(Extended) Interval Analysis

Achim D. Brucker   Amy Stell

January 22, 2024

Department of Computer Science
University of Exeter
Exeter, UK
{a.brucker,a.stell}@exeter.ac.uk

Abstract

Interval analysis (also called interval arithmetic) is a well known mathematical technique to analyse or mitigate rounding errors or measurement errors. Thus, it is promising to integrate interval analysis into program verification environments. Such an integration is not only useful for the verification of numerical algorithms: the need to ensure that computations stay within certain bounds is common. For example to show that computations stay within the hardware bounds of a given number representation.

Another application is the verification of cyber-physical systems, where a discretised implementation approximates a system described in physical quantities expressed using perfect mathematical reals, and perfect ordinary differential equations.

In this AFP entry, we formalise extended interval analysis, including the concept of inclusion isotone (or inclusion isotonic) (extended) interval analysis. The main result is the formal proof that interval-splitting converges for Lipschitz-continuous interval isotone functions. From pragmatic perspective, we provide the datatypes and theory required for integrating interval analysis into other formalisations and applications.

Keywords: Extended Interval Analysis, Formalising Mathematics, Isabelle/HOL

Contents

1 Introduction

Interval analysis [4] in general and, in particular, inclusion isotone extended interval arithmetic [5] are well known mathematical techniques to analyse or mitigate rounding errors or measurement errors. Thus, it is promising to integrate interval analysis into program verification environments. Such an integration is not only useful for the verification of numerical algorithms: the need to ensure that computations stay within certain bounds is common. For example to show that computations stay within the hardware bounds of a given number representation.

Another application is the verification of cyber-physical systems, were a discretised implementation approximates a system described in physical quantities expressed using perfect mathematical reals, and perfect ordinary differential equations. Moreover, first applications of interval analysis to the security analysis of neural networks are promising [6, 3] and, when combined with existing works of verifying neural networks in Isabelle [2] provide a verification approach using a "correct-by-construction" verification tool for neural networks.

In this AFP entry, we formalise extended interval analysis, including the concept of inclusion isotone (extended) interval analysis. The main result is the formal proof that interval-splitting converges for Lipschitz-continuous inclusion isotone functions. From pragmatic perspective, we provide the datatypes and theory required for integrating interval analysis into other formalisations and applications. In more detail, our contributions are:

1. a conservative formalisation of (extended) interval arithmetic in Isabelle/HOL, including inclusion isotonicity;

2. we formally prove that interval-splitting converges for Lipschitz-continuous inclusion isotone functions;

From an end-user's perspective, the main entry points into this session are the following three theories:

- `Interval_Analysis` (Chapter 10): This theory provides interval analysis over standard types such as real or integer. All operations work over (closed) intervals.

- `Extended_Interval_Analysis` (Chapter 12): This theory provides extended interval analysis over the type extended reals. All operations work over (closed) intervals.

- `Extended_Multi_Interval_Analysis` (Chapter 14): This theory provides extended multi-interval analysis over the type extended reals. All operations work over multi-intervals, i.e., lists of (closed) intervals.

The following publication [1] gives a high-level overview of this AFP entry:

> A. D. Brucker, T. Cameron-Burke, and A. Stell. Formally verified interval arithmetic and its application to program verification. In 13th IEEE/ACM International Conference on Formal Methods in Software Engineering (FormaliSE 2024). IEEE, 2024.

The rest of this document is automatically generated from the formalisation in Isabelle/HOL, i.e., all content is checked by Isabelle. Overall, the structure of this document follows the theory dependencies (see Figure 1.1).

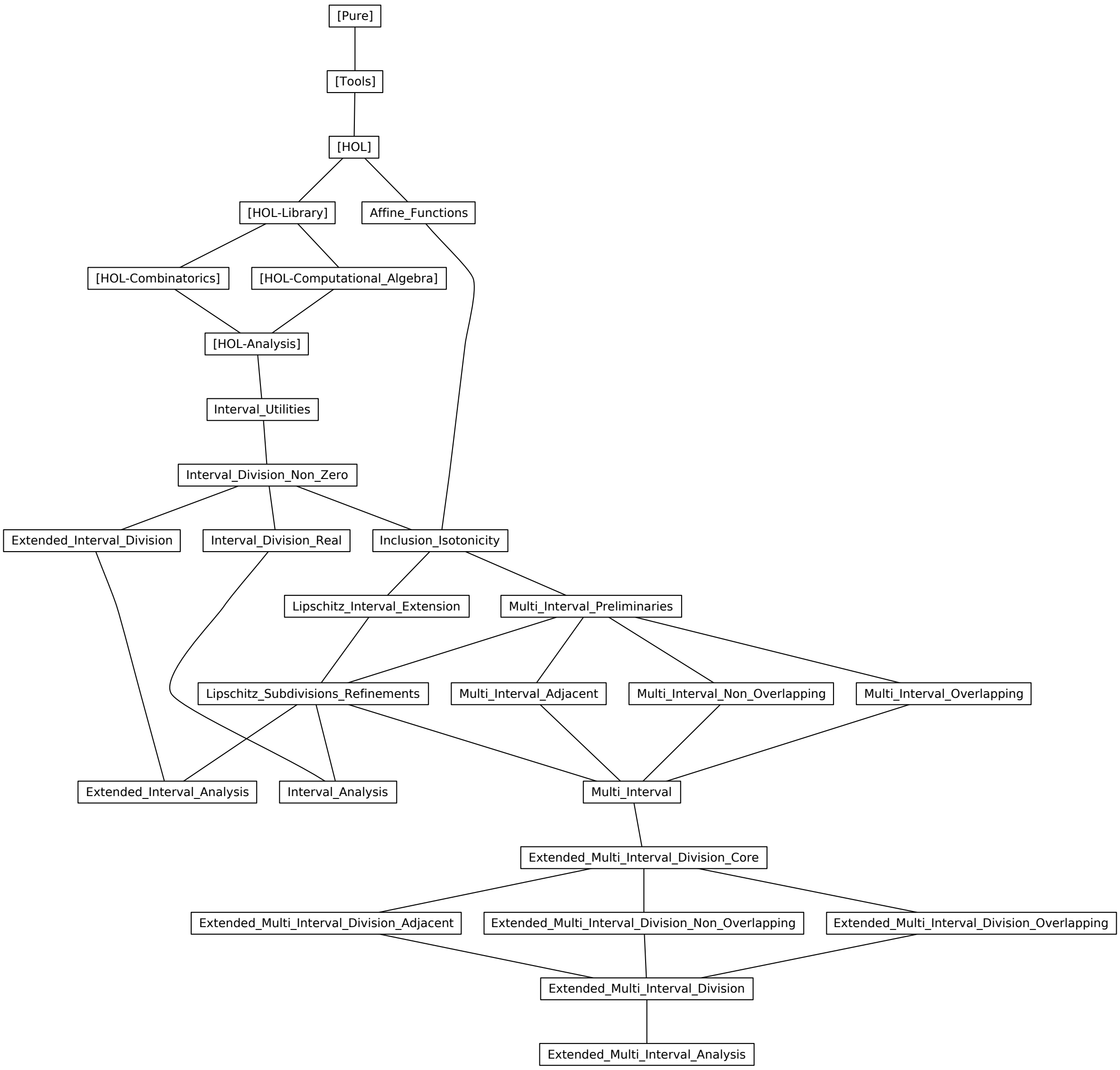

Figure 1.1: The Dependency Graph of the Isabelle Theories.

Generated Sessions

2 Interval Utilities (Interval_Utilities)

theory
Interval_Utilities
imports
HOL−Library.Interval
HOL−Analysis.Analysis
HOL−Library.Interval_Float
begin

2.1 Preliminaries

lemma *compact_set_of*:
fixes *X*::‹′*a*::{*preorder*,*topological_space*, *ordered_euclidean_space*} *interval*›
shows ‹*compact* (*set_of X*)›
⟨*proof*⟩

lemma *bounded_set_of*:
fixes *X*::‹′*a*::{*preorder*,*topological_space*, *ordered_euclidean_space*} *interval*›
shows ‹*bounded* (*set_of X*)›
⟨*proof*⟩

lemma *compact_img_set_of*:
fixes *X* :: ‹*real interval*› **and** *f*::‹*real* ⇒ *real*›
assumes ‹*continuous_on* (*set_of X*) *f*›
shows ‹*compact* (*f* ‘ *set_of X*)›
⟨*proof*⟩

lemma *sup_inf_dist_bounded*:
fixes *X*::‹*real set*›
shows ‹*bdd_below X* ⟹ *bdd_above X* ==> ∀ *x* ∈ *X*. ∀ *x*′ ∈ *X*. *dist x x*′ ≤ *Sup X* − *Inf X*›
⟨*proof*⟩

lemma *set_of_nonempty*[*simp*]: ‹*set_of X* ≠ {}›
⟨*proof*⟩

lemma *lower_in_interval*[*simp*]:‹*lower X* $\in_i$ *X*›
⟨*proof*⟩

lemma *upper_in_interval*[*simp*]:‹*upper X* $\in_i$ *X*›
⟨*proof*⟩

lemma *bdd_below_set_of*: ‹*bdd_below* (*set_of X*)›
⟨*proof*⟩

lemma *bdd_above_set_of*: ‹*bdd_above* (*set_of X*)›
⟨*proof*⟩

lemma *closed_set_of*: ‹*closed* (*set_of* (*X*::*real interval*))›
 ⟨*proof*⟩

lemma *set_f_nonempty*: ‹*f* ‘ *set_of X* ≠ {}›
 ⟨*proof*⟩

lemma *interval_linorder_case_split*[*case_names LeftOf Including RightOf*]:
assumes ‹(*upper x* < *c* ⟹ *P* (*x*::(′*a*::*linorder interval*)))›
 ‹(*c* $\in_i$ *x* ⟹ *P x*)›
 ‹(*c* < *lower x* ⟹ *P x*)›
 shows‹ *P x*›
⟨*proof*⟩

lemma *foldl_conj_True*:
‹*foldl* (∧) *x xs* = *list_all* (λ *e*. *e* = *True*) (*x*#*xs*)›
 ⟨*proof*⟩

lemma *foldl_conj_set_True*:
‹*foldl* (∧) *x xs* = (∀ *e* ∈ *set* (*x*#*xs*) . *e* = *True*)›
 ⟨*proof*⟩

2.2 Interval Bounds and Set Conversion

lemma *sup_set_of*:
 fixes *X* :: ′*a*::{*conditionally_complete_lattice*} *interval*
 shows *Sup* (*set_of X*) = *upper X*
 ⟨*proof*⟩

lemma *inf_set_of*:
 fixes *X* :: ′*a*::{*conditionally_complete_lattice*} *interval*
 shows *Inf* (*set_of X*) = *lower X*
 ⟨*proof*⟩

lemma *inf_le_sup_set_of*:
 fixes *X* :: ′*a*::{*conditionally_complete_lattice*} *interval*
 shows*Inf* (*set_of X*) ≤ *Sup* (*set_of X*)
 ⟨*proof*⟩

lemma *in_bounds*: ‹*x* $\in_i$ *X* ⟹ *lower X* ≤ *x* ∧ *x* ≤ *upper X*›
 ⟨*proof*⟩

lemma *lower_bounds*[*simp*]:
 assumes ‹*L* ≤ *U*›
 shows ‹*lower* (*Interval*(*L*,*U*)) = *L* ›
 ⟨*proof*⟩

lemma *upper_bounds* [*simp*]:
 assumes ‹*L* ≤ *U*›
 shows ‹*upper* (*Interval*(*L*,*U*)) = *U*›
 ⟨*proof*⟩

lemma *lower_point_interval*[*simp*]: ‹*lower* (*Interval* (*x*,*x*)) = *x*›
 ⟨*proof*⟩

lemma *upper_point_interval*[*simp*]: ‹*upper* (*Interval* (*x*,*x*)) = *x*›
⟨*proof*⟩

lemma *map2_nth*:
assumes ‹*length xs* = *length ys*›
and ‹*n* < *length xs*›
shows ‹(*map2 f xs ys*)!*n* = *f* (*xs*!*n*) (*ys*!*n*)›
⟨*proof*⟩

lemma *map_set*: ‹*a* ∈ *set* (*map f X*) ⟹ (∃ *x* ∈ *set X* . *f x* = *a*)›
⟨*proof*⟩
lemma *map_pair_set_left*: ‹(*a*,*b*) ∈ *set* (*zip* (*map f X*) (*map f Y*)) ⟹ (∃ *x* ∈ *set X* . *f x* = *a*)›
⟨*proof*⟩
lemma *map_pair_set_right*: ‹(*a*,*b*) ∈ *set* (*zip* (*map f X*) (*map f Y*)) ⟹ (∃ *y* ∈ *set Y* . *f y* = *b*)›
⟨*proof*⟩
lemma *map_pair_set*: ‹(*a*,*b*) ∈ *set* (*zip* (*map f X*) (*map f Y*)) ⟹ (∃ *x* ∈ *set X* . *f x* = *a*) ∧ (∃ *y* ∈ *set Y* . *f y* = *b*)›
⟨*proof*⟩

lemma *map_pair_f_all*:
assumes ‹*length X* = *length Y*›
shows ‹(∀ (*x*, *y*)∈*set* (*zip* (*map f X*) (*map f Y*)). *x* ≤ *y*) = (∀ (*x*, *y*)∈*set* (*zip X Y*). *f x* ≤ *f y*)›
⟨*proof*⟩

definition *map_interval_swap* :: ‹(*'a*::*linorder* × *'a*) *list* ⇒ *'a interval list*› **where**
‹*map_interval_swap* = *map* (λ (*x*,*y*). *Interval* (*if x* ≤ *y then* (*x*,*y*) *else* (*y*,*x*)))›

definition *mk_interval* :: ‹(*'a*::*linorder* × *'a*) ⇒ *'a interval* › **where**
‹*mk_interval* = (λ (*x*,*y*). *Interval* (*if x* ≤ *y then* (*x*,*y*) *else* (*y*,*x*)))›

definition *mk_interval'* :: ‹(*'a*::*linorder* × *'a*) ⇒ *'a interval* › **where**
‹*mk_interval'* = (λ (*x*,*y*). (*if x* ≤ *y then Interval*(*x*,*y*) *else Interval*(*y*,*x*)))›

lemma *map_interval_swap_code*[*code*]:
‹ *map_interval_swap* = *map* (λ (*x*,*y*). *the* (*if x* ≤ *y then Interval' x y else Interval' y x*))›
⟨*proof*⟩

lemma *mk_interval_code*[*code*]:
‹*mk_interval* = (λ (*x*,*y*). *the* (*if x* ≤ *y then Interval' x y else Interval' y x*))›
⟨*proof*⟩

lemma *mk_interval'*:
‹*mk_interval* = (λ (*x*,*y*). (*if x* ≤ *y then Interval*(*x*, *y*) *else Interval*(*y*, *x*)))›
⟨*proof*⟩

lemma *mk_interval_lower*[*simp*]: ‹ *lower* (*mk_interval* (*x*,*y*)) = (*if x* ≤ *y then x else y*)›
⟨*proof*⟩

lemma *mk_interval_upper*[*simp*]: ‹ *upper* (*mk_interval* (*x*,*y*)) = (*if x* ≤ *y then y else x*)›
⟨*proof*⟩

2.3 Linear Order on List of Intervals

definition

le_interval_list :: ‹(′*a*::*linorder*) *interval list* ⇒ ′*a interval list* ⇒ *bool*› ((_/ $\leq_I$ _) [51, 51] 50)
where
‹*le_interval_list Xs Ys* ≡ (*length Xs* = *length Ys*) ∧ (*foldl* (∧) *True* (*map2* (≤) *Xs Ys*))›

lemma *le_interval_single*: ‹(*x* ≤ *y*) = ([*x*] $\leq_I$ [*y*])›
⟨*proof*⟩

lemma *le_intervall_empty*[*simp*]: ‹[] $\leq_I$ []›
⟨*proof*⟩

lemma *le_interval_list_rev*: ‹(*is* $\leq_I$ *js*) = (*rev is* $\leq_I$ *rev js*)›
⟨*proof*⟩

lemma *le_interval_list_imp_length*:
assumes ‹*Xs* $\leq_I$ *Ys*› **shows** ‹*length Xs* = *length Ys*›
⟨*proof*⟩

lemma *lsplit_left*: **assumes** ‹*length* (*xs*) = *length* (*ys*)›
and ‹(∀ *n*<*length* (*x* # *xs*). (*x* # *xs*) ! *n* ≤ (*y* # *ys*) ! *n*)› **shows** ‹
((∀ *n* < *length xs*. *xs*!*n* ≤ *ys*!*n*) ∧ *x* ≤ *y*)›
⟨*proof*⟩

lemma *lsplit_right*: **assumes** ‹*length* (*xs*) = *length* (*ys*)›
and ‹((∀ *n* < *length xs*. *xs*!*n* ≤ *ys*!*n*) ∧ *x* ≤ *y*)›
shows ‹(*n*<*length* (*x* # *xs*) ⟶ (*x* # *xs*) ! *n* ≤ (*y* # *ys*) ! *n*)›
⟨*proof*⟩

lemma *lsplit*: **assumes** ‹*length* (*xs*) = *length* (*ys*)›
shows ‹(∀ *n*<*length* (*x* # *xs*). (*x* # *xs*) ! *n* ≤ (*y* # *ys*) ! *n*) =
((∀ *n* < *length xs*. *xs*!*n* ≤ *ys*!*n*) ∧ *x* ≤ *y*)›
⟨*proof*⟩

lemma *le_interval_list_all*′:
assumes ‹*length Xs* = *length Ys*› **and** ‹*Xs* $\leq_I$ *Ys*› **shows** ‹∀ *n* < *length Xs*. *Xs*!*n* ≤ *Ys*!*n*›
⟨*proof*⟩

lemma *le_interval_list_all2*:
assumes ‹*length Xs* = *length Ys*›
and ‹∀ *n*<*length Xs* . (*Xs*!*n* ≤ *Ys*!*n*)›
shows ‹*Xs* $\leq_I$ *Ys*›
⟨*proof*⟩

lemma *le_interval_list_all*:
assumes ‹*Xs* $\leq_I$ *Ys*› **shows** ‹∀ *n* < *length Xs*. *Xs*!*n* ≤ *Ys*!*n*›
⟨*proof*⟩

lemma *le_interval_list_imp*:
assumes ‹*Xs* $\leq_I$ *Ys*› **shows** ‹*n* < *length Xs* ⟶ *Xs*!*n* ≤ *Ys*!*n*›
⟨*proof*⟩

lemma *interval_set_leq_eq*: ‹(*X* ≤ *Y*) = (*set_of X* ⊆ *set_of Y*)›
for *X* :: ‹′*a*::*linordered_ring interval*›
⟨*proof*⟩

lemma times_interval_right:
fixes X Y C ::‹'a::linordered_ring interval›
assumes ‹$X \leq Y$›
shows ‹$C * X \leq C * Y$›
$\langle proof \rangle$

lemma times_interval_left:
fixes X Y C ::‹'a::{real_normed_algebra,linordered_ring,linear_continuum_topology} interval›
assumes ‹$X \leq Y$›
shows ‹$X * C \leq Y * C$›
$\langle proof \rangle$

2.4 Support for Lists of Intervals

abbreviation in_interval_list::‹('a::preorder) list ⇒ 'a interval list ⇒ bool› ((_/ $\in_I$ _) [51, 51] 50)
where ‹in_interval_list xs Xs ≡ foldl (∧) True (map2 (in_interval) xs Xs)›

lemma interval_of_in_interval_list[simp]: ‹xs $\in_I$ map interval_of xs›
$\langle proof \rangle$

lemma interval_of_in_eq: ‹interval_of x $\leq$ X = (x $\in_i$ X)›
$\langle proof \rangle$

lemma interval_of_list_in:
assumes ‹length inputs = length Inputs›
shows ‹(map interval_of inputs $\leq_I$ Inputs) = (inputs $\in_I$ Inputs)›
$\langle proof \rangle$

2.5 Interval Width and Arithmetic Operations

lemma interval_width_addition:
fixes A:: 'a::{linordered_ring} interval
shows ‹width (A + B) = width A + width B›
$\langle proof \rangle$

lemma interval_width_times:
fixes a :: 'a::{linordered_ring} **and** A :: 'a interval
shows width (interval_of a * A) = |a| * width A
$\langle proof \rangle$

lemma interval_sup_width:
fixes X Y :: 'a::{linordered_ring, lattice} interval
shows width (sup X Y) = max (upper X) (upper Y) − min (lower X) (lower Y)
$\langle proof \rangle$

lemma width_expanded: ‹interval_of (width Y) = Interval(upper Y − lower Y, upper Y − lower Y)›
$\langle proof \rangle$

lemma interval_width_positive:
fixes X :: ‹'a::{linordered_ring} interval›
shows ‹0 $\leq$ width X›

$\langle proof \rangle$

2.6 Interval Multiplication

lemma *interval_interval_times*:
‹$X * Y = Interval(Min \{(lower\ X * lower\ Y), (lower\ X * upper\ Y), (upper\ X * lower\ Y), (upper\ X * upper\ Y)\},$
$Max \{(lower\ X * lower\ Y), (lower\ X * upper\ Y), (upper\ X * lower\ Y), (upper\ X * upper\ Y)\})$›
$\langle proof \rangle$

lemma *interval_times_scalar*: ‹$interval\_of\ a * A = mk\_interval(a * lower\ A, a * upper\ A)$›
$\langle proof \rangle$

lemma *interval_times_scalar_pos_l*:
assumes ‹$0 \leq a$ ›
shows ‹$interval\_of\ a * A = Interval(a * lower\ A, a * upper\ A)$›
$\langle proof \rangle$

lemma *interval_times_scalar_pos_r*:
fixes $a :: 'a::\{linordered\_idom\}$
assumes ‹$0 \leq a$›
shows ‹$A * interval\_of\ a = Interval(a * lower\ A, a * upper\ A)$›
$\langle proof \rangle$

2.7 Distance-based Properties of Intervals

Given two real intervals *X* and *Y*, and two real numbers *a* and *b*, the width of the sum of the scaled intervals is equivalent to the width of the two individual intervals.

lemma *width_of_scaled_interval_sum*:
fixes $X :: 'a::\{linordered\_ring\}\ interval$
shows ‹$width\ (interval\_of\ a * X + interval\_of\ b * Y) = |a| * width\ X + |b| * width\ Y$›
$\langle proof \rangle$

lemma *width_of_product_interval_bound_real*:
fixes $X :: real\ interval$
shows ‹$interval\_of\ (width\ (X * Y)) \leq abs\_interval(X) * interval\_of\ (width\ Y) + abs\_interval(Y) * interval\_of\ (width\ X)$›
$\langle proof \rangle$

lemma *width_of_product_interval_bound_int*:
fixes $X :: int\ interval$
shows ‹$interval\_of\ (width\ (X * Y)) \leq abs\_interval(X) * interval\_of\ (width\ Y) + abs\_interval(Y) * interval\_of\ (width\ X)$›
$\langle proof \rangle$

end

3 Basic Properties of Interval Division (Interval_Division_Non_Zero)

theory
Interval_Division_Non_Zero
imports
Interval_Utilities
begin

The theory *HOL−Library.Interval* does not define a division operation on intervals. In the following we define a locale capturing the core properties of division by an interval that does not contain zero.

3.1 Preliminaries

lemma *division_leq_neg*:
fixes $x :: 'a::\{linordered\_field\}$
assumes $0 < x$ **and** $y < 0$ **and** $z < 0$ **and** $y \leq z$
shows $x / z \leq x / y$
$\langle proof \rangle$

lemma *division_leq*:
fixes $x :: 'a::\{linordered\_field\}$
assumes $0 < x$ **and** $y \leq z$ **and** ‹$y \neq 0 \wedge z \neq 0$› **and** ‹$(y < 0 \wedge z < 0) \vee (0 < y \wedge 0 < z)$ ›
shows $x / z \leq x / y$
$\langle proof \rangle$

lemma *upper_leq_lower_div*:
fixes $Y :: 'a::\{linordered\_field\}$ *interval*
assumes ‹$lower\ Y \leq upper\ Y$› **and** ‹$\neg\ 0 \in_i Y$›
shows ‹$1 / upper\ Y \leq 1 / lower\ Y$›
$\langle proof \rangle$

3.2 A Locale for Interval Division Where the Quotient-Interval does not Contain Zero

locale *interval_division* = *inverse* +
constrains *inverse* :: ‹$'a::\{linordered\_field, real\_normed\_algebra, linear\_continuum\_topology\}$ *interval* ⇒ *'a interval*›
and *divide* :: ‹$'a::\{linordered\_field, real\_normed\_algebra, linear\_continuum\_topology\}$ *interval* ⇒ *'a interval* ⇒ *'a interval*›
assumes *inverse_left*: ‹$\neg\ 0 \in_i x \Longrightarrow 1 \leq (inverse\ x) * x$›
and *divide*: ‹$\neg\ 0 \in_i y \Longrightarrow x \leq (divide\ x\ y) * y$›
begin
end

lemma *interval_non_zero_eq*:
‹$\neg\ 0 \in_i (i::'a::\{linorder, zero\}\ interval) = (lower\ i < 0 \wedge upper\ i < 0) \vee (lower\ i > 0 \wedge upper\ i > 0)$›

$\langle proof \rangle$

lemma *inverse_includes_one*:
assumes ‹¬ *0* $\in_i$ (*i*::′*a*::{*division_ring*,*linordered_ring*} *interval*)›
shows ‹*1* $\in_i$ (*mk_interval* (*1* / *upper i*, *1* / *lower i*)) ∗ *i*›
$\langle proof \rangle$

lemma *inverse_includes_one′*:
assumes ‹¬ *0* $\in_i$ (*i*::′*a*::{*division_ring*,*linordered_ring*} *interval*)›
shows ‹*1* ≤ (*mk_interval* (*1* / *upper i*, *1* / *lower i*)) ∗ *i*›
$\langle proof \rangle$

locale *interval_division_inverse* = *inverse* +
constrains *inverse* :: ‹′*a*::{*linordered_field*, *real_normed_algebra*,*linear_continuum_topology*} *interval* ⇒ ′*a interval*›
and *divide* :: ‹′*a*::{*linordered_field*, *real_normed_algebra*,*linear_continuum_topology*} *interval* ⇒ ′*a interval* ⇒ ′*a interval*›
assumes *inverse_non_zero_def*: ‹¬ *0* $\in_i$ *x* ⟹ (*inverse x*) = *mk_interval*(*1* / (*upper x*), *1* / (*lower x*))›
and *divide_non_zero_def*: ‹¬ *0* $\in_i$ *y* ⟹ (*divide x y*) = *inverse y* ∗ *x*›
begin

sublocale *interval_division divide inverse*
$\langle proof \rangle$

lemma *inverse_left_ge_one*:
assumes ‹¬ *0* $\in_i$ *x*›
shows ‹*1* ≤ (*inverse x*) ∗ *x*›
$\langle proof \rangle$

lemma *division_right_ge_refl*:
assumes ‹¬ *0* $\in_i$ *y*›
shows ‹*x* ≤ *x* ∗ ((*inverse y*) ∗ *y*)›
$\langle proof \rangle$

lemma *division_right_ge_refl′*:
assumes ‹¬ *0* $\in_i$ *y*›
shows ‹*x* ≤ *x* ∗ *inverse y* ∗ *y*›
$\langle proof \rangle$

lemma *interval_div_constant*:
assumes ‹*0* ∉ *set_of Y*› **and** ‹*0* ≤ *x*›
shows ‹*divide* (*interval_of x*) *Y* = *Interval*(*x* / *upper Y*, *x* / *lower Y*)›
$\langle proof \rangle$

lemma *interval_of_width*:
assumes ‹¬ *0* $\in_i$ *Y*›
shows ‹*interval_of*(*width* (*divide* (*interval_of 1*) *Y*)) = *Interval*(*1* / *lower Y* − *1* / *upper Y*, *1* / *lower Y* − *1* / *upper Y*) ›
$\langle proof \rangle$

lemma *abs_pos*:
assumes ‹*0* < *lower Y* › **and** ‹¬ *0* $\in_i$ *Y*›
shows ‹*abs_interval*(*divide* (*interval_of 1*) *Y*) = *Interval*(*1* / *upper Y*, *1* / *lower Y*)›
$\langle proof \rangle$

lemma *abs_neg*:
  assumes ‹*upper Y* $<$ *0* › **and** ‹$\neg$ *0* $\in_i$ *Y*›
  shows ‹*abs_interval*(*divide* (*interval_of 1*) *Y*) $=$ *Interval*(*1* / |*lower Y*|, *1* / |*upper Y*|)›
$\langle proof \rangle$

end

end

4 A Naive Interval Division for Real Intervals (Interval_Division_Real)

theory
Interval_Division_Real
imports
Interval_Division_Non_Zero
begin

The theory *HOL−Library.Interval* does not define a division operation on intervals. Actually, In the following we define division in a straight forward way. This is possible, as in HOL, the property $?a\ /\ (0::?'a) = (0::?'a)$ holds. Therefore, we do not need to use, in the first instance, extended interval analysis (e.g., based on the type *ereal*. As a consequence, results obtained using this definition might differ from results obtained using definitions of divisions using extended reals (e.g., [4]).

```
instantiation interval :: ({linordered_field, real_normed_algebra,linear_continuum_topology}) inverse
begin
 definition inverse_interval :: 'a interval ⇒ 'a interval
  where inverse_interval a = mk_interval ( 1 / (upper a), 1 / (lower a))
 definition divide_interval :: 'a interval ⇒ 'a interval ⇒ 'a interval
  where divide_interval a b = inverse b ∗ a
 instance ⟨proof⟩
end

interpretation interval_division_inverse divide inverse
 ⟨proof⟩

lemma width_of_reciprocal_interval_bound_real:
 fixes Y :: real interval
 assumes ‹¬ 0 ∈i Y›
 shows ‹interval_of(width ((interval_of 1) / Y)) ≤
     (abs_interval((interval_of 1) / Y) ∗ abs_interval((interval_of 1) / Y)) ∗ interval_of(width Y)›
⟨proof⟩

end
```

5 Affine Functions (Affine_Functions)

In this theory, we provide formalisation of affine functions (sometimes also called linear polynomial functions).

theory
Affine_Functions
imports
Complex_Main
begin

5.1 Definition of Affine Functions, Alternative Definitions, and Special Cases

locale *affine_fun* $=$
fixes f
assumes ‹$\exists\ b.\ linear\ (\lambda\, x.\ f\, x - b)$›

lemma *affine_fun_alt*:
‹$affine\_fun\ f = (\exists\ c\ g.\ (f = (\lambda\, x.\ g\ x + c)) \wedge linear\ g)$›
$\langle proof \rangle$

lemma *affine_fun_real_linfun*:
‹$affine\_fun\ (f{::}(real \Rightarrow real)) = (\exists\ a\ b\ .\ f = (\lambda\, x.\ a * x + b))$›
$\langle proof \rangle$

lemma *linear_is_affine_fun*: ‹$linear\ f \Longrightarrow affine\_fun\ f$›
$\langle proof \rangle$

lemma *affine_zero_is_linear*: **assumes** ‹$affine\_fun\ f$› **and** ‹$f\ 0 = 0$› **shows** ‹$linear\ f$›
$\langle proof \rangle$

lemma *affine_add*:
assumes ‹$affine\_fun\ f$› **and** ‹$affine\_fun\ g$›
shows ‹$affine\_fun\ (\lambda\, x.\ f\, x + g\ x)$›
$\langle proof \rangle$

lemma *scaleR*:
assumes ‹$affine\_fun\ f$ › **shows** ‹$affine\_fun\ (\lambda\, x.\ a *_R (f\, x))$›
$\langle proof \rangle$

lemma *real_affine_funD*:
fixes $f :: real \Rightarrow real$
assumes $affine\_fun\ f$ **obtains** $c\ b$ **where** $f = (\lambda\, x.\ c * x + b)$
$\langle proof \rangle$

5.2 Common Linear Polynomial Functions

lemma *affine_fun_const*[*simp*]: ‹$affine\_fun\ (\lambda\, x.\ c)$›
$\langle proof \rangle$

lemma *affine_fun_id*[*simp*]: ‹*affine_fun* $(\lambda\, x.\ x)$›
⟨*proof*⟩

lemma *affine_fun_mult*[*simp*]: ‹*affine_fun* $(\lambda\, x.\ (c{::}'a{::}real\_algebra) * x)$›
⟨*proof*⟩

lemma *affine_fun_scaled*[*simp*]: ‹*affine_fun* $(\lambda\, x.\ x\ /c\ )$›
for *c* :: *'a::real_normed_field*
⟨*proof*⟩

lemma *affine_fun_add*[*simp*]: ‹*affine_fun* $(\lambda\, x.\ x + c)$›
⟨*proof*⟩

lemma *affine_fun_diff*[*simp*]: ‹*affine_fun* $(\lambda\, x.\ x - c)$›
⟨*proof*⟩

lemma *affine_fun_triv*[*simp*]: ‹*affine_fun* $(\lambda\, x.\ a *_R x + c)$›
⟨*proof*⟩

lemma *affine_fun_add_const*[*simp*]: **assumes** ‹*affine_fun f*› **shows** ‹*affine_fun* $(\lambda\, x.\ (f\, x) + c)$›
⟨*proof*⟩

lemma *affine_fun_diff_const*[*simp*]: **assumes** ‹*affine_fun f*› **shows** ‹*affine_fun* $(\lambda\, x.\ (f\, x) - c)$›
⟨*proof*⟩

lemma *affine_fun_comp*[*simp*]: **assumes** ‹*affine_fun* (f)›
and ‹*affine_fun* (g)› **shows** ‹*affine_fun* $(f \circ g)$›
⟨*proof*⟩

lemma *affine_fun_linear*[*simp*]: **assumes** ‹*affine_fun f*› **shows** ‹*affine_fun* $(\lambda\, x.\ a *_R (f\, x) + c)$›
⟨*proof*⟩

5.3 Linear Polynomial Functions and Orderings

lemma *affine_fun_leq_pos*:
assumes ‹*affine_fun* $(f{::}real \Rightarrow real)$› **and** ‹*affine_fun g*›
and ‹$x \in \{0..u\}$› **and** ‹$(f\, 0 \leq g\, 0\ )$› **and** ‹$(f\, u \leq g\, u)$›
shows ‹$f\, x \leq g\, x$›
⟨*proof*⟩

lemma *affine_fun_leq_neg*:
assumes ‹*affine_fun* $(f{::}real \Rightarrow real)$› **and** ‹*affine_fun g*›
and ‹$x \in \{l..0\}$› **and** ‹$(f\, l \leq g\, l\ )$› **and** ‹$(f\, 0 \leq g\, 0)$›
shows ‹$f\, x \leq g\, x$›
⟨*proof*⟩

lemma *affine_fun_leq*:
assumes ‹*affine_fun* $(f{::}real \Rightarrow real)$› **and** ‹*affine_fun g*›
and ‹$x \in \{l..u\}$› **and** ‹$(f\, l \leq g\, l\ )$› **and** ‹$(f\, u \leq g\, u)$›
shows ‹$f\, x \leq g\ (x{::}real)$›
⟨*proof*⟩

lemma *affine_fun_le_pos*:
assumes ‹*affine_fun* (f::*real* ⇒ *real*)› **and** ‹*affine_fun g*›
and ‹x∈{*o*..*u*}› **and** ‹(f *o* < g *o*)› **and** ‹(f *u* < g *u*)›
shows ‹f x < g (x::*real*)›
⟨*proof*⟩

lemma *affine_fun_le_neg*:
assumes ‹*affine_fun* (f::*real* ⇒ *real*)› **and** ‹*affine_fun g*›
and ‹x∈{*l*..*o*}› **and** ‹(f *l* < g *l*)› **and** ‹(f *o* < g *o*)›
shows ‹f x < g (x::*real*)›
⟨*proof*⟩

lemma *affine_fun_le*:
assumes ‹*affine_fun* (f::*real* ⇒ *real*)› **and** ‹*affine_fun g*›
and ‹x∈{*l*..*u*}› **and** ‹(f *l* < g *l*)› **and** ‹(f *u* < g *u*)›
shows ‹f x < g (x::*real*)›
⟨*proof*⟩

end

6 Interval Inclusion Isotonicity (Inclusion_Isotonicity)

theory
Inclusion_Isotonicity
imports
Interval_Utilities
Affine_Functions
Interval_Division_Non_Zero
begin

6.1 Interval Extension

6.1.1 Textbook Definition of Interval Extension

definition
is_interval_extension_of :: ‹(′a::preorder interval ⇒ ′b::preorder interval) ⇒ (′a ⇒ ′b) ⇒ bool›
(**infix** (is′_interval′_extension′_of) 50)
where
‹((F is_interval_extension_of f)) = (∀ x. (F (interval_of x)) = interval_of (f x))›

definition‹extend_natural f = (λ X. mk_interval (f (lower X), f(upper X)))›

lemma *interval_extension_comp*:
assumes *interval_ext_F*: ‹F is_interval_extension_of f›
and *interval_ext_G*: ‹G is_interval_extension_of g›
shows ‹(F o G) is_interval_extension_of (f o g)›
⟨*proof*⟩

lemma *interval_extension_const*: ‹(λ x. interval_of c) is_interval_extension_of (λ x. c)›
⟨*proof*⟩

lemma *interval_extension_id*: ‹(λ x. x) is_interval_extension_of (λ x. x)›
⟨*proof*⟩

6.1.2 A Stronger Definition of Interval Extension

definition
is_natural_interval_extension_of :: ‹(′a::linorder interval ⇒ ′b::linorder interval) ⇒ (′a ⇒ ′b) ⇒ bool›
(**infix** ‹(is′_natural′_interval′_extension′_of)› 50)
where
‹((F is_natural_interval_extension_of f)) = (∀ l u. (F (mk_interval (l,u))) = mk_interval (f l, f u))›

lemma ‹(extend_natural f) is_interval_extension_of f›
⟨*proof*⟩

lemma ‹(extend_natural f) is_natural_interval_extension_of f›
⟨*proof*⟩

lemma *natural_interval_extension_implies_interval_extension*:

assumes ‹F *is_natural_interval_extension_of* f› **shows** ‹F *is_interval_extension_of* f›
⟨*proof*⟩

lemma *const_add_is_natural_interval_extensions*:
‹$(\lambda\, x.\ (interval\_of\ c) + x)$ *is_natural_interval_extension_of* $(\lambda\, x.\ c + x)$›
⟨*proof*⟩

lemma *const_times_is_natural_interval_extensions*:
‹$(\lambda\, x.\ (interval\_of\ c) * x)$ *is_natural_interval_extension_of* $(\lambda\, x.\ c * x)$›
⟨*proof*⟩

lemma *const_is_interval_extension*: ‹F *is_natural_interval_extension_of* $(\lambda\, x.\ b) \Longrightarrow F = (\lambda\, x.(interval\_of\ b))$›
⟨*proof*⟩

lemma *id_is_interval_extension*: ‹F *is_natural_interval_extension_of* $(\lambda\, x.\ x) \Longrightarrow F = (\lambda\, x.\ x)$›
⟨*proof*⟩

lemma *extend_natural_is_interval_extension*:
‹$(extend\_natural\ f)$ *is_natural_interval_extension_of* f›
⟨*proof*⟩

lemma *is_natural_interval_extension_implies_bounds*:
fixes F :: ‹*real interval* ⇒ *real interval*›
assumes ‹F *is_natural_interval_extension_of* f› **and** ‹$F\ x \leq F\ x'$›
shows
‹$((f\ (lower\ x')) \leq\ (f\ (lower\ x))\ ) \vee\ ((f\ (upper\ x')) \leq\ (f\ (upper\ x))\ )$›
⟨*proof*⟩

lemma *interval_extension_lower*:
‹$((F\ is\_interval\_extension\_of\ f)) \Longrightarrow lower\ (F\ (interval\_of\ x)) = (f\ x)$›
⟨*proof*⟩

lemma *interval_extension_upper*:
‹$((F\ is\_interval\_extension\_of\ f)) \Longrightarrow upper\ (F\ (interval\_of\ x)) = (f\ x)$›
⟨*proof*⟩

lemma *is_interval_extension_eq_upper_and_lower*:
‹$((F\ is\_interval\_extension\_of\ f))$
$= (\forall\ x.\ lower\ (F\ (interval\_of\ x)) = (f\ x) \wedge upper\ (F\ (interval\_of\ x)) = (f\ x))$›
⟨*proof*⟩

lemma *interval_extension_lower_simp*[*simp*]:
assumes ‹F *is_interval_extension_of* f› **and** ‹$X = Interval(x,x)$›
shows ‹$lower\ (F\ X) = f\ x$ ›
⟨*proof*⟩

lemma *interval_extension_upper_simp*[*simp*]:
assumes ‹F *is_interval_extension_of* f› **and** ‹$X = Interval(x,x)$›
shows ‹$upper\ (F\ X) = f\ x$ ›
⟨*proof*⟩

6.2 Interval Inclusion Isotonicity

In this section, we introduce the concept of inclusion isotonicity. The formalization in this theory generalises the definitions from [5]:

definition
inclusion_isotonic :: ‹('a::preorder interval ⇒ 'b::preorder interval) ⇒ bool›
where
‹*inclusion_isotonic f* = (∀ x x'. x ≤ x' ⟶ (f x) ≤ (f x'))›

We can immediately show that any natural extension of an affine function of from type *real* to *real* is interval inclusion isotonic:

lemma *real_affine_fun_is_inclusion_isotonicM*:
assumes ‹*affine_fun* (*f*::*real* => *real*)›
shows ‹*inclusion_isotonic* (*extend_natural f*)›
⟨*proof*⟩

definition
inclusion_isotonic_on :: ‹('a interval ⇒ bool) ⇒ ('a::preorder interval ⇒ 'b::preorder interval) ⇒ bool›
where
‹*inclusion_isotonic_on P f* = (∀ x x'. P x ∧ P x' ∧ x ≤ x' ⟶ (f x) ≤ (f x'))›

lemma *inclusion_isotonic_eq*: ‹*inclusion_isotonic_on* (λ x . *True*) = *inclusion_isotonic*›
⟨*proof*⟩

definition *inclusion_isotonic_binary* :: ‹('a::preorder interval ⇒ 'a interval ⇒ 'b::preorder interval) ⇒ bool›
where
‹*inclusion_isotonic_binary f* = (∀ x x' y y'. x ≤ x' ∧ y ≤ y' ⟶ (f x y) ≤ (f x' y'))›

definition *inclusion_isotonic_binary_on* :: ‹('a interval ⇒ bool) ⇒ ('a::preorder interval ⇒ 'a interval ⇒ 'b::preorder interval) ⇒ bool›
where
‹*inclusion_isotonic_binary_on P f* = (∀ x x' y y'. P x ∧ P x' ∧ P y ∧ P y' ∧ x ≤ x' ∧ y ≤ y' ⟶ (f x y) ≤ (f x' y'))›

lemma *inclusion_isotonic_binary_eq*: ‹*inclusion_isotonic_binary_on* (λ x . *True*) = *inclusion_isotonic_binary*›
⟨*proof*⟩

definition *inclusion_isotonicM_n* :: ‹*nat* ⇒ ('a::linorder interval list ⇒ 'a::linorder interval) ⇒ bool› **where**
‹*inclusion_isotonicM_n n f* = (∀ is js. (length is = n ∧ length js = n ∧ (is $\leq_I$ js)) ⟶ f is ≤ f js)›

definition *inclusion_isotonicM_n_on* :: ‹('a interval ⇒ bool) ⇒ nat ⇒ ('a::linorder interval list ⇒ 'a::linorder interval) ⇒ bool› **where**
‹*inclusion_isotonicM_n_on P n f* = (∀ is js. (∀ i ∈ set is. P i) ∧ (∀ j ∈ set js. P j) ∧ (length is = n ∧ length js = n ∧ (is $\leq_I$ js)) ⟶ f is ≤ f js)›

lemma *inclusion_isotonicM_n_eq*: ‹*inclusion_isotonicM_n_on* (λ x . *True*) = *inclusion_isotonicM_n*›
⟨*proof*⟩

Finally, we extend the definition to functions over lists and show that the three definitions of interval inclusion isotonicity are, for their corresponding types, equivalent:

locale *inclusion_isotonicM* =
fixes *n* :: *nat*
and *f* :: ‹('a::linorder) interval list ⇒ 'a interval list›

assumes *inclusion_isotonicM* :
‹($\forall$ *is js*. (*length is* = *n*) ∧(*length js* = *n*) ∧ (*is* $\leq_I$ *js*) ⟶ *f is* $\leq_I$ *f js*)›
begin
end

locale *inclusion_isotonicM_on* =
fixes *P* :: ‹′*a*::*linorder interval* ⇒ *bool*›
and *n* :: *nat*
and *f* :: ‹(′*a*::*linorder*) *interval list* ⇒ ′*a interval list*›
assumes *inclusion_isotonicM* :
‹($\forall$ *is js*. ($\forall$ *i* ∈ *set is*. *P i*) ∧ ($\forall$ *j* ∈ *set js*. *P j*) ∧ (*length is* = *n*) ∧(*length js* = *n*) ∧ (*is* $\leq_I$ *js*) ⟶ *f is* $\leq_I$ *f js*)›
begin
end

lemma *inclusion_isotonicM_on_eq*: ‹*inclusion_isotonicM_on* (λ *x*. *True*) = *inclusion_isotonicM*›
⟨*proof*⟩

lemma *inclusion_isotonic_conv*:
‹*inclusion_isotonic g* = *inclusion_isotonicM* 1 (λ *xs* . *case xs of* [*x*] ⇒ [*g x*])›
⟨*proof*⟩

lemma *inclusion_isotonicM_n_conv1*:
‹*inclusion_isotonicM_n n f* = *inclusion_isotonicM n* (λ *xs*. [*f xs*])›
⟨*proof*⟩

lemma *inclusion_isotonicM_conv2*:
assumes ‹*inclusion_isotonicM n f*›
and ‹$\forall$ *xs*. (*length xs* = *n*) ⟶ *N* = (*length* (*f xs*))›
shows ‹*inclusion_isotonicM n* (λ *xs*. *if n*′ < *N then* [(*f xs*)!*n*′] *else* [])›
⟨*proof*⟩

lemma *inclusion_isotonicM_n_on_conv1*:
‹*inclusion_isotonicM_n_on P n f* = *inclusion_isotonicM_on P n* (λ *xs*. [*f xs*])›
⟨*proof*⟩

lemma *inclusion_isotonicM_on_conv2*:
assumes ‹*inclusion_isotonicM_on P n f*›
and ‹$\forall$ *xs*. (*length xs* = *n*) ⟶ *N* = (*length* (*f xs*))›
shows ‹*inclusion_isotonicM_on P n* (λ *xs*. *if n*′ < *N then* [(*f xs*)!*n*′] *else* [])›
⟨*proof*⟩

lemma *inclusion_isotonic_binary_conv1*:
‹*inclusion_isotonic_binary f* = *inclusion_isotonicM_n* 2 (λ *xs* . *case xs of* [*x*,*y*] ⇒ *f x y*)›
⟨*proof*⟩

lemma *inclusion_isotonic_binary_conv2*:
‹*inclusion_isotonic_binary f* = *inclusion_isotonicM* 2 (λ *xs* . *case xs of* [*x*,*y*] ⇒ [*f x y*])›
⟨*proof*⟩

lemma *inclusion_isotonic_binary_on_conv1*:
‹*inclusion_isotonic_binary_on P f* = *inclusion_isotonicM_n_on P* 2 (λ *xs* . *case xs of* [*x*,*y*] ⇒ *f x y*)›
⟨*proof*⟩

lemma *inclusion_isotonic_binary_on_conv2*:

‹inclusion_isotonic_binary_on P f = inclusion_isotonicM_on P 2 (λ xs . case xs of [x,y] ⇒ [f x y])›
⟨proof⟩

6.2.1 Compositionality of Interval Inclusion Isotonicy

lemma *inclusion_isotonicM_comp*:
 assumes ‹inclusion_isotonicM n f›
  and ‹∀ xs. length xs = n ⟶ length (f xs) = m›
 shows ‹inclusion_isotonicM n (g o f)›
 ⟨proof⟩

lemma *inclusion_isotonicM_on_comp*:
 assumes ‹inclusion_isotonicM_on P n f›
  and ‹inclusion_isotonicM_on Q m g›
  and ‹∀ xs. length xs = n ⟶ length (f xs) = m›
  and ‹∀ is. (∀ i ∈ set is. P i) ⟶ (∀x∈set (f is). Q x)›
shows ‹inclusion_isotonicM_on P n (g o f)›
 ⟨proof⟩

lemma *inclusion_isotonic_comp*:
 assumes ‹inclusion_isotonic f›
  and ‹inclusion_isotonic g›
 shows ‹inclusion_isotonic (g o f)›
 ⟨proof⟩

lemma *inclusion_isotonic_on_comp*:
 assumes ‹inclusion_isotonic_on P f›
  and ‹inclusion_isotonic_on Q g›
  and ‹∀ x. P x ⟶ Q (f x)›
shows ‹inclusion_isotonic_on P (g o f)›
 ⟨proof⟩

6.2.2 Interval Inclusion Isontonicity of the Core Operator

Unary Minus (Negation)

lemma *inclusion_isotonicM_uminus[simp]*: ‹inclusion_isotonic uminus›
 ⟨proof⟩

Addition

lemma *inclusion_isotonicM_plus[simp]*: ‹inclusion_isotonic_binary (+)›
 ⟨proof⟩

Substraction

lemma *inclusion_isotonicM_minus[simp]*: ‹inclusion_isotonic_binary (−)›
 ⟨proof⟩

Multiplication

lemma *inclusion_isotonicM_times[simp]*:
 ‹inclusion_isotonic_binary (λ x y. (x::'a::{linordered_ring, real_normed_algebra, linear_continuum_topology} interval) ∗ y)›

⟨proof⟩

6.2.3 Interval Inclusion Isotonicity of Various Functions

lemma *inclusion_isotonicM_n_empty*[*simp*]: ‹*inclusion_isotonicM n* (λ *xs*. [])›
⟨proof⟩

lemma *inclusion_isotonic_id*[*simp*]: ‹*inclusion_isotonic id*›
⟨proof⟩

lemma *inclusion_isotonicM_id*[*simp*]: ‹*inclusion_isotonicM n id*›
⟨proof⟩

lemma *inclusion_isotonicM_hd*[*simp*]:
assumes ‹$0 < n$›
shows ‹*inclusion_isotonicM_n n hd*›
⟨proof⟩

lemma *inclusion_isotonic_add_const1*[*simp*]:
‹*inclusion_isotonic* (λ *x*. *x* + *c*)›
⟨proof⟩

lemma *inclusion_isotonicM_1_add_const2*[*simp*]:
‹*inclusion_isotonic* (λ *x*. *c* + *x*)›
⟨proof⟩

lemma *inclusion_isotonic_times_right*[*simp*]:
‹*inclusion_isotonic* (λ *x*. *C*∗ (*x*::′*a*::*linordered_ring interval*))›
⟨proof⟩

lemma *inclusion_isotonic_times_left*[*simp*]:
‹*inclusion_isotonic* (λ *x*. (*x*::′*a*::{*real_normed_algebra*,*linordered_ring*,*linear_continuum_topology*} *interval*) ∗ *C*)›
⟨proof⟩

lemma *map_inclusion_isotonicM*:
assumes ‹*inclusion_isotonic f*›
shows ‹*inclusion_isotonicM n* (*map f*)›
⟨proof⟩

lemma *inclusion_isotonicM_fun_plus*:
assumes ‹*inclusion_isotonic f*› **and** ‹*inclusion_isotonic g*›
shows ‹*inclusion_isotonic* (λ *x*. *f x* + *g x*)›
⟨proof⟩

lemma *inclusion_isotonic_plus_const*:
assumes ‹*inclusion_isotonic f*› **and** ‹*inclusion_isotonic g*›
shows ‹*inclusion_isotonic* (λ *x*. *g x* + *c*)›
⟨proof⟩

lemma *inclusion_isotonic_times_const_real*:
assumes ‹*inclusion_isotonic f*›
shows ‹*inclusion_isotonic* (λ *x*. (*c*::*real interval*) ∗ (*f x*))›
⟨proof⟩

lemma *intervall_le_foldr*:
assumes ‹*inclusion_isotonic_binary f*›
shows ‹*length js* = *length is* ⟹ *is* $\leq_I$ *js* ⟹ (*foldr f is e*) ≤ (*foldr f js e*)›
⟨*proof*⟩

lemma *intervall_le_foldr_map*:
assumes ‹*inclusion_isotonic_binary f*›
and ‹*inclusion_isotonic g*›
shows ‹*length js* = *length is* ⟹ *is* $\leq_I$ *js* ⟹ (*foldr f* (*map g is*) *e*) ≤ (*foldr f* (*map g js*) *e*)›
⟨*proof*⟩

lemma *intervall_le_foldr_e*:
assumes ‹*inclusion_isotonic_binary f*›
and ‹*is* ≤ *js*›
shows ‹(*foldr f xs is*) ≤ (*foldr f xs js*)›
⟨*proof*⟩

lemma *foldr_inclusion_isotonicM_e*:
assumes ‹*inclusion_isotonic_binary f*›
shows ‹∀ *x*. *inclusion_isotonic* (*foldr f x*)›
⟨*proof*⟩

lemma *foldr_inclusion_isotonicM*:
assumes ‹*inclusion_isotonic_binary f*›
shows ‹*inclusion_isotonicM_n n* (λ *x*. *foldr f x e*)›
⟨*proof*⟩

lemma *foldr_inclusion_isotonicM_g*:
assumes ‹*inclusion_isotonic_binary f*›
and ‹*inclusion_isotonicM n g*›
shows ‹*inclusion_isotonicM_n n* (λ *x*. *foldr f* ((*g x*)) *e*)›
⟨*proof*⟩

lemma *foldr_map_inclusion_isotonicM_g*:
assumes ‹*inclusion_isotonic_binary f*›
and ‹*inclusion_isotonic g* ›
and ‹*inclusion_isotonicM n P*›
shows ‹*inclusion_isotonicM_n n* (λ *x*. *foldr f* (*map g* (*P x*)) *e*)›
⟨*proof*⟩

lemma *foldl_inclusion_isotonicM*:
assumes ‹*inclusion_isotonic_binary f*›
shows ‹*inclusion_isotonicM_n n* (*foldl f e*)›
⟨*proof*⟩

lemma *fold_inclusion_isotonicM*:
assumes ‹*inclusion_isotonic_binary f*›
shows ‹*inclusion_isotonicM_n n* (λ *x*. *fold f x e*)›
⟨*proof*⟩

lemma *map2_inclusion_isotonicM_left*: **assumes** ‹*inclusion_isotonic_binary f*›
shows ‹*inclusion_isotonicM n* (*map2 f xs*)›
⟨*proof*⟩

lemma *map2_inclusion_isotonicM_right*: **assumes** ‹*inclusion_isotonic_binary f*›
  shows ‹*inclusion_isotonicM n* $(\lambda$ *ys. map2 f ys xs*$)$›
$\langle proof \rangle$

lemma *map2_inclusion_isotonicM_right_g*:
assumes ‹*inclusion_isotonic_binary f*›
and ‹$\forall$ *xs. length* $($*g xs*$)$ $=$ *length xs*›
and ‹*inclusion_isotonicM n g*›
and ‹*length xs* $=$ *n*›
and ‹*length is* $=$ *n*›
and ‹*length js* $=$ *n*›
and ‹*is* $\leq_I$ *js*›
shows‹*map2 f* $($*g is*$)$ $($*h xs*$)$ $\leq_I$ *map2 f* $($*g js*$)$ $($*h xs*$)$›
$\langle proof \rangle$

lemma *inclusion_isotonicM_map*:
assumes ‹$\forall$ *x* $\in$ *set xs . g x* $\leq$ *h x*›
shows ‹$($*map g xs*$)$ $\leq_I$ $($*map h xs*$)$›
$\langle proof \rangle$

6.3 Interval Extension and Inclusion Properties

lemma *interval_extension_inclusion*:
assumes ‹*F is_interval_extension_of f*›
shows ‹$\forall$ *X . interval_of* $($*f X*$)$ $\leq$ *F* $($*interval_of X*$)$›
$\langle proof \rangle$

lemma *interval_extension_subset_const*:
assumes *interval_ext_F*: ‹*F is_interval_extension_of f*›
shows ‹$\forall$ *X . set_of* $($*interval_of* $($*f X*$))$ $\subseteq$ *set_of* $($*F* $($*interval_of X*$))$›
$\langle proof \rangle$

lemma *fundamental_theorem_of_interval_analysis*:
fixes *F* :: ‹*real interval* $\Rightarrow$ *real interval*›
assumes *interval_ext_F*: ‹*F is_interval_extension_of f*›
and *inc_isontonic_F*: ‹*inclusion_isotonic F*›
shows ‹$\forall$ *X . f* ‘ $($*set_of X*$)$ $\subseteq$ *set_of* $($*F X*$)$›
$\langle proof \rangle$

lemma *interval_extension_bounds*:
assumes *inclusion_isotonic F*
and *F is_interval_extension_of f*
shows $(\forall$ *x* $\in$ *set_of X. lower* $($*F X*$)$ $\leq$ *f x*$)$ $\vee$ $(\forall$ *x* $\in$ *set_of X. f x* $\leq$ *lower* $($*F X*$))$
$\langle proof \rangle$

lemma *inclusion_isotonic_extension_subset*:
assumes *inclusion_isotonic F*
and *F is_interval_extension_of f*
shows ‹$($*f* ‘ *set_of X*$)$ $\subseteq$ *set_of* $($*F X*$)$›
$\langle proof \rangle$

lemma *inclusion_isotonic_extension_includes*:
assumes *inclusion_isotonic F*
and *F is_interval_extension_of f*
shows ‹$\forall$ *x* $\in$ *set_of X*. *f x* $\in$ *set_of* (*F X*)›
$\langle proof \rangle$

lemma *inclusion_isotonic_extension_lower_bound*:
assumes *inclusion_isotonic F*
and *F is_interval_extension_of f*
shows ‹$\forall$ *x* $\in$ *set_of X*. *lower* (*F X*) $\leq$ *f x*›
$\langle proof \rangle$

lemma *inclusion_isotonic_extension_upper_bound*:
assumes *inclusion_isotonic F*
and *F is_interval_extension_of f*
shows ‹$\forall$ *x* $\in$ *set_of X*. *f x* $\leq$ *upper* (*F X*)›
$\langle proof \rangle$

lemma *inclusion_isotonic_inf*:
fixes *F*::‹*real interval* $\Rightarrow$ *real interval*›
assumes *inclusion_isotonic F*
and *F is_interval_extension_of f*
shows ‹*lower* (*F* (*X*::*real interval*)) $\leq$ *Inf* (*f* ‘ *set_of X*)›
$\langle proof \rangle$

lemma *inclusion_isotonic_sup*:
fixes *F*::‹*real interval* $\Rightarrow$ *real interval*›
assumes *inclusion_isotonic F*
and *F is_interval_extension_of f*
shows ‹*Sup* (*f* ‘ *set_of X*) $\leq$ *upper* (*F X*)›
$\langle proof \rangle$

lemma *lower_inf*:
fixes *F*::‹*real interval* $\Rightarrow$ *real interval*›
assumes *inclusion_isotonic F*
and *F is_interval_extension_of f*
shows *Inf* (*f* ‘ *set_of X*) $\leq$ *f* (*lower X*)
$\langle proof \rangle$
lemma *upper_sup*:
fixes *F*::‹*real interval* $\Rightarrow$ *real interval*›
assumes *inclusion_isotonic F*
and *F is_interval_extension_of f*
shows *f* (*upper X*) $\leq$ *Sup* (*f* ‘ *set_of X*)
$\langle proof \rangle$

lemma *lower_F_f*:
fixes *F*::‹*real interval* $\Rightarrow$ *real interval*›
assumes *inclusion_isotonic F*
and *F is_interval_extension_of f*
shows *lower* (*F X*) $\leq$ *f* (*lower X*)
$\langle proof \rangle$

lemma *upper_F_f*:

fixes *F*::‹*real interval* ⇒ *real interval*›
assumes *inclusion_isotonic F*
and *F is_interval_extension_of f*
shows *f* (*upper X*) ≤ *upper* (*F X*)
⟨*proof*⟩

lemma *inclusion_isotonic_interval_extension_le*:
assumes *inclusion_isotonic*: ‹*inclusion_isotonic F*›
and *interval_extension*: ‹*F is_interval_extension_of f*›
and *adjacent*: ‹*upper a* = *lower b*›
shows ‹*lower* (*F b*) ≤ *upper* (*F a*)›
⟨*proof*⟩

6.4 Division

context *interval_division_inverse*
begin
lemma *inclusion_isotonic_on_inverse*[*simp*]:
‹*inclusion_isotonic_on* (λ *x* . ¬ *0* $\in_i$ *x*) ((*inverse*::(′*a interval* ⇒ ′*a interval*)))›
⟨*proof*⟩

lemma *inclusion_isotonic_on_division*[*simp*]:
‹*inclusion_isotonic_binary_on* (λ *x* . ¬ *0* $\in_i$ *x*) (λ *x y*. *divide x y*)›
⟨*proof*⟩

end

end

7 Lipschitz Continuity of Intervals (Lipschitz_Interval_Extension)

An extension of of Lipschitz Continuity, developed for the Lipschitz Continuity of intervals.

```
theory
  Lipschitz_Interval_Extension
  imports
  Inclusion_Isotonicity
  HOL−Analysis.Lipschitz
  Interval_Utilities
begin
```

7.1 Definition of Lipschitz Continuity on Intervals

An interval extension F is said to be lipschitz in X if there is a constant L such that $w\ (F\ X) \leq Lw\ X$ for every $X \subseteq X$. Hence the width of $F\ X$ approaches $0::'a$ at least linearly with the width of X.

```
definition lipschitzI_on :: 'a::{zero,minus,preorder,times} ⇒ 'a interval set ⇒ ('a interval ⇒ 'a interval) ⇒ bool
  where lipschitzI_on C U F ⟷ (0 ≤ C ∧ (∀ x ∈ U. width (F x) ≤ C ∗ width x))
```

The definition *lipschitzI_on* refers to definition 6.1 in[4]

```
bundle lipschitzI_syntax begin
notation lipschitzI_on (_−lipschitzI′_on [1000])
end
bundle no_lipschitzI_syntax begin
no_notation lipschitzI_on (_−lipschitzI′_on [1000])
end

unbundle lipschitzI_syntax

lemma lipschitzI_onI: C−lipschitzI_on U F
  if ⋀x . x ∈ U ⟹ width (F x) ≤ C ∗ width x 0 ≤ C
  ⟨proof⟩

lemma lipschitzI_onD:
  width (F x) ≤ C ∗ width x
  if C−lipschitzI_on U F x ∈ U
  ⟨proof⟩

lemma lipschitzI_on_nonneg:
  0 ≤ C if C−lipschitzI_on U F
  ⟨proof⟩

lemma lipschitz_on_normD:
  norm (width (F x)) ≤ C ∗ norm (width x)
  if C−lipschitzI_on U F x ∈ U
  ⟨proof⟩
```

lemma *lipschitzI_on_mono*: $L-lipschitzI\_on\ D\ (F:: 'a::\{linordered\_ring\}\ interval \Rightarrow 'a\ interval)$
if $M-lipschitzI\_on\ E\ F\ M \leq L\ D \subseteq E$
$\langle proof \rangle$

lemmas *lipschitzI_on_subset*
$= lipschitzI\_on\_mono[OF\ \_\ \_\ order\_refl]$
and $lipschitzI\_on\_le = lipschitzI\_on\_mono[OF\ \_\ order\_refl]$

lemma *lipschitzI_on_leI*:
$C-lipschitzI\_on\ U\ F$
if $\bigwedge x.\ x \in U \Longrightarrow width\ (F\ x) \leq C * width\ x$
$0 \leq C$
for $F::'a::\{minus,preorder,times,zero\}\ interval \Rightarrow 'a\ interval$
$\langle proof \rangle$

7.1.1 Lipschitz Continuity of Operations

Let F and G be inclusion isotonic interval extensions with F Lipschitz in Y and G Lipschitz in X and $G\ X \subseteq Y$. Then the composition $H\ X = F\ (G\ X)$ is Lipschitz in X and is inclusion isotonic

lemma *lipschitzI_on_compose*:
fixes U :: ‹$'a::\{linordered\_ring\}\ interval\ set$›
assumes f: $C-lipschitzI\_on\ U\ F$ **and** g: $D-lipschitzI\_on\ (F`U)\ G$
shows $(D * C)-lipschitzI\_on\ U\ (G\ o\ F)$
$\langle proof \rangle$

The definition $[\![?C-lipschitzI\_on\ ?U\ ?F;\ ?D-lipschitzI\_on\ (?F\ `\ ?U)\ ?G]\!] \Longrightarrow (?D * ?C)-lipschitzI\_on\ ?U\ (?G \circ ?F)$ refers to lemma 6.3 in [4]

lemma *lipschitzI_on_compose2*:
fixes U :: ‹$'a::\{linordered\_ring\}\ interval\ set$›
assumes F: $C-lipschitzI\_on\ U\ F$ **and** G: $D-lipschitzI\_on\ (F`U)\ G$
shows $(D * C)-lipschitzI\_on\ U\ (\lambda x.\ G\ (F\ x))$
$\langle proof \rangle$

lemma *lipschitzI_on_cong*:
$C-lipschitzI\_on\ U\ G \longleftrightarrow D-lipschitzI\_on\ V\ F$
if $C = D\ U = V\ \bigwedge x.\ x \in V \Longrightarrow G\ x = F\ x$
$\langle proof \rangle$

lemma *lipschitzI_on_transform*:
$C-lipschitzI\_on\ U\ G$
if $C-lipschitzI\_on\ U\ F$
$\bigwedge x.\ x \in U \Longrightarrow G\ x = F\ x$
$\langle proof \rangle$

lemma *lipschitz_on_empty_iff*: $C-lipschitzI\_on\ \{\}\ f \longleftrightarrow C \geq 0$
$\langle proof \rangle$

lemma *lipschitz_on_id*: $(1::real)-lipschitzI\_on\ U\ (\lambda x.\ x)$
$\langle proof \rangle$

lemma *lipschitz_on_constant*:
assumes ‹$lower\ c = upper\ c$›

shows $(0::real)-lipschitzI\_on\ U\ (\lambda x.\ c)$
⟨proof⟩

lemma *lipschitzI_on_mult*:
fixes $X :: 'a::\{linordered\_idom\}$
assumes *lipschitzI_on C U f*
and $1 \le X$
shows $lipschitzI\_on\ (X*C)\ U\ f$
⟨proof⟩

7.1.2 Interval bounds on reals

lemma *bounded_image_real*:
fixes $X :: real\ interval$ **and** $f :: real \Rightarrow real$
assumes $\forall x\in\{lower\ X..upper\ X\}.$
$f\ (lower\ X) - L * (upper\ X - lower\ X) \le f\ x \wedge f\ x \le f\ (lower\ X) + L * (upper\ X - lower\ X)$
shows $\exists x\ e.\ \forall y\in f\ `\ \{lower\ X..upper\ X\}.\ dist\ x\ y \le e$
⟨proof⟩

lemma *lipschitz_bounded_image_real*:
fixes $X :: real\ set$ **and** $f :: real \Rightarrow real$
assumes *bounded X* **and** *L−lipschitz_on X f*
shows $bounded\ (f\ `\ X)$
⟨proof⟩

lemma *inf_le_sup_image_real*:
fixes $X :: real\ interval$ **and** $f :: real \Rightarrow real$
assumes $L-lipschitz\_on\ (set\_of\ X)\ f$
shows $Inf\ (f\ `\ set\_of\ X) \le Sup\ (f\ `\ set\_of\ X)$
⟨proof⟩

lemma *sup_image_le_real*:
fixes $f :: real \Rightarrow real$ **and** $F :: real\ interval \Rightarrow real\ interval$ **and** $X :: real\ interval$
assumes $f\ `\ set\_of\ X \subseteq set\_of\ (F\ X)$
and $L-lipschitz\_on\ (set\_of\ X)\ f$
shows $Sup\ (f\ `\ set\_of\ X) \le Sup\ (set\_of\ (F\ X))$
⟨proof⟩

lemma *inf_image_le_real*:
fixes $f :: real \Rightarrow real$ **and** $F :: real\ interval \Rightarrow real\ interval$ **and** $X :: real\ interval$
assumes $f\ `\ set\_of\ X \subseteq set\_of\ (F\ X)$
and $L-lipschitz\_on\ (set\_of\ X)\ f$
shows $Inf\ (set\_of\ (F\ X)) \le Inf\ (f\ `\ (set\_of\ X))$
⟨proof⟩

end

8 Multi-Intervals (Multi_Interval_Preliminaries)

8.1 Preliminaries

```
theory
  Multi_Interval_Preliminaries
imports
  HOL−Library.Interval
  HOL−Analysis.Analysis
  Inclusion_Isotonicity
begin
```

8.1.1 A Class for Capturing Monotonicity of Minus

We try to keep our formalisation of interval arithmetic as generic as possible. In particular, we want to support intervals of type *nat*, *int*, *real*, and *ereal*. For all these types, minus (substraction) is monotonous. Sadly, Isabelle lacks a type class capturing this. Luckily, it is very easy to define our own:

```
class minus_mono = minus + linorder +
  assumes minus_mono: A ≤ B ⟹ D ≤ C ⟹ A − C ≤ B − D
begin end

instance nat::minus_mono
  ⟨proof⟩
instance int::minus_mono
  ⟨proof⟩
instance real::minus_mono
  ⟨proof⟩
instance integer::minus_mono
  ⟨proof⟩
instance ereal::minus_mono
  ⟨proof⟩
```

8.1.2 Infrastructure for Lifting Interval Operations to Lists of Intervals

```
definition un_op_interval_list::‹('a interval ⇒ 'a interval)
                        ⇒ 'a interval list ⇒ 'a interval list›
  where
  ‹un_op_interval_list op xs = map op xs›


definition bin_op_interval_list::‹('a interval ⇒ 'a interval ⇒ 'a interval)
                       ⇒ 'a interval list ⇒ 'a interval list ⇒ 'a interval list›
  where
  ‹bin_op_interval_list op xs ys = concat (map (λ x . map (op x) xs) ys)›

lemma bin_op_interval_list_non_empty: ‹(xs ≠ [] ∧ ys ≠ []) = (bin_op_interval_list op xs ys ≠ [])›
  ⟨proof⟩
```

lemma *bin_op_interval_list_empty*: ‹$(xs = [] \vee ys = []) = (bin\_op\_interval\_list\ op\ xs\ ys = [])$›
⟨*proof*⟩

lemma *bin_op_interval_unroll*: ‹$bin\_op\_interval\_list\ op\ (xs)\ (y\#ys) = (map\ (op\ y)\ xs)@(\ bin\_op\_interval\_list\ op\ xs\ ys)$›
⟨*proof*⟩

lemma *bin_op_interval_list_commute*:
assumes *op_commute*: ‹$\bigwedge x\ y.\ op\ x\ y = op\ y\ x$›
shows ‹$set\ (bin\_op\_interval\_list\ (op)\ xs\ ys) = set\ (bin\_op\_interval\_list\ (op)\ ys\ xs)$›
⟨*proof*⟩

lemma *bin_op_interval_list_assoc*:
assumes *op_assoc*: ‹$\bigwedge x\ y\ z.\ op\ (op\ x\ y)\ z = op\ x\ (op\ y\ z)$›
shows ‹$set\ (bin\_op\_interval\_list\ (op)\ ((bin\_op\_interval\_list\ (op)\ xs\ ys))\ zs) = set\ (bin\_op\_interval\_list\ (op)\ xs\ ((bin\_op\_interval\_list\ (op)\ ys\ zs)))$›
⟨*proof*⟩

Lifting Unary Minus, Addition, and Multiplication

definition ‹$iList\_uminus = un\_op\_interval\_list\ (\lambda x.\ - x)$›
definition ‹$iList\_plus\ \ = bin\_op\_interval\_list\ (+)$›
definition ‹$iList\_times\ \ = bin\_op\_interval\_list\ (*)$›

lemma *iList_plus_lower*:
assumes ‹$A \neq []$› **and** ‹$B \neq []$›
shows ‹$lower\ (hd\ (iList\_plus\ A\ B)) = lower\ (hd\ A) + lower\ (hd\ B)$›
⟨*proof*⟩

lemma *iList_plus_upper*:
assumes ‹$A \neq []$› **and** ‹$B \neq []$›
shows ‹$upper\ (hd\ (iList\_plus\ A\ B)) = upper\ (hd\ A) + upper\ (hd\ B)$›
⟨*proof*⟩

lemma *iList_plus_unroll*:
‹$iList\_plus\ ys\ (x\ \#\ xs) = map\ ((+)\ x)\ ys\ @\ iList\_plus\ ys\ xs$›
⟨*proof*⟩

lemma $a \neq [] \Longrightarrow (iList\_plus\ [Interval\ (0, 0)]\ a) = (a::'a::\{ordered\_ab\_group\_add,zero\}\ interval\ list)$
⟨*proof*⟩

lemma *iList_plus_commute*:
‹$set\ (iList\_plus\ xs\ ys) = set\ (iList\_plus\ ys\ xs)$›
⟨*proof*⟩

lemma *iList_plus_assoc*:
‹$set\ (iList\_plus\ xs\ (iList\_plus\ ys\ zs)) = set\ (iList\_plus\ (iList\_plus\ xs\ ys)\ zs)$›
⟨*proof*⟩

lemma *remdups_append1*:
$remdups\ (remdups\ xs\ @\ ys) = remdups\ (xs\ @\ ys)$
⟨*proof*⟩

lemma *bin_op_interval_remdups_left*:
‹*remdups* (*bin_op_interval_list op* (*remdups x*) *y*) = *remdups* (*bin_op_interval_list op x y*)›
⟨*proof*⟩

lemma *iList_plus_remdups_left*:
remdups (*iList_plus* (*remdups a*) *b*) = *remdups* (*iList_plus a b*)
for *a*::'*a*::{*minus_mono*,*ordered_ab_semigroup_add*,*linorder*,*linordered_field*} *interval list*
⟨*proof*⟩

lemma *interval_add_eq*: ‹*a* + *b* = *Interval*(*lower a* + *lower b*, *upper a* + *upper b*)›
⟨*proof*⟩

lemma *iList_plus_lower_upper_eq*:
‹*iList_plus* = *bin_op_interval_list* (λ *a b*. *Interval*(*lower a* + *lower b*, *upper a* + *upper b*))›
⟨*proof*⟩

A Locale for Multi-Interval Division Where the Quotient does not Contain *0*

context *interval_division*
begin

definition ‹*iList_inverse* = *un_op_interval_list inverse*›
definition ‹*iList_divide* = *bin_op_interval_list divide*›

end

8.1.3 Utilities for (Sorted) Lists of Intervals

definition ‹*cmp_lower_width* = (λ *x y*. *if lower x* = *lower y then width x* ≤ *width y else lower x* < *lower y*)›

definition ‹*sorted_wrt_lower* = *sorted_wrt cmp_lower_width*›

lemma *interval_lower_width_eq*:
‹(*lower x* = *lower y* ∧ *width x* = *width y*) = (*x* = (*y*::'*a*::{*minus_mono*, *linordered_field*} *interval*))›
⟨*proof*⟩

lemma *sorted_wrt_lower_distinct_lists_eq*:
assumes ‹*set xs* = *set* (*ys*::'*a*::{*minus_mono*, *linordered_field*} *interval list*)›
and ‹*distinct xs*› **and** ‹*distinct ys*›
and ‹*sorted_wrt_lower xs*› **and** ‹*sorted_wrt_lower ys*›
shows‹ *xs* = *ys*›
⟨*proof*⟩

definition ‹*sorted_wrt_upper* = *sorted_wrt* (λ *x y*. *upper x* ≤ *upper y*)›

definition ‹*cmp_non_overlapping* = (λ *x y*. *upper x* ≤ *lower y*)›

lemma *cmp_non_overlapping_lower*: ‹*cmp_non_overlapping x y* ⟹ *lower x* ≤ *lower y*›
⟨*proof*⟩

lemma *cmp_non_overlapping_upper*: ‹*cmp_non_overlapping x y* ⟹ *upper x* ≤ *upper y*›
⟨*proof*⟩

definition ‹*non_overlapping_sorted* = *sorted_wrt cmp_non_overlapping*›

definition ‹contiguous xs = (∀ i < length xs − 1 . upper (xs ! i) = lower (xs ! (i + 1)))›

lemma *non_overlapping_sorted_empty*: ‹non_overlapping_sorted []›
⟨proof⟩

lemma *non_overlapping_sorted_unroll*:
assumes xs ≠ [] **shows** non_overlapping_sorted (x # xs) = (upper x ≤ lower (hd xs) ∧ non_overlapping_sorted xs)
⟨proof⟩

lemma *contiguous_non_overlapping*: ‹contiguous (is::'a::{preorder} interval list) ⟹ non_overlapping_sorted is›
⟨proof⟩

definition ‹cmp_non_adjacent = (λ x y. upper x < lower y)›

lemma *cmp_non_adjacent_lower*: ‹cmp_non_adjacent x y ⟹ lower x < lower y›
⟨proof⟩

lemma *cmp_non_adjacent_upper*: ‹cmp_non_adjacent x y ⟹ upper x < upper y›
⟨proof⟩

definition ‹non_adjacent_sorted_wrt_lower = sorted_wrt cmp_non_adjacent›

lemma *non_adjacent_sorted_wrt_lower_unroll*:
assumes xs ≠ []
shows non_adjacent_sorted_wrt_lower (x # xs) =
((upper x < lower (hd xs)) ∧ non_adjacent_sorted_wrt_lower xs)
⟨proof⟩

lemma *non_adjacent_implies_non_overlapping*:
assumes ‹non_adjacent_sorted_wrt_lower is› **shows** ‹non_overlapping_sorted is›
⟨proof⟩

lemma *non_overlapping_implies_sorted_wrt_lower*:
assumes ‹non_overlapping_sorted (is::'a::{minus_mono} interval list)›
shows ‹sorted_wrt_lower is›
⟨proof⟩

lemma *non_overlapping_implies_sorted_wrt_upper*:
assumes ‹non_overlapping_sorted is›
shows ‹sorted_wrt_upper is›
⟨proof⟩

lemma *non_adjacent_implies_sorted_wrt_lower*:
assumes ‹non_adjacent_sorted_wrt_lower is›
shows ‹sorted_wrt_lower is›
⟨proof⟩

lemma *non_adjacent_implies_distinct*:
assumes ‹non_adjacent_sorted_wrt_lower is›
shows ‹distinct is›
⟨proof⟩

fun *merge_overlapping_intervals_sorted_wrt_lower* :: *'a::linorder interval list* ⇒ *'a interval list* **where**
merge_overlapping_intervals_sorted_wrt_lower [] = [] |
merge_overlapping_intervals_sorted_wrt_lower [x] = [x] |
merge_overlapping_intervals_sorted_wrt_lower (x#y#ys) =
(*if upper x* ≤ *lower y*
then x#(*merge_overlapping_intervals_sorted_wrt_lower* (y#ys))
else merge_overlapping_intervals_sorted_wrt_lower ((*mk_interval*(*lower x*, *max* (*upper x*) (*upper y*)))#ys)
)

lemma *sorted_wrt_lower_unroll*:
assumes *xs* ≠ []
shows *sorted_wrt_lower* (x # xs) = ((*if lower x* ≠ *lower* (*hd xs*)
then lower x < *lower* (*hd xs*)
else width x ≤ *width* (*hd xs*)) ∧ *sorted_wrt_lower* (xs))
⟨*proof*⟩

lemma *sorted_wrt_upper_unroll*:
assumes *xs* ≠ []
shows *sorted_wrt_upper* (x # xs) = ((*upper x* ≤ *upper* (*hd xs*))∧ *sorted_wrt_upper* (xs))
⟨*proof*⟩

lemma *sorted_wrt_lower_tail*: *sorted_wrt_lower* (x # xs) ⟹ *sorted_wrt_lower* (xs)
⟨*proof*⟩

lemma *sorted_wrt_lower_tail'*:*sorted_wrt_lower* (x # y # ys) ⟹ *sorted_wrt_lower* (x # ys)
⟨*proof*⟩

lemma *iList_plus_leq_B*:
assumes *sorted_wrt_lower A* **and** *sorted_wrt_lower B* **and** *1* < *length B*
shows *hd* (*map lower* (*iList_plus A B*)) ≤ *hd* (*map lower* (*iList_plus A* (*tl B*)))
⟨*proof*⟩

lemma *iList_plus_leq_A*:
assumes *sorted_wrt_lower A* **and** *sorted_wrt_lower B* **and** *1* < *length A*
shows *hd* (*map lower* (*iList_plus A B*)) ≤ *hd* (*map lower* (*iList_plus* (*tl A*) *B*))
⟨*proof*⟩

value ‹*merge_overlapping_intervals_sorted_wrt_lower* [*mk_interval*(1::*int*,2), *mk_interval*(2,3), *mk_interval*(5,7), *mk_interval*(6,10)]›

lemma *merge_overlapping_intervals_sorted_wrt_lower_non_nil*:
assumes ‹*xs* ≠ []›
shows ‹(*merge_overlapping_intervals_sorted_wrt_lower xs*) ≠ []›
⟨*proof*⟩

lemma *merge_overlapping_intervals_sorted_hd_lower*:
assumes ‹*xs* ≠ []›
shows *lower* (*hd* (*merge_overlapping_intervals_sorted_wrt_lower* (xs))) = *lower* (*hd xs*)
⟨*proof*⟩

lemma *merge_overlapping_intervals_sorted_hd_upper*:
assumes ‹*xs* ≠ []›

```
 shows upper (hd xs) ≤ upper (hd (merge_overlapping_intervals_sorted_wrt_lower xs))
⟨proof⟩

lemma Interval_id[simp]: ‹Interval (lower x, upper x) = x ›
 ⟨proof⟩

lemma mk_interval_id[simp]: ‹(mk_interval (lower x, upper x)) = x›
 ⟨proof⟩

lemma merge_overlapping_intervals_sorted_hd_width:
 assumes ‹xs ≠ []›
 shows width (hd xs) ≤ width (hd (merge_overlapping_intervals_sorted_wrt_lower (xs::'a::{minus_mono} interval
list)))
⟨proof⟩



lemma merge_overlapping_intervals_sorted_wrt_lower_sorted_lower:
 assumes ‹sorted_wrt_lower (xs::'a::{minus_mono} interval list)›
 shows ‹sorted_wrt_lower (merge_overlapping_intervals_sorted_wrt_lower xs)›
⟨proof⟩

lemma merge_overlapping_intervals_sorted_sorted_non_non_overlapping:
 assumes ‹sorted_wrt_lower (xs::'a::{minus_mono} interval list)›
 shows ‹non_overlapping_sorted (merge_overlapping_intervals_sorted_wrt_lower xs)›
 ⟨proof⟩



fun merge_adjacent_intervals_sorted_wrt_lower :: 'a::linorder interval list ⇒ 'a interval list  where
merge_adjacent_intervals_sorted_wrt_lower [] = [] |
merge_adjacent_intervals_sorted_wrt_lower [x] = [x] |
merge_adjacent_intervals_sorted_wrt_lower (x#y#ys) =
   (  if upper x < lower y
     then x#(merge_adjacent_intervals_sorted_wrt_lower (y#ys))
     else merge_adjacent_intervals_sorted_wrt_lower ((mk_interval(lower x, max (upper y) (upper x)))#ys)
    )

value ‹merge_adjacent_intervals_sorted_wrt_lower [mk_interval(1::int,2), mk_interval(2,3), mk_interval(5,7),
mk_interval(6,10)]›

lemma merge_adjacent_intervals_sorted_wrt_lower_non_nil:
 assumes ‹xs ≠ []›
 shows ‹(merge_adjacent_intervals_sorted_wrt_lower xs) ≠ []›
 ⟨proof⟩

lemma merge_adjacent_intervals_sorted_wrt_lower_non_nil':
 shows ‹(merge_adjacent_intervals_sorted_wrt_lower (x#xs)) ≠ []›
 ⟨proof⟩

lemma merge_adjacent_intervals_sorted_wrt_lower_sorted_lower_lower_hd:
 assumes ‹sorted_wrt_lower xs›
 shows ‹lower (hd (merge_adjacent_intervals_sorted_wrt_lower xs)) = lower (hd xs)›
 ⟨proof⟩
```

lemma *merge_adjacent_intervals_sorted_wrt_lower_sorted_lower_lower_subset*:
‹*set* (*map lower* (*merge_adjacent_intervals_sorted_wrt_lower xs*)) ⊆ *set* (*map lower xs*)›
⟨*proof*⟩

lemma *merge_adjacent_intervals_sorted_wrt_lower_set_eq*:
assumes ‹*set* (*xs*::*real interval list*) = *set ys*›
and ‹*distinct xs*› **and** ‹*distinct ys*›
and ‹*sorted_wrt_lower xs*› **and** ‹*sorted_wrt_lower ys*›
shows ‹*merge_adjacent_intervals_sorted_wrt_lower xs* = *merge_adjacent_intervals_sorted_wrt_lower ys*›
⟨*proof*⟩

lemma *merge_adjacent_intervals_sorted_wrt_lower_lower_upper*:
assumes *sorted_wrt_lower xs*
shows *x* ∈ *set* (*merge_adjacent_intervals_sorted_wrt_lower xs*) ⟹ ∃ *l* ∈ *set xs*. ∃ *u* ∈ *set xs*. *lower l* = *lower x* ∧ *upper u* = *upper x*
⟨*proof*⟩

primrec *interval_insert_sort_lower_width* :: (*'a*::{*linorder*,*minus*}) *interval* ⇒ *'a interval list* ⇒ *'a interval list* **where**
interval_insert_sort_lower_width x [] = [*x*] |
interval_insert_sort_lower_width x (*y*#*ys*) =
(*if cmp_lower_width x y then* (*x*#*y*#*ys*) *else y*#(*interval_insert_sort_lower_width x ys*))

lemma *interval_insert_sort_lower_width_length*:
‹*length* (*interval_insert_sort_lower_width x xs*) = 1 + *length xs* ›
⟨*proof*⟩

lemma *interval_insert_sort_lower_width_nonempty*:
‹*interval_insert_sort_lower_width x xs* ≠ []›
⟨*proof*⟩

lemma *interval_insert_sort_wrt_lower*:
‹*sorted_wrt_lower xs* ⟹ *sorted_wrt_lower* (*interval_insert_sort_lower_width x xs*)›
⟨*proof*⟩

lemma *interval_isort_elements*: *set* (*interval_insert_sort_lower_width x xs*) = {*x*} ∪ *set xs*
⟨*proof*⟩

lemma *foldr_isort_elements*: *set* (*foldr interval_insert_sort_lower_width xs* []) = *set xs*
⟨*proof*⟩

definition *interval_sort_lower_width* :: (*'a*::{*linorder*,*minus*}) *interval list* ⇒ *'a interval list* **where**
interval_sort_lower_width xs = *foldr interval_insert_sort_lower_width xs* []

lemma *interval_sort_lower_width_length*: ‹*length* (*interval_sort_lower_width xs*) = (*length xs*)›
⟨*proof*⟩

lemma *interval_sort_lower_width_sorted*: ‹*sorted_wrt_lower* (*interval_sort_lower_width xs*)›
⟨*proof*⟩

lemma *interval_sort_lower_width_set_eq*:
‹*set* (*interval_sort_lower_width x*) = *set x*›
$\langle proof \rangle$

lemma *interval_sort_lower_width_remdups*:
‹*remdups* (*interval_sort_lower_width* (*remdups xs*)) = *interval_sort_lower_width* (*remdups xs*)›
$\langle proof \rangle$

lemma *interval_sort_lower_width_distinct*:
assumes ‹*distinct xs*› **shows**
‹*distinct* (*interval_sort_lower_width* (*remdups xs*))›
$\langle proof \rangle$

lemma *foldr_interval_insert_sort_lower_width_distinct*:
assumes ‹*distinct zs*›
shows ‹*distinct* (*foldr interval_insert_sort_lower_width zs* [])›
$\langle proof \rangle$

lemma *non_overlapping_sorted_remdups*:
non_overlapping_sorted xs $\Longrightarrow$ *non_overlapping_sorted* (*remdups xs*)
$\langle proof \rangle$

lemma *insert_in_lower_width*: $x \in$ *set* (*interval_insert_sort_lower_width a list*) = ($x = a \vee x \in$ *set list*)
$\langle proof \rangle$

lemma *remdups_set_eq*:
assumes ‹*set xs* = *set ys*›
shows ‹*set* (*remdups xs*) = *set* (*remdups ys*)›
$\langle proof \rangle$

lemma *remdups_lower_hd*:
assumes *xs* $\neq$ [] **and** *sorted_wrt_lower xs*
shows (*lower* $\circ$ *hd*) (*remdups xs*) = (*lower* $\circ$ *hd*) *xs*
$\langle proof \rangle$

8.1.4 Various Notions of Validity of Sorted Lists of Intervals

Validity Tests

definition ‹*valid_mInterval_ovl is* = (*sorted_wrt_lower is* $\wedge$ *distinct is* $\wedge$ *is* $\neq$ [])›

The predicate *valid_mInterval_ovl* requires that a list of intervals is distinct and sorted with respect to the lower bound of each interval.

definition *valid_mInterval_adj* :: *'a::minus_mono interval list* $\Rightarrow$ *bool*
where ‹*valid_mInterval_adj is* = (*non_overlapping_sorted is* $\wedge$ *distinct is* $\wedge$ *is* $\neq$ [])›

The predicate *valid_mInterval_adj* is strictly stronger than *valid_mInterval_ovl*:

lemma *valid_adj_imp_ovl*: ‹*valid_mInterval_adj x* $\Longrightarrow$ *valid_mInterval_ovl x*›
$\langle proof \rangle$

Informally,*valid_mInterval_ovl* further limits the list of intervals to be non-overlapping. Note that adjacent intervals (i.e., intervals that share the same bounds) are allowed. For example:

lemma *valid_mInterval_adj* [*Interval*(1::*int*,2), *Interval*(2,3)]
⟨*proof*⟩

definition ‹*valid_mInterval_non_ovl is* = (*valid_mInterval_ovl is* ∧ *non_adjacent_sorted_wrt_lower is*)›

Informally,*valid_mInterval_non_ovl* further limits the list of intervals to also forbid adjacent intervals (i.e., intervals that share the same bounds) are allowed. It is strictly stronger than the other two predicates:

lemma *valid_non_ovl_imp_ovl*: ‹*valid_mInterval_non_ovl x* ⟹ *valid_mInterval_ovl x*›
⟨*proof*⟩
lemma *valid_non_ovl_imp_adj*: ‹*valid_mInterval_non_ovl x* ⟹ *valid_mInterval_adj x*›
⟨*proof*⟩

lemma *valid_mInterval_non_ovl_sorted*: *valid_mInterval_non_ovl xs* ⟹ *sorted_wrt_lower xs*
⟨*proof*⟩

lemma *valid_mInterval_non_ovl_unroll*:
‹*ys* ≠ [] ⟹ *valid_mInterval_non_ovl* (*y* # *ys*) ⟹ *valid_mInterval_non_ovl ys*›
⟨*proof*⟩

lemma *valid_mInterval_non_ovl_eqI*:
assumes ‹*valid_mInterval_non_ovl xs*›
and ‹*valid_mInterval_non_ovl ys*›
and ‹*set xs* = *set ys*›
shows ‹*xs* = *ys*›
⟨*proof*⟩

Constructors

Overlapping Intervals **definition** ‹ *mk_mInterval_ovl* = *remdups o interval_sort_lower_width*›

lemma *mk_mInterval_ovl_non_empty*: ‹*is* ≠ [] ⟹ (*mk_mInterval_ovl is*) ≠ []›
⟨*proof*⟩

lemma *mk_mInterval_ovl_empty*[*simp*]:
mk_mInterval_ovl [] = []
⟨*proof*⟩

lemma *mk_mInterval_ovl_distinct*: ‹*distinct* (*mk_mInterval_ovl is*)›
⟨*proof*⟩

lemma *sorted_wrt_lower_remdups*:
sorted_wrt_lower xs ⟹ *sorted_wrt_lower* (*remdups xs*)
⟨*proof*⟩

lemma *interval_sort_lower_width_swap_remdups*:
‹*remdups* (*interval_sort_lower_width xs*) = *interval_sort_lower_width* (*remdups xs*)›
for *xs*::′*a*::{*minus_mono*, *linordered_field*} *interval list*
⟨*proof*⟩

lemma *mk_mInterval_ovl_sorted*: ‹*sorted_wrt_lower* (*mk_mInterval_ovl is*)›
⟨*proof*⟩

theorem *mk_mInterval_ovl_valid*: ‹*is* ≠ [] ⟹ *valid_mInterval_ovl* (*mk_mInterval_ovl is*)›
⟨*proof*⟩

lemma *valid_mk_mInterval_ovl_id*:
assumes ‹*valid_mInterval_ovl xs*›
shows ‹*mk_mInterval_ovl xs* = *xs*›
⟨*proof*⟩

lemma *mk_mInterval_ovl_eq*:
assumes ‹*set xs* = *set* (*ys*::'*a*::{*minus_mono*, *linordered_field*} *interval list*)›
shows ‹*mk_mInterval_ovl xs* = *mk_mInterval_ovl ys* ›
⟨*proof*⟩

Adjacent Intervals **definition** *mk_mInterval_adj* :: ('*a*::{*minus*, *linorder*,*linorder*}) *interval list* ⇒ '*a interval list*
where ‹*mk_mInterval_adj* = *remdups o merge_overlapping_intervals_sorted_wrt_lower o mk_mInterval_ovl*›

lemma *mk_mInterval_adj_non_overlapping_sorted*: ‹*non_overlapping_sorted* (*mk_mInterval_adj* (*is*::'*a*::{*minus_mono*} *interval list*))›
⟨*proof*⟩

lemma *mk_mInterval_adj_sorted*: ‹*sorted_wrt_lower* (*mk_mInterval_adj* (*is*::'*a*::*minus_mono interval list*))›
⟨*proof*⟩

lemma *mk_mInterval_adj_non_empty*: ‹*is* ≠ [] ⟹ (*mk_mInterval_adj is*) ≠ []›
⟨*proof*⟩

lemma *mk_mInterval_adj_empty*[*simp*]:
mk_mInterval_adj [] = []
⟨*proof*⟩

lemma *mk_mInterval_adj_distinct*: ‹*distinct* (*mk_mInterval_adj is*)›
⟨*proof*⟩

theorem *mk_mInterval_adj_valid*: ‹*is* ≠ [] ⟹ *valid_mInterval_adj* (*mk_mInterval_adj is*)›
⟨*proof*⟩

lemma *valid_mk_mInterval_adj_id*:
assumes ‹*valid_mInterval_adj xs*›
shows ‹*mk_mInterval_adj xs* = *xs*›
⟨*proof*⟩

lemma *mk_mInterval_adj_eq*:
assumes ‹*set xs* = *set* (*ys*::'*a*::{*minus_mono*, *linordered_field*} *interval list*)›
shows ‹*mk_mInterval_adj xs* = *mk_mInterval_adj ys* ›
⟨*proof*⟩

lemma *mk_mInterval_ovl_id*:
mk_mInterval_ovl (*mk_mInterval_ovl x*) = *mk_mInterval_ovl x*
⟨*proof*⟩

value *valid_mInterval_adj* (*mk_mInterval_adj* ([*Ivl* (1::*int*) 2, *Ivl* 1 3, *Ivl* 1 1]))

value *valid_mInterval_adj* $($*mk_mInterval_adj* $([$*Ivl* $(1::int)$ *1*, *Ivl 1 2*, *Ivl 2 3*$]))$

Non-Overlapping Intervals definition ‹*mk_mInterval_non_ovl* = *remdups* o *merge_adjacent_intervals_sorted_wrt_lower* o *mk_mInterval_ovl*›

lemma *mk_mInterval_non_ovl_distinct*:
distinct (*mk_mInterval_non_ovl is*)
⟨*proof*⟩

lemma *mk_mInterval_non_ovl_non_empty*:
is ≠ [] ⟹ *mk_mInterval_non_ovl is* ≠ []
⟨*proof*⟩

lemma *mk_mInterval_non_ovl_empty*[*simp*]:
mk_mInterval_non_ovl [] = []
⟨*proof*⟩

lemma *mk_mInterval_non_ovl_eq*:
assumes ‹*set xs* = *set* (*ys*::'*a*::{*minus_mono*, *linordered_field*} *interval list*)›
shows ‹*mk_mInterval_non_ovl xs* = *mk_mInterval_non_ovl ys* ›
⟨*proof*⟩

lemma *sorted_wrt_lower_merge_adjacent_intervals_sorted_wrt_lower*:
sorted_wrt_lower xs ⟹ *sorted_wrt_lower* (*merge_adjacent_intervals_sorted_wrt_lower xs*)
⟨*proof*⟩

lemma *mk_mInterval_non_ovl_sorted_wrt_lower*:
is ≠ [] ⟹ *sorted_wrt_lower* (*mk_mInterval_non_ovl* (*is*::*int interval list*))
⟨*proof*⟩

lemma *valid_ovl_mkInterval_non_ovl*: *is* ≠ [] ⟹ *valid_mInterval_ovl* (*mk_mInterval_non_ovl is*)
⟨*proof*⟩

lemma *non_adj_sorted_mkInterval_non_ovl*:
sorted_wrt_lower xs
⟹ *non_adjacent_sorted_wrt_lower* (*merge_adjacent_intervals_sorted_wrt_lower xs*)
⟨*proof*⟩

lemma *bin_op_mInterval_commute*:
assumes *op_commute*: ‹⋀ *x y*. *op x y* = *op y x*›
shows ‹*mk_mInterval_non_ovl* (*bin_op_interval_list op x y*) = *mk_mInterval_non_ovl* (*bin_op_interval_list op y* (*x*::'*a*::{*minus_mono*, *linordered_field*} *interval list*))›
⟨*proof*⟩

lemma *iList_plus_mInterval_ovl_assoc*:
‹*mk_mInterval_ovl* (*iList_plus x* (*iList_plus y z*)) = *mk_mInterval_ovl* (*iList_plus* (*iList_plus x* (*y*::'*a*::{*minus_mono*, *linordered_field*} *interval list*)) *z*)›
⟨*proof*⟩

lemma *iList_plus_mInterval_adj_commute*:
‹*mk_mInterval_adj* (*iList_plus x y*) = *mk_mInterval_adj* (*iList_plus y* (*x*::'*a*::{*minus_mono*, *linordered_field*} *interval*

list))›
⟨*proof*⟩

lemma *iList_plus_mInterval_non_ovl_assoc*:
‹*mk_mInterval_non_ovl* (*iList_plus* *x* (*iList_plus* *y* *z*)) = *mk_mInterval_non_ovl* (*iList_plus* (*iList_plus* *x* (*y*::′*a*::{*minus_mono*, *linordered_field*} *interval list*)) *z*)›
⟨*proof*⟩

lemma *iList_plus_mInterval_non_ovl_commute*:
‹*mk_mInterval_non_ovl* (*iList_plus x y*) = *mk_mInterval_non_ovl* (*iList_plus y* (*x*::′*a*::{*minus_mono*, *linordered_field*} *interval list*))›
⟨*proof*⟩

lemma *iList_plus_mInterval_adj_assoc*:
‹*mk_mInterval_non_ovl* (*iList_plus* *x* (*iList_plus* *y* *z*)) = *mk_mInterval_non_ovl* (*iList_plus* (*iList_plus* *x* (*y*::′*a*::{*minus_mono*, *linordered_field*} *interval list*)) *z*)›
⟨*proof*⟩

lemma *sorted_wrt_lower_mk_mInterval_non_ovl*: *sorted_wrt_lower* (*mk_mInterval_non_ovl xs*)
⟨*proof*⟩

theorem *mk_mInterval_non_ovl_valid*: ‹*sorted_wrt_lower is* ⟹ *is* ≠ [] ⟹ *valid_mInterval_non_ovl* (*mk_mInterval_non_ovl is*)›
⟨*proof*⟩

lemma *valid_mk_mInterval_non_ovl_id*:
assumes ‹*valid_mInterval_non_ovl xs*›
shows ‹*mk_mInterval_non_ovl xs* = *xs*›
⟨*proof*⟩

lemma *mk_mInterval_non_ovl_single*:
mk_mInterval_non_ovl [*x*] = [*x*]
⟨*proof*⟩

lemma *mk_mInterval_non_ovl_id*:
mk_mInterval_non_ovl (*mk_mInterval_non_ovl x*) = *mk_mInterval_non_ovl x*
⟨*proof*⟩

value *valid_mInterval_non_ovl* (*mk_mInterval_non_ovl* ([*Ivl* (1::*int*) 2, *Ivl* 1 3, *Ivl* 1 1]))
value *valid_mInterval_non_ovl* (*mk_mInterval_non_ovl* ([*Ivl* (1::*int*) 1, *Ivl* 1 2, *Ivl* 2 3]))

value‹*mk_mInterval_ovl* [*mk_interval* ((1::*int*), 4), *mk_interval* (0,2), *mk_interval* (3,5), *mk_interval* (5,7),
mk_interval (7,7), *mk_interval* (8,8)]›
value‹*mk_mInterval_adj* [*mk_interval* ((1::*int*), 4), *mk_interval* (0,2), *mk_interval* (3,5), *mk_interval* (5,7),
mk_interval (7,7), *mk_interval* (8,8)]›
value‹*mk_mInterval_non_ovl* [*mk_interval* ((1::*int*), 4), *mk_interval* (0,2), *mk_interval* (3,5), *mk_interval* (5,7),
mk_interval (7,7), *mk_interval* (8,8)]›

8.1.5 Union over a List of Intervals

definition ‹*set_of_interval_list XS* = *foldr* (λ*x a*. *set_of x* ∪ *a*) *XS* {}›

lemma *set_of_interval_list_nonempty*:
assumes *non_empty*: ‹$XS \neq ([]::real\ interval\ list)$›
shows ‹$(set\_of\_interval\_list\ XS) \neq \{\}$›
$\langle proof \rangle$

lemma *set_of_interval_list_bdd_below*:
assumes *non_empty*: ‹$XS \neq ([]::real\ interval\ list)$›
shows ‹$bdd\_below\ (set\_of\_interval\_list\ XS)$›
$\langle proof \rangle$

lemma *set_of_interval_list_bdd_above*:
assumes *non_empty*: ‹$XS \neq ([]::real\ interval\ list)$›
shows ‹$bdd\_above\ (set\_of\_interval\_list\ XS)$›
$\langle proof \rangle$

lemma *inf_set_of_interval_list_lower*:
assumes *non_empty*: ‹$XS \neq ([]::real\ interval\ list)$›
and *sorted*: ‹$sorted\_wrt\_lower\ XS$›
shows ‹$Inf\ (set\_of\_interval\_list\ XS) = lower\ (hd\ XS)$›
$\langle proof \rangle$

lemma *contiguous_sorted_wrt_upper*:
assumes $contiguous\ (xs::\ real\ interval\ list)$
shows $sorted\_wrt\_upper\ xs$
$\langle proof \rangle$

lemma *contiguous_sorted_wrt_lower*:
assumes $contiguous\ (XS::\ real\ interval\ list)$
shows $sorted\_wrt\_lower\ XS$
$\langle proof \rangle$

lemma *max_last_sorted_wrt_upper*:
assumes $XS \neq []\ sorted\_wrt\_upper\ (XS::\ 'a::\{linorder\}\ interval\ list)$
shows $Max\ (set(map\ upper\ XS)) = upper\ (last\ XS)$
$\langle proof \rangle$

lemma *min_hd_sorted_wrt_lower*:
assumes $XS \neq []\ sorted\_wrt\_lower\ (XS::\ 'a::\{linorder,minus,preorder\}\ interval\ list)$
shows $Min\ (set(map\ lower\ XS)) = lower\ (hd\ XS)$
$\langle proof \rangle$

lemma *lower_isort*:
assumes ‹$xs \neq []$›**and** ‹$(lower\ o\ hd)\ xs = Min\ (lower\ `\ (set\ xs))$›
shows ‹$(lower \circ hd)\ (interval\_sort\_lower\_width\ xs) = (lower \circ hd)\ xs$›
$\langle proof \rangle$

lemma *min_sort*:
$Min\ (set\ (map\ lower\ (foldr\ interval\_insert\_sort\_lower\_width\ xs\ []))) = Min\ (set\ (map\ lower\ xs))$
$\langle proof \rangle$

lemma *mk_mInterval_lower*:

assumes *xs* $\neq$ []
shows *Min* (*set* (*map lower* (*mk_mInterval_non_ovl xs*))) = *Min* (*set* (*map lower xs*))
$\langle proof \rangle$

lemma *sup_set_of_interval_list_upper*:
assumes *non_empty*: ‹*XS* $\neq$ ([]::*real interval list*)›
and *sorted*: ‹*sorted_wrt_upper XS*›
shows ‹*Sup* (*set_of_interval_list XS*) = *upper* (*last XS*)›
$\langle proof \rangle$

lemma *compact_set_of_interval_list*:
‹*compact* (*set_of_interval_list* (*XS*::('*a*::{*preorder*,*ordered_euclidean_space*,*topological_space*} *interval list*)))›
$\langle proof \rangle$

lemma *lower_le_upper_aux*: ‹*xs* $\neq$ [] $\Longrightarrow$ *non_overlapping_sorted xs* $\Longrightarrow$ *lower* (*hd xs*) $\leq$ *upper* (*last xs*)›
$\langle proof \rangle$

lemma *contiguous_lower_le_upper*:
assumes *non_empty*: ‹*XS* $\neq$ ([]::*real interval list*)›
and *contiguous*: ‹*contiguous XS*›
shows ‹(*lower* (*hd XS*)) $\leq$ (*upper* (*last XS*))›
$\langle proof \rangle$

lemma *diameter_Sup_Inf*:
assumes ‹*compact X*› ‹*X* $\neq$ {}›
shows ‹*diameter X* $\leq$ *Sup X* − *Inf X*›
$\langle proof \rangle$

lemma *diameter_width_compact*:
assumes ‹*compact X*› ‹*bdd_below X*› ‹*bdd_above X*› ‹*X* $\neq$ {}›
shows ‹*diameter X* = *Sup X* − *Inf X*›
$\langle proof \rangle$

lemma *diameter_contiguous*:
assumes *non_empty*: ‹*XS* $\neq$ ([]::*real interval list*)›
and *contiguous*: ‹*contiguous XS*›
shows ‹*diameter* (*set_of_interval_list XS*) = *dist* (*lower* (*hd XS*)) (*upper* (*last XS*))›
$\langle proof \rangle$

lemma *interval_list_union_contiguous_lower*:
assumes *non_empty*: ‹*XS* $\neq$ []›
and *sorted*: ‹*sorted_wrt_lower XS*›
shows ‹*lower* (*interval_list_union XS*) = *lower* (*hd XS*)›
$\langle proof \rangle$

lemma *interval_list_union_contiguous_upper*:
assumes *non_empty*: ‹*XS* $\neq$ []›
and *sorted*: ‹*sorted_wrt_upper XS*›
shows ‹*upper* (*interval_list_union XS*) = *upper* (*last XS*)›
$\langle proof \rangle$

lemma *interval_list_union_contiguous*:
assumes *non_empty*: ‹$XS \neq []$›
and *sorted_lower*: ‹*sorted_wrt_lower XS*›
and *sorted_upper*: ‹*sorted_wrt_upper XS*›
shows ‹*interval_list_union XS* = *Interval* (*lower* (*hd XS*), *upper* (*last XS*))›
⟨*proof*⟩

lemma *contiguous_bounds_lower*:
assumes *non_empty*: ‹$XS \neq []$›
and *contiguous*: ‹*contiguous* (*XS*::*real interval list*)›
shows *lower* (*hd XS*) = *Min* (*set* (*map lower XS*))
⟨*proof*⟩

lemma *contiguous_bounds_upper*:
assumes *non_empty*: ‹$XS \neq []$›
and *contiguous*: ‹*contiguous* (*XS*::*real interval list*)›
shows *upper* (*last XS*) = *Max* (*set* (*map upper XS*))
⟨*proof*⟩

lemma *set_of_interval_list_contiguous*:
assumes *non_empty*: ‹$XS \neq$ ([]::*real interval list*)›
and *contiguous*: ‹*contiguous XS*›
shows ‹*set_of_interval_list XS* = {*lower* (*hd XS*)..*upper* (*last XS*)}›
⟨*proof*⟩

lemma *set_of_interval_list_set_eq_interval_list_union_contiguous*:
assumes *non_empty*: ‹$XS \neq$ ([]::*real interval list*)›
and *contiguous*: ‹*contiguous XS*›
shows ‹*set_of_interval_list XS* = *set_of* (*interval_list_union XS*)›
⟨*proof*⟩

lemma *mInterval_ovl_lower_hd_min*:
‹*valid_mInterval_ovl x* $\Longrightarrow$ *Min* (*set* (*map lower x*)) = (*lower o hd*) *x*›
⟨*proof*⟩

lemma *mInterval_adj_lower_hd_min*:
‹*valid_mInterval_adj x* $\Longrightarrow$ *Min* (*set* (*map lower x*)) = (*lower o hd*) *x*›
⟨*proof*⟩

lemma *mInterval_non_ovl_lower_hd_min*:
‹*valid_mInterval_non_ovl x* $\Longrightarrow$ *Min* (*set* (*map lower x*)) = (*lower o hd*) *x*›
⟨*proof*⟩

lemma *mInterval_ovl_lower_last_max*:
‹*valid_mInterval_ovl x* $\Longrightarrow$ (*Max* (*set* (*map lower x*))) = (*lower o last*) *x*›
⟨*proof*⟩

lemma *mInterval_adj_upper_hd_min*:
‹ *valid_mInterval_adj x* $\Longrightarrow$ *Min* (*set* (*map upper x*)) = (*upper o hd*) *x*›
⟨*proof*⟩

lemma *mInterval_adj_upper_last_max*:

‹ valid_mInterval_adj x ⟹ Max (set (map upper x)) = (upper o last) x›
⟨proof⟩

lemma *set_of_subeq_aux*:
‹(⋃x∈set is. {lower x..upper x}) ⊆ {Min (lower ‘ (set is)) .. Max (upper ‘ (set is))}›
⟨proof⟩

lemma *lower_merge_adjacent_intervals*:
assumes xs ≠ []
and ‹sorted_wrt_lower xs›
shows (lower ∘ hd) (merge_adjacent_intervals_sorted_wrt_lower xs) = (lower ∘ hd) xs
⟨proof⟩

lemma *sorted_wrt_lower_hd_min*:
‹x ≠ [] ⟹ sorted_wrt_lower x ⟹ Min (set (map lower x)) = (lower o hd) x›
⟨proof⟩

lemma *lower_hd_min_over_mk_mInterval_non_ovl*:
xs ≠ [] ⟹ (lower o hd) xs = Min (lower ‘ (set xs)) ⟹ (lower ∘ hd) (mk_mInterval_non_ovl xs) = (lower ∘ hd) xs
⟨proof⟩

theorem *valid_mInterval_non_ovl_nempty*: valid_mInterval_non_ovl x ⟹ x ≠ []
⟨proof⟩

end

9 Subdivisions and Refinements (Lipschitz_Subdivisions_Refinements)

theory
Lipschitz_Subdivisions_Refinements
imports
Lipschitz_Interval_Extension
Multi_Interval_Preliminaries
begin

9.1 Subdivisions

A uniform subdivision of an interval X splits X into a vector of equal length, contiguous intervals.

definition *uniform_subdivision* :: *'a::linordered_field interval* ⇒ *nat* ⇒ *'a interval list* **where**
uniform_subdivision A n = *map* (λ*i*. *let i'* = *of_nat i in*
mk_interval (*lower A* + (*upper A* − *lower A*) ∗ *i'* / *of_nat n*,
lower A + (*upper A* − *lower A*) ∗ (*i'* + 1) / *of_nat n*)) [0..<*n*]

The definition *uniform_subdivision* refers to definition 6.2 in [4]

definition *overlapping_ordered* :: *'a*::{*preorder*} *interval list* ⇒ *bool* **where**
overlapping_ordered xs = (∀ *i*. *i* < *length xs* − 1 ⟶ *lower* (*xs* ! (*i* + 1)) ≤ *upper* (*xs* ! *i*))

definition *overlapping_non_zero_width* :: *'a*::{*preorder*, *minus*, *zero*, *ord*} *interval list* ⇒ *bool* **where**
overlapping_non_zero_width xs = (∀ *i* < *length xs* − 1 . ∃ *e*. *e* $\in_i$ (*xs* ! (*i* + 1)) ∧ *e* $\in_i$ (*xs* ! *i*) ∧ 0 < *width* (*xs* ! (*i* + 1)) ∧ 0 < *width* (*xs* ! *i*))

definition *overlapping* :: *'a*::{*preorder*} *interval list* ⇒ *bool* **where**
overlapping xs = (∀ *i* < *length xs* − 1 . ∃ *e*. *e* $\in_i$ (*xs* ! (*i* + 1)) ∧ *e* $\in_i$ (*xs* ! *i*))

definition *check_is_uniform_subdivision* :: *'a::linordered_field interval* ⇒ *'a interval list* ⇒ *bool* **where**
check_is_uniform_subdivision A xs = (*let n* = *length xs in*
if n = 0 *then True*
else
let d = *width A* / *of_nat n in*
list_all (λ *x*. *width x* = *d*) *xs* ∧
contiguous xs ∧
lower (*hd xs*) = *lower A* ∧
upper (*last xs*) = *upper A*)

lemma *non_empty_subdivision*:
assumes 0 < *n*
shows *uniform_subdivision A n* ≠ []
⟨*proof*⟩

lemma *uniform_subdivision_id*: *uniform_subdivision X* 1 = [*X*]
⟨*proof*⟩

lemma *subdivision_length_n*:
assumes $0 < n$
shows $length(uniform\_subdivision\ A\ n) = n$
⟨*proof*⟩

lemma *contiguous_uniform_subdivision*: $contiguous\ (uniform\_subdivision\ A\ n)$
⟨*proof*⟩

lemma *overlapping_ordered_uniform_subdivision*:
assumes $0 < n$
shows $overlapping\_ordered\ (uniform\_subdivision\ A\ n)$
⟨*proof*⟩

lemma *overlapping_uniform_subdivision*:
assumes $0 < N$
shows $overlapping\ (uniform\_subdivision\ X\ N)$
⟨*proof*⟩

lemma *hd_lower_uniform_subdivision*:
assumes $0 < n$
shows $lower\ (\ hd\ (uniform\_subdivision\ A\ n)) = lower\ A$
⟨*proof*⟩

lemma *last_upper_uniform_subdivision*:
assumes $0 < n$
shows $upper\ (\ last\ (uniform\_subdivision\ A\ n)) = upper\ A$
⟨*proof*⟩

lemma *uniform_subdivisions_width_single*:
fixes $A :: 'a::linordered\_field\ interval$
shows ‹$width\ (Interval\ (lower\ A + (upper\ A - lower\ A) * \ x\ /\ of\_nat\ n,$
$lower\ A + (upper\ A - lower\ A) * (x + 1)\ /\ \ of\_nat\ n)) = \ width\ A\ /\ of\_nat\ n$›
⟨*proof*⟩

lemma *uniform_subdivisions_width*:
assumes $0 < n$
shows ‹$\forall\ A.\ A \in set\ (uniform\_subdivision\ X\ n) \longrightarrow width\ A = width\ X\ /\ of\_nat\ n$›
⟨*proof*⟩

lemma *uniform_subdivision_sum_width*:
assumes ‹$0 < n$›
shows ‹$sum\_list\ (map\ width\ (uniform\_subdivision\ X\ n)) = width\ X$›
⟨*proof*⟩

lemma *uniform_subdivisions_distinct*:
assumes $0 < n\ 0 < width\ A$
shows $distinct\ (uniform\_subdivision\ A\ n)$
⟨*proof*⟩

lemma *uniform_subdivisions_non_overlapping*:
assumes $0 < n$
shows $non\_overlapping\_sorted\ (uniform\_subdivision\ A\ n)$

$\langle proof \rangle$

We prove that our uniform subdivision meets the multi-interval type

lemma *uniform_subdivisions_valid_ainterval*:
assumes $0 < n$ $0 < width\ A$
shows $valid\_mInterval\_adj(uniform\_subdivision\ A\ n)$
$\langle proof \rangle$

lemma *uniform_subdivisions_valid*:
assumes $0 < n$
shows $check\_is\_uniform\_subdivision\ A\ (uniform\_subdivision\ A\ n)$
$\langle proof \rangle$

9.2 Refinement

Let $F\ X$ be an inclusion isotonic, Lipschitz, interval extension for $X \subseteq Y$. A refinement $F_N\ X$ of $F\ X$ is the union of interval values of X over the elements of a uniform subdivision of X

definition *refinement* :: $('a::\{linordered\_field,lattice\}\ interval \Rightarrow\ 'a\ interval) \Rightarrow nat \Rightarrow\ 'a\ interval \Rightarrow\ 'a\ interval$ **where**
‹$refinement\ F\ N\ X = (interval\_list\_union\ (map\ F\ (uniform\_subdivision\ X\ N)))$›

definition *check_is_refinement* **where**
‹$check\_is\_refinement\ F\ n\ As\ B = (let\ I = refinement\ F\ n\ As\ in\ lower\ B \leq lower\ I \wedge upper\ I \leq upper\ B)$›

definition *refinement_set* :: $('a::\{linordered\_field,lattice\}\ interval \Rightarrow\ 'a\ interval) \Rightarrow nat \Rightarrow\ 'a\ interval \Rightarrow\ 'a\ set$ **where**
‹$refinement\_set\ F\ N\ X = (set\_of\_interval\_list\ (map\ F\ (uniform\_subdivision\ X\ N)))$›

The definition *refinement* refers to definition 6.3 in [4].

The excess width of $F\ X$ is $w\ (E\ X) = w\ (F\ X) - w\ (f\ X)$. The united extension $f\ x$ for $x \in X$ has zero excess width and we can compute $f\ x$ as closely as desired by computing refinements of an extension $F\ X$.

definition $width\_set\ s = Sup\ s - Inf\ s$

lemma *width_set_bounded*:
fixes X :: ‹$real\ set$›
assumes ‹$bdd\_below\ X$› ‹$bdd\_above\ X$›
shows ‹$\forall\ x \in X.\ \forall\ x' \in X.\ dist\ x\ x' \leq width\_set\ X$›
$\langle proof \rangle$

lemma *width_inclusion_isotonic_approx*:
fixes F::‹$real\ interval \Rightarrow real\ interval$›
assumes $inclusion\_isotonic\ F$ $F\ is\_interval\_extension\_of\ f$
shows ‹$0 \leq width\ (F\ X) - width\_set\ (f\ `\ set\_of\ X)$›
$\langle proof \rangle$

lemma *diameter_width*:
assumes ‹$a \leq b$›
shows ‹$diameter\ \{a..b\} = width\_set\ \{a..b\}$›
$\langle proof \rangle$

lemma *lipschitz_dist_diameter_limit*:
fixes S::‹$'a::\{metric\_space, heine\_borel\}\ set$›
and f::‹$'a::\{metric\_space, heine\_borel\} \Rightarrow\ 'b::\{metric\_space, heine\_borel\}$›
assumes ‹$C{-}lipschitz\_on\ S\ f$› **and** ‹$bounded\ S$›

shows ‹$x \in (f\,`S) \Longrightarrow y \in (f\,`S) \Longrightarrow$ *dist x y* $\leq$ *diameter* $(f`\,S)$›
$\langle proof \rangle$

definition *excess_width_diameter* :: $('a$::*preorder interval* $\Rightarrow$ *real interval*$) \Rightarrow ('a \Rightarrow 'b$::*metric_space*$) \Rightarrow 'a$ *interval* $\Rightarrow$ *real* **where**
‹*excess_width_diameter F f X* = *width*$(F\,X)$ − *diameter* $(f\,`$ *set_of X*$)$›

definition *excess_width_set* :: $('a$::$\{$*minus*,*linorder*,*Inf*,*Sup*$\}$ *interval* $\Rightarrow 'a$ *set*$) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a$ *interval* $\Rightarrow 'a$ **where**
‹*excess_width_set F f X* = *width_set*$(F\,X)$ − *width_set* $(f\,`$ *set_of X*$)$›

definition *excess_width* :: $('a$::$\{$*minus*,*linorder*,*Inf*,*Sup*$\}$ *interval* $\Rightarrow 'a$ *interval*$) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a$ *interval* $\Rightarrow 'a$ **where**
‹*excess_width F f X* = *width*$(F\,X)$ − *width_set* $(f\,`$ *set_of X*$)$›

The definition *excess_width* refers to definition 6.4 in [4]

lemma *width_set_of*: **fixes** *X* :: *real interval*
shows *width_set_upper_lower*: ‹*width_set* $($*set_of X*$) = |($*lower X*$) - ($*upper X*$)|$›
$\langle proof \rangle$

lemma *width_set_dist*:
fixes *f* :: *real* $\Rightarrow$ *real*
shows *width_set* $($ *set_of X*$) = ($*dist* $($*lower X*$)$ $($*upper X*$))$
$\langle proof \rangle$

lemma *diameter_of*: **fixes** *X* :: *real interval*
shows *diameter_upper_lower*: ‹*diameter* $($*set_of X*$) = |($*lower X*$) - ($*upper X*$)|$›
$\langle proof \rangle$

lemma *diameter_dist*:
fixes *X* :: *real interval*
shows *diameter* $($ *set_of X*$) = ($*dist* $($*lower X*$)$ $($*upper X*$))$
$\langle proof \rangle$

lemma *bdd_below_f_set_of*:
fixes *f* :: *real* $\Rightarrow$ *real*
assumes *C*−*lipschitz_on X f*
and ‹*bounded X*› **and** ‹$X \neq \{\}$›
shows ‹*bdd_below* $(f\,`\,X)$›
$\langle proof \rangle$

lemma *bdd_above_f_set_of*:
fixes *f* :: *real* $\Rightarrow$ *real*
assumes *C*−*lipschitz_on* (X) *f*
and ‹*bounded X*› **and** ‹$X \neq \{\}$›
shows ‹*bdd_above* $(f\,`\,X)$›
$\langle proof \rangle$

lemma *diameter_image_dist*:
fixes *f*::‹*real* $\Rightarrow$ *real*›
assumes ‹*continuous_on* $($*set_of X*$)$ *f*›
shows ‹$(\exists\ x\in$ *set_of X*. $\exists\ x'\in$ *set_of X* . *diameter* $(f\,`$ *set_of X*$) =$ *dist* $(f\,x)$ $(f\,x'))$›
$\langle proof \rangle$

lemma *excess_width_inf_diameter*:
fixes *F*::‹*real interval* $\Rightarrow$ *real interval*›

assumes *inclusion_isotonic F F is_interval_extension_of f* ‹ $C-$*lipschitz_on* (*set_of X*) *f*›
shows ‹*dist* (*Inf* (*f* ‘ *set_of X*)) (*lower* (*F X*)) $\leq$ *excess_width_diameter F f X*›
⟨*proof*⟩

lemma *excess_width_inf*:
fixes *F*::‹*real interval* ⇒ *real interval*›
assumes *inclusion_isotonic F F is_interval_extension_of f* ‹ $C-$*lipschitz_on* (*set_of X*) *f*›
shows ‹*dist* (*Inf* (*f* ‘ *set_of X*)) (*lower* (*F X*)) $\leq$ *excess_width F f X*›
⟨*proof*⟩

lemma *excess_width_sup_diameter*:
fixes *F*::‹*real interval* ⇒ *real interval*›
assumes *inclusion_isotonic F F is_interval_extension_of f* ‹ $C-$*lipschitz_on* (*set_of X*) *f*›
shows ‹*dist* (*Sup* (*f* ‘ *set_of X*)) (*upper* (*F X*)) $\leq$ *excess_width F f X*›
⟨*proof*⟩

lemma *excess_width_sup*:
fixes *F*::‹*real interval* ⇒ *real interval*›
assumes *inclusion_isotonic F F is_interval_extension_of f* ‹ $C-$*lipschitz_on* (*set_of X*) *f*›
shows ‹*dist* (*Sup* (*f* ‘ *set_of X*)) (*upper* (*F X*)) $\leq$ *excess_width F f X*›
⟨*proof*⟩

If *F X* is an inclusion isotonic, Lipschitz, interval extension then the excess width of a refinement is of order (1::'*a*) / *N*

If *X* and *X* are intervals such that $X \subseteq Y$, then there is an interval *E* with *lower E* $\leq$ (0::'*a*) $\wedge$ (0::'*a*) $\leq$ *upper E* such that $Y = X + E$ and $w\ Y = w\ X + w\ E$.

lemma *interval_subset_width*:
fixes *X Y* :: '*a*::{*linordered_ring*, *lattice*} *interval*
assumes $X \leq Y$
and *X_def*: $X = Interval(a, b)$ **and** *x_valid*: $a \leq b$
and *Y_def*: $Y = Interval(c, d)$ **and** *y_valid*: $c \leq d$
shows $\exists$ *E*. $0 \in_i E \wedge Y = X + E \wedge$ *width Y* = *width X* + *width E*
⟨*proof*⟩

lemma *excess_width_incl*:
fixes *F* :: *real interval* ⇒ *real interval* **and** *X* :: *real interval*
assumes *int*: ‹*F is_interval_extension_of f*›
and *iso*: *inclusion_isotonic F*
and $L-$*lipschitz_on* (*set_of X*) *f*
shows ‹ $\exists$ *E* . *F X* = *Interval*(*Inf* (*f* ‘ *set_of X*), *Sup* (*f* ‘ *set_of X*)) + *E* ›
⟨*proof*⟩

lemma *excess_interval_superset_interval*:
fixes *F* :: *real interval* ⇒ *real interval* **and** *X* :: *real interval*
assumes *int*: ‹*F is_interval_extension_of f*›
and *iso*: *inclusion_isotonic F*
and $L-$*lipschitz_on* (*set_of X*) *f*
and *ex*: ‹ $\exists$ *E* . *F X* = *Interval*(*Inf* (*f* ‘ *set_of X*), *Sup* (*f* ‘ *set_of X*)) + *E* ›
shows ‹*Interval*(*Inf* (*f* ‘ *set_of X*), *Sup* (*f* ‘ *set_of X*)) $\leq$ *F X*›
⟨*proof*⟩

lemma *each_subdivision_width_order*:

fixes X :: $'a$::{*linordered_field*,*lattice*,*metric_space*} *interval*
assumes *inclusion_isotonic F lipschitzI_on C U F F is_interval_extension_of f*
and *set* (*uniform_subdivision X N*) $\subseteq$ *U* $0 < N$ *Xs* $\in$ *set* (*uniform_subdivision X N*)
shows *width*(*F Xs*) $\leq$ *C* $*$ *width* (*X*) / *of_nat N*
⟨*proof*⟩

lemma *each_subdivision_excess_width_order*:
fixes *X* :: *real interval*
assumes *inclusion_isotonic F lipschitzI_on C U F F is_interval_extension_of f*
and *set* (*uniform_subdivision X N*) $\subseteq$ *U* $0 < N$
and *L*−*lipschitz_on* (*set_of* (*interval_list_union* (*uniform_subdivision X N*))) *f*
shows ∀ *Xs* ∈ *set* (*uniform_subdivision X N*) . *excess_width F f Xs* $\leq$ *C* $*$ *width* (*X*) / *of_nat N*
⟨*proof*⟩

The theorem ⟦*inclusion_isotonic ?F*; *?C*−*lipschitzI_on ?U ?F*; *?F is_interval_extension_of ?f*; *set* (*uniform_subdivision ?X ?N*) ⊆ *?U*; $0 <$ *?N*; *?Xs* ∈ *set* (*uniform_subdivision ?X ?N*)⟧ ⟹ *width* (*?F ?Xs*) ≤ *?C* $*$ *width ?X* / *of_nat ?N* refers to Theorem 6.1 in [4].

lemma *sup_interval_max*:
fixes *X Y* :: $'a$::{*linordered_ring*, *lattice*} *interval*
shows *sup X Y* = *Interval*(*min* (*lower X*) (*lower Y*), *max* (*upper X*) (*upper Y*))
⟨*proof*⟩

lemma *interval_inf_sup_lower*: *inf* (*lower I1*) (*lower I2*) = *lower* (*sup I1 I2*)
⟨*proof*⟩

lemma *interval_sup_sup_upper*: *sup* (*upper I1*) (*upper I2*) = *upper* (*sup I1 I2*)
⟨*proof*⟩

lemma *interval_union_lower*:
assumes *contiguous Xs Xs* ≠ []
shows *lower* (*interval_list_union Xs*) = *lower* (*Xs*!0)
⟨*proof*⟩

lemma *interval_union_upper*:
assumes *contiguous Xs Xs* ≠ []
shows *upper* (*interval_list_union Xs*) = *upper* (*last Xs*)
⟨*proof*⟩

lemma *union_set*:
assumes $0 < n$
shows *interval_list_union* (*uniform_subdivision X n*) = *X*
⟨*proof*⟩

lemma *sum_list_less*:
assumes *list_all* (λ*n*. *n* ≤ (*y*::*real*)) *xs*
shows *sum_list xs* ≤ *y* $*$ *length xs*
⟨*proof*⟩

lemma *in_bounds2*:
fixes *X Y* :: $'a$::{*linordered_ring*} *interval*
shows $x \in_i X \wedge x \in_i Y \Longrightarrow$
(*lower Y* ≤ *lower X* ∧ *upper Y* ≤ *upper X* ∧ *lower X* ≤ *upper Y* ∧ *lower Y* ≤ *upper X*) ∨
(*lower X* ≤ *lower Y* ∧ *upper X* ≤ *upper Y* ∧ *lower X* ≤ *upper Y* ∧ *lower Y* ≤ *upper X*) ∨
(*lower Y* ≤ *lower X* ∧ *upper X* ≤ *upper Y* ∧ *lower Y* ≤ *lower X* ∧ *lower Y* ≤ *upper X*) ∨

$(lower\ X \leq lower\ Y \wedge upper\ Y \leq upper\ X \wedge lower\ X \leq upper\ Y \wedge lower\ Y \leq upper\ X)$
⟨proof⟩

lemma *overlapping_width_sum*:
fixes $X\ Y :: 'a::\{linordered\_ring, lattice\}\ interval$
assumes $overlapping\ [X,Y]$
shows $width\ (sup\ X\ Y) \leq width\ X + width\ Y$
⟨proof⟩

lemma *interval_list_union_width*:
fixes $xs :: 'a::\{linordered\_ring, lattice\}\ interval\ list$
assumes $overlapping\ xs\ xs \neq []$
shows $overlapping\ xs \Longrightarrow width\ (interval\_list\_union\ xs) \leq sum\_list\ (map\ width\ xs)$
⟨proof⟩

lemma *map_non_zero_width*:
fixes $U :: 'a::\{linordered\_idom\}\ interval\ set$
assumes $C{-}lipschitzI\_on\ U\ F\ inclusion\_isotonic\ F\ set\ xs \subseteq U$
shows $\forall x \in set\ xs.\ 0 \leq width\ x \longrightarrow 0 \leq width\ (F\ x)$
⟨proof⟩

lemma *inclusion_isotonic_preserves_overlapping*:
assumes $inclusion\_isotonic\ F\ \ xs \neq []\ \ F\ is\_interval\_extension\_of\ f$
shows $contiguous\ xs \Longrightarrow overlapping\ (map\ F\ xs)$
⟨proof⟩

lemma *bounded_image_of*:
fixes f::‹$real \Rightarrow real$›
assumes ‹$L{-}lipschitz\_on\ (set\_of\ X)\ f$›
shows ‹$bounded\ (f\ `\ set\_of\ X)$›
⟨proof⟩

lemma *dist_le_diameter*:
fixes f::‹$real \Rightarrow real$ ›
assumes ‹ $C{-}lipschitz\_on\ (set\_of\ X)\ f$›
shows ‹$dist\ (f\ (upper\ X))\ (f\ (lower\ X)) \leq diameter\ (f\ `\ set\_of\ X)$›
⟨proof⟩

lemma *excess_width_inf_sup*:
fixes $X :: real\ interval$ **and** f::‹$real \Rightarrow real$›
assumes ‹$continuous\_on\ (set\_of\ X)\ f$›
shows $Inf\ (f\ `\ set\_of\ X) - lower\ (F\ X) + upper\ (F\ X) - Sup\ (f\ `\ set\_of\ X) \leq excess\_width\ F\ f\ X$
⟨proof⟩

lemma *excess_width_lower_bound*:
fixes $X :: real\ interval$
assumes $inclusion\_isotonic\ F\ \ F\ is\_interval\_extension\_of\ f$ ‹$continuous\_on\ (set\_of\ X)\ f$›
shows $Inf\ (f\ `\ set\_of\ X) - lower\ (F\ X) \leq excess\_width\ F\ f\ X$
⟨proof⟩

lemma *excess_width_upper_bound*:
fixes $X :: real\ interval$
assumes $inclusion\_isotonic\ F\ \ F\ is\_interval\_extension\_of\ f$ ‹$continuous\_on\ (set\_of\ X)\ f$›

shows *upper* $(F\,X) - Sup\ (f\ `\ set\_of\ X) \leq excess\_width\ F\ f\ X$
⟨*proof*⟩

lemma *lipschitz_excess_width_lower_bound*:
fixes *X* :: *real interval*
assumes *inclusion_isotonic F lipschitzI_on C U F F is_interval_extension_of f*
and *set* $(uniform\_subdivision\ X\ N) \subseteq U\ \ N = 1$
and *L*$-$*lipschitz_on* $(set\_of\ (interval\_list\_union\ (uniform\_subdivision\ X\ N)))\ f$
shows *Inf* $(f\ `\ set\_of\ X) - lower\ (F\ X) \leq C * width\ X$
⟨*proof*⟩

lemma *lipschitz_excess_width_upper_bound*:
fixes *X* :: *real interval*
assumes *inclusion_isotonic F lipschitzI_on C U F F is_interval_extension_of f*
and *set* $(uniform\_subdivision\ X\ N) \subseteq U\ \ N = 1$
and *L*$-$*lipschitz_on* $(set\_of\ (interval\_list\_union\ (uniform\_subdivision\ X\ N)))\ f$
shows *upper* $(F\,X) - Sup\ (f\ `\ set\_of\ X) \leq C * width\ X$
⟨*proof*⟩

lemma *excess_width_bound_inf*:
fixes *X* :: *real interval*
assumes *excess_width_bound*: ‹$excess\_width\ F\ f\ X \leq k$›
and *inclusion_isotonic*: ‹*inclusion_isotonic F*›
and *interval_extension*: ‹*F is_interval_extension_of f*›
shows ‹$Inf\ (f\ `\ set\_of\ X) - k \leq lower\ (F\ X)$›
⟨*proof*⟩

lemma *excess_width_bound_sup*:
fixes *X* :: *real interval*
assumes *excess_width_bound*: ‹$excess\_width\ F\ f\ X \leq k$›
and *inclusion_isotonic*: ‹*inclusion_isotonic F*›
and *interval_extension*: ‹*F is_interval_extension_of f*›
shows ‹$upper\ (F\ X) \leq Sup\ (f\ `\ set\_of\ X) + k$›
⟨*proof*⟩

lemma *set_of_interval_list_subset_inf_sup*:
assumes *non_empty*: ‹$XS \neq ([]::real\ interval\ list)$›
shows ‹$set\_of\_interval\_list\ XS \subseteq \{Min(set\ (map\ lower\ XS))..Max(set\ (map\ upper\ XS))\}$›
⟨*proof*⟩

lemma *lower_bound_F_inf*:
assumes *non_empty*: ‹$XS \neq ([]::real\ interval\ list)$›
and *inclusion_isotonic*: ‹*inclusion_isotonic F*›
and *interval_extension*: ‹*F is_interval_extension_of f*›
and *sorted_lower*: ‹*sorted_wrt_lower XS*›
and *lipschitz*: ‹$0 \leq C$› ‹C−*lipschitz_on* $((set\_of\_interval\_list\ XS))\ f$›
and *excess_width_bounded*: ‹$(Max\ (set\ ((map\ (excess\_width\ F\ f))\ XS))) \leq k$›
shows ‹$(Inf\ (f\ `\ (set\_of\_interval\_list\ XS))) - k \leq Inf\ (set\_of\_interval\_list\ (map\ F\ XS))$›
⟨*proof*⟩

lemma *upper_bound_F_sup*:
assumes *non_empty*: ‹$XS \neq ([]::real\ interval\ list)$›
and *inclusion_isotonic*: ‹*inclusion_isotonic F*›
and *interval_extension*: ‹*F is_interval_extension_of f*›

```
 and    sorted_upper: ‹sorted_wrt_upper XS›
 and    lipschitz: ‹0 ≤ C› ‹C−lipschitz_on ((set_of_interval_list XS)) f›
 and    excess_width_bounded: ‹(Max (set ((map (excess_width F f)) XS))) ≤ k›
shows ‹Sup (set_of_interval_list (map F XS)) ≤ (Sup (f ‘ (set_of_interval_list XS))) + k ›
 ⟨proof⟩

lemma Inf_interval_list_approx: assumes non_empty: ‹XS ≠ ([]::real interval list)›
 and    inclusion_isotonic: ‹inclusion_isotonic F›
 and    interval_extension: ‹F is_interval_extension_of f›
 and    sorted_upper: ‹sorted_wrt_upper XS›
 and    lipschitz: ‹0 ≤ C› ‹C−lipschitz_on ((set_of_interval_list XS)) f›
 and    excess_width_bounded: ‹(Max (set ((map (excess_width F f)) XS))) ≤ k›
shows  Inf (set_of_interval_list (map F XS)) ≤ Inf (f ‘ set_of_interval_list XS)
 ⟨proof⟩

lemma Sup_interval_list_approx: assumes non_empty: ‹XS ≠ ([]::real interval list)›
 and    inclusion_isotonic: ‹inclusion_isotonic F›
 and    interval_extension: ‹F is_interval_extension_of f›
 and    sorted_lower: ‹sorted_wrt_lower XS›
 and    lipschitz: ‹0 ≤ C› ‹C−lipschitz_on ((set_of_interval_list XS)) f›
 and    excess_width_bounded: ‹(Max (set ((map (excess_width F f)) XS))) ≤ k›
shows Sup (f ‘ set_of_interval_list XS) ≤ Sup (set_of_interval_list (map F XS))
 ⟨proof⟩


lemma map_inclusion_isotonic_excess_width_bounded:
 assumes non_empty: ‹XS ≠ ([]::real interval list)›
 and    inclusion_isotonic: ‹inclusion_isotonic F›
 and    interval_extension: ‹F is_interval_extension_of f›
 and    sorted_lower: ‹sorted_wrt_lower XS›
 and    sorted_upper: ‹sorted_wrt_upper XS›
 and    lipschitz: ‹C−lipschitz_on ((set_of_interval_list XS)) f›
 and    excess_width_bounded: ‹(Max (set ((map (excess_width F f)) XS))) ≤ k›
shows ‹width_set (set_of_interval_list (map F XS)) − width_set (f ‘ (set_of_interval_list XS)) ≤ 2 ∗ k›
 and  ‹width_set (set_of_interval_list (map F XS)) − width_set (f ‘ (set_of_interval_list XS)) ≥ 0›
 ⟨proof⟩

lemma max_subdivision_excess_width_order:
 fixes X :: real interval
 assumes inclusion_isotonic F lipschitzI_on C U F F is_interval_extension_of f
 and set (uniform_subdivision X N) ⊆ U  0 < N
 and  L−lipschitz_on (set_of_interval_list (uniform_subdivision X N)) f
shows ‹Max (set (map (excess_width F f) (uniform_subdivision X N))) ≤ C ∗ width X / real N›
⟨proof⟩

lemma set_of_interval_list_set_eq_interval_list_union_contiguous:
 assumes non_empty: ‹XS ≠ ([]::real interval list)›
 and    contiguous: ‹contiguous XS›
shows ‹set_of_interval_list XS = set_of (interval_list_union XS)›
 ⟨proof⟩


lemma width_eq_wdith_set:
 fixes X :: ‹('a::{conditionally_complete_lattice, minus, preorder}) interval›
```

shows ‹width X = width_set (set_of X)›
⟨proof⟩

lemma *width_zero_lower_upper*:
fixes *X* :: *real interval*
assumes ‹width X = 0›
shows ‹lower X = upper X›
⟨proof⟩

lemma *width_zero_mk_interval*:
fixes *X* :: *real interval*
assumes ‹width X = 0›
shows ‹∃ x. X = mk_interval(x,x)›
⟨proof⟩

lemma *width_zero_interval_of*:
fixes *X* :: *real interval*
assumes ‹width X = 0›
shows ‹∃ x. X = interval_of x›
⟨proof⟩

lemma *width_zero_interval_extension*:
fixes *F* :: *real interval ⇒ real interval*
assumes ‹F is_interval_extension_of f›
and ‹width X = 0›
shows ‹width (F X) = 0›
⟨proof⟩

9.3 Lipschitz Interval Inclusive

If F is a natural interval extension of a real valued rational function with $F X$ defined for $X \subseteq Y$ where X and Y are intervals or n-dimentional interval vectors then F is Lipschitz in Y

lemma *interval_extension_bounded*:
fixes *F* :: *real interval ⇒ real interval*
assumes ‹F is_interval_extension_of f›
and ‹(width (F X)) / (width X) ≤ L›
shows width (F X) ≤ L ∗ width X
⟨proof⟩

lemma *lipschitz_on_implies_lipschitzI_on*:
fixes *F* :: *real interval ⇒ real interval*
assumes ‹F is_interval_extension_of f›
and ‹C−lipschitz_on X f›
and ‹⋃ (set_of ‘ Y) ⊆ X›
and ‹0 ≤ L›
and ‹∀ y ∈ Y. (width (F y)) / (width y) ≤ L›
shows L−lipschitzI_on Y F
⟨proof⟩

lemma *lipschitz_on_implies_lipschitzI_on2*:
fixes *f* :: ‹real ⇒ real›
assumes ‹S ≠ []› and ‹0 ≤ C›
and ‹F is_interval_extension_of f›

and ‹0 ≤ L›
and ‹∀ y ∈ (set S). (width (F y)) / (width y) ≤ L›
and ‹C−lipschitz_on (set_of (interval_list_union (S))) f ›
shows ‹L−lipschitzI_on (set (S)) F›
⟨proof⟩

lemma *width_img_Max*:
assumes ‹finite S›
shows ‹∀ x∈S. width (F x) ≤ Max (width ‘ F ‘ S)›
⟨proof⟩
lemma *width_Min*:
assumes ‹finite S›
shows ‹∀ x∈S. Min (width ‘ S) ≤ width x›
⟨proof⟩

lemma *lipschitzI_on_le_interval*:
fixes F :: ‹real interval ⇒ real interval›
assumes inc_isontonic_F: ‹inclusion_isotonic F›
and lipschitzI_F: ‹C−lipschitzI_on {X} F›
and interval_inc: ‹x ≤ X›
shows ‹width (F x) ≤ C ∗ width X›
⟨proof⟩

lemma *lipschitzI_on_le_lipschitzI_on*:
fixes F :: ‹real interval ⇒ real interval›
assumes inc_isontonic_F: ‹inclusion_isotonic F›
and lipschitzI_F: ‹C−lipschitzI_on {X} F›
and interval_inc: ‹x ≤ X›
and interval_ext: ‹F is_interval_extension_of f ›
shows ‹∃ L. L−lipschitzI_on {x} F›
⟨proof⟩

lemma *uniform_subdivision_le*:
fixes X :: ‹real interval›
assumes ‹N>0›
shows ‹∀ x ∈ set (uniform_subdivision X N). x ≤ X›
⟨proof⟩

lemma *lipschitzI_on_uniform_subdivision*:
fixes F :: ‹real interval ⇒ real interval›
assumes inc_isontonic_F: ‹inclusion_isotonic F›
and lipschitzI_F: ‹C−lipschitzI_on ({X}) F›
and ‹N>0›
shows ‹∀ x∈(set (uniform_subdivision X N)). width (F x) ≤ C ∗ width X›
⟨proof⟩

lemma *division_leq_pos*:
fixes x :: 'a::{linordered_field}
assumes x > 0 and y > 0 and z > 0 and y ≤ z
shows x / z ≤ x / y
⟨proof⟩

lemma *each_subdivision_width_order'*:

fixes X :: *real interval*
assumes F *is_interval_extension_of* f
and $0 < N$
and $Xs \in$ *set* (*uniform_subdivision* X N)
shows $\exists$ L. *width*(F Xs) $\leq L *$ *width* (X) / *of_nat* N
⟨proof⟩

lemma *uniform_subdivision_min_nonzero*:
assumes ‹$N > 0$›
and ‹*width* $X > 0$›
shows ‹$0 <$ *Min* (*width* ‘ *set* (*uniform_subdivision* X N))›
⟨proof⟩

lemma *uniform_subdivision_width_zero_replicate_eq*:
fixes X::‹*real interval*›
assumes *positive_N*: ‹$0 < N$›
and *zero_width_X*: ‹$0 =$ *width* X ›
shows‹*replicate* N X = (*uniform_subdivision* X N)›
⟨proof⟩

lemma *set_of_interval_list_zero_width*:
fixes X::‹*real interval*›
assumes *positive_N*: ‹$0 < N$›
and *zero_width_X*: ‹$0 =$ *width* X ›
shows‹*set_of_interval_list* (*uniform_subdivision* X N) = {*lower* X..*upper* X}›
⟨proof⟩

lemma *width_zero_subdivision*: *width* X = (0::*real*) ==> $N > 0 \Longrightarrow$ *set* (*uniform_subdivision* X N) = {X}
⟨proof⟩

lemma *lipschitz_on_implies_lipschitzI_on_pre*:
fixes f :: ‹*real* $\Rightarrow$ *real*›
and F :: ‹*real interval* $\Rightarrow$ *real interval*›
assumes ‹*finite* S›
and ‹$0 <$ *Min* (*width* ‘ S)›
shows ‹*let max_F_width* = *Max* (*width* ‘ (F ‘ S));
min_f_width = *Min* (*width* ‘ S)
in $\forall$ $x \in S$. *width* (F x) $\leq$ (*max_F_width*/*min_f_width*) $*$ *width* x›
⟨proof⟩

lemma *lipschitz_on_implies_lipschitzI_on*′:
fixes f :: ‹*real* $\Rightarrow$ *real*›
and F :: ‹*real interval* $\Rightarrow$ *real interval*›
assumes *non_empty*: ‹$S \neq \{\}$›
and *finite*: ‹*finite* S›
and *non_zero_width*: ‹$0 <$ *Min* (*width* ‘ S)›
and *interval_ext_F*: ‹F *is_interval_extension_of* f›
shows ‹$\exists$ L. L—*lipschitzI_on* S F›
⟨proof⟩

lemma *natural_extension_transfer_lipschitz*:
assumes *positive_N*: ‹$0 < N$›
and *inc_isontonic_F*: ‹*inclusion_isotonic* F›

and *interval_ext_F*: ‹*F is_natural_interval_extension_of f*›
and *lipschitz_f*: ‹*C*−*lipschitz_on* (*set_of X*) *f*›
shows ‹*C*−*lipschitzI_on* (*set* (*uniform_subdivision X N*)) *F*›
⟨*proof*⟩

lemma *lipschitz_on_division_lipschitz_on*:
assumes *lipschitz_f*: *C*−*lipschitz_on* (*set_of X*) *f*
and *non_empty*: *uniform_subdivision X N* ≠ []
and *subdivision*: *Xs* ∈ *set*(*uniform_subdivision* (*X*::*real interval*) *N*)
shows ∃ *L* . *L*−*lipschitz_on* (*set_of Xs*) *f*
⟨*proof*⟩

lemma *lipschitz_on_lischitz_on_subdivisions*:
assumes *lipschitz_f*: *C*−*lipschitz_on* (*set_of X*) *f*
and *non_empty*: *uniform_subdivision X N* ≠ []
and *non_zero*: $0 < N$
shows ∃ *L* . ∀ *Xs* ∈ *set*(*uniform_subdivision* (*X*::*real interval*) *N*). *L*−*lipschitz_on* (*set_of Xs*) *f*
⟨*proof*⟩

lemma *lipschitz_on_lischitz_on_subdivisions_n*:
assumes *lipschitz_f*: *C*−*lipschitz_on* (*set_of X*) *f*
and *non_empty*: *uniform_subdivision X N* ≠ []
and *non_zero*: $0 < N$
shows ∃ *L* . ∀ *N* > 0 . ∀ *Xs* ∈ *set*(*uniform_subdivision* (*X*::*real interval*) *N*). *L*−*lipschitz_on* (*set_of Xs*) *f*
⟨*proof*⟩

lemma *lipschitzI_on_division_lipschitzI_on*:
assumes *lipschitz_f*: *C*−*lipschitzI_on* (*set*(*uniform_subdivision X N*)) *F*
and *non_empty*: *uniform_subdivision X N* ≠ []
and *subdivision*: *Xs* ∈ *set*(*uniform_subdivision* (*X*::*real interval*) *N*)
shows ∃ *L* . *L*−*lipschitzI_on* {*Xs*} *F*
⟨*proof*⟩

lemma *lipschitzI_on_lipschitzI_on_subdivisions*:
fixes *X* :: *real interval*
assumes *lipschitz_f*: *C*−*lipschitzI_on* (*set*(*uniform_subdivision X N*)) *F*
and *non_zero*: $0 < N$
shows ∃ *L* . ∀ *Xs* ∈ *set*(*uniform_subdivision X N*). *L*−*lipschitzI_on* {*Xs*} *F*
⟨*proof*⟩

9.4 Lipschitz Convergence

lemma *isotonic_lipschitz_refinement'*:
assumes *positive_N*: ‹$0 < N$›
and *inc_isontonic_F*: ‹*inclusion_isotonic F*›
and *interval_ext_F*: ‹*F is_interval_extension_of f*›
and *lipschitz_f*: ‹*C*−*lipschitz_on* (*set_of X*) *f*›
shows ‹∃ *L*. *width_set* (*set_of_interval_list* (*map F* (*uniform_subdivision X N*))) − *width_set* (*f* ‘ (*set_of X*)) ≤ 2 ∗ (*L* ∗ *width X* / *real N*)›
⟨*proof*⟩

lemma *isotonic_lipschitz_refinementI*:

```
 assumes positive_N: ‹0 < N›
 and inc_isontonic_F: ‹inclusion_isotonic F›
 and interval_ext_F: ‹F is_interval_extension_of f›
 and lipschitz_f:   ‹L-lipschitz_on (set_of X) f›
 and lipschitz_F:   ‹C-lipschitzI_on (set (uniform_subdivision X N)) F›
shows ‹width_set (set_of_interval_list (map F (uniform_subdivision X N))) - width_set (f ` (set_of X)) ≤ 2 * (C * width
X / real N)›
⟨proof⟩

lemma isotonic_lipschitz_refinement:
 assumes positive_N: ‹0 < N›
 and inc_isontonic_F: ‹inclusion_isotonic F›
 and interval_ext_F: ‹F is_interval_extension_of f›
 and lipschitz_f:   ‹L-lipschitz_on (set_of X) f›
 and lipschitz_F:   ‹C-lipschitzI_on (set (uniform_subdivision X N)) F›
shows ‹excess_width_set (refinement_set F N) f X ≤ 2 * (C * width X / real N)›
 ⟨proof⟩

end
```

10 Interval Analysis (Interval_Analysis)

theory
Interval_Analysis
imports
Interval_Division_Real
Lipschitz_Subdivisions_Refinements
begin

This theory provides interval analysis over standard types such as real or integer. All operations work over (closed) intervals.

end

11 Extended Division on Intervals (Extended_Interval_Division)

theory
Extended_Interval_Division
imports
Interval_Division_Non_Zero
begin

In this theory, we define an extended division operation on intervals. This definition is inspired by the definition given in [4], but we use an over-approximation for the case in which zero is an element of the divisor interval. By this, we avoid the need for multi-intervals.

```
instantiation interval :: ({infinity, linordered_field, real_normed_algebra,linear_continuum_topology}) inverse
begin
 definition inverse_interval :: 'a interval ⇒ 'a interval
  where inverse_interval a = (
                if (¬ 0 ∈i a) then mk_interval ( 1 / (upper a), 1 / (lower a))
                else if lower a = 0 ∧ 0 < upper a then mk_interval (1/ upper a, ∞)
                else if lower a < 0 ∧ 0 < upper a then mk_interval (−∞, ∞)
                else if lower a < upper a ∧ upper a = 0 then mk_interval(−∞, 1 / lower a)
                else undefined
               )

 definition divide_interval :: 'a interval ⇒ 'a interval ⇒ 'a interval
  where divide_interval a b = inverse b ∗ a
 instance ⟨proof⟩
end

interpretation interval_division_inverse divide inverse
 ⟨proof⟩

end
```

12 Extended Interval Analysis (Extended_Interval_Analysis)

theory
Extended_Interval_Analysis
imports
Extended_Interval_Division
Lipschitz_Subdivisions_Refinements
begin

This theory provides extended interval analysis over the type extended reals. All operations work over (closed) intervals.

end

12.1 Overlapping Multi-Intervals (Multi_Interval_Overlapping)

theory
Multi_Interval_Overlapping
imports
Multi_Interval_Preliminaries
begin

12.1.1 Type Definition

typedef (**overloaded**) *'a minterval_ovl* =
{*is*::*'a*::{*minus_mono*} *interval list*. *valid_mInterval_ovl is*}
morphisms *bounds_of_minterval_ovl mInterval_ovl*
⟨*proof*⟩

setup_lifting *type_definition_minterval_ovl*

lift_definition *mlower_ovl*::(*'a*::{*minus_mono*}) *minterval_ovl* ⇒ *'a* **is** ‹*lower o hd*› ⟨*proof*⟩
lift_definition *mupper_ovl*::(*'a*::{*minus_mono*}) *minterval_ovl* ⇒ *'a* **is** ‹*upper o last*› ⟨*proof*⟩
lift_definition *mlist_ovl*::(*'a*::{*minus_mono*}) *minterval_ovl* ⇒ *'a interval list* **is** ‹*id*› ⟨*proof*⟩

12.1.2 Equality and Orderings

lemma *minterval_ovl_eq_iff*: $a = b \longleftrightarrow$ *mlist_ovl a* = *mlist_ovl b*
⟨*proof*⟩

lemma *ainterval_eqI*: *mlist_ovl a* = *mlist_ovl b* $\Longrightarrow a = b$
⟨*proof*⟩

lemma *minterval_ovl_imp_upper_lower_eq* :
$a = b \longrightarrow$ *mlower_ovl a* = *mlower_ovl b* ∧ *mupper_ovl a* = *mupper_ovl b*
⟨*proof*⟩

lemma *valid_mInterval_ovl_lower_le_upper*:
valid_mInterval_ovl i $\Longrightarrow$ (*lower* $\circ$ *hd*) *i* $\leq$ (*upper* $\circ$ *last*) *i*
$\langle proof \rangle$

lemma *mlower_non_ovl_le_mupper_non_ovl*[*simp*]: *mlower_ovl i* $\leq$ *mupper_ovl i*
$\langle proof \rangle$

lift_definition *set_of_ovl* :: *'a*::{*minus_mono*} *minterval_ovl* $\Rightarrow$ *'a set*
is λ *is*. $\bigcup$*x*$\in$*set is*. {*lower x*..*upper x*} $\langle proof \rangle$

lemma *not_in_ovl_eq*:
‹($\neg$ *e* $\in$ *set_of_ovl xs*) = ($\forall$ *x* $\in$ *set* (*mlist_ovl xs*). $\neg$ *e* $\in$ *set_of x*)›
$\langle proof \rangle$

lemma *in_ovl_eq*:
‹(*e* $\in$ *set_of_ovl xs*) = ($\exists$ *x* $\in$ *set* (*mlist_ovl xs*). *e* $\in$ *set_of x*)›
$\langle proof \rangle$

context notes [[*typedef_overloaded*]] **begin**

lift_definition(*code_dt*) *mInterval_ovl'*::*'a*::*minus_mono interval list* $\Rightarrow$ *'a minterval_ovl option*
is λ *is*. *if valid_mInterval_ovl is then Some is else None*
$\langle proof \rangle$

lemma *mInterval_ovl'_split*:
P (*mInterval_ovl' is*) $\longleftrightarrow$
($\forall$ *ivl*. *valid_mInterval_ovl is* $\longrightarrow$ *mlist_ovl ivl* = *is* $\longrightarrow$ *P* (*Some ivl*)) $\wedge$ ($\neg$ *valid_mInterval_ovl is* $\longrightarrow$ *P None*)
$\langle proof \rangle$

lemma *mInterval_ovl'_split_asm*:
P (*mInterval_ovl' is*) $\longleftrightarrow$
$\neg$(($\exists$ *ivl*. *valid_mInterval_ovl is* $\wedge$ *mlist_ovl ivl* = *is* $\wedge$ $\neg$*P* (*Some ivl*)) $\vee$ ($\neg$ *valid_mInterval_ovl is* $\wedge$ $\neg$*P None*))
$\langle proof \rangle$

lemmas *mInterval_ovl'_splits* = *mInterval_ovl'_split mInterval_ovl'_split_asm*

lemma *mInterval'_eq_Some*: *mInterval_ovl' is* = *Some i* $\Longrightarrow$ *mlist_ovl i* = *is*
$\langle proof \rangle$

end

lemma *set_of_ovl_non_zero_list_all*:
‹*o* $\notin$ *set_of_ovl xs* $\Longrightarrow$ $\forall$ *x* $\in$ *set* (*mlist_ovl xs*). $\neg$ *o* $\in_i$ *x*›
$\langle proof \rangle$

instantiation *minterval_ovl* :: ({*minus_mono*}) *equal*
begin

definition *equal_class.equal a b* $\equiv$ (*mlist_ovl a* = *mlist_ovl b*)

instance $\langle proof \rangle$
end

instantiation *minterval_ovl* :: ({*minus_mono*}) *ord* **begin**

definition *less_eq_minterval_ovl* :: $'a$ *minterval_ovl* $\Rightarrow$ $'a$ *minterval_ovl* $\Rightarrow$ *bool*
where *less_eq_minterval_ovl a b* $\longleftrightarrow$ *mlower_ovl b* $\leq$ *mlower_ovl a* $\wedge$ *mupper_ovl a* $\leq$ *mupper_ovl b*

definition *less_minterval_ovl* :: $'a$ *minterval_ovl* $\Rightarrow$ $'a$ *minterval_ovl* $\Rightarrow$ *bool*
where *less_minterval_ovl x y* $= (x \leq y \wedge \neg\, y \leq x)$

instance $\langle proof \rangle$
end

instantiation *minterval_ovl* :: $(\{$*minus_mono*,*lattice*$\})$ *sup*
begin

lift_definition *sup_minterval_non_ovl* :: $'a$ *minterval_ovl* $\Rightarrow$ $'a$ *minterval_ovl* $\Rightarrow$ $'a$ *minterval_ovl*
is λ *a b*. [*Interval* (*inf* (*lower* (*hd a*)) (*lower* (*hd b*)), *sup* (*upper* (*last a*)) (*upper* (*last b*)))]
$\langle proof \rangle$
instance
$\langle proof \rangle$
end

instantiation *minterval_ovl* :: $(\{$*lattice*,*minus_mono*$\})$ *preorder*
begin
instance
$\langle proof \rangle$
end

lift_definition *minterval_ovl_of* :: $'a$::$\{$*minus_mono*$\}$ $\Rightarrow$ $'a$ *minterval_ovl* **is** λx. [*Interval*(x, x)]
$\langle proof \rangle$

lemma *mlower_ovl_minterval_ovl_of*[*simp*]: *mlower_ovl* (*minterval_ovl_of a*) $=$ *a*
$\langle proof \rangle$

lemma *mupper_ovl_minterval_ovl_of*[*simp*]: *mupper_ovl* (*minterval_ovl_of a*) $=$ *a*
$\langle proof \rangle$

definition *width_ovl* :: $'a$::$\{$*minus_mono*$\}$ *minterval_ovl* $\Rightarrow$ $'a$
where *width_ovl i* $=$ *mupper_ovl i* $-$ *mlower_ovl i*

12.1.3 Zero and One

instantiation *minterval_ovl* :: $(\{$*minus_mono*,*zero*$\})$ *zero*
begin

lift_definition *zero_minterval_ovl*::$'a$ *minterval_ovl* **is** *mk_mInterval_ovl* [*Interval* (0, 0)]
$\langle proof \rangle$

lemma *mlower_ovl_zero*[*simp*]: *mlower_ovl* 0 $=$ 0
$\langle proof \rangle$

lemma *mupper_ovl_zero*[*simp*]: *mupper_ovl* 0 $=$ 0
$\langle proof \rangle$

instance $\langle proof \rangle$
end

instantiation *minterval_ovl* :: $(\{$*minus_mono*,*one*$\})$ *one*
begin

lift_definition *one_minterval_ovl*::*'a minterval_ovl* **is** *mk_mInterval_ovl* [*Interval* (1, 1)]
$\langle proof \rangle$

lemma *mlower_ovl_one*[*simp*]: *mlower_ovl* 1 = 1
$\langle proof \rangle$

lemma *mupper_ovl_one*[*simp*]: *mupper_ovl* 1 = 1
$\langle proof \rangle$

instance $\langle proof \rangle$
end

12.1.4 Addition

instantiation *minterval_ovl* :: $(\{$*minus_mono*,*ordered_ab_semigroup_add*,*linordered_field*$\})$ *plus*
begin

lift_definition *plus_minterval_ovl*::*'a minterval_ovl* ⇒ *'a minterval_ovl* ⇒ *'a minterval_ovl*
is λ *a b* . *mk_mInterval_ovl* (*iList_plus a b*)
$\langle proof \rangle$

lemma *valid_mk_interval_iList_plus*:
assumes *valid_mInterval_ovl a* **and** *valid_mInterval_ovl b*
shows *valid_mInterval_ovl* (*mk_mInterval_ovl* (*iList_plus a b*))
$\langle proof \rangle$

lift_definition *plus_minterval_non_ovl*::*'a minterval_ovl* ⇒ *'a minterval_ovl* ⇒ *'a minterval_ovl*
is λ *a b* . *mk_mInterval_ovl* (*iList_plus a b*)
$\langle proof \rangle$

lemma *interval_plus_com*:
‹*a* + *b* = *b* + *a*› **for** *a*::*'a*::{*minus_mono*,*ordered_ab_semigroup_add*,*linordered_field*} *minterval_ovl*
$\langle proof \rangle$

instance $\langle proof \rangle$
end

12.1.5 Unary Minus

lemma *a*: (*x*::*'a*::*ordered_ab_group_add interval*) ≠ *y* ⟹ −*x* ≠ −*y*
$\langle proof \rangle$

lemma *b*: *distinct* (*is*::*'a*::*ordered_ab_group_add interval list*) ⟹ *distinct* (*map* (λ *i*. −*i*) *is*)
$\langle proof \rangle$

instantiation *minterval_ovl* :: $(\{$*minus_mono*, *ordered_ab_group_add*$\})$ *uminus*
begin

lift_definition *uminus_minterval_ovl*::*'a minterval_ovl* ⇒ *'a minterval_ovl*

is λ *is* . *mk_mInterval_ovl* (*rev* (*map* (λ *i*. $-$*i*) *is*))
⟨*proof*⟩
instance ⟨*proof*⟩
end

12.1.6 Subtraction

instantiation *minterval_ovl* :: ({*minus_mono*, *linordered_field*, *ordered_ab_group_add*}) *minus*
begin

definition *minus_minterval_ovl*::′*a minterval_ovl* ⇒ ′*a minterval_ovl* ⇒ ′*a minterval_ovl*
where *minus_minterval_ovl a b* = *a* + − *b*

instance ⟨*proof*⟩
end

12.1.7 Multiplication

instantiation *minterval_ovl* :: ({*minus_mono*,*linordered_semiring*}) *times*
begin

lift_definition *times_minterval_ovl* :: ′*a minterval_ovl* ⇒ ′*a minterval_ovl* ⇒ ′*a minterval_ovl*
is λ *a b* . *mk_mInterval_ovl* (*iList_times a b*)
⟨*proof*⟩

instance ⟨*proof*⟩
end

12.1.8 Multiplicative Inverse and Division

locale *minterval_ovl_division* = *inverse* +
constrains *inverse* :: ‹′*a*::{*linordered_field*, *zero*, *minus*, *minus_mono*, *real_normed_algebra*,*linear_continuum_topology*} *minterval_ovl* ⇒ ′*a minterval_ovl*›
and *divide* :: ‹′*a*::{*linordered_field*, *zero*, *minus*, *minus_mono*, *real_normed_algebra*,*linear_continuum_topology*} *minterval_ovl* ⇒ ′*a minterval_ovl* ⇒ ′*a minterval_ovl*›
assumes *inverse_left*: ‹¬ 0 ∈ *set_of_ovl x* ⟹ 1 ≤ (*inverse x*) ∗ *x*›
and *divide*: ‹¬ 0 ∈ *set_of_ovl y* ⟹ *x* ≤ (*divide x y*) ∗ *y*›
begin
end

locale *minterval_ovl_division_inverse* = *inverse* +
constrains *inverse* :: ‹′*a*::{*linordered_field*, *zero*, *minus*, *minus_mono*, *real_normed_algebra*,*linear_continuum_topology*} *minterval_ovl* ⇒ ′*a minterval_ovl*›
and *divide* :: ‹′*a*::{*linordered_field*, *zero*, *minus*, *minus_mono*, *real_normed_algebra*,*linear_continuum_topology*} *minterval_ovl* ⇒ ′*a minterval_ovl* ⇒ ′*a minterval_ovl*›
assumes *inverse_non_zero_def*: ‹¬ 0 ∈ *set_of_ovl x* ⟹ (*inverse x*)
= *mInterval_ovl* (*mk_mInterval_ovl*(*un_op_interval_list* (λ *i*. *mk_interval* (1 / (*upper i*), 1 / (*lower i*))) (*mlist_ovl x*)))›
and *divide_non_zero_def*: ‹¬ 0 ∈ *set_of_ovl y* ⟹ (*divide x y*) = *inverse y* ∗ *x*›
begin

end

12.1.9 Membership

abbreviation (**in** *preorder*) *in_minterval_ovl* ((_/ $\in_{no}$ _) [51, 51] 50)
where *in_minterval_ovl x X* ≡ *x* ∈ *set_of_ovl X*

lemma *in_minterval_ovl_to_minterval_ovl*[*intro*!]: *a* $\in_{no}$ *minterval_ovl_of a*
⟨*proof*⟩

instance *minterval_ovl* :: ({*one*, *preorder*, *minus_mono*, *linordered_semiring*}) *power*
⟨*proof*⟩

lemma *set_of_one_ovl*[*simp*]: *set_of_ovl* (1::'*a*::{*one*, *order*, *minus_mono*} *minterval_ovl*) = {1}
⟨*proof*⟩

lifting_update *minterval_ovl.lifting*
lifting_forget *minterval_ovl.lifting*

end

12.2 Non-Overlapping Multi-Intervals (Multi_Interval_Non_Overlapping)

theory
Multi_Interval_Non_Overlapping
imports
Multi_Interval_Preliminaries
begin

12.2.1 Type Definition

typedef (**overloaded**) '*a minterval_non_ovl* =
{*is*::'*a*::{*minus_mono*} *interval list*. *valid_mInterval_non_ovl is*}
morphisms *bounds_of_minterval_non_ovl mInterval_non_ovl*
⟨*proof*⟩

setup_lifting *type_definition_minterval_non_ovl*

lift_definition *mlower_non_ovl*::('*a*::{*minus_mono*}) *minterval_non_ovl* ⇒ '*a* **is** ‹*lower o hd*› ⟨*proof*⟩
lift_definition *mupper_non_ovl*::('*a*::{*minus_mono*}) *minterval_non_ovl* ⇒ '*a* **is** ‹*upper o last*› ⟨*proof*⟩
lift_definition *mlist_non_ovl*::('*a*::{*minus_mono*}) *minterval_non_ovl* ⇒ '*a interval list* **is** ‹*id*› ⟨*proof*⟩

12.2.2 Equality and Orderings

lemma *minterval_non_ovl_eq_iff*: *a* = *b* ⟷ *mlist_non_ovl a* = *mlist_non_ovl b*
⟨*proof*⟩

lemma *ainterval_eqI*: *mlist_non_ovl a* = *mlist_non_ovl b* ⟹ *a* = *b*
⟨*proof*⟩

lemma *minterval_non_ovl_imp_upper_lower_eq* :
a = *b* ⟶ *mlower_non_ovl a* = *mlower_non_ovl b* ∧ *mupper_non_ovl a* = *mupper_non_ovl b*
⟨*proof*⟩

lemma *mlower_non_ovl_le_mupper_non_ovl*[*simp*]: *mlower_non_ovl i* $\leq$ *mupper_non_ovl i*
⟨*proof*⟩

lift_definition *set_of_non_ovl* :: *'a*::{*minus_mono*} *minterval_non_ovl* ⇒ *'a set*
is λ *is*. ⋃*x*∈*set is*. {*lower x*..*upper x*} ⟨*proof*⟩

lemma *set_non_ovl_of_subset*: *set_of_non_ovl* (*x*::*'a*::*minus_mono minterval_non_ovl*) ⊆ {*mlower_non_ovl x* .. *mupper_non_ovl x*}
⟨*proof*⟩

lemma *not_in_non_ovl_eq*:
‹(¬ *e* ∈ *set_of_non_ovl xs*) = (∀ *x* ∈ *set* (*mlist_non_ovl xs*). ¬ *e* ∈ *set_of x*)›
⟨*proof*⟩

lemma *in_non_ovl_eq*:
‹(*e* ∈ *set_of_non_ovl xs*) = (∃ *x* ∈ *set* (*mlist_non_ovl xs*). *e* ∈ *set_of x*)›
⟨*proof*⟩

lemma *set_of_non_ovl_non_zero_list_all*:
‹*o* ∉ *set_of_non_ovl xs* ⟹ ∀ *x* ∈ *set* (*mlist_non_ovl xs*). ¬ *o* $\in_i$ *x*›
⟨*proof*⟩

context notes [[*typedef_overloaded*]] **begin**

lift_definition(*code_dt*) *mInterval_non_ovl'*::*'a*::*minus_mono interval list* ⇒ *'a minterval_non_ovl option*
is λ *is*. *if valid_mInterval_non_ovl is then Some is else None*
⟨*proof*⟩

lemma *mInterval_non_ovl'_split*:
P (*mInterval_non_ovl' is*) ⟷
(∀ *ivl*. *valid_mInterval_non_ovl is* ⟶ *mlist_non_ovl ivl* = *is* ⟶ *P* (*Some ivl*)) ∧ (¬ *valid_mInterval_non_ovl is* ⟶ *P None*)
⟨*proof*⟩

lemma *mInterval_non_ovl'_split_asm*:
P (*mInterval_non_ovl' is*) ⟷
¬((∃ *ivl*. *valid_mInterval_non_ovl is* ∧ *mlist_non_ovl ivl* = *is* ∧ ¬*P* (*Some ivl*)) ∨ (¬ *valid_mInterval_non_ovl is* ∧ ¬*P None*))
⟨*proof*⟩

lemmas *mInterval_non_ovl'_splits* = *mInterval_non_ovl'_split mInterval_non_ovl'_split_asm*

lemma *mInterval'_eq_Some*: *mInterval_non_ovl' is* = *Some i* ⟹ *mlist_non_ovl i* = *is*
⟨*proof*⟩

end

instantiation *minterval_non_ovl* :: ({*minus_mono*}) *equal*
begin

definition *equal_class.equal a b* ≡ (*mlist_non_ovl a* = *mlist_non_ovl b*)

instance $\langle proof \rangle$
end

instantiation *minterval_non_ovl* :: ({*minus_mono*}) *ord* **begin**

definition *less_eq_minterval_non_ovl* :: *'a minterval_non_ovl* ⇒ *'a minterval_non_ovl* ⇒ *bool*
where *less_eq_minterval_non_ovl a b* ⟷ *mlower_non_ovl b* ≤ *mlower_non_ovl a* ∧ *mupper_non_ovl a* ≤ *mupper_non_ovl b*

definition *less_minterval_non_ovl* :: *'a minterval_non_ovl* ⇒ *'a minterval_non_ovl* ⇒ *bool*
where *less_minterval_non_ovl x y* = (*x* ≤ *y* ∧ ¬ *y* ≤ *x*)

instance $\langle proof \rangle$
end

instantiation *minterval_non_ovl* :: ({*minus_mono*,*lattice*}) *sup*
begin

lift_definition *sup_minterval_non_ovl* :: *'a minterval_non_ovl* ⇒ *'a minterval_non_ovl* ⇒ *'a minterval_non_ovl*
is λ *a b*. [*Interval* (*inf* (*lower* (*hd a*)) (*lower* (*hd b*)), *sup* (*upper* (*last a*)) (*upper* (*last b*)))]
$\langle proof \rangle$

lemma *mlower_non_ovl_sup*[*simp*]: *mlower_non_ovl* (*sup A B*) = *inf* (*mlower_non_ovl A*) (*mlower_non_ovl B*)
$\langle proof \rangle$

lemma *mupper_non_ovl_sup*[*simp*]: *mupper_non_ovl* (*sup A B*) = *sup* (*mupper_non_ovl A*) (*mupper_non_ovl B*)
$\langle proof \rangle$
instance
$\langle proof \rangle$
end

instantiation *minterval_non_ovl* :: ({*lattice*,*minus_mono*}) *preorder*
begin
instance
$\langle proof \rangle$
end

lemma *set_of_minterval_non_ovl_union*: *set_of_non_ovl A* ∪ *set_of_non_ovl B* ⊆ *set_of_non_ovl* (*sup A B*)
for *A*::*'a*::{*lattice*, *minus_mono*} *minterval_non_ovl*
$\langle proof \rangle$

lemma *minterval_non_ovl_union_commute*: *sup A B* = *sup B A* **for** *A*::*'a*::{*minus_mono*,*lattice*} *minterval_non_ovl*
$\langle proof \rangle$

lemma *minterval_non_ovl_union_mono1*: *set_of_non_ovl a* ⊆ *set_of_non_ovl* (*sup a A*)
for *A* :: *'a*::{*minus_mono*,*lattice*} *minterval_non_ovl*
$\langle proof \rangle$

lemma *minterval_non_ovl_union_mono2*: *set_of_non_ovl A* ⊆ *set_of_non_ovl* (*sup a A*) **for** *A* :: *'a*::{*lattice*, *minus_mono*} *minterval_non_ovl*
$\langle proof \rangle$

lift_definition *minterval_non_ovl_of* :: *'a*::{*minus_mono*} ⇒ *'a minterval_non_ovl* **is** λ*x*. [*Interval*(*x*, *x*)]

⟨proof⟩

lemma *mlower_non_ovl_minterval_non_ovl_of*[*simp*]: *mlower_non_ovl* (*minterval_non_ovl_of a*) = *a*
⟨proof⟩

lemma *mupper_non_ovl_minterval_non_ovl_of*[*simp*]: *mupper_non_ovl* (*minterval_non_ovl_of a*) = *a*
⟨proof⟩

definition *width_non_ovl* :: ′*a*::{*minus_mono*} *minterval_non_ovl* ⇒ ′*a*
where *width_non_ovl i* = *mupper_non_ovl i* − *mlower_non_ovl i*

12.2.3 Zero and One

instantiation *minterval_non_ovl* :: ({*minus_mono*,*zero*}) *zero*
begin

lift_definition *zero_minterval_non_ovl*::′*a minterval_non_ovl* **is** *mk_mInterval_non_ovl* [*Interval* (0, 0)]
⟨proof⟩

lemma *mlower_non_ovl_zero*[*simp*]: *mlower_non_ovl* 0 = 0
⟨proof⟩

lemma *mupper_non_ovl_zero*[*simp*]: *mupper_non_ovl* 0 = 0
⟨proof⟩

instance ⟨proof⟩
end

instantiation *minterval_non_ovl* :: ({*minus_mono*,*one*}) *one*
begin

lift_definition *one_minterval_non_ovl*::′*a minterval_non_ovl* **is** *mk_mInterval_non_ovl* [*Interval* (1, 1)]
⟨proof⟩

lemma *mlower_non_ovl_one*[*simp*]: *mlower_non_ovl* 1 = 1
⟨proof⟩

lemma *mupper_non_ovl_one*[*simp*]: *mupper_non_ovl* 1 = 1
⟨proof⟩

instance ⟨proof⟩
end

12.2.4 Addition

instantiation *minterval_non_ovl* :: ({*minus_mono*,*ordered_ab_semigroup_add*,*linordered_field*}) *plus*
begin

lemma *valid_mk_interval_iList_plus*:
assumes *valid_mInterval_non_ovl a* **and** *valid_mInterval_non_ovl b*
shows *valid_mInterval_non_ovl* (*mk_mInterval_non_ovl* (*iList_plus a b*))
⟨proof⟩

lift_definition *plus_minterval_non_ovl*::′*a minterval_non_ovl* ⇒ ′*a minterval_non_ovl* ⇒ ′*a minterval_non_ovl*

is λ *a b . mk_mInterval_non_ovl* (*iList_plus a b*)
$\langle proof \rangle$

lemma *interval_plus_com*:
‹$a + b = b + a$› **for** a::$'a$::{*minus_mono*,*ordered_ab_semigroup_add*,*linordered_field*} *minterval_non_ovl*
$\langle proof \rangle$

instance $\langle proof \rangle$

end

12.2.5 Unary Minus

lemma *a*: (x::$'a$::*ordered_ab_group_add interval*) $\neq y \Longrightarrow -x \neq -y$
$\langle proof \rangle$

lemma *b*: *distinct* (*is*::$'a$::*ordered_ab_group_add interval list*) $\Longrightarrow$ *distinct* (*map* (λ *i*. $-i$) *is*)
$\langle proof \rangle$

instantiation *minterval_non_ovl* :: ({*minus_mono*, *ordered_ab_group_add*}) *uminus*
begin

lift_definition *uminus_minterval_non_ovl*::$'a$ *minterval_non_ovl* $\Rightarrow$ $'a$ *minterval_non_ovl*
is λ *is . mk_mInterval_non_ovl* (*rev* (*map* (λ *i*. $-i$) *is*))
$\langle proof \rangle$

instance $\langle proof \rangle$
end

12.2.6 Subtraction

instantiation *minterval_non_ovl* :: ({*minus_mono*, *linordered_field*, *ordered_ab_group_add*}) *minus*
begin

definition *minus_minterval_non_ovl*::$'a$ *minterval_non_ovl* $\Rightarrow$ $'a$ *minterval_non_ovl* $\Rightarrow$ $'a$ *minterval_non_ovl*
where *minus_minterval_non_ovl a b* $= a + - b$

instance $\langle proof \rangle$
end

12.2.7 Multiplication

instantiation *minterval_non_ovl* :: ({*minus_mono*, *linordered_field*}) *times*
begin

lift_definition *times_minterval_non_ovl*::$'a$ *minterval_non_ovl* $\Rightarrow$ $'a$ *minterval_non_ovl* $\Rightarrow$ $'a$ *minterval_non_ovl*
is λ *a b . mk_mInterval_non_ovl* (*iList_times a b*)
$\langle proof \rangle$

instance $\langle proof \rangle$
end

12.2.8 Multiplicative Inverse and Division

locale *minterval_non_ovl_division* = *inverse* +
constrains *inverse* :: ‹′*a*::{*linordered_field*, *zero*, *minus*, *minus_mono*, *real_normed_algebra*,*linear_continuum_topology*} *minterval_non_ovl* ⇒ ′*a minterval_non_ovl*›
and *divide* :: ‹′*a*::{*linordered_field*, *zero*, *minus*, *minus_mono*, *real_normed_algebra*,*linear_continuum_topology*} *minterval_non_ovl* ⇒ ′*a minterval_non_ovl* ⇒ ′*a minterval_non_ovl*›
assumes *inverse_left*: ‹¬ 0 ∈ *set_of_non_ovl x* ⟹ 1 ≤ (*inverse x*) ∗ *x*›
and *divide*: ‹¬ 0 ∈ *set_of_non_ovl y* ⟹ *x* ≤ (*divide x y*) ∗ *y*›
begin
end

locale *minterval_non_ovl_division_inverse* = *inverse* +
constrains *inverse* :: ‹′*a*::{*linordered_field*, *zero*, *minus*, *minus_mono*, *real_normed_algebra*,*linear_continuum_topology*} *minterval_non_ovl* ⇒ ′*a minterval_non_ovl*›
and *divide* :: ‹′*a*::{*linordered_field*, *zero*, *minus*, *minus_mono*, *real_normed_algebra*,*linear_continuum_topology*} *minterval_non_ovl* ⇒ ′*a minterval_non_ovl* ⇒ ′*a minterval_non_ovl*›
assumes *inverse_non_zero_def*: ‹¬ 0 ∈ *set_of_non_ovl x* ⟹ (*inverse x*) = *mInterval_non_ovl* (*mk_mInterval_non_ovl*(*un_op_interval_list* (λ *i*. *mk_interval* (1 / (*upper i*), 1 / (*lower i*))) (*mlist_non_ovl x*)))›
and *divide_non_zero_def*: ‹¬ 0 ∈ *set_of_non_ovl y* ⟹ (*divide x y*) = *inverse y* ∗ *x*›
begin

end

12.2.9 Membership

abbreviation (**in** *preorder*) *in_minterval_non_ovl* ((_/ $\in_{no}$ _) [51, 51] 50)
where *in_minterval_non_ovl x X* ≡ *x* ∈ *set_of_non_ovl X*

lemma *in_minterval_non_ovl_to_minterval_non_ovl*[*intro*!]: *a* $\in_{no}$ *minterval_non_ovl_of a*
⟨*proof*⟩

instance *minterval_non_ovl* :: ({*one*, *preorder*, *minus_mono*, *linordered_semiring*}) *power*
⟨*proof*⟩

lemma *set_of_one_non_ovl*[*simp*]: *set_of_non_ovl* (1::′*a*::{*one*, *minus_mono*, *order*} *minterval_non_ovl*) = {1}
⟨*proof*⟩

lifting_update *minterval_non_ovl.lifting*
lifting_forget *minterval_non_ovl.lifting*

end

12.3 Adjacent Multi-Intervals (Multi_Interval_Adjacent)

theory
Multi_Interval_Adjacent
imports
Multi_Interval_Preliminaries
begin

12.3.1 A Type For Non Overlapping Multi Intervals

typedef (**overloaded**) *'a minterval_adj* =
{*is*::'*a*::{*minus_mono*,*linorder*} *interval list*. *valid_mInterval_adj is*}
morphisms *bounds_of_minterval_adj mInterval_adj*
⟨*proof*⟩

setup_lifting *type_definition_minterval_adj*

lift_definition *mlower_adj*::('*a*::{*minus_mono*}) *minterval_adj* ⇒ '*a* **is** ‹*lower o hd*› ⟨*proof*⟩
lift_definition *mupper_adj*::('*a*::{*minus_mono*}) *minterval_adj* ⇒ '*a* **is** ‹*upper o last*› ⟨*proof*⟩
lift_definition *mlist_adj*::('*a*::{*minus_mono*}) *minterval_adj* ⇒ '*a interval list* **is** ‹*id*› ⟨*proof*⟩

12.3.2 Equality and Orderings

lemma *minterval_adj_eq_iff*: *a* = *b* ⟷ *mlist_adj a* = *mlist_adj b*
⟨*proof*⟩

lemma *ainterval_eqI*: *mlist_adj a* = *mlist_adj b* ⟹ *a* = *b*
⟨*proof*⟩

lemma *minterval_adj_imp_upper_lower_eq* :
a = *b* ⟶ *mlower_adj a* = *mlower_adj b* ∧ *mupper_adj a* = *mupper_adj b*
⟨*proof*⟩

lemma *mlower_adj_le_mupper_adj*[*simp*]: *mlower_adj i* ≤ *mupper_adj i*
⟨*proof*⟩

lift_definition *set_of_adj* :: '*a*::{*minus_mono*} *minterval_adj* ⇒ '*a set*
is λ *is*. ⋃*x*∈*set is*. {*lower x*..*upper x*} ⟨*proof*⟩

lemma *set_adj_of_subset*: *set_of_adj* (*x*::'*a*::*minus_mono minterval_adj*) ⊆ {*mlower_adj x* .. *mupper_adj x*}
⟨*proof*⟩

lemma *not_in_adj_eq*:
‹(¬ *e* ∈ *set_of_adj xs*) = (∀ *x* ∈ *set* (*mlist_adj xs*). ¬ *e* ∈ *set_of x*)›
⟨*proof*⟩

lemma *in_adj_eq*:
‹(*e* ∈ *set_of_adj xs*) = (∃ *x* ∈ *set* (*mlist_adj xs*). *e* ∈ *set_of x*)›
⟨*proof*⟩

lemma *set_of_adj_non_zero_list_all*:
‹*o* ∉ *set_of_adj xs* ⟹ ∀ *x* ∈ *set* (*mlist_adj xs*). ¬ *o* $\in_i$ *x*›
⟨*proof*⟩

context notes [[*typedef_overloaded*]] **begin**

lift_definition(*code_dt*) *mInterval_adj*'::'*a*::*minus_mono interval list* ⇒ '*a minterval_adj option*
is λ *is*. *if valid_mInterval_adj is then Some is else None*
⟨*proof*⟩

lemma *mInterval_adj′_split*:
P (*mInterval_adj′ is*) $\longleftrightarrow$
($\forall$ *ivl. valid_mInterval_adj is* $\longrightarrow$ *mlist_adj ivl* = *is* $\longrightarrow$ *P* (*Some ivl*)) $\wedge$ ($\neg$ *valid_mInterval_adj is* $\longrightarrow$ *P None*)
$\langle proof \rangle$

lemma *mInterval_adj′_split_asm*:
P (*mInterval_adj′ is*) $\longleftrightarrow$
$\neg$(($\exists$ *ivl. valid_mInterval_adj is* $\wedge$ *mlist_adj ivl* = *is* $\wedge$ $\neg$*P* (*Some ivl*)) $\vee$ ($\neg$ *valid_mInterval_adj is* $\wedge$ $\neg$*P None*))
$\langle proof \rangle$

lemmas *mInterval_adj′_splits* = *mInterval_adj′_split mInterval_adj′_split_asm*

lemma *mInterval′_eq_Some*: *mInterval_adj′ is* = *Some i* $\Longrightarrow$ *mlist_adj i* = *is*
$\langle proof \rangle$

end

instantiation *minterval_adj* :: ({*minus_mono*}) *equal*
begin

definition *equal_class.equal a b* $\equiv$ (*mlist_adj a* = *mlist_adj b*)

instance $\langle proof \rangle$
end

instantiation *minterval_adj* :: ({*minus_mono*}) *ord* **begin**

definition *less_eq_minterval_adj* :: *′a minterval_adj* $\Rightarrow$ *′a minterval_adj* $\Rightarrow$ *bool*
where *less_eq_minterval_adj a b* $\longleftrightarrow$ *mlower_adj b* $\leq$ *mlower_adj a* $\wedge$ *mupper_adj a* $\leq$ *mupper_adj b*

definition *less_minterval_adj* :: *′a minterval_adj* $\Rightarrow$ *′a minterval_adj* $\Rightarrow$ *bool*
where *less_minterval_adj x y* = ($x \leq y \wedge \neg\, y \leq x$)

instance $\langle proof \rangle$
end

instantiation *minterval_adj* :: ({*minus_mono,lattice*}) *sup*
begin

lift_definition *sup_minterval_adj* :: *′a minterval_adj* $\Rightarrow$ *′a minterval_adj* $\Rightarrow$ *′a minterval_adj*
is λ *a b*. [*Interval* (*inf* (*lower* (*hd a*)) (*lower* (*hd b*)), *sup* (*upper* (*last a*)) (*upper* (*last b*)))]
$\langle proof \rangle$

lemma *mlower_adj_sup*[*simp*]: *mlower_adj* (*sup A B*) = *inf* (*mlower_adj A*) (*mlower_adj B*)
$\langle proof \rangle$

lemma *mupper_adj_sup*[*simp*]: *mupper_adj* (*sup A B*) = *sup* (*mupper_adj A*) (*mupper_adj B*)
$\langle proof \rangle$
instance
$\langle proof \rangle$
end

instantiation *minterval_adj* :: ({*lattice,minus_mono*}) *preorder*
begin

instance
⟨proof⟩
end

lemma *set_of_minterval_adj_union*: *set_of_adj A* ∪ *set_of_adj B* ⊆ *set_of_adj* (*sup A B*)
for *A*::′*a*::{*lattice*, *minus_mono*} *minterval_adj*
⟨proof⟩

lemma *minterval_adj_union_commute*: *sup A B* = *sup B A* **for** *A*::′*a*::{*minus_mono*,*lattice*} *minterval_adj*
⟨proof⟩

lemma *minterval_adj_union_mono1*: *set_of_adj a* ⊆ *set_of_adj* (*sup a A*)
for *A* :: ′*a*::{*minus_mono*,*lattice*} *minterval_adj*
⟨proof⟩

lemma *minterval_adj_union_mono2*: *set_of_adj A* ⊆ *set_of_adj* (*sup a A*) **for** *A* :: ′*a*::{*lattice*, *minus_mono*} *minterval_adj*
⟨proof⟩

lift_definition *minterval_adj_of* :: ′*a*::{*minus_mono*} ⇒ ′*a minterval_adj* **is** λ*x*. [*Interval*(*x*, *x*)]
⟨proof⟩

lemma *mlower_adj_minterval_adj_of*[*simp*]: *mlower_adj* (*minterval_adj_of a*) = *a*
⟨proof⟩

lemma *mupper_adj_minterval_adj_of*[*simp*]: *mupper_adj* (*minterval_adj_of a*) = *a*
⟨proof⟩

definition *width_adj* :: ′*a*::{*minus_mono*} *minterval_adj* ⇒ ′*a*
where *width_adj i* = *mupper_adj i* − *mlower_adj i*

12.3.3 Zero and One

instantiation *minterval_adj* :: ({*minus_mono*,*zero*}) *zero*
begin

lift_definition *zero_minterval_adj*::′*a minterval_adj* **is** *mk_mInterval_adj* [*Interval* (0, 0)]
⟨proof⟩

lemma *mlower_adj_zero*[*simp*]: *mlower_adj* 0 = 0
⟨proof⟩

lemma *mupper_adj_zero*[*simp*]: *mupper_adj* 0 = 0
⟨proof⟩

instance ⟨proof⟩
end

instantiation *minterval_adj* :: ({*minus_mono*,*one*}) *one*
begin

lift_definition *one_minterval_adj*::′*a minterval_adj* **is** *mk_mInterval_adj* [*Interval* (1, 1)]
⟨proof⟩

lemma *mlower_adj_one*[*simp*]: *mlower_adj* 1 = 1

⟨proof⟩

lemma *mupper_adj_one*[*simp*]: *mupper_adj* 1 = 1
⟨proof⟩

instance ⟨proof⟩
end

12.3.4 Addition

instantiation *minterval_adj* :: ({*minus_mono*,*ordered_ab_semigroup_add*,*linordered_field*}) *plus*
begin

lift_definition *plus_minterval_adj*::'*a minterval_adj* ⇒ '*a minterval_adj* ⇒ '*a minterval_adj*
is λ *a b* . *mk_mInterval_adj* (*iList_plus a b*)
⟨proof⟩

instance ⟨proof⟩

lemma *interval_plus_com*:
‹*a* + *b* = *b* + *a*› **for** *a*::'*a minterval_adj*
⟨proof⟩

end

12.3.5 Unary Minus

lemma *a*: (*x*::'*a*::*ordered_ab_group_add interval*) ≠ *y* ⟹ −*x* ≠ −*y*
⟨proof⟩

lemma *b*: *distinct* (*is*::'*a*::*ordered_ab_group_add interval list*) ⟹ *distinct* (*map* (λ *i*. −*i*) *is*)
⟨proof⟩

instantiation *minterval_adj* :: ({*minus_mono*, *ordered_ab_group_add*}) *uminus*
begin

lift_definition *uminus_minterval_non_ovl*::'*a minterval_adj* ⇒ '*a minterval_adj*
is λ *is* . *mk_mInterval_non_ovl* (*rev* (*map* (λ *i*. −*i*) *is*))
⟨proof⟩

instance ⟨proof⟩
end

12.3.6 Subtraction

instantiation *minterval_adj* :: ({*minus_mono*, *linordered_field*, *ordered_ab_group_add*}) *minus*
begin

definition *minus_minterval_non_ovl*::'*a minterval_adj* ⇒ '*a minterval_adj* ⇒ '*a minterval_adj*
where *minus_minterval_non_ovl a b* = *a* + − *b*

instance ⟨proof⟩
end

12.3.7 Multiplication

instantiation *minterval_adj* :: ({*minus_mono*, *linordered_field*}) *times*
begin

lift_definition *times_minterval_non_ovl*::'*a minterval_adj* ⇒ '*a minterval_adj* ⇒ '*a minterval_adj*
is λ *a b* . *mk_mInterval_non_ovl* (*iList_times a b*)
⟨*proof*⟩

instance ⟨*proof*⟩
end

12.3.8 Multiplicative Inverse and Division

locale *minterval_adj_division* = *inverse* +
constrains *inverse* :: ‹'*a*::{*linordered_field*, *zero*, *minus*, *minus_mono*, *real_normed_algebra*,*linear_continuum_topology*} *minterval_adj* ⇒ '*a minterval_adj*›
and *divide* :: ‹'*a*::{*linordered_field*, *zero*, *minus*, *minus_mono*, *real_normed_algebra*,*linear_continuum_topology*} *minterval_adj* ⇒ '*a minterval_adj* ⇒ '*a minterval_adj*›
assumes *inverse_left*: ‹¬ 0 ∈ *set_of_adj x* ⟹ 1 ≤ (*inverse x*) ∗ *x*›
and *divide*: ‹¬ 0 ∈ *set_of_adj y* ⟹ *x* ≤ (*divide x y*) ∗ *y*›
begin
end

locale *minterval_adj_division_inverse* = *inverse* +
constrains *inverse* :: ‹'*a*::{*linordered_field*, *zero*, *minus*, *minus_mono*, *real_normed_algebra*,*linear_continuum_topology*} *minterval_adj* ⇒ '*a minterval_adj*›
and *divide* :: ‹'*a*::{*linordered_field*, *zero*, *minus*, *minus_mono*, *real_normed_algebra*,*linear_continuum_topology*} *minterval_adj* ⇒ '*a minterval_adj* ⇒ '*a minterval_adj*›
assumes *inverse_non_zero_def*: ‹¬ 0 ∈ *set_of_adj x* ⟹ (*inverse x*)
= *mInterval_adj* (*mk_mInterval_adj*(*un_op_interval_list* (λ *i*. *mk_interval* (1 / (*upper i*), 1 / (*lower i*))) (*mlist_adj x*)))›
and *divide_non_zero_def*: ‹¬ 0 ∈ *set_of_adj y* ⟹ (*divide x y*) = *inverse y* ∗ *x*›
begin
end

12.3.9 Membership

abbreviation (**in** *preorder*) *in_minterval_adj* ((_/ $\in_{adj}$ _) [51, 51] 50)
where *in_minterval_adj x X* ≡ *x* ∈ *set_of_adj X*

lemma *in_minterval_adj_to_minterval_adj*[*intro*!]: *a* $\in_{adj}$ *minterval_adj_of a*
⟨*proof*⟩

instance *minterval_adj* :: ({*one*, *preorder*, *minus_mono*, *linordered_semiring*}) *power*
⟨*proof*⟩

lemma *set_of_one_adj*[*simp*]: *set_of_adj* (1::'*a*::{*one*, *minus_mono*, *order*} *minterval_adj*) = {1}
⟨*proof*⟩

lifting_update *minterval_adj.lifting*
lifting_forget *minterval_adj.lifting*

end

12.4 Bringing Everything Together (Multi_Interval)

theory
Multi_Interval
imports
Multi_Interval_Overlapping
Multi_Interval_Non_Overlapping
Multi_Interval_Adjacent
Lipschitz_Subdivisions_Refinements
begin

For convince, we provide a theory that provides all three variants of multi-intervals. Note that the order in which these theories are imported is important: importing the theory Multi_Interval_Adjacent last ensures that it provides the default definitions and lemmas.

end

13 Extended Division on Multi-Intervals (Extended_Multi_Interval_Division_Core)

theory
Extended_Multi_Interval_Division_Core
imports
Interval_Division_Non_Zero
Multi_Interval
begin

13.1 Division over List of Intervals

In this theory, we define an extended division operation on intervals. This is a formalization of the interval division given in [4].

definition *inverse_interval* :: ('a::{linorder,minus_mono,zero,one,inverse,infinity,uminus}) interval ⇒ ('a interval) list
where *inverse_interval a* = (
if (¬ 0 ∈ᵢ a) then [mk_interval (1 / (upper a), 1 / (lower a))]
else if lower a = 0 ∧ 0 < upper a then [mk_interval (1/ upper a, ∞)]
else if lower a < 0 ∧ 0 < upper a then [mk_interval (−∞, 1/lower a), mk_interval (1/upper a, ∞)]
else if lower a < upper a ∧ upper a = 0 then [mk_interval(−∞, 1 / lower a)]
else undefined
)
definition ‹minverse = concat o (map inverse_interval)›

13.2 Multi-Interval Division

end

13.2.1 Overlapping Multi-Intervals (Extended_Multi_Interval_Division_Overlapping)

theory
Extended_Multi_Interval_Division_Overlapping
imports
Extended_Multi_Interval_Division_Core
begin

definition ‹minterval_ovl_inverse x = mInterval_ovl (mk_mInterval_ovl(minverse (mlist_ovl x)))›
definition ‹minterval_ovl_divide x y = (minterval_ovl_inverse y) ∗ x›

lemma *set_of_ovl_non_zero_map_inverse*:
assumes ‹0 ∉ set_of_ovl xs›
shows ‹concat (map inverse_interval (mlist_ovl xs)) = map (λi. mk_interval (1 / upper i, 1 / lower i)) (mlist_ovl xs)›
⟨proof⟩

interpretation *minterval_ovl_division_inverse minterval_ovl_divide minterval_ovl_inverse*
⟨proof⟩

end

13.2.2 Non Overlapping Multi-Intervals (Extended_Multi_Interval_Division_Non_Overlapping)

```
theory
  Extended_Multi_Interval_Division_Non_Overlapping
imports
  Extended_Multi_Interval_Division_Core
begin


definition ‹minterval_non_ovl_inverse x = mInterval_non_ovl (mk_mInterval_non_ovl(minverse (mlist_non_ovl x)))›
definition ‹minterval_non_ovl_divide x y = (minterval_non_ovl_inverse y) ∗ x›

lemma set_of_non_ovl_non_zero_map_inverse:
  assumes ‹0 ∉ set_of_non_ovl xs›
    shows ‹concat (map inverse_interval (mlist_non_ovl xs)) = map (λi. mk_interval (1 / upper i, 1 / lower i))
(mlist_non_ovl xs)›
⟨proof⟩

interpretation minterval_non_ovl_division_inverse minterval_non_ovl_divide minterval_non_ovl_inverse
  ⟨proof⟩

end
```

13.2.3 Adjacent Multi-Intervals (Extended_Multi_Interval_Division_Adjacent)

```
theory
  Extended_Multi_Interval_Division_Adjacent
imports
  Extended_Multi_Interval_Division_Core
begin

definition ‹minterval_adj_inverse x = mInterval_adj (mk_mInterval_adj(minverse (mlist_adj x)))›
definition ‹minterval_adj_divide x y = (minterval_adj_inverse y) ∗ x›

lemma set_of_adj_non_zero_map_inverse:
  assumes ‹0 ∉ set_of_adj xs›
  shows ‹concat (map inverse_interval (mlist_adj xs)) = map (λi. mk_interval (1 / upper i, 1 / lower i)) (mlist_adj xs)›
⟨proof⟩

interpretation minterval_adj_division_inverse minterval_adj_divide minterval_adj_inverse
  ⟨proof⟩

end
```

13.3 Bringing Everything Together (Extended_Multi_Interval_Division)

```
theory
  Extended_Multi_Interval_Division
imports
  Extended_Multi_Interval_Division_Overlapping
```

```
  Extended_Multi_Interval_Division_Non_Overlapping
  Extended_Multi_Interval_Division_Adjacent
begin

end
```

14 Extended Multi-Interval Analysis (Extended_Multi_Interval_Analysis)

theory
Extended_Multi_Interval_Analysis
imports
Extended_Multi_Interval_Division
begin

This theory provides extended multi-interval analysis over the type extended reals. All operations work over multi-intervals, i.e., lists of (closed) intervals.

end

Bibliography

[1] A. D. Brucker, T. Cameron-Burke, and A. Stell. Formally verified interval arithmetic and its application to program verification. In *13th IEEE/ACM International Conference on Formal Methods in Software Engineering (FormaliSE 2024)*. IEEE, 2024.

[2] A. D. Brucker and A. Stell. Verifying feedforward neural networks for classification in Isabelle/HOL. In M. Chechik, J.-P. Katoen, and M. Leucker, editors, *Formal Methods (FM 2023)*. Lübeck, Germany, 2023. ISBN: 978-3-642-38915-3. URL: http://www.brucker.ch/bibliography/abstract/brucker.ea-feedforward-nn-verification-2023.

[3] A. Harapanahalli, S. Jafarpour, and S. Coogan. A toolbox for fast interval arithmetic in numpy with an application to formal verification of neural network controlled systems. *CoRR*, abs/2306.15340, 2023. DOI: 10.48550/ARXIV.2306.15340. arXiv: 2306.15340. URL: https://doi.org/10.48550/arXiv.2306.15340.

[4] R. E. Moore, R. B. Kearfott, and M. J. Cloud. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, USA, 2009. ISBN: 0898716691.

[5] D. Ratz. *Inclusion isotone extended interval arithmetic. A toolbox update*. 1997. DOI: 10.5445/IR/67997. Karlsruhe 1996. (Berichte aus dem Forschungsschwerpunkt Computerarithmetik, Intervallrechnung und Numerische Algorithmen mit Ergebnisverifikation. 1996,5.)

[6] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal security analysis of neural networks using symbolic intervals. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 1599–1614, Baltimore, MD, USA. USENIX Association, 2018. ISBN: 9781931971461.