$'a$ *matrix* => $'a$ *matrix*)
= ($\forall$ *x*. *terminates* (λ *y*. $y \neq 0$, β, (*x*::$'a$::*lordered-ring matrix*)))
$\langle proof \rangle$

The following lemma transfers the condition of termination from matrices to sparse matrices.

lemma *local-bounded-func-beta-impl-terminates-loop-spmat*:
assumes *ssm*: *sorted-sparse-matrix* (*A*::$'a$::*lordered-ring spmat*)
shows *terminates* (λy. $y \neq 0$, β, *sparse-row-matrix A*)
= *terminates* (λy. *non-zero-spmat* $y \neq$ (*0*::*nat*), β*-spmat*, *A*)
$\langle proof \rangle$

lemma *local-bounded-func-impl-terminates-beta-spmat-loop*:
assumes *ssm*: *sorted-sparse-matrix* (*A*::$'a$::*lordered-ring spmat*)
and *locbf*: *local-bounded-func* (β::$'a$::*lordered-ring matrix* => $'a$ *matrix*)
shows *terminates* (λ *y*. *non-zero-spmat* $y \neq$ (*0*::*nat*), β*-spmat*, *A*)
$\langle proof \rangle$

lemma *LEAST-local-bound-beta-non-zero-spmat*:
assumes *is-zero*: *non-zero-spmat A* = *0*
shows (*LEAST n*::*nat*. *non-zero-spmat* ((β*-spmat* ^ *n*) *A*)
= (*0*::*nat*)) = *0*
$\langle proof \rangle$

lemma *local-bound-gen-spmat-beta-correct*:
shows *terminates* (λ *y*. *non-zero-spmat* $y \neq$ (*0*::*nat*), β*-spmat*, *A*)
$\Longrightarrow$ *local-bound-gen-spmat* β*-spmat A m*
= *m* + (*LEAST n*::*nat*. *non-zero-spmat* ((β*-spmat*^*n*) *A*) = *0*)
$\langle proof \rangle$

The following lemma exactly represents the difference between our old definitions, with which we proved the BPL, and the new ones, from which we are trying to generate code; under the termination premise, both *LEAST n*. (*f* ^ *n*) *x* = (*0*::$'b$), the old definition of local nilpotency, and *local-bound f x*, the loop computing the lower bound, are equivalent

Whereas *LEAST n*. (*f* ^ *n*) *x* = (*0*::$'b$) does not have a computable interpretation, *local-bound f x* does have it, and code can be generated from it.

lemma *local-bound-beta-spmat-correct*:
terminates (λ *y*. *non-zero-spmat y* $\neq$ (*0*::*nat*), β*-spmat*, *A*) $\Longrightarrow$
local-bound-spmat β*-spmat A*
= (*LEAST n*::*nat*. *non-zero-spmat* ((β*-spmat*ˆ*n*) *A*) = *0*)
⟨*proof*⟩

lemma *nrows-hom-oper-pert-decre*:
assumes *A-not-null*: (*A*::′*a*::*lordered-ring matrix*) $\neq$ *0*
shows *nrows* ((− (*hom-oper-matrix1* ∘ *pert-matrix1*)) *A*) < *nrows A*
(**is** *nrows* ((− *?f*) *A*) < *nrows A*)
⟨*proof*⟩

corollary *nrows-beta-decre*:
assumes *A-not-null*: (*A*::′*a*::*lordered-ring matrix*) $\neq$ *0*
shows *nrows* (β *A*) < *nrows A*
⟨*proof*⟩

lemma *local-bounded-func-*β*-matrix*:
shows *local-bounded-func* (β::′*a*::*lordered-ring matrix* => ′*a matrix*)
⟨*proof*⟩

lemma *local-bounded-func-beta-impl-local-bound-spmat-is-Least*:
assumes *ssm*: *sorted-sparse-matrix* (*A*::′*a*::*lordered-ring spmat*)
shows *local-bound-spmat* β*-spmat A*
= (*LEAST n*::*nat*. *non-zero-spmat* ((β*-spmat*ˆ*n*) *A*) = *0*)
⟨*proof*⟩

lemma *local-bound-eq-local-bound-spmat-beta*:
assumes *ssm*: *sorted-sparse-matrix* (*A*::′*a*::*lordered-ring spmat*)
shows *local-bound-spmat* β*-spmat A* = *local-bound* β (*sparse-row-matrix A*)
⟨*proof*⟩

lemma *psi-execute-sparse*:
assumes *ssm*: *sorted-sparse-matrix* (*A*::′*a*::*lordered-ring spmat*)
shows *sparse-row-matrix* (Ψ*-spmat A*)
= (Ψ::′*a*::*lordered-ring matrix* => ′*a matrix*) (*sparse-row-matrix A*)
⟨*proof*⟩

lemmas *compute-phi-psi-sparse* =

— The set of lemmas we use from this theory
— The lemma computing Φ
sym [*OF phi-execute-sparse*]
Φ*-spmat-def*

— The lemma computing Ψ
sym [*OF psi-execute-sparse*]
Ψ*-spmat-def*

— The lemma computing the bound for α
sym [*OF local-bound-eq-local-bound-spmat*]

— The lemma computing the bound for β
sym [*OF local-bound-eq-local-bound-spmat-beta*]

— The generic lemmas computing the bound
local-bound-spmat-def
local-bound-gen-spmat-def
For-simp
non-zero-spmat.simps
non-zero-spvec.simps

— The lemmas dealing with the termination predicate
terminates-rec

— The lemmas computing the finite sum of sparse matrices
fin-sum-spmat-rec
fin-sum-spmat-rec-calc

— The lemma computing addition of matrices
add-spmat.simps

— The lemmas computing function powers
funpow-rec-calc
comp-calc
id-calc
alpha-nth-power-execute-sparse
beta-nth-power-execute-sparse

— The lemmas computing beta
alpha-execute-sparse
α*-spmat-def*

— The lemmas computing beta
beta-execute-sparse
β*-spmat-def*

— The lemmas computing the differential
diff-matrix1-execute-sparse
sym [*OF sparse-row-matrix-differ-spmat*]
sparse-row-matrix-differ-spmat-differ-spmat-sort
differ-spmat-sort.simps

— The lemmas computing the homotopy operator
hom-oper-matrix1-execute-sparse-simp
hom-oper-spmat.simps

— The lemmas computing the perturbation operator

pert-matrix1-execute-sparse-simp
pert-spmat-sort.simps

— Lemma for the minus operation over matrices
minus-spmat.simps

— Some aditional library lemmas also required
sort-spvec.simps
insert-sorted.simps
compute-hol
compute-numeral
NatBin.mod-nat-number-of
NatBin.div-nat-number-of
NatBin.nat-number-of
neg-int
sorted-sp-simps
sparse-row-matrix-arith-simps
map.simps
split-apply

— And of course, some matrices:
example-one-def
example-one'-def
example-zero-def

The BARRAS mode seems to be a more flexible way for code generation, where, for instance, lambda abstractions are allowed in the definitions.

Apparently is also slower.

⟨*ML*⟩

Different ways to compute the local bound:

⟨*ML*⟩

Different ways to compute the result of applying Φ to the matrix *example-one*.

⟨*ML*⟩

lemma *funpow-equiv*:
shows $(g \circ f)\hat{}(Suc\ n) = g \circ ((f \circ g)\hat{}n) \circ f$
⟨*proof*⟩

lemma *f-id*: $f \circ id = f$
⟨*proof*⟩

lemma *fun-Compl-equiv*: $-(f \circ g) = (-f) \circ g$
⟨*proof*⟩

lemma *funpow-zip-next*:
$f\hat{}(n + 1) = f\hat{}n \circ f$

⟨*proof*⟩

code-datatype *sparse-row-vector sparse-row-matrix*

declare *zero-matrix-def* [*code func del*]

lemmas [*code func*] = *sparse-row-vector-empty* [*symmetric*]

declare *plus-matrix-def* [*code func del*]
lemmas [*code func*] = *sparse-row-add-spmat* [*symmetric*]
lemmas [*code func*] = *sparse-row-vector-add* [*symmetric*]

term *0*::*'a*::*zero matrix*

export-code
0::*'a*::*zero matrix*
op +::(*'a*::{*plus*,*zero*} *matrix* => *'a matrix* => *'a matrix*)
in *Haskell* **file** −

declare *pert-matrix1-def* [*code func del*]

lemmas [*code func*] =
pert-matrix1-execute-sparse-simp
pert-spmat-sort.simps

declare *hom-oper-matrix1-def* [*code func del*]

lemmas [*code func*] =
hom-oper-matrix1-execute-sparse-simp
hom-oper-spmat.simps

declare *fun-Compl-def* [*code func del*]

lemmas [*code func*] = *fun-Compl-def*

As far as alpha is defined for multiple types, we have to somehow restrict its definition to the appropiate one

Thus, we first delete its definition:

lemma [*code func*, *code func del*]: $\alpha = \alpha$ ⟨*proof*⟩

Then we provide a new definition for it with the appropriate data type

definition *alpha-matrix* ::
'a::*lordered-ring matrix* ⇒ *'a matrix*
where [*code post*, *code func del*]: *alpha-matrix* = α

Then we add the inline tag that will make our definition display each time we use alpha

lemmas [*code inline*] = *alpha-matrix-def* [*symmetric*]

And finally we make use of the required lemmas for code generation

lemmas [*code func*] = *alpha-execute-sparse* [*folded alpha-matrix-def*]
lemmas [*code func*] = α-*spmat-def*

declare Φ-*def* [*code func del*]

At the following lemma we cannot go further using code generation's tool by Haftmann, since we have a conditional rule, not an equation.

Contrarily, HCL can execute the premises, and succeed with the computation.

definition *foox* = (α :: *'a::lordered-ring matrix* $\Rightarrow$ -) *o* α

code-thms *foox*

export-code
  0::'a::zero matrix
  op +::(*'a::{plus,zero} matrix* => *'a matrix* => *'a matrix*)
  pert-matrix1
  op − ::*'a::lordered-ring* => *'a* => *'a*
  hom-oper-matrix1
  alpha-matrix::(*'a::lordered-ring matrix* => *'a matrix*)
  in *Haskell* **file** −

end