

(Extended) Interval Analysis

Achim D. Brucker^{ORCID}

Amy Stell^{ORCID}

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	9
2	Interval Utilities (Interval_Utilities)	13
2.1	Preliminaries	13
2.2	Interval Bounds and Set Conversion	14
2.3	Linear Order on List of Intervals	16
2.4	Support for Lists of Intervals	18
2.5	Interval Width and Arithmetic Operations	19
2.6	Interval Multiplication	20
2.7	Distance-based Properties of Intervals	21
3	Basic Properties of Interval Division (Interval_Division_Non_Zero)	23
3.1	Preliminaries	23
3.2	A Locale for Interval Division Where the Quotient-Interval does not Contain Zero	24
4	A Naive Interval Division for Real Intervals (Interval_Division_Real)	29
5	Affine Functions (Affine_Functions)	33
5.1	Definition of Affine Functions, Alternative Definitions, and Special Cases	33
5.2	Common Linear Polynomial Functions	34
5.3	Linear Polynomial Functions and Orderings	34
6	Interval Inclusion Isotonicity (Inclusion_Isotonicity)	37
6.1	Interval Extension	37
6.1.1	Textbook Definition of Interval Extension	37
6.1.2	A Stronger Definition of Interval Extension	37
6.2	Interval Inclusion Isotonicity	39
6.2.1	Compositionality of Interval Inclusion Isotonicity	41
6.2.2	Interval Inclusion Isotonicity of the Core Operator	42
6.2.3	Interval Inclusion Isotonicity of Various Functions	43
6.3	Interval Extension and Inclusion Properties	46
6.4	Division	49
7	Lipschitz Continuity of Intervals (Lipschitz_Interval_Extension)	51
7.1	Definition of Lipschitz Continuity on Intervals	51
7.1.1	Lipschitz Continuity of Operations	52
7.1.2	Interval bounds on reals	53
8	Multi-Intervals (Multi_Interval_Preliminaries)	57
8.1	Preliminaries	57
8.1.1	A Class for Capturing Monotonicity of Minus	57
8.1.2	Infrastructure for Lifting Interval Operations to Lists of Intervals	57
8.1.3	Utilities for (Sorted) Lists of Intervals	61

8.1.4	Various Notions of Validity of Sorted Lists of Intervals	77
8.1.5	Union over a List of Intervals	86
9	Subdivisions and Refinements (📖 Lipschitz_Subdivisions_Refinements)	97
9.1	Subdivisions	97
9.2	Refinement	102
9.3	Lipschitz Interval Inclusive	123
9.4	Lipschitz Convergence	129
10	Interval Analysis (📖 Interval_Analysis)	135
11	Extended Division on Intervals (📖 Extended_Interval_Division)	137
12	Extended Interval Analysis (📖 Extended_Interval_Analysis)	139
12.1	Overlapping Multi-Intervals (📖 Multi_Interval_Overlapping)	139
12.1.1	Type Definition	139
12.1.2	Equality and Orderings	139
12.1.3	Zero and One	142
12.1.4	Addition	143
12.1.5	Unary Minus	143
12.1.6	Subtraction	144
12.1.7	Multiplication	144
12.1.8	Multiplicative Inverse and Division	144
12.1.9	Membership	145
12.2	Non-Overlapping Multi-Intervals (📖 Multi_Interval_Non_Overlapping)	145
12.2.1	Type Definition	145
12.2.2	Equality and Orderings	146
12.2.3	Zero and One	149
12.2.4	Addition	150
12.2.5	Unary Minus	150
12.2.6	Subtraction	151
12.2.7	Multiplication	151
12.2.8	Multiplicative Inverse and Division	152
12.2.9	Membership	152
12.3	Adjacent Multi-Intervals (📖 Multi_Interval_Adjacent)	152
12.3.1	A Type For Non Overlapping Multi Intervals	153
12.3.2	Equality and Orderings	153
12.3.3	Zero and One	156
12.3.4	Addition	157
12.3.5	Unary Minus	157
12.3.6	Subtraction	158
12.3.7	Multiplication	158
12.3.8	Multiplicative Inverse and Division	158
12.3.9	Membership	159
12.4	Bringing Everything Together (📖 Multi_Interval)	159
13	Extended Division on Multi-Intervals (📖 Extended_Multi_Interval_Division_Core)	161
13.1	Division over List of Intervals	161

13.2	Multi-Interval Division	161
13.2.1	Overlapping Multi-Intervals (Extended_Multi_Interval_Division_Overlapping)	161
13.2.2	Non Overlapping Multi-Intervals (Extended_Multi_Interval_Division_Non_Overlapping)	162
13.2.3	Adjacent Multi-Intervals (Extended_Multi_Interval_Division_Adjacent)	163
13.3	Bringing Everything Together (Extended_Multi_Interval_Division)	163
14	Extended Multi-Interval Analysis (Extended_Multi_Interval_Analysis)	165

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, where 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).

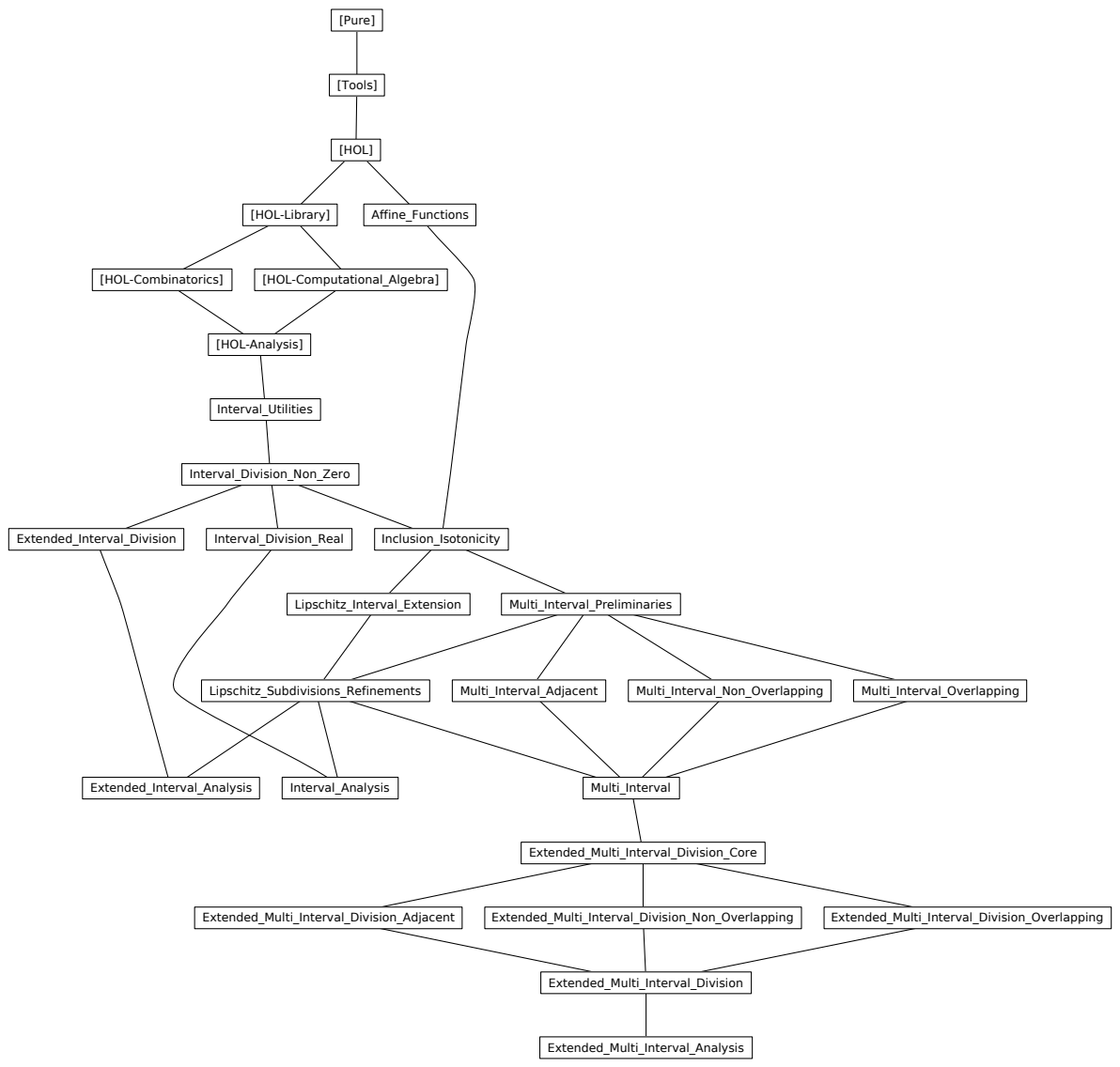


Figure 1.1: The Dependency Graph of the Isabelle Theories.

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 $X :: \langle 'a :: \{preorder, topological_space, ordered_euclidean_space\} \text{ interval} \rangle$

shows $\langle compact \ (set_of\ X) \rangle$

by (*simp add: set_of_eq compact_interval[of lower X upper X]*)

lemma *bounded_set_of*:

fixes $X :: \langle 'a :: \{preorder, topological_space, ordered_euclidean_space\} \text{ interval} \rangle$

shows $\langle bounded \ (set_of\ X) \rangle$

by (*simp add: set_of_eq compact_interval[of lower X upper X]*)

lemma *compact_img_set_of*:

fixes $X :: \langle real \ interval \rangle$ **and** $f :: \langle real \Rightarrow real \rangle$

assumes $\langle continuous_on \ (set_of\ X) \ f \rangle$

shows $\langle compact \ (f \ ' \ set_of\ X) \rangle$

using *compact_continuous_image[of set_of X f, simplified assms] compact_interval*

unfolding *set_of_eq*

by *simp*

lemma *sup_inf_dist_bounded*:

fixes $X :: \langle real \ set \rangle$

shows $\langle bdd_below\ X \implies bdd_above\ X \implies \forall x \in X. \forall x' \in X. dist\ x\ x' \leq Sup\ X - Inf\ X \rangle$

using *cInf_lower[of _ X] cSup_upper[of _ X]*

apply (*auto simp add: dist_real_def*) [1]

by (*smt (z3)*)

lemma *set_of_nonempty[simp]*: $\langle set_of\ X \neq \{\} \rangle$

by (*simp add: set_of_eq*)

lemma *lower_in_interval[simp]*: $\langle lower\ X \in_i\ X \rangle$

by (*simp add: in_intervalI*)

lemma *upper_in_interval[simp]*: $\langle upper\ X \in_i\ X \rangle$

by (*simp add: in_intervalI*)

lemma *bdd_below_set_of*: $\langle bdd_below \ (set_of\ X) \rangle$

by (*metis atLeastAtMost_iff bdd_below.unfold set_of_eq*)

lemma *bdd_above_set_of*: $\langle \text{bdd_above } (\text{set_of } X) \rangle$
by (*metis atLeastAtMost_iff bdd_above.unfold set_of_eq*)

lemma *closed_set_of*: $\langle \text{closed } (\text{set_of } (X::\text{real interval})) \rangle$
by (*metis closed_real_atLeastAtMost set_of_eq*)

lemma *set_f_nonempty*: $\langle f ' \text{set_of } X \neq \{\} \rangle$
apply (*simp add:image_def set_of_eq*)
by *fastforce*

lemma *interval_linorder_case_split*[*case_names LeftOf Including RightOf*]:
assumes $\langle \text{upper } x < c \implies P (x::('a::\text{linorder interval})) \rangle$
 $\langle (c \in_i x \implies P x) \rangle$
 $\langle (c < \text{lower } x \implies P x) \rangle$
shows $\langle P x \rangle$
proof (*insert assms, cases upper x < c*)
case *True*
then show *?thesis*
using *assms(1) by blast*
next
case *False*
then show *?thesis*
proof (*cases c \in_i x*)
case *True*
then show *?thesis*
using *assms(2) by blast*
next
case *False*
then show *?thesis*
by (*meson assms(1) assms(3) in_intervall not_le_imp_less*)
qed
qed

lemma *foldl_conj_True*:
 $\langle \text{foldl } (\wedge) x xs = \text{list_all } (\lambda e. e = \text{True}) (x\#xs) \rangle$
by (*induction xs rule:rev_induct, auto*)

lemma *foldl_conj_set_True*:
 $\langle \text{foldl } (\wedge) x xs = (\forall e \in \text{set } (x\#xs). e = \text{True}) \rangle$
by (*induction xs rule:rev_induct, auto*)

2.2 Interval Bounds and Set Conversion

lemma *sup_set_of*:
fixes $X :: 'a::\{\text{conditionally_complete_lattice}\}$ *interval*
shows $\text{Sup } (\text{set_of } X) = \text{upper } X$
unfolding *set_of_def upper_def*
using *cSup_atLeastAtMost lower_le_upper[of X]*
by (*simp add: upper_def lower_def*)

lemma *inf_set_of*:
fixes $X :: 'a::\{\text{conditionally_complete_lattice}\}$ *interval*

```

shows Inf (set_of X) = lower X
unfolding set_of_def lower_def
using cInf_atLeastAtMost lower_le_upper[of X]
by (simp add: upper_def lower_def)

```

```

lemma inf_le_sup_set_of:
  fixes X :: 'a::{conditionally_complete_lattice} interval
  shows Inf (set_of X) ≤ Sup (set_of X)
  using sup_set_of inf_set_of lower_le_upper
  by metis

```

```

lemma in_bounds: ⟨x ∈i X ⟹ lower X ≤ x ∧ x ≤ upper X⟩
  by (simp add: set_of_eq)

```

```

lemma lower_bounds[simp]:
  assumes ⟨L ≤ U⟩
  shows ⟨lower (Interval(L,U)) = L⟩
  using assms
  apply (simp add: lower.rep_eq)
  by (simp add: bounds_of_interval_eq_lower_upper lower_Interval upper_Interval)

```

```

lemma upper_bounds [simp]:
  assumes ⟨L ≤ U⟩
  shows ⟨upper (Interval(L,U)) = U⟩
  using assms
  apply (simp add: upper.rep_eq)
  by (simp add: bounds_of_interval_eq_lower_upper lower_Interval upper_Interval)

```

```

lemma lower_point_interval[simp]: ⟨lower (Interval (x,x)) = x⟩
  by (simp)

```

```

lemma upper_point_interval[simp]: ⟨upper (Interval (x,x)) = x⟩
  by (simp)

```

```

lemma map2_nth:
  assumes ⟨length xs = length ys⟩
  and   ⟨n < length xs⟩
  shows ⟨(map2 f xs ys)!n = f (xs!n) (ys!n)⟩
  using assms by simp

```

```

lemma map_set: ⟨a ∈ set (map f X) ⟹ (∃ x ∈ set X . f x = a)⟩
  by auto

```

```

lemma map_pair_set_left: ⟨(a,b) ∈ set (zip (map f X) (map f Y)) ⟹ (∃ x ∈ set X . f x = a)⟩
  by (meson map_set set_zip_leftD)

```

```

lemma map_pair_set_right: ⟨(a,b) ∈ set (zip (map f X) (map f Y)) ⟹ (∃ y ∈ set Y . f y = b)⟩
  by (meson map_set set_zip_rightD)

```

```

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)⟩
  by (meson map_pair_set_left set_zip_rightD zip_same)

```

```

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)⟩
  by (insert assms(1), induction X Y rule: list_induct2, auto)

```

```

definition map_interval_swap :: ⟨('a::linorder × 'a) list ⟹ 'a interval list⟩ where

```

$\langle \text{map_interval_swap} = \text{map } (\lambda (x,y). \text{Interval } (\text{if } x \leq y \text{ then } (x,y) \text{ else } (y,x))) \rangle$

definition $\text{mk_interval} :: \langle 'a::\text{linorder} \times 'a \rangle \Rightarrow 'a \text{ interval} \rangle$ **where**
 $\langle \text{mk_interval} = (\lambda (x,y). \text{Interval } (\text{if } x \leq y \text{ then } (x,y) \text{ else } (y,x))) \rangle$

definition $\text{mk_interval}' :: \langle 'a::\text{linorder} \times 'a \rangle \Rightarrow 'a \text{ interval} \rangle$ **where**
 $\langle \text{mk_interval}' = (\lambda (x,y). (\text{if } x \leq y \text{ then Interval}(x,y) \text{ else Interval}(y,x))) \rangle$

lemma $\text{map_interval_swap_code}[\text{code}]$:
 $\langle \text{map_interval_swap} = \text{map } (\lambda (x,y). \text{the } (\text{if } x \leq y \text{ then Interval}' x y \text{ else Interval}' y x)) \rangle$
unfolding $\text{map_interval_swap_def}$ **by** (rule ext, rule arg_cong, auto simp add: Interval'.abs_eq)

lemma $\text{mk_interval_code}[\text{code}]$:
 $\langle \text{mk_interval} = (\lambda (x,y). \text{the } (\text{if } x \leq y \text{ then Interval}' x y \text{ else Interval}' y x)) \rangle$
unfolding mk_interval_def **by** (rule ext, rule arg_cong, auto simp add: Interval'.abs_eq)

lemma $\text{mk_interval}'$:
 $\langle \text{mk_interval} = (\lambda (x,y). (\text{if } x \leq y \text{ then Interval}(x,y) \text{ else Interval}(y,x))) \rangle$
unfolding mk_interval_def **by** (rule ext, rule arg_cong, auto)

lemma $\text{mk_interval_lower}[\text{simp}]$: $\langle \text{lower } (\text{mk_interval } (x,y)) = (\text{if } x \leq y \text{ then } x \text{ else } y) \rangle$
by (simp add: lower_def Interval_inverse mk_interval_def)

lemma $\text{mk_interval_upper}[\text{simp}]$: $\langle \text{upper } (\text{mk_interval } (x,y)) = (\text{if } x \leq y \text{ then } y \text{ else } x) \rangle$
by (simp add: upper_def Interval_inverse mk_interval_def)

2.3 Linear Order on List of Intervals

definition
 $\text{le_interval_list} :: \langle 'a::\text{linorder} \rangle \text{ interval list} \Rightarrow 'a \text{ interval list} \Rightarrow \text{bool} \langle (_ / \leq_I _) [51, 51] 50 \rangle$
where
 $\langle \text{le_interval_list } Xs Ys \equiv (\text{length } Xs = \text{length } Ys) \wedge (\text{foldl } (\wedge) \text{True } (\text{map2 } (\leq) Xs Ys)) \rangle$

lemma $\text{le_interval_single}$: $\langle (x \leq y) = ([x] \leq_I [y]) \rangle$
unfolding $\text{le_interval_list_def}$ **by** simp

lemma $\text{le_interval_empty}[\text{simp}]$: $\langle [] \leq_I [] \rangle$
unfolding $\text{le_interval_list_def}$ **by** simp

lemma $\text{le_interval_list_rev}$: $\langle (is \leq_I js) = (\text{rev } is \leq_I \text{rev } js) \rangle$
unfolding $\text{le_interval_list_def}$
by (safe, simp_all add: foldl_conj_set_True zip_rev)

lemma $\text{le_interval_list_imp_length}$:
assumes $\langle Xs \leq_I Ys \rangle$ **shows** $\langle \text{length } Xs = \text{length } Ys \rangle$
using **assms** **unfolding** $\text{le_interval_list_def}$
by simp

lemma lsplit_left : **assumes** $\langle \text{length } (xs) = \text{length } (ys) \rangle$
and $\langle (\forall n < \text{length } (x \# xs). (x \# xs) ! n \leq (y \# ys) ! n) \rangle$ **shows** \langle
 $(\forall n < \text{length } xs. xs ! n \leq ys ! n) \wedge x \leq y \rangle$
using **assms** **by** auto


```

lemma lsplit_right: assumes  $\langle \text{length } xs = \text{length } ys \rangle$ 
  and  $\langle (\forall n < \text{length } xs. xs!n \leq ys!n) \wedge x \leq y \rangle$ 
shows  $\langle n < \text{length } (x \# xs) \longrightarrow (x \# xs)!n \leq (y \# ys)!n \rangle$ 
proof(cases  $n=0$ )
  case True
  then show ?thesis using assms by (simp)
next
  case False
  then show ?thesis using assms by simp
qed

```

```

lemma lsplit: assumes  $\langle \text{length } xs = \text{length } ys \rangle$ 
shows  $\langle (\forall n < \text{length } (x \# xs). (x \# xs)!n \leq (y \# ys)!n) =$ 
   $(\langle (\forall n < \text{length } xs. xs!n \leq ys!n) \wedge x \leq y \rangle)$ 
using assms lsplit_left lsplit_right by metis

```

```

lemma le_interval_list_all':
  assumes  $\langle \text{length } Xs = \text{length } Ys \rangle$  and  $\langle Xs \leq_I Ys \rangle$  shows  $\langle \forall n < \text{length } Xs. Xs!n \leq Ys!n \rangle$ 
proof(insert assms, induction rule:list_induct2)
  case Nil
  then show ?case by simp
next
  case (Cons  $x$   $xs$   $y$   $ys$ )
  then show ?case
    using lsplit[of  $xs$   $ys$   $x$   $y$ ] le_interval_list_def
    unfolding le_interval_list_def
    by(auto simp add: foldl_conj_True)
qed

```

```

lemma le_interval_list_all2:
  assumes  $\langle \text{length } Xs = \text{length } Ys \rangle$ 
  and  $\langle \forall n < \text{length } Xs. (Xs!n \leq Ys!n) \rangle$ 
shows  $\langle Xs \leq_I Ys \rangle$ 
proof(insert assms, induction rule:list_induct2)
  case Nil
  then show ?case by simp
next
  case (Cons  $x$   $xs$   $y$   $ys$ )
  then show ?case
    using lsplit[of  $xs$   $ys$   $x$   $y$ ] le_interval_list_def
    unfolding le_interval_list_def
    by(auto simp add: foldl_conj_True)
qed

```

```

lemma le_interval_list_all:
  assumes  $\langle Xs \leq_I Ys \rangle$  shows  $\langle \forall n < \text{length } Xs. Xs!n \leq Ys!n \rangle$ 
using assms le_interval_list_all' le_interval_list_imp_length
by auto

```

```

lemma le_interval_list_imp:
  assumes  $\langle Xs \leq_I Ys \rangle$  shows  $\langle n < \text{length } Xs \longrightarrow Xs!n \leq Ys!n \rangle$ 
using assms le_interval_list_all' le_interval_list_imp_length
by auto

```

lemma *interval_set_leq_eq*: $\langle X \leq Y \rangle = (\text{set_of } X \subseteq \text{set_of } Y)$
for $X :: \langle 'a :: \text{linordered_ring interval} \rangle$
by (*simp add: less_eq_interval_def set_of_eq*)

lemma *times_interval_right*:
fixes $X Y C :: \langle 'a :: \text{linordered_ring interval} \rangle$
assumes $\langle X \leq Y \rangle$
shows $\langle C * X \leq C * Y \rangle$
using *assms[simplified interval_set_leq_eq]*
apply(*subst interval_set_leq_eq*)
by (*simp add: set_of_mul_inc_right*)

lemma *times_interval_left*:
fixes $X Y C :: \langle 'a :: \{\text{real_normed_algebra, linordered_ring, linear_continuum_topology}\} \text{ interval} \rangle$
assumes $\langle X \leq Y \rangle$
shows $\langle X * C \leq Y * C \rangle$
using *assms[simplified interval_set_leq_eq]*
apply(*subst interval_set_leq_eq*)
by (*simp add: set_of_mul_inc_left*)

2.4 Support for Lists of Intervals

abbreviation *in_interval_list*:: $\langle ('a :: \text{preorder}) \text{ list} \Rightarrow 'a \text{ interval list} \Rightarrow \text{bool} \rangle ((_ / \in_I _) [51, 51] 50)$
where $\langle \text{in_interval_list } xs \ Xs \equiv \text{foldl } (\wedge) \ \text{True } (\text{map2 } (\text{in_interval}) \ xs \ Xs) \rangle$

lemma *interval_of_in_interval_list[simp]*: $\langle xs \in_I \ \text{map } \text{interval_of } xs \rangle$
proof(*induction xs*)
case *Nil*
then show *?case by simp*
next
case (*Cons a xs*)
then show *?case*
by (*simp add: in_interval_to_interval*)
qed

lemma *interval_of_in_eq*: $\langle \text{interval_of } x \leq X = (x \in_i X) \rangle$
by (*simp add: less_eq_interval_def set_of_eq*)

lemma *interval_of_list_in*:
assumes $\langle \text{length } \text{inputs} = \text{length } \text{Inputs} \rangle$
shows $\langle (\text{map } \text{interval_of } \text{inputs} \leq_I \ \text{Inputs}) = (\text{inputs} \in_I \ \text{Inputs}) \rangle$
unfolding *le_interval_list_def*
proof(*insert assms, induction inputs Inputs rule: list_induct2*)
case *Nil*
then show *?case by simp*
next
case (*Cons x xs y ys*)
then show *?case*
proof(*cases interval_of x ≤ y*)
case *True*
then show *?thesis*

```

    using Cons by (simp add: interval_of_in_eq)
next
case False
then show ?thesis using foldl_conj_True interval_of_in_eq by auto
qed
qed

```

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>
by (simp add: width_def)

```

```

lemma interval_width_times:
fixes a :: 'a::{linordered_ring} and A :: 'a interval
shows width (interval_of a * A) = |a| * width A
proof -
have width (interval_of a * A) = upper (interval_of a * A) - lower (interval_of a * A)
by (simp add: width_def)
also have ... = (if a > 0 then a * upper A - a * lower A else a * lower A - a * upper A)
proof -
have upper (interval_of a * A) = max (a * lower A) (a * upper A)
by (simp add: upper_times)
moreover have lower (interval_of a * A) = min (a * lower A) (a * upper A)
by (simp add: lower_times)
ultimately show ?thesis
by (simp add: mult_left_mono mult_left_mono_neg)
qed
also have ... = |a| * (upper A - lower A)
by (simp add: right_diff_distrib)
finally show ?thesis
by (simp add: width_def)
qed

```

```

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)
proof -
have a0:  $\forall i ia. \text{lower } (\text{sup } i ia) = \text{min } (\text{lower } i::\text{real}) (\text{lower } ia)$ 
by (simp add: inf_min)
have a1:  $\forall i ia. \text{upper } (\text{sup } i ia) = \text{max } (\text{upper } i::\text{real}) (\text{upper } ia)$ 
by (simp add: sup_max)
show ?thesis
using a0 a1 by (simp add: inf_min sup_max width_def)
qed

```

```

lemma width_expanded: <interval_of (width Y) = Interval(upper Y - lower Y, upper Y - lower Y)>
unfolding width_def interval_of_def by simp

```

```

lemma interval_width_positive:
fixes X :: 'a::{linordered_ring} interval
shows <0 ≤ width X>

```

```

using lower_le_upper
by (simp add: width_def)

```

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)})⟩
by (simp add: times_interval_def bounds_of_interval_eq_lower_upper Let_def min_def)

```

```

lemma interval_times_scalar: ⟨interval_of a * A = mk_interval(a * lower A, a * upper A)⟩
proof –
  have upper (interval_of a * A) = max (a * lower A) (a * upper A)
  by (simp add: upper_times)
  moreover have lower (interval_of a * A) = min (a * lower A) (a * upper A)
  by (simp add: lower_times)
  ultimately show ?thesis
  by (simp add: interval_eq_iff)
qed

```

```

lemma interval_times_scalar_pos_l:
assumes ⟨0 ≤ a⟩
shows ⟨interval_of a * A = Interval(a * lower A, a * upper A)⟩
proof –
  have upper (interval_of a * A) = a * upper A
  using assms
  by (simp add: upper_times mult_left_mono)
  moreover have lower (interval_of a * A) = a * lower A
  using assms
  by (simp add: lower_times mult_left_mono)
  ultimately show ?thesis
  using assms
  by (metis bounds_of_interval_eq_lower_upper bounds_of_interval_inverse lower_le_upper)
qed

```

```

lemma interval_times_scalar_pos_r:
fixes a :: 'a::{linordered_idom}
assumes ⟨0 ≤ a⟩
shows ⟨A * interval_of a = Interval(a * lower A, a * upper A)⟩
proof –
  have upper (A * interval_of a) = a * upper A
  using assms
  by (simp add: mult.commute mult_left_mono upper_times)
  moreover have lower (A * interval_of a) = a * lower A
  using assms
  by (simp add: lower_times mult_right_mono)
  ultimately show ?thesis
  using assms
  by (metis bounds_of_interval_eq_lower_upper bounds_of_interval_inverse lower_le_upper)
qed

```

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 $X :: 'a::\{\text{linordered_ring}\}$ interval

shows $\langle \text{width} (\text{interval_of } a * X + \text{interval_of } b * Y) = |a| * \text{width } X + |b| * \text{width } Y \rangle$

using *interval_width_addition interval_width_times* **by** *metis*

lemma *width_of_product_interval_bound_real*:

fixes $X :: \text{real interval}$

shows $\langle \text{interval_of} (\text{width } (X * Y)) \leq \text{abs_interval}(X) * \text{interval_of} (\text{width } Y) + \text{abs_interval}(Y) * \text{interval_of} (\text{width } X) \rangle$

proof –

have a_0 : $\text{lower } X \leq \text{upper } X \langle \text{lower } Y \leq \text{upper } Y \rangle$

using *lower_le_upper* **by** *simp_all*

have a_1 : $\langle \text{width } Y \geq 0 \rangle$ **and** a_1' : $\langle \text{width } X \geq 0 \rangle$

using a_0 *interval_width_positive* **by** *simp_all*

have a_2 : $\langle \text{abs_interval}(X) * \text{interval_of} (\text{width } Y) = \text{Interval} (\text{width } Y * \text{lower} (\text{abs_interval } X), \text{width } Y * \text{upper} (\text{abs_interval } X)) \rangle$

using *interval_times_scalar_pos_r* a_0 a_1 **by** *blast*

have a_3 : $\langle \text{abs_interval}(Y) * \text{interval_of} (\text{width } X) = \text{Interval} (\text{width } X * \text{lower} (\text{abs_interval } Y), \text{width } X * \text{upper} (\text{abs_interval } Y)) \rangle$

using *interval_times_scalar_pos_r* a_0 *interval_width_positive* **by** *blast*

have a_4 : $\langle \text{abs_interval}(X) * \text{interval_of} (\text{width } Y) + \text{abs_interval}(Y) * \text{interval_of} (\text{width } X) = \text{Interval} (\text{width } Y * \text{lower} (\text{abs_interval } X) + \text{width } X * \text{lower} (\text{abs_interval } Y), \text{width } Y * \text{upper} (\text{abs_interval } X) + \text{width } X * \text{upper} (\text{abs_interval } Y)) \rangle$

using a_0 a_2 a_3 *unfolding plus_interval_def*

apply (*simp add:bounds_of_interval_eq_lower_upper mult_left_mono interval_width_positive*)

by (*auto simp add:bounds_of_interval_eq_lower_upper mult_left_mono interval_width_positive*)

have a_5 : $\langle \text{width}(X * Y) = \text{Max} \{(\text{lower } X * \text{lower } Y), (\text{lower } X * \text{upper } Y), (\text{upper } X * \text{lower } Y), (\text{upper } X * \text{upper } Y)\} - \text{Min} \{(\text{lower } X * \text{lower } Y), (\text{lower } X * \text{upper } Y), (\text{upper } X * \text{lower } Y), (\text{upper } X * \text{upper } Y)\} \rangle$

using *lower_times_upper_times* *unfolding width_def* **by** *metis*

have a_6 : $\langle \text{Max} \{(\text{lower } X * \text{lower } Y), (\text{lower } X * \text{upper } Y), (\text{upper } X * \text{lower } Y), (\text{upper } X * \text{upper } Y)\} - \text{Min} \{(\text{lower } X * \text{lower } Y), (\text{lower } X * \text{upper } Y), (\text{upper } X * \text{lower } Y), (\text{upper } X * \text{upper } Y)\} \leq \text{width } Y * \text{upper} (\text{abs_interval } X) + \text{width } X * \text{upper} (\text{abs_interval } Y) \rangle$

using a_0 a_1 a_1' *unfolding width_def*

by (*simp, smt (verit) a_0 mult.commute mult_less_cancel_left_pos mult_minus_left right_diff_distrib*)

have a_7 : $\langle \text{width } Y * \text{lower} (\text{abs_interval } X) + \text{width } X * \text{lower} (\text{abs_interval } Y) \leq \text{Max} \{(\text{lower } X * \text{lower } Y), (\text{lower } X * \text{upper } Y), (\text{upper } X * \text{lower } Y), (\text{upper } X * \text{upper } Y)\} - \text{Min} \{(\text{lower } X * \text{lower } Y), (\text{lower } X * \text{upper } Y), (\text{upper } X * \text{lower } Y), (\text{upper } X * \text{upper } Y)\} \rangle$

using a_0 a_1 a_1' *unfolding width_def*

by (*simp, smt (verit) a_0 mult.commute mult_less_cancel_left_pos mult_minus_left right_diff_distrib*)

have $\langle \text{interval_of} (\text{Max} \{(\text{lower } X * \text{lower } Y), (\text{lower } X * \text{upper } Y), (\text{upper } X * \text{lower } Y), (\text{upper } X * \text{upper } Y)\} - \text{Min} \{(\text{lower } X * \text{lower } Y), (\text{lower } X * \text{upper } Y), (\text{upper } X * \text{lower } Y), (\text{upper } X * \text{upper } Y)\}) \leq$

$\text{Interval} (\text{width } Y * \text{lower} (\text{abs_interval } X) + \text{width } X * \text{lower} (\text{abs_interval } Y),$

$\text{width } Y * \text{upper} (\text{abs_interval } X) + \text{width } X * \text{upper} (\text{abs_interval } Y)) \rangle$

using a_6 a_7 *unfolding less_eq_interval_def*

by (*smt (verit, ccfv_threshold) lower_bounds_lower_interval_of_upper_bounds_upper_interval_of*)

then show *?thesis* **using** a_4 a_5 a_6 **by** *presburger*

qed

lemma *width_of_product_interval_bound_int*:

```

fixes X :: int interval
shows <interval_of (width (X * Y)) ≤ abs_interval(X) * interval_of (width Y) + abs_interval(Y) * interval_of (width X)>
proof –
  have a0: lower X ≤ upper X <lower Y ≤ upper Y>
    using lower_le_upper by simp_all
  have a1: <width Y ≥ 0> and <width X ≥ 0>
    using a0 interval_width_positive by simp_all
  have a2: <abs_interval(X) * interval_of (width Y) = Interval (width Y * lower (abs_interval X), width Y * upper
(abs_interval X))>
    using interval_times_scalar_pos_r a0 a1 by blast
  have a3: <abs_interval(Y) * interval_of (width X) = Interval (width X * lower (abs_interval Y), width X * upper
(abs_interval Y))>
    using interval_times_scalar_pos_r a0 interval_width_positive by blast
  have a4: <abs_interval(X) * interval_of (width Y) + abs_interval(Y) * interval_of (width X) = Interval (width Y * lower
(abs_interval X) + width X * lower (abs_interval Y), width Y * upper (abs_interval X) + width X * upper (abs_interval
Y))>
    using a0 a2 a3 unfolding plus_interval_def
    by (auto simp add: bounds_of_interval_eq_lower_upper mult_left_mono interval_width_positive)
  have a5: <width(X * Y) = Max {(lower X * lower Y), (lower X * upper Y), (upper X * lower Y), (upper X * upper Y)} –
Min {(lower X * lower Y), (lower X * upper Y), (upper X * lower Y), (upper X * upper Y)}>
    using lower_times_upper_times unfolding width_def by metis
  have a6: <Max {(lower X * lower Y), (lower X * upper Y), (upper X * lower Y), (upper X * upper Y)} – Min {(lower X *
lower Y), (lower X * upper Y), (upper X * lower Y), (upper X * upper Y)} ≤ width Y * upper (abs_interval X) + width X
* upper (abs_interval Y)>
    using a0 unfolding width_def
    by (simp, smt (verit) a0 mult_commute mult_less_cancel_left_pos mult_minus_left right_diff_distrib)
  have a7: <width Y * lower (abs_interval X) + width X * lower (abs_interval Y) ≤ Max {(lower X * lower Y), (lower X *
upper Y), (upper X * lower Y), (upper X * upper Y)} – Min {(lower X * lower Y), (lower X * upper Y), (upper X * lower
Y), (upper X * upper Y)}>
    using a0 unfolding width_def
    apply (auto)[1]
    by (smt (verit) a0(1) a0(2) left_diff_distrib mult_less_cancel_left_pos int_distrib(3)
mult_less_cancel_left mult_minus_right mult_commute mult_le_o_iff right_diff_distrib'
mult_left_mono mult_le_cancel_right right_diff_distrib zero_less_mult_iff )+
  have <interval_of (Max {(lower X * lower Y), (lower X * upper Y), (upper X * lower Y), (upper X * upper Y)} –
Min {(lower X * lower Y), (lower X * upper Y), (upper X * lower Y), (upper X * upper Y)}) ≤
Interval (width Y * lower (abs_interval X) + width X * lower (abs_interval Y),
width Y * upper (abs_interval X) + width X * upper (abs_interval Y))>
    using a6 a7 unfolding less_eq_interval_def
    by (smt (verit, ccfv_threshold) lower_bounds lower_interval_of upper_bounds upper_interval_of)
then show ?thesis using a4 a5 a6 by presburger
qed

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 $x / z \leq x / y$

proof —

have $x * y \leq x * z$ **using** *assms* **by** *simp*

hence $(x * y) / (y * z) \leq (x * z) / (y * z)$

using *assms*

by (*simp add: divide_right_mono zero_less_mult_iff neg_divide_le_eq*)

thus *?thesis* **using** *assms* **by** *auto*[1]

qed

lemma *division_leq*:

fixes $x :: 'a::\{linordered_field\}$

assumes $0 < x$ and $y \leq z$ and $\langle y \neq 0 \wedge z \neq 0 \rangle$ and $\langle (y < 0 \wedge z < 0) \vee (0 < y \wedge 0 < z) \rangle$

shows $x / z \leq x / y$

proof (*cases* $\langle (y < 0 \wedge z < 0) \rangle$)

case *True*

then show *?thesis* **using** *assms* *division_leq_neg* **by** *blast*

next

case *False*

have $\langle (0 < y \wedge 0 < z) \rangle$ **using** *assms* *False* **by** *blast*

then show *?thesis*

using *assms*

by (*simp add: frac_le*)

qed

lemma *upper_leq_lower_div*:

fixes $Y :: 'a::\{linordered_field\}$ *interval*

assumes $\langle \text{lower } Y \leq \text{upper } Y \rangle$ and $\langle \neg 0 \in_i Y \rangle$

shows $\langle 1 / \text{upper } Y \leq 1 / \text{lower } Y \rangle$

using *assms* *division_leq* *frac_le*

by (*metis* *atLeastAtMost_iff* *inverse_eq_divide* *le_imp_inverse_le*)

`le_imp_inverse_le_neg linorder_not_less set_of_eq`)

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: <¬ 0 ∈i x => 1 ≤ (inverse x) * x>`

and `divide: <¬ 0 ∈i y => x ≤ (divide x y) * y>`

begin

end

lemma `interval_non_zero_eq:`

`<¬ 0 ∈i (i::'a::{\linorder, zero} interval) = (lower i < 0 ∧ upper i < 0) ∨ (lower i > 0 ∧ upper i > 0)>`

by `(metis in_bounds in_interval linorder_not_less lower_le_upper order_le_less_trans)`

lemma `inverse_includes_one:`

assumes `<¬ 0 ∈i (i::'a::{\division_ring, linordered_ring} interval)>`

shows `<1 ∈i (mk_interval (1 / upper i, 1 / lower i)) * i>`

using `assms interval_non_zero_eq[of i]`

apply `(simp add: set_of_eq)`

apply `(safe, simp_all)`

by `(metis in_bounds lower_in_interval mk_interval_upper nonzero_eq_divide_eq times_in_interval upper_in_interval)+`

lemma `inverse_includes_one':`

assumes `<¬ 0 ∈i (i::'a::{\division_ring, linordered_ring} interval)>`

shows `<1 ≤ (mk_interval (1 / upper i, 1 / lower i)) * i>`

by `(simp add: assms in_bounds inverse_includes_one less_eq_interval_def)`

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 ∈i x => (inverse x) = mk_interval(1 / (upper x), 1 / (lower x))>`

and `divide_non_zero_def: <¬ 0 ∈i y => (divide x y) = inverse y * x>`

begin

sublocale `interval_division divide inverse`

apply `(standard)`

subgoal

by `(simp add: inverse_includes_one' inverse_non_zero_def)`

subgoal

by `(metis (no_types, opaque_lifting) divide_non_zero_def interval_mul_commute inverse_includes_one' inverse_non_zero_def mult.assoc one_times_ivl_right times_interval_right)`

done

lemma `inverse_left_ge_one:`

assumes `<¬ 0 ∈i x>`

shows `<1 ≤ (inverse x) * x>`


```

proof –
  have lower_ne_zero: ⟨lower x ≠ 0⟩
    using assms lower_in_interval by metis
  have upper_ne_zero: ⟨upper x ≠ 0⟩
    using assms lower_in_interval by metis
  have ⟨1 ≤ (mk_interval (1 / (upper x), 1 / (lower x))) * x⟩
  proof(cases 1 / upper x ≤ 1 / lower x)
  case True note * = this
  then show ?thesis
  proof(cases upper x = lower x)
  case True
  then show ?thesis
    using upper_times[of mk_interval (1 / upper x, 1 / lower x) x]
      lower_times[of mk_interval (1 / upper x, 1 / lower x) x]
      interval_eq_iff[of mk_interval (1 / upper x, 1 / lower x) * x 1]
      lower_ne_zero upper_ne_zero
    unfolding mk_interval'
    by simp
  next
  case False
  then show ?thesis
    using interval_eq_iff[of mk_interval (1 / upper x, 1 / lower x) * x 1]
      upper_times[of mk_interval (1 / upper x, 1 / lower x) x]
      lower_times[of mk_interval (1 / upper x, 1 / lower x) x]
  proof –
  have 1 / lower x = upper (mk_interval (1 / upper x, 1 / lower x))
    by (simp add: *)
  then show ?thesis
    by (metis (no_types) in_bounds less_eq_interval_def lower_in_interval lower_one
      nonzero_divide_eq_eq times_in_interval upper_in_interval upper_ne_zero upper_one)
  qed
qed
next
  case False
  then show ?thesis
    using interval_eq_iff[of mk_interval (1 / upper x, 1 / lower x) * x 1]
      upper_times[of mk_interval (1 / upper x, 1 / lower x) x]
      lower_times[of mk_interval (1 / upper x, 1 / lower x) x]
    using assms lower_le_upper upper_leq_lower_div by blast
  qed
  then show ?thesis
by (simp add: assms inverse_non_zero_def)
qed

lemma division_right_ge_refl:
  assumes ⟨¬ 0 ∈i y⟩
  shows ⟨x ≤ x * ((inverse y) * y)⟩
proof –
  have a1: ⟨set_of 1 ⊆ set_of ((inverse y) * y)⟩
    using inverse_left_ge_one[of y, simplified assms, simplified]
  by (simp add: interval_set_leq_eq)
show ?thesis
  using set_of_mul_inc_right[of 1 mk_interval (1 / upper y, 1 / lower y) * y x,
    simplified one_times_ivl_right[of x] a1, simplified]

```

by (metis a1 interval_set_leq_eq one_times_ivl_right times_interval_right)
qed

lemma division_right_ge_refl':
assumes $\langle \neg 0 \in_i y \rangle$
shows $\langle x \leq x * \text{inverse } y * y \rangle$
by (simp add: assms division_right_ge_refl mult.assoc)

lemma interval_div_constant:
assumes $\langle 0 \notin \text{set_of } Y \rangle$ and $\langle 0 \leq x \rangle$
shows $\langle \text{divide } (\text{interval_of } x) \ Y = \text{Interval}(x / \text{upper } Y, x / \text{lower } Y) \rangle$
proof –
have l: $\langle \text{lower } Y \leq \text{upper } Y \rangle$ **using** lower_le_upper **by** simp
have $\langle 1 / \text{upper } Y \leq 1 / \text{lower } Y \rangle$ **using** assms l
by (metis divide_left_mono frac_le in_intervall linorder_not_less mult_neg_neg order_less_le zero_less_one_class.zero_le_one)
then show ?thesis
using
interval_of.abs_eq[of x]
assms divide_non_zero_def[of Y interval_of x, simplified] assms, simplified]
inverse_non_zero_def[of Y, simplified] assms, simplified]
interval_times_scalar_pos_l interval_times_scalar_pos_r **by** fastforce
qed

lemma interval_of_width:
assumes $\langle \neg 0 \in_i Y \rangle$
shows $\langle \text{interval_of}(\text{width } (\text{divide } (\text{interval_of } 1) \ Y)) = \text{Interval}(1 / \text{lower } Y - 1 / \text{upper } Y, 1 / \text{lower } Y - 1 / \text{upper } Y) \rangle$
proof(cases Y rule:interval_linorder_case_split[of _ 0 λ Y. interval_of(width (divide (interval_of 1) Y))
= Interval(1 / lower Y - 1 / upper Y, 1 / lower Y - 1 / upper Y)])
case LeftOf
have $\langle 1 / \text{upper } Y \leq 1 / \text{lower } Y \rangle$
using assms division_leq_neg LeftOf
by (simp add: le_divide_eq)
then show ?case
using interval_div_constant upper_bounds lower_bounds assms
unfolding width_def interval_of_def **by** fastforce
next
case Including
then show ?case **using** assms **by** simp
next
case RightOf
have $\langle 1 / \text{upper } Y \leq 1 / \text{lower } Y \rangle$
by (simp add: assms upper_leq_lower_div)
then show ?case
using interval_div_constant upper_bounds lower_bounds assms
unfolding width_def interval_of_def **by** fastforce
qed

lemma abs_pos:
assumes $\langle 0 < \text{lower } Y \rangle$ and $\langle \neg 0 \in_i Y \rangle$
shows $\langle \text{abs_interval}(\text{divide } (\text{interval_of } 1) \ Y) = \text{Interval}(1 / \text{upper } Y, 1 / \text{lower } Y) \rangle$
proof –
have l: $\langle \text{lower } Y \leq \text{upper } Y \rangle$ **using** lower_le_upper **by** simp
have $\langle 0 < 1 / \text{upper } Y \rangle$

```

by (metis assms(1) l dual_order.strict_trans1 zero_less_divide_1_iff)
moreover have  $\langle 0 < 1 / \text{lower } Y \rangle$ 
by (metis assms(1) zero_less_divide_1_iff)
moreover have  $\langle 1 / \text{upper } Y \leq 1 / \text{lower } Y \rangle$ 
using assms by (simp add: frac_le)
moreover have  $\langle \text{divide } (\text{interval\_of } 1) Y = \text{Interval}(1 / \text{upper } Y, 1 / \text{lower } Y) \rangle$ 
using assms interval_div_constant[of Y 1] by simp
ultimately show ?thesis
unfolding abs_interval_def by (simp add: bounds_of_interval_eq_lower_upper)
qed

```

lemma abs_neg:

```

assumes  $\langle \text{upper } Y < 0 \rangle$  and  $\langle \neg 0 \in_i Y \rangle$ 
shows  $\langle \text{abs\_interval}(\text{divide } (\text{interval\_of } 1) Y) = \text{Interval}(1 / |\text{lower } Y|, 1 / |\text{upper } Y|) \rangle$ 
proof –
have l:  $\langle \text{lower } Y \leq \text{upper } Y \rangle$  using lower_le_upper by simp
have i0:  $\langle 1 / \text{upper } Y < 0 \rangle$  and i1:  $\langle 1 / \text{lower } Y < 0 \rangle$ 
using assms by (simp, meson assms(1) divide_less_0_1_iff lower_le_upper order_le_less_trans)
moreover have i2:  $\langle |\text{upper } Y| \leq |\text{lower } Y| \rangle$ 
using assms l by linarith
then have i3:  $\langle |1 / \text{lower } Y| \leq |1 / \text{upper } Y| \rangle$ 
using assms division_leq_neg i1
by (simp add: division_leq)
moreover have  $\langle \text{divide } (\text{interval\_of } 1) Y = \text{Interval}(1 / \text{upper } Y, 1 / \text{lower } Y) \rangle$ 
using assms interval_div_constant[of Y 1] by simp
moreover have  $\langle \text{abs\_interval}(\text{Interval}(1 / \text{upper } Y, 1 / \text{lower } Y)) = \text{Interval}(|1 / \text{lower } Y|, |1 / \text{upper } Y|) \rangle$ 
using assms i0 i1 i2 i3 unfolding abs_interval_def min_def max_def
by (simp add: bounds_of_interval_eq_lower_upper)
moreover have  $\langle \dots = \text{Interval}(1 / |\text{lower } Y|, 1 / |\text{upper } Y|) \rangle$ 
by auto[1]
ultimately show ?thesis
using assms interval_div_constant by force
qed

```

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 $?a / (o::?'a) = (o::?'a)$ 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* \Rightarrow 'a *interval*

where *inverse_interval* *a* = *mk_interval* (1 / (upper *a*), 1 / (lower *a*))

definition *divide_interval* :: 'a *interval* \Rightarrow 'a *interval* \Rightarrow 'a *interval*

where *divide_interval* *a b* = *inverse* *b* * *a*

instance ..

end

interpretation *interval_division_inverse* *divide* *inverse*

apply (*unfold_locales*)

subgoal **by** (*simp* *add*: *inverse_interval_def*)

subgoal **by** (*simp* *add*: *divide_interval_def*)

done

lemma *width_of_reciprocal_interval_bound_real*:

fixes *Y* :: *real interval*

assumes $\langle \neg 0 \in_i Y \rangle$

shows $\langle \text{interval_of}(\text{width}((\text{interval_of } 1) / Y)) \leq$

$(\text{abs_interval}((\text{interval_of } 1) / Y) * \text{abs_interval}((\text{interval_of } 1) / Y)) * \text{interval_of}(\text{width } Y) \rangle$

proof (*cases* *Y* *rule*: *interval_linorder_case_split* [*of* *o* $\lambda Y. \text{interval_of}(\text{width}((\text{interval_of } 1) / Y)) \leq$

$(\text{abs_interval}((\text{interval_of } 1) / Y)) * (\text{abs_interval}((\text{interval_of } 1) / Y)) * \text{interval_of}(\text{width } Y)]$)

case *LeftOf*

have *ao*: $\langle \text{lower } Y \leq \text{upper } Y \rangle$ **using** *lower_le_upper* **by** *simp*

have *a1*: $\langle \text{interval_of}(\text{width}((\text{interval_of } 1) / Y)) = \text{Interval}(1 / \text{lower } Y - 1 / \text{upper } Y, 1 / \text{lower } Y - 1 / \text{upper } Y) \rangle$

using *assms* *interval_of_width* **by** *blast*

have *a2*: $\langle (\text{abs_interval}((\text{interval_of } 1) / Y) * \text{abs_interval}((\text{interval_of } 1) / Y)) * \text{interval_of}(\text{width } Y) = \text{Interval}((\text{upper } Y - \text{lower } Y) * 1 / |\text{lower } Y| * 1 / |\text{lower } Y|, (\text{upper } Y - \text{lower } Y) * 1 / |\text{upper } Y| * 1 / |\text{upper } Y|) \rangle$

proof –

have *bo*: $\langle 1 / |\text{lower } Y| \leq 1 / |\text{upper } Y| \rangle$ **using** *assms* *ao* *LeftOf*

by (*smt* (*verit*, *best*) *frac_le*)

have *b1*: $\langle 0 \leq \text{upper } Y - \text{lower } Y \rangle$

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

```

moreover have b2: ⟨(abs_interval((interval_of 1) / Y) * abs_interval((interval_of 1) / Y)) * interval_of(width Y) =
Interval(1/|lower Y|, 1/|upper Y|) * Interval(1/|lower Y|, 1/|upper Y|) * Interval(upper Y - lower Y, upper Y - lower Y)⟩
using assms LeftOf abs_neg[of Y] width_expanded[of Y] by simp
moreover have b3: ⟨... = Interval(Min { (1 / |lower Y| * 1 / |lower Y|), (1 / |lower Y| * 1 / |upper Y|), (1 / |upper Y| * 1
/ |lower Y|), (1 / |upper Y| * 1 / |upper Y|) }, Max { (1 / |lower Y| * 1 / |lower Y|), (1 / |lower Y| * 1 / |upper Y|), (1 / |upper
Y| * 1 / |lower Y|), (1 / |upper Y| * 1 / |upper Y|) }) * Interval(upper Y - lower Y, upper Y - lower Y)⟩
using assms ao LeftOf interval_interval_times[of Interval(1/|lower Y|, 1/|upper Y|) Interval(1/|lower Y|, 1/|upper Y|)]
using lower_bounds upper_bounds bo by simp
have b4: ⟨Min { (1 / |lower Y| * 1 / |lower Y|), (1 / |lower Y| * 1 / |upper Y|), (1 / |upper Y| * 1 / |lower Y|), (1 / |upper Y|
* 1 / |upper Y|) } = (1 / |lower Y| * 1 / |lower Y|)⟩
using assms ao LeftOf upper_leq_lower_div
apply (simp add: min_def, safe)
apply (smt (z3) divide_divide_eq_left' divide_less_cancel divide_minus_right nonzero_minus_divide_divide)
apply argo+
by (smt (z3) ao divide_divide_eq_left' divide_le_cancel divide_nonneg_neg frac_le minus_divide_right mult commute
mult_left_mono mult_minus_left)
have b5: ⟨Max { (1 / |lower Y| * 1 / |lower Y|), (1 / |lower Y| * 1 / |upper Y|), (1 / |upper Y| * 1 / |lower Y|), (1 / |upper Y|
* 1 / |upper Y|) } = (1 / |upper Y| * 1 / |upper Y|)⟩
using assms ao LeftOf upper_leq_lower_div
apply (simp add: max_def, safe)
apply (smt (verit, del_insts) frac_le mult_mono_nonpos_nonpos mult_neg_neg)
apply argo
apply argo
apply (smt (z3) bo divide_divide_eq_left' divide_less_cancel minus_divide_right nonzero_minus_divide_divide)
apply (smt (verit, best) division_leq mult_mono_nonpos_nonpos not_real_square_gt_zero)
by argo+
moreover have ⟨(abs_interval((interval_of 1) / Y) * abs_interval((interval_of 1) / Y)) * interval_of(width Y) = Inter-
val(1 / |lower Y| * 1 / |lower Y|, 1 / |upper Y| * 1 / |upper Y|) * Interval(upper Y - lower Y, upper Y - lower Y)⟩
using b2 b3 b4 b5 by presburger
moreover have ⟨... = Interval((upper Y - lower Y) * 1 / |lower Y| * 1 / |lower Y|, (upper Y - lower Y) * 1 / |upper Y| *
1 / |upper Y|)⟩
using assms ao LeftOf b4 b5 interval_times_scalar_pos_r[of upper Y - lower Y Interval(1 / |lower Y| * 1 / |lower Y|, 1
/ |upper Y| * 1 / |upper Y|)]
unfolding interval_of_def by simp
ultimately show ?thesis by metis
qed
have a3: ⟨(upper Y - lower Y) * 1 / |lower Y| * 1 / |lower Y| ≤ 1 / lower Y - 1 / upper Y⟩
proof -
have (1 / lower Y) - (1 / upper Y) = (upper Y - lower Y) / (lower Y * upper Y)
using assms ao LeftOf
by (smt (verit) diff_divide_distrib nonzero_divide_mult_cancel_left nonzero_divide_mult_cancel_right)
moreover have ⟨(upper Y - lower Y) * 1 / |lower Y| * 1 / |lower Y| = (upper Y - lower Y) / |lower Y|^2⟩
using assms LeftOf by (simp add: power2_eq_square)
moreover have ⟨((upper Y - lower Y) / |lower Y|^2) ≤ (upper Y - lower Y) / (lower Y * upper Y)⟩
using assms ao LeftOf
by (smt (verit) frac_le minus_mult_minus mult_left_mono_neg mult_neg_neg power2_eq_square)
ultimately show ?thesis by metis
qed
have a4: ⟨1 / lower Y - 1 / upper Y ≤ (upper Y - lower Y) * 1 / |upper Y| * 1 / |upper Y|⟩
proof -
have (1 / lower Y) - (1 / upper Y) = (upper Y - lower Y) / (lower Y * upper Y)
using assms ao LeftOf
by (smt (verit) diff_divide_distrib nonzero_divide_mult_cancel_left nonzero_divide_mult_cancel_right)
moreover have ⟨(upper Y - lower Y) * 1 / |upper Y| * 1 / |upper Y| = (upper Y - lower Y) / |upper Y|^2⟩

```

```

    using assms LeftOf by (simp add: power2_eq_square)
    moreover have  $\langle (upper\ Y - lower\ Y) / (lower\ Y * upper\ Y) \leq ((upper\ Y - lower\ Y) / |upper\ Y|^2) \rangle$ 
    using assms LeftOf
    by (simp add: frac_le power2_eq_square zero_less_mult_iff)
    ultimately show ?thesis by metis
qed
have  $\langle Interval(1 / lower\ Y - 1 / upper\ Y, 1 / lower\ Y - 1 / upper\ Y) \leq Interval((upper\ Y - lower\ Y) * 1 / |lower\ Y| * 1 / |lower\ Y|, (upper\ Y - lower\ Y) * 1 / |upper\ Y| * 1 / |upper\ Y|) \rangle$ 
    using a3 a4 unfolding less_eq_interval_def by simp
then show ?case using a1 a2 a3 a4 by simp
next
case Including
then show ?case using assms by simp
next
case RightOf
have ao:  $\langle lower\ Y \leq upper\ Y \rangle$  using lower_le_upper by simp
have a1:  $\langle interval\_of(width\ ((interval\_of\ 1) / Y)) = Interval(1 / lower\ Y - 1 / upper\ Y, 1 / lower\ Y - 1 / upper\ Y) \rangle$ 
    using assms interval_of_width by blast
have a2:  $\langle (abs\_interval((interval\_of\ 1) / Y) * abs\_interval((interval\_of\ 1) / Y)) * interval\_of(width\ Y) = Interval((upper\ Y - lower\ Y) * 1 / upper\ Y * 1 / upper\ Y, (upper\ Y - lower\ Y) * 1 / lower\ Y * 1 / lower\ Y) \rangle$ 
proof -
    have bo:  $\langle 0 \leq upper\ Y - lower\ Y \rangle$ 
    using assms RightOf by simp
    moreover have b1:  $\langle (abs\_interval((interval\_of\ 1) / Y) * abs\_interval((interval\_of\ 1) / Y)) * interval\_of(width\ Y) = Interval(1 / upper\ Y, 1 / lower\ Y) * Interval(1 / upper\ Y, 1 / lower\ Y) * Interval(upper\ Y - lower\ Y, upper\ Y - lower\ Y) \rangle$ 
        using assms RightOf abs_pos[of Y] width_expanded[of Y] by simp
    moreover have b2:  $\langle \dots = Interval(\text{Min } \{ (1 / upper\ Y * 1 / upper\ Y), (1 / upper\ Y * 1 / lower\ Y), (1 / lower\ Y * 1 / upper\ Y), (1 / lower\ Y * 1 / lower\ Y) \}, \text{Max } \{ (1 / upper\ Y * 1 / upper\ Y), (1 / upper\ Y * 1 / lower\ Y), (1 / lower\ Y * 1 / upper\ Y), (1 / lower\ Y * 1 / lower\ Y) \}) * Interval(upper\ Y - lower\ Y, upper\ Y - lower\ Y) \rangle$ 
        using assms RightOf upper_leq_lower_div interval_interval_times[of Interval(1 / upper\ Y, 1 / lower\ Y) Interval(1 / upper\ Y, 1 / lower\ Y)]
        by fastforce
    have b3:  $\langle \text{Min } \{ (1 / upper\ Y * 1 / upper\ Y), (1 / upper\ Y * 1 / lower\ Y), (1 / lower\ Y * 1 / upper\ Y), (1 / lower\ Y * 1 / lower\ Y) \} = (1 / upper\ Y * 1 / upper\ Y) \rangle$ 
        using assms ao RightOf upper_leq_lower_div
        apply (simp add: min_def, safe)
        apply (smt (verit, ccfv_SIG) frac_le mult_less_cancel_left_pos zero_less_mult_iff)
        apply argo+
        by (simp add: frac_le)
    have b4:  $\langle \text{Max } \{ (1 / upper\ Y * 1 / upper\ Y), (1 / upper\ Y * 1 / lower\ Y), (1 / lower\ Y * 1 / upper\ Y), (1 / lower\ Y * 1 / lower\ Y) \} = (1 / lower\ Y * 1 / lower\ Y) \rangle$ 
        using assms ao RightOf upper_leq_lower_div
        apply (simp add: max_def, safe)
    subgoal
        using frac_le le_numeral_extra(4) less_numeral_extra(3) linordered_nonzero_semiring_class.zero_le_one
        mult_mono' not_real_square_gt_zero order_less_imp_le by metis
    apply argo+
    apply (simp add: frac_le)
    subgoal
        using frac_le linordered_nonzero_semiring_class.zero_le_one mult_mono' mult_pos_pos
        order.refl order_less_imp_le by meson
    by argo+
    moreover have  $\langle (abs\_interval((interval\_of\ 1) / Y) * abs\_interval((interval\_of\ 1) / Y)) * interval\_of(width\ Y) = Interval(1 / upper\ Y * 1 / upper\ Y, 1 / lower\ Y * 1 / lower\ Y) * Interval(upper\ Y - lower\ Y, upper\ Y - lower\ Y) \rangle$ 

```

```

using b1 b2 b3 b4 by presburger
moreover have <... = Interval((upper Y - lower Y) * 1 / upper Y * 1 / upper Y, (upper Y - lower Y) * 1 / lower Y * 1 / lower Y)>
using assms RightOf bo interval_times_scalar_pos_r[of upper Y - lower Y Interval(1 / upper Y * 1 / upper Y, 1 / lower Y * 1 / lower Y)]
unfolding interval_of_def
by (smt (verit, del_insts) divide_cancel_left divide_nonneg_nonneg frac_le interval_times_scalar_pos_r lower_bounds nonzero_divide_mult_cancel_right times_divide_eq_right upper_bounds width_def width_expanded)
ultimately show ?thesis by simp
qed
have a3: <1 / upper Y * 1 / upper Y * (upper Y - lower Y) ≤ 1 / lower Y - 1 / upper Y>
proof -
have bo: <0 < upper Y> using assms ao RightOf by argo
then have (1 / lower Y) - (1 / upper Y) = (upper Y - lower Y) / (lower Y * upper Y)
using assms RightOf bo by (simp add: diff_divide_distrib)
then have b1: <((1 / upper Y) * (1 / upper Y) * (upper Y - lower Y) * (lower Y * upper Y) ≤ (upper Y - lower Y)) = ((1 / upper Y) * (1 / upper Y) * (upper Y - lower Y) ≤ 1 / lower Y - 1 / upper Y)>
using assms RightOf bo by (simp add: pos_le_divide_eq)
then have b2: <((upper Y - lower Y) * lower Y / upper Y) ≤ upper Y - lower Y>
using assms ao RightOf bo by (smt (verit) mult_left_less_imp_less pos_less_divide_eq)
moreover have <(upper Y - lower Y) * lower Y ≤ (upper Y - lower Y) * upper Y>
using assms RightOf bo calculation pos_divide_le_eq by blast
show ?thesis using assms bo b1 b2 by force
qed
have a4: <1 / lower Y - 1 / upper Y ≤ 1 / lower Y * 1 / lower Y * (upper Y - lower Y)>
proof -
have bo: <0 < upper Y> using assms ao RightOf by argo
then have (1 / lower Y) - (1 / upper Y) = (upper Y - lower Y) / (lower Y * upper Y)
using assms RightOf bo by (simp add: diff_divide_distrib)
then have b1: <(upper Y - lower Y ≤ (1 / lower Y) * (1 / lower Y) * (upper Y - lower Y) * (lower Y * upper Y)) = (1 / lower Y - 1 / upper Y ≤ ((1 / lower Y) * (1 / lower Y) * (upper Y - lower Y)))>
using assms RightOf bo by (simp add: pos_divide_le_eq)
then have b2: <upper Y - lower Y ≤ ((upper Y - lower Y) * upper Y / lower Y)>
using assms ao RightOf bo by (smt (verit) division_leq mult_pos_pos nonzero_mult_div_cancel_right)
moreover have <(upper Y - lower Y) * lower Y ≤ (upper Y - lower Y) * upper Y>
using calculation pos_divide_le_eq by (simp add: RightOf pos_le_divide_eq)
show ?thesis using assms bo b1 b2 RightOf by simp
qed
have <Interval(1 / lower Y - 1 / upper Y, 1 / lower Y - 1 / upper Y) ≤ Interval((upper Y - lower Y) * 1 / upper Y * 1 / upper Y, (upper Y - lower Y) * 1 / lower Y * 1 / lower Y)>
using a3 a4 unfolding less_eq_interval_def by simp
then show ?case using a1 a2 a3 a4 by simp
qed
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  $\langle \exists b. \text{linear } (\lambda x. f x - b) \rangle$ 

lemma affine_fun_alt:
 $\langle \text{affine\_fun } f = (\exists c g. (f = (\lambda x. g x + c)) \wedge \text{linear } g) \rangle$ 
  unfolding affine_fun_def
  by (safe, force, simp add: Real_Vector_Spaces.linear_iff)

lemma affine_fun_real_linfun:
 $\langle \text{affine\_fun } (f :: \text{real} \Rightarrow \text{real}) = (\exists a b. f = (\lambda x. a * x + b)) \rangle$ 
  by (simp add: affine_fun_alt, metis real_linearD linear_times)

lemma linear_is_affine_fun:  $\langle \text{linear } f \Longrightarrow \text{affine\_fun } f \rangle$ 
  by (standard, simp add: Real_Vector_Spaces.linear_iff)

lemma affine_zero_is_linear: assumes  $\langle \text{affine\_fun } f \rangle$  and  $\langle f 0 = 0 \rangle$  shows  $\langle \text{linear } f \rangle$ 
  apply (insert assms)
  unfolding affine_fun_def
  using linear_o by fastforce

lemma affine_add:
  assumes  $\langle \text{affine\_fun } f \rangle$  and  $\langle \text{affine\_fun } g \rangle$ 
  shows  $\langle \text{affine\_fun } (\lambda x. f x + g x) \rangle$ 
  using assms linear_compose_add
  by (auto simp add: affine_fun_alt, force)

lemma scaleR:
  assumes  $\langle \text{affine\_fun } f \rangle$  shows  $\langle \text{affine\_fun } (\lambda x. a *_R (f x)) \rangle$ 
  using assms
  apply (simp add: affine_fun_alt)
  by (metis linear_compose_scale_right scaleR_right_distrib)

lemma real_affine_funD:
  fixes  $f :: \text{real} \Rightarrow \text{real}$ 
  assumes  $\text{affine\_fun } f$  obtains  $c b$  where  $f = (\lambda x. c * x + b)$ 
  using assms
```

```

apply(simp add: affine_fun_alt)
by (metis real_linearD)

```

5.2 Common Linear Polynomial Functions

```

lemma affine_fun_const[simp]: <affine_fun (λ x. c)>

```

```

apply(simp add: affine_fun_alt)
using module_hom_zero by force

```

```

lemma affine_fun_id[simp]: <affine_fun (λ x. x)>

```

```

by (simp add: linear_is_affine_fun module_hom_ident)

```

```

lemma affine_fun_mult[simp]: <affine_fun (λ x. (c::'a::real_algebra) * x)>

```

```

by(simp add: linear_is_affine_fun)

```

```

lemma affine_fun_scaled[simp]: <affine_fun (λ x. x / c)>

```

```

for c :: 'a::real_normed_field

```

```

using bounded_linear_divide[of c] linear_is_affine_fun[of (λ x. x / c)] bounded_linear.linear
by blast

```

```

lemma affine_fun_add[simp]: <affine_fun (λ x. x + c)>

```

```

apply(simp add: affine_fun_alt)
using module_hom_ident by force

```

```

lemma affine_fun_diff[simp]: <affine_fun (λ x. x - c)>

```

```

apply(simp add: affine_fun_alt)
using module_hom_ident by force

```

```

lemma affine_fun_triv[simp]: <affine_fun (λ x. a *R x + c)>

```

```

apply(simp add: affine_fun_alt)
by fastforce

```

```

lemma affine_fun_add_const[simp]: assumes <affine_fun f> shows <affine_fun (λ x. (f x) + c)>

```

```

using assms apply(simp add:affine_fun_alt)
by (metis add.commute add.left_commute)

```

```

lemma affine_fun_diff_const[simp]: assumes <affine_fun f> shows <affine_fun (λ x. (f x) - c)>

```

```

using assms apply(simp add:affine_fun_alt) by force

```

```

lemma affine_fun_comp[simp]: assumes <affine_fun (f)>

```

```

and <affine_fun (g)> shows <affine_fun (f ∘ g)>

```

```

using assms
unfolding affine_fun_alt
apply(simp add:o_def)
using linear_add linear_compose[simplified o_def] add.assoc
by metis

```

```

lemma affine_fun_linear[simp]: assumes <affine_fun f> shows <affine_fun (λ x. a *R (f x) + c)>

```

```

by(rule affine_fun_comp[of λ x. a *R x + c f, simplified o_def], simp_all add: assms)

```

5.3 Linear Polynomial Functions and Orderings

```

lemma affine_fun_leq_pos:

```

```

assumes <affine_fun (f::real ⇒ real)> and <affine_fun g>
and <x∈{0..u}> and <(f o ≤ g o)> and <(f u ≤ g u)>
shows <f x ≤ g x>
using assms
apply(auto simp add: linear_o affine_fun_real_linfun)[1]
by (smt (verit) left_diff_distrib mult_left_mono_neg mult_right_mono)

```

```

lemma affine_fun_leq_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>
using assms
apply(auto simp add: linear_o affine_fun_real_linfun)[1]
using Groups.mult_ac(2) left_diff_distrib mult_right_mono
by (smt (verit, ccfv_SIG))

```

```

lemma affine_fun_leq:
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)>
using assms affine_fun_leq_neg[of f g x l] affine_fun_leq_pos[of f g x u]
apply(simp add: linear_o affine_fun_real_linfun)
by (smt (verit, ccfv_SIG) left_diff_distrib mult_left_mono_neg)

```

```

lemma affine_fun_le_pos:
assumes <affine_fun (f::real ⇒ real)> and <affine_fun g>
and <x∈{0..u}> and <(f o < g o)> and <(f u < g u)>
shows <f x < g (x::real)>
using assms
apply(auto simp add: linear_o affine_fun_real_linfun)[1]
using Groups.mult_ac(2) left_diff_distrib mult_right_mono
by (smt (verit, best))

```

```

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)>
using assms
apply(auto simp add: linear_o affine_fun_real_linfun)[1]
using Groups.mult_ac(2) left_diff_distrib mult_right_mono
by (smt (verit, ccfv_SIG))

```

```

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)>
using assms affine_fun_le_neg[of f g x l] affine_fun_le_pos[of f g x u]
apply(simp add: linear_o affine_fun_real_linfun)
by (smt (verit, ccfv_SIG) left_diff_distrib mult_left_mono_neg)

```

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 :: $\langle ('a::preorder\ interval \Rightarrow 'b::preorder\ interval) \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool \rangle$
(*infix* (*is'_interval'_extension'_of*) 50)

where

$\langle ((F\ is_interval_extension_of\ f)) = (\forall x. (F\ (interval_of\ x)) = interval_of\ (f\ x)) \rangle$

definition *extend_natural* $f = (\lambda X. mk_interval\ (f\ (lower\ X), f\ (upper\ X)))$

lemma *interval_extension_comp*:

assumes *interval_ext_F*: $\langle F\ is_interval_extension_of\ f \rangle$

and *interval_ext_G*: $\langle G\ is_interval_extension_of\ g \rangle$

shows $\langle (F\ o\ G)\ is_interval_extension_of\ (f\ o\ g) \rangle$

using *assms* **unfolding** *is_interval_extension_of_def*

by *simp*

lemma *interval_extension_const*: $\langle (\lambda x. interval_of\ c)\ is_interval_extension_of\ (\lambda x. c) \rangle$

unfolding *is_interval_extension_of_def*

by (*simp* *add: interval_eq_iff*)

lemma *interval_extension_id*: $\langle (\lambda x. x)\ is_interval_extension_of\ (\lambda x. x) \rangle$

unfolding *is_interval_extension_of_def*

by (*simp* *add: interval_eq_iff*)

6.1.2 A Stronger Definition of Interval Extension

definition

is_natural_interval_extension_of :: $\langle ('a::linorder\ interval \Rightarrow 'b::linorder\ interval) \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool \rangle$
(*infix* (*is'_natural'_interval'_extension'_of*) 50)

where

$\langle ((F\ is_natural_interval_extension_of\ f)) = (\forall l\ u. (F\ (mk_interval\ (l,u))) = mk_interval\ (f\ l, f\ u)) \rangle$

lemma $\langle (extend_natural\ f)\ is_interval_extension_of\ f \rangle$

unfolding *is_interval_extension_of_def* *extend_natural_def*

by (*auto* *simp* *add: mk_interval'_interval_of_def*)

lemma $\langle \text{extend_natural } f \rangle \text{ is_natural_interval_extension_of } f$
unfolding $\text{is_natural_interval_extension_of_def extend_natural_def}$
by $(\text{auto simp add: mk_interval' interval_of_def})$

lemma $\text{natural_interval_extension_implies_interval_extension}$:
assumes $\langle F \text{ is_natural_interval_extension_of } f \rangle$ **shows** $\langle F \text{ is_interval_extension_of } f \rangle$
using $\text{assms unfolding is_natural_interval_extension_of_def is_interval_extension_of_def}$
 $\text{mk_interval_def interval_of_def}$
by $(\text{auto split:if_splits})$

lemma $\text{const_add_is_natural_interval_extensions}$:
 $\langle (\lambda x. (\text{interval_of } c) + x) \text{ is_natural_interval_extension_of } (\lambda x. c + x) \rangle$
unfolding $\text{is_natural_interval_extension_of_def mk_interval_def}$
by $(\text{simp add: add_mono_thms_linordered_semiring(1) antisym interval_eq_iff add_mono}$
 $\text{split:if_splits})$

lemma $\text{const_times_is_natural_interval_extensions}$:
 $\langle (\lambda x. (\text{interval_of } c) * x) \text{ is_natural_interval_extension_of } (\lambda x. c * x) \rangle$
unfolding $\text{is_natural_interval_extension_of_def mk_interval_def}$
unfolding $\text{times_interval_def Let_def}$
by $(\text{simp add: interval_eq_iff Interval_inverse interval_of.rep_eq add_mono}$
 $\text{split:if_splits})$

lemma $\text{const_is_interval_extension}$: $\langle F \text{ is_natural_interval_extension_of } (\lambda x. b) \implies F = (\lambda x. (\text{interval_of } b)) \rangle$
unfolding $\text{is_natural_interval_extension_of_def}$
apply $(\text{auto simp add:mk_interval' interval_of_def split:if_splits})[1]$
by $(\text{metis bounds_of_interval_eq_lower_upper bounds_of_interval_inverse lower_le_upper})$

lemma $\text{id_is_interval_extension}$: $\langle F \text{ is_natural_interval_extension_of } (\lambda x. x) \implies F = (\lambda x. x) \rangle$
unfolding $\text{is_natural_interval_extension_of_def}$
apply $(\text{auto simp add:mk_interval' interval_of_def split:if_splits})[1]$
by $(\text{metis bounds_of_interval_eq_lower_upper bounds_of_interval_inverse lower_le_upper})$

lemma $\text{extend_natural_is_interval_extension}$:
 $\langle \text{extend_natural } f \rangle \text{ is_natural_interval_extension_of } f$
unfolding $\text{extend_natural_def is_natural_interval_extension_of_def mk_interval'_def}$
by $(\text{smt (z3) case_prod_conv mk_interval_def lower_bounds nle_le upper_bounds})$

lemma $\text{is_natural_interval_extension_implies_bounds}$:
fixes $F :: \langle \text{real interval} \Rightarrow \text{real interval} \rangle$
assumes $\langle F \text{ is_natural_interval_extension_of } f \rangle$ **and** $\langle F x \leq F x' \rangle$
shows
 $\langle (f (\text{lower } x')) \leq (f (\text{lower } x)) \rangle \vee \langle (f (\text{upper } x')) \leq (f (\text{upper } x)) \rangle$
by $(\text{smt (verit) assms interval_eq1 is_natural_interval_extension_of_def less_eq_interval_def}$
 $\text{lower_le_upper mk_interval_lower mk_interval_upper})$

lemma $\text{interval_extension_lower}$:
 $\langle (F \text{ is_interval_extension_of } f) \implies \text{lower } (F (\text{interval_of } x)) = (f x) \rangle$
unfolding $\text{is_interval_extension_of_def}$ **by** simp

lemma $\text{interval_extension_upper}$:
 $\langle (F \text{ is_interval_extension_of } f) \implies \text{upper } (F (\text{interval_of } x)) = (f x) \rangle$
unfolding $\text{is_interval_extension_of_def}$ **by** simp

lemma *is_interval_extension_eq_upper_and_lower*:
 $\langle (F \text{ is_interval_extension_of } f) \Rightarrow (\forall x. \text{lower } (F (\text{interval_of } x)) = (f x) \wedge \text{upper } (F (\text{interval_of } x)) = (f x)) \rangle$
unfolding *is_interval_extension_of_def*
by (*simp add: interval_eq_iff*)

lemma *interval_extension_lower_simp*[*simp*]:
assumes $\langle F \text{ is_interval_extension_of } f \rangle$ **and** $\langle X = \text{Interval}(x, x) \rangle$
shows $\langle \text{lower } (F X) = f x \rangle$
by (*metis assms interval_extension_lower interval_of.abs_eq*)

lemma *interval_extension_upper_simp*[*simp*]:
assumes $\langle F \text{ is_interval_extension_of } f \rangle$ **and** $\langle X = \text{Interval}(x, x) \rangle$
shows $\langle \text{upper } (F X) = f x \rangle$
by (*metis assms interval_extension_upper interval_of.abs_eq*)

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 :: $\langle ('a::\text{preorder interval} \Rightarrow 'b::\text{preorder interval}) \Rightarrow \text{bool} \rangle$
where
 $\langle \text{inclusion_isotonic } f = (\forall x x'. x \leq x' \longrightarrow (f x) \leq (f x')) \rangle$

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 $\langle \text{affine_fun } (f::\text{real} \Rightarrow \text{real}) \rangle$
shows $\langle \text{inclusion_isotonic } (\text{extend_natural } f) \rangle$
using *assms*
unfolding *inclusion_isotonic_def extend_natural_def Interval.less_eq_interval_def affine_fun_real_linfun*
by (*auto, (metis lower_le_upper mult_le_cancel_left nle_le) +*)

definition
inclusion_isotonic_on :: $\langle ('a \text{ interval} \Rightarrow \text{bool}) \Rightarrow ('a::\text{preorder interval} \Rightarrow 'b::\text{preorder interval}) \Rightarrow \text{bool} \rangle$
where
 $\langle \text{inclusion_isotonic_on } P f = (\forall x x'. P x \wedge P x' \wedge x \leq x' \longrightarrow (f x) \leq (f x')) \rangle$

lemma *inclusion_isotonic_eq*: $\langle \text{inclusion_isotonic_on } (\lambda x. \text{True}) = \text{inclusion_isotonic} \rangle$
unfolding *inclusion_isotonic_on_def inclusion_isotonic_def*
by *simp*

definition *inclusion_isotonic_binary* :: $\langle ('a::\text{preorder interval} \Rightarrow 'a \text{ interval} \Rightarrow 'b::\text{preorder interval}) \Rightarrow \text{bool} \rangle$
where
 $\langle \text{inclusion_isotonic_binary } f = (\forall x x' y y'. x \leq x' \wedge y \leq y' \longrightarrow (f x y) \leq (f x' y')) \rangle$

definition *inclusion_isotonic_binary_on* :: $\langle ('a \text{ interval} \Rightarrow \text{bool}) \Rightarrow ('a::\text{preorder interval} \Rightarrow 'a \text{ interval} \Rightarrow 'b::\text{preorder interval}) \Rightarrow \text{bool} \rangle$
where
 $\langle \text{inclusion_isotonic_binary_on } P f = (\forall x x' y y'. P x \wedge P x' \wedge P y \wedge P y' \wedge x \leq x' \wedge y \leq y' \longrightarrow (f x y) \leq (f x' y')) \rangle$

lemma *inclusion_isotonic_binary_eq*: $\langle \text{inclusion_isotonic_binary_on } (\lambda x . \text{True}) = \text{inclusion_isotonic_binary} \rangle$
unfolding *inclusion_isotonic_binary_on_def inclusion_isotonic_binary_def*
by *simp*

definition *inclusion_isotonicM_n* :: $\langle \text{nat} \Rightarrow ('a::\text{linorder } \text{interval list} \Rightarrow 'a::\text{linorder } \text{interval}) \Rightarrow \text{bool} \rangle$ **where**
 $\langle \text{inclusion_isotonicM_n } n f = (\forall \text{ is js. } (\text{length is} = n \wedge \text{length js} = n \wedge (\text{is} \leq_I \text{js})) \longrightarrow f \text{ is} \leq f \text{ js}) \rangle$

definition *inclusion_isotonicM_n_on* :: $\langle ('a \text{ interval} \Rightarrow \text{bool}) \Rightarrow \text{nat} \Rightarrow ('a::\text{linorder } \text{interval list} \Rightarrow 'a::\text{linorder } \text{interval}) \Rightarrow \text{bool} \rangle$ **where**
 $\langle \text{inclusion_isotonicM_n_on } P n f = (\forall \text{ is js. } (\forall i \in \text{set is. } P i) \wedge (\forall j \in \text{set js. } P j) \wedge (\text{length is} = n \wedge \text{length js} = n \wedge (\text{is} \leq_I \text{js})) \longrightarrow f \text{ is} \leq f \text{ js}) \rangle$

lemma *inclusion_isotonicM_n_eq*: $\langle \text{inclusion_isotonicM_n_on } (\lambda x . \text{True}) = \text{inclusion_isotonicM_n} \rangle$
unfolding *inclusion_isotonicM_n_on_def inclusion_isotonicM_n_def*
by *simp*

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* :: $\langle ('a::\text{linorder}) \text{ interval list} \Rightarrow 'a \text{ interval list} \rangle$
assumes *inclusion_isotonicM* :
 $\langle (\forall \text{ is js. } (\text{length is} = n) \wedge (\text{length js} = n) \wedge (\text{is} \leq_I \text{js}) \longrightarrow f \text{ is} \leq_I f \text{ js}) \rangle$
begin
end

locale *inclusion_isotonicM_on* =
fixes *P* :: $\langle 'a::\text{linorder } \text{interval} \Rightarrow \text{bool} \rangle$
and *n* :: *nat*
and *f* :: $\langle ('a::\text{linorder}) \text{ interval list} \Rightarrow 'a \text{ interval list} \rangle$
assumes *inclusion_isotonicM* :
 $\langle (\forall \text{ is js. } (\forall i \in \text{set is. } P i) \wedge (\forall j \in \text{set js. } P j) \wedge (\text{length is} = n) \wedge (\text{length js} = n) \wedge (\text{is} \leq_I \text{js}) \longrightarrow f \text{ is} \leq_I f \text{ js}) \rangle$
begin
end

lemma *inclusion_isotonicM_on_eq*: $\langle \text{inclusion_isotonicM_on } (\lambda x . \text{True}) = \text{inclusion_isotonicM} \rangle$
unfolding *inclusion_isotonicM_on_def inclusion_isotonicM_def*
by *simp*

lemma *inclusion_isotonic_conv*:
 $\langle \text{inclusion_isotonic } g = \text{inclusion_isotonicM } 1 (\lambda \text{xs} . \text{case } \text{xs} \text{ of } [x] \Rightarrow [g x]) \rangle$
unfolding *inclusion_isotonic_def inclusion_isotonicM_def*
by *(auto simp add: le_interval_single split:list.split)*

lemma *inclusion_isotonicM_n_conv1*:
 $\langle \text{inclusion_isotonicM_n } n f = \text{inclusion_isotonicM } n (\lambda \text{xs} . [f \text{xs}]) \rangle$
unfolding *inclusion_isotonicM_n_def inclusion_isotonicM_def le_interval_list_def*
by *(auto)*

lemma *inclusion_isotonicM_conv2*:
assumes $\langle \text{inclusion_isotonicM } n f \rangle$
and $\langle \forall \text{xs. } (\text{length xs} = n) \longrightarrow N = (\text{length } (f \text{xs})) \rangle$
shows $\langle \text{inclusion_isotonicM } n (\lambda \text{xs} . \text{if } n' < N \text{ then } [(f \text{xs})!n'] \text{ else } []) \rangle$


```

using assms unfolding inclusion_isotonicM_def
apply(simp split:if_splits, safe)
apply(subst le_interval_single[symmetric])
apply(subst le_interval_list_all)
subgoal by blast
subgoal by blast
subgoal by(rule TrueI)
done

```

```

lemma inclusion_isotonicM_n_on_conv1:
  <inclusion_isotonicM_n_on P n f = inclusion_isotonicM_on P n (λ xs. [f xs])>
  unfolding inclusion_isotonicM_n_on_def inclusion_isotonicM_on_def le_interval_list_def
  by(auto)

```

```

lemma inclusion_isotonicM_on_conv2:
  assumes <inclusion_isotonicM_on P n f>
  and < $\forall$  xs. (length xs = n)  $\longrightarrow$  N = (length (f xs))>
  shows <inclusion_isotonicM_on P n (λ xs. if n' < N then [(f xs)!n'] else [])>
  using assms unfolding inclusion_isotonicM_on_def
  apply(simp split:if_splits, safe)
  apply(subst le_interval_single[symmetric])
  apply(subst le_interval_list_all)
  subgoal by blast
  subgoal by blast
  subgoal by(rule TrueI)
done

```

```

lemma inclusion_isotonic_binary_conv1:
  <inclusion_isotonic_binary f = inclusion_isotonicM_n 2 (λ xs . case xs of [x,y]  $\Rightarrow$  f x y)>
  unfolding inclusion_isotonic_binary_def inclusion_isotonicM_n_def le_interval_list_def
  by(auto simp add: le_interval_list_def split:list.split)

```

```

lemma inclusion_isotonic_binary_conv2:
  <inclusion_isotonic_binary f = inclusion_isotonicM 2 (λ xs . case xs of [x,y]  $\Rightarrow$  [f x y])>
  apply(simp add: inclusion_isotonic_binary_conv1)
  apply(simp add: inclusion_isotonicM_n_conv1)
  unfolding inclusion_isotonicM_def
  by(auto split:list.split)

```

```

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]  $\Rightarrow$  f x y)>
  unfolding inclusion_isotonic_binary_on_def inclusion_isotonicM_n_on_def le_interval_list_def
  by(auto simp add: le_interval_list_def split:list.split)

```

```

lemma inclusion_isotonic_binary_on_conv2:
  <inclusion_isotonic_binary_on P f = inclusion_isotonicM_on P 2 (λ xs . case xs of [x,y]  $\Rightarrow$  [f x y])>
  apply(simp add: inclusion_isotonic_binary_on_conv1)
  apply(simp add: inclusion_isotonicM_n_on_conv1)
  unfolding inclusion_isotonicM_on_def
  by(auto split:list.split)

```

6.2.1 Compositionality of Interval Inclusion Isotonicity

```

lemma inclusion_isotonicM_comp:

```

assumes $\langle \text{inclusion_isotonicM } n \ f \rangle$
and $\langle \text{inclusion_isotonicM } m \ g \rangle$
and $\langle \forall xs. \text{length } xs = n \longrightarrow \text{length } (f \ xs) = m \rangle$
shows $\langle \text{inclusion_isotonicM } n \ (g \ o \ f) \rangle$
using *assms* **unfolding** *inclusion_isotonicM_def*
by(*simp*)

lemma *inclusion_isotonicM_on_comp*:
assumes $\langle \text{inclusion_isotonicM_on } P \ n \ f \rangle$
and $\langle \text{inclusion_isotonicM_on } Q \ m \ g \rangle$
and $\langle \forall xs. \text{length } xs = n \longrightarrow \text{length } (f \ xs) = m \rangle$
and $\langle \forall is. (\forall i \in \text{set } is. P \ i) \longrightarrow (\forall x \in \text{set } (f \ is). Q \ x) \rangle$
shows $\langle \text{inclusion_isotonicM_on } P \ n \ (g \ o \ f) \rangle$
using *assms* **unfolding** *inclusion_isotonicM_on_def* *o_def*
by(*auto*)

lemma *inclusion_isotonic_comp*:
assumes $\langle \text{inclusion_isotonic } f \rangle$
and $\langle \text{inclusion_isotonic } g \rangle$
shows $\langle \text{inclusion_isotonic } (g \ o \ f) \rangle$
using *assms* **unfolding** *inclusion_isotonic_def*
by(*simp*)

lemma *inclusion_isotonic_on_comp*:
assumes $\langle \text{inclusion_isotonic_on } P \ f \rangle$
and $\langle \text{inclusion_isotonic_on } Q \ g \rangle$
and $\langle \forall x. P \ x \longrightarrow Q \ (f \ x) \rangle$
shows $\langle \text{inclusion_isotonic_on } P \ (g \ o \ f) \rangle$
using *assms* **unfolding** *inclusion_isotonic_on_def*
by(*simp*)

6.2.2 Interval Inclusion Isotonicity of the Core Operator

Unary Minus (Negation)

lemma *inclusion_isotonicM_uminus*[*simp*]: $\langle \text{inclusion_isotonic } \text{uminus} \rangle$
by (*simp* *add*: *inclusion_isotonic_def* *less_eq_interval_def*)

Addition

lemma *inclusion_isotonicM_plus*[*simp*]: $\langle \text{inclusion_isotonic_binary } (+) \rangle$
by (*simp* *add*: *inclusion_isotonic_binary_def* *less_eq_interval_def* *add_mono*)

Substraction

lemma *inclusion_isotonicM_minus*[*simp*]: $\langle \text{inclusion_isotonic_binary } (-) \rangle$
by (*simp* *add*: *inclusion_isotonic_binary_def* *less_eq_interval_def* *diff_mono*)

Multiplication

lemma *inclusion_isotonicM_times*[*simp*]:
 $\langle \text{inclusion_isotonic_binary } (\lambda x \ y. (x :: 'a :: \{\text{linordered_ring, real_normed_algebra, linear_continuum_topology}\}) \text{interval}) * y) \rangle$
unfolding *inclusion_isotonic_binary_def* **using** *set_of_mul_inc* *interval_set_leq_eq*
by *metis*

6.2.3 Interval Inclusion Isotonicity of Various Functions

lemma *inclusion_isotonicM_n_empty*[simp]: $\langle \text{inclusion_isotonicM } n \ (\lambda \text{ xs. } []) \rangle$
unfolding *inclusion_isotonicM_def* **by**(simp)

lemma *inclusion_isotonic_id*[simp]: $\langle \text{inclusion_isotonic } \text{id} \rangle$
unfolding *inclusion_isotonic_def le_interval_list_def*
by(simp)

lemma *inclusion_isotonicM_id*[simp]: $\langle \text{inclusion_isotonicM } n \ \text{id} \rangle$
unfolding *inclusion_isotonicM_def le_interval_list_def*
by(simp)

lemma *inclusion_isotonicM_hd*[simp]:
assumes $\langle 0 < n \rangle$
shows $\langle \text{inclusion_isotonicM_n } n \ \text{hd} \rangle$
unfolding *inclusion_isotonicM_n_def le_interval_list_def*
proof(*insert assms, induction n rule:nat_induct_non_zero*)
case 1
then show ?case
by (*auto, metis (no_types, lifting) Nil_eq_zip_iff case_proxD foldl_conj_set_True hd_zip insert_iff length_o_conv list.map_disc_iff list.map_sel(1) list.set(2) list.set_sel(1) nat.distinct(1)*)
next
case (*Suc n*)
then show ?case
by (*auto, metis (no_types, lifting) Nil_eq_zip_iff Zero_not_Suc case_proxD foldl_conj_set_True hd_zip insert_iff length_o_conv list.map_disc_iff list.map_sel(1) list.set(2) list.set_sel(1)*)
qed

lemma *inclusion_isotonic_add_const1*[simp]:
 $\langle \text{inclusion_isotonic } (\lambda x. x + c) \rangle$
unfolding *inclusion_isotonic_def*
by (*simp add: add_mono_thms_linordered_semiring(3) less_eq_interval_def*)

lemma *inclusion_isotonicM_1_add_const2*[simp]:
 $\langle \text{inclusion_isotonic } (\lambda x. c + x) \rangle$
unfolding *inclusion_isotonic_def*
by (*simp add: add_mono_thms_linordered_semiring(2) less_eq_interval_def*)

lemma *inclusion_isotonic_times_right*[simp]:
 $\langle \text{inclusion_isotonic } (\lambda x. C * (x :: 'a :: \text{linordered_ring interval})) \rangle$
unfolding *inclusion_isotonic_def*
using *times_interval_right* **by** *auto*

lemma *inclusion_isotonic_times_left*[simp]:
 $\langle \text{inclusion_isotonic } (\lambda x. (x :: 'a :: \{\text{real_normed_algebra, linordered_ring, linear_continuum_topology}\} \text{interval}) * C) \rangle$
unfolding *inclusion_isotonic_def*
using *times_interval_left* **by** *auto*

lemma *map_inclusion_isotonicM*:
assumes $\langle \text{inclusion_isotonic } f \rangle$
shows $\langle \text{inclusion_isotonicM } n \ (\text{map } f) \rangle$
using *assms map_set map_pair_set_left map_pair_set_right map_pair_set*
unfolding *inclusion_isotonicM_def le_interval_list_def inclusion_isotonic_def*

by(simp add: foldl_conj_set_True map_pair_f_all, blast)

lemma inclusion_isotonicM_fun_plus:

assumes $\langle \text{inclusion_isotonic } f \rangle$ **and** $\langle \text{inclusion_isotonic } g \rangle$

shows $\langle \text{inclusion_isotonic } (\lambda x. f x + g x) \rangle$

using assms **unfolding** inclusion_isotonic_def

by (simp add: add_mono_thms_linordered_semiring(1) less_eq_interval_def)

lemma inclusion_isotonic_plus_const:

assumes $\langle \text{inclusion_isotonic } f \rangle$ **and** $\langle \text{inclusion_isotonic } g \rangle$

shows $\langle \text{inclusion_isotonic } (\lambda x. g x + c) \rangle$

using assms **unfolding** inclusion_isotonic_def

by (simp add: add_mono_thms_linordered_semiring(1) less_eq_interval_def)

lemma inclusion_isotonic_times_const_real:

assumes $\langle \text{inclusion_isotonic } f \rangle$

shows $\langle \text{inclusion_isotonic } (\lambda x. (c::\text{real interval}) * (f x)) \rangle$

using inclusion_isotonic_comp assms

unfolding inclusion_isotonic_def

by (simp add: times_interval_right)

lemma intervall_le_foldr:

assumes $\langle \text{inclusion_isotonic_binary } f \rangle$

shows $\langle \text{length } js = \text{length } is \implies is \leq_I js \implies (\text{foldr } f \text{ is } e) \leq (\text{foldr } f \text{ js } e) \rangle$

proof (induction rule:list_induct2)

case Nil

then show ?case

by (simp add: less_eq_interval_def)

next

case (Cons x xs y ys)

then show ?case

unfolding le_interval_list_def

by (simp, metis (no_types, lifting) assms foldl_conj_True inclusion_isotonic_binary_def list_all_simps(1))

qed

lemma intervall_le_foldr_map:

assumes $\langle \text{inclusion_isotonic_binary } f \rangle$

and $\langle \text{inclusion_isotonic } g \rangle$

shows $\langle \text{length } js = \text{length } is \implies is \leq_I js \implies (\text{foldr } f (\text{map } g \text{ is}) e) \leq (\text{foldr } f (\text{map } g \text{ js}) e) \rangle$

proof (induction rule:list_induct2)

case Nil

then show ?case

by (simp add: less_eq_interval_def)

next

case (Cons x xs y ys)

then show ?case

using assms **unfolding** inclusion_isotonicM_n_def le_interval_list_def

by(simp add: foldl_conj_True inclusion_isotonic_def inclusion_isotonic_binary_def)

qed

lemma intervall_le_foldr_e:

assumes $\langle \text{inclusion_isotonic_binary } f \rangle$

and $\langle is \leq js \rangle$

```

  shows <(foldr f xs is) ≤ (foldr f xs js)>
proof (induction xs)
  case Nil
  then show ?case using assms by(simp)
next
  case (Cons x)
  then show ?case
    using assms unfolding le_interval_list_def inclusion_isotonic_binary_def
    by (simp add: less_eq_interval_def)
qed

```

```

lemma foldr_inclusion_isotonicM_e:
  assumes <inclusion_isotonic_binary f>
  shows <∀ x. inclusion_isotonic (foldr f x)>
  using assms
  unfolding inclusion_isotonic_def
  by(simp add: intervall_le_foldr_e)

```

```

lemma foldr_inclusion_isotonicM:
  assumes <inclusion_isotonic_binary f>
  shows <inclusion_isotonicM_n n (λ x. foldr f x e)>
  using assms
  unfolding inclusion_isotonicM_n_def using intervall_le_foldr
  by auto

```

```

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)>
  using assms(2)
  unfolding inclusion_isotonicM_n_def inclusion_isotonicM_def
  by (metis assms(1) intervall_le_foldr le_interval_list_imp_length)

```

```

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)>
  using intervall_le_foldr_map assms(3)
  unfolding inclusion_isotonicM_n_def inclusion_isotonicM_def
  by (metis (no_types, lifting) assms(1) assms(2) intervall_le_foldr_map le_interval_list_imp_length)

```

```

lemma foldl_inclusion_isotonicM:
  assumes <inclusion_isotonic_binary f>
  shows <inclusion_isotonicM_n n (foldl f e)>
  unfolding inclusion_isotonicM_n_def
  apply(subst foldl_conv_foldr)
  apply(subst foldl_conv_foldr)
  using assms foldr_inclusion_isotonicM[simplified inclusion_isotonicM_def]
  le_interval_list_rev length_rev
  unfolding inclusion_isotonicM_n_def inclusion_isotonic_binary_def
  by (metis)

```

```

lemma fold_inclusion_isotonicM:

```

```

assumes <inclusion_isotonic_binary f>
shows <inclusion_isotonicM_n n (λ x. fold f x e)>
unfolding inclusion_isotonicM_n_def
apply(subst foldL_conv_fold[symmetric])
apply(subst foldL_conv_fold[symmetric])
using assms foldL_inclusion_isotonicM[simplified inclusion_isotonicM_def]
unfolding inclusion_isotonic_binary_def inclusion_isotonicM_n_def
by (metis)

```

```

lemma map2_inclusion_isotonicM_left: assumes <inclusion_isotonic_binary f>
  shows <inclusion_isotonicM n (map2 f xs)>
unfolding inclusion_isotonicM_def inclusion_isotonic_binary_def
apply(safe,subst le_interval_list_all2, simp_all)
using le_interval_list_imp le_interval_list_all
by (metis assms dual_order.refl inclusion_isotonic_binary_def less_eq_interval_def)

```

```

lemma map2_inclusion_isotonicM_right: assumes <inclusion_isotonic_binary f>
  shows <inclusion_isotonicM n (λ ys. map2 f ys xs)>
unfolding inclusion_isotonicM_def inclusion_isotonic_binary_def
apply(safe,subst le_interval_list_all2, simp_all)
using le_interval_list_imp le_interval_list_all
by (metis assms dual_order.refl inclusion_isotonic_binary_def less_eq_interval_def)

```

```

lemma map2_inclusion_isotonicM_right_g:
assumes <inclusion_isotonic_binary f>
and <∀ 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 ≤I js>
shows<map2 f (g is) (h xs) ≤I map2 f (g js) (h xs)>
using assms unfolding inclusion_isotonic_binary_def
apply(subst le_interval_list_all2, simp_all)
using assms unfolding inclusion_isotonic_binary_def
by (metis dual_order.refl inclusion_isotonicM_def
  le_interval_list_imp less_eq_interval_def)

```

```

lemma inclusion_isotonicM_map:
assumes <∀ x ∈ set xs . g x ≤ h x>
shows <(map g xs) ≤I (map h xs)>
using assms by(subst le_interval_list_all2, simp_all)

```

6.3 Interval Extension and Inclusion Properties

```

lemma interval_extension_inclusion:
assumes <F is_interval_extension_of f>
shows <∀ X . interval_of (f X) ≤ F (interval_of X)>
using assms
unfolding is_interval_extension_of_def
by (simp add: in_interval_to_interval interval_of_in_eq)

```

lemma *interval_extension_subset_const*:
assumes *interval_ext_F*: $\langle F \text{ is_interval_extension_of } f \rangle$
shows $\langle \forall X . \text{set_of } (f X) \subseteq \text{set_of } (F (f X)) \rangle$
using *assms*
unfolding *is_interval_extension_of_def set_of_def* **by** *auto*

lemma *fundamental_theorem_of_interval_analysis*:
fixes *F* :: $\langle \text{real interval} \Rightarrow \text{real interval} \rangle$
assumes *interval_ext_F*: $\langle F \text{ is_interval_extension_of } f \rangle$
and *inc_isotonic_F*: $\langle \text{inclusion_isotonic } F \rangle$
shows $\langle \forall X . f' (\text{set_of } X) \subseteq \text{set_of } (F X) \rangle$
proof
fix *X*
show $\langle f' (\text{set_of } X) \subseteq \text{set_of } (F X) \rangle$
proof
fix *y*
assume $\langle y \in f' (\text{set_of } X) \rangle$
then obtain *x* **where** *i*: $\langle x \in \text{set_of } X \rangle$ **and** $\langle y = f x \rangle$ **by** *auto*
then have $\langle \text{interval_of } (f x) = F (\text{interval_of } x) \rangle$ **using** *assms* **unfolding** *is_interval_extension_of_def* **by** *simp*
then have $\langle y \in \text{set_of } (F (\text{interval_of } x)) \rangle$ **using** $\langle y = f x \rangle$ **by** (*metis in_interval_to_interval*)
then have $\langle \text{interval_of } x \leq X \rangle$ **using** $\langle x \in \text{set_of } X \rangle$ **using** *interval_of_in_eq* **by** *blast*
then have $\langle F (\text{interval_of } x) \leq F X \rangle$ **using** *inc_isotonic_F* **unfolding** *inclusion_isotonic_def* **by** *simp*
then show $\langle y \in \text{set_of } (F X) \rangle$ **using** $\langle y \in \text{set_of } (F (\text{interval_of } x)) \rangle$ **using** *set_of_subset_iff'* **by** *auto*
qed
qed

lemma *interval_extension_bounds*:
assumes *inclusion_isotonic F*
and *F is_interval_extension_of f*
shows $\langle \forall x \in \text{set_of } X . \text{lower } (F X) \leq f x \rangle \vee \langle \forall x \in \text{set_of } X . f x \leq \text{lower } (F X) \rangle$
using *assms*
by (*metis (no_types, lifting) in_bounds inclusion_isotonic_def interval_of_in_eq is_interval_extension_of_def*)

lemma *inclusion_isotonic_extension_subset*:
assumes *inclusion_isotonic F*
and *F is_interval_extension_of f*
shows $\langle f' \text{set_of } X \subseteq \text{set_of } (F X) \rangle$
using *assms interval_of_in_eq*
unfolding *inclusion_isotonic_def set_of_eq is_interval_extension_of_def*
by (*metis (mono_tags, lifting) image_subsetI*)

lemma *inclusion_isotonic_extension_includes*:
assumes *inclusion_isotonic F*
and *F is_interval_extension_of f*
shows $\langle \forall x \in \text{set_of } X . f x \in \text{set_of } (F X) \rangle$
using *assms inclusion_isotonic_extension_subset*
by *blast*

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

using *assms inclusion_isotonic_extension_includes*
using *in_bounds* **by** *blast*

lemma *inclusion_isotonic_extension_upper_bound*:
assumes *inclusion_isotonic F*
and *F is_interval_extension_of f*
shows $\langle \forall x \in \text{set_of } X. f\ x \leq \text{upper } (F\ X) \rangle$
using *assms inclusion_isotonic_extension_includes*
using *in_bounds* **by** *blast*

lemma *inclusion_isotonic_inf*:
fixes *F::<real interval \Rightarrow real interval>*
assumes *inclusion_isotonic F*
and *F is_interval_extension_of f*
shows $\langle \text{lower } (F\ (X::\text{real interval})) \leq \text{Inf } (f'\ \text{set_of } X) \rangle$
using *inclusion_isotonic_extension_subset[of F f X, simplified assms]*
clnf_superset_mono[of f' set_of X set_of (F X)]
by (*simp add: bdd_below_set_of inf_set_of*)

lemma *inclusion_isotonic_sup*:
fixes *F::<real interval \Rightarrow real interval>*
assumes *inclusion_isotonic F*
and *F is_interval_extension_of f*
shows $\langle \text{Sup } (f'\ \text{set_of } X) \leq \text{upper } (F\ X) \rangle$
using *inclusion_isotonic_extension_subset[of F f X, simplified assms]*
cSup_subset_mono[of f' set_of X set_of (F X)]
by (*simp add: bdd_above_set_of sup_set_of*)

lemma *lower_inf*:
fixes *F::<real interval \Rightarrow real interval>*
assumes *inclusion_isotonic F*
and *F is_interval_extension_of f*
shows $\text{Inf } (f'\ \text{set_of } X) \leq f\ (\text{lower } X)$
using *clnf_superset_mono[of {f (lower X)} f' set_of X]*
by (*metis assms(1) assms(2) bdd_below_mono bdd_below_set_of bot.extremum*
clnf_singleton imageI insert_not_empty insert_subsetI
fundamental_theorem_of_interval_analysis lower_in_interval)

lemma *upper_sup*:
fixes *F::<real interval \Rightarrow real interval>*
assumes *inclusion_isotonic F*
and *F is_interval_extension_of f*
shows $f\ (\text{upper } X) \leq \text{Sup } (f'\ \text{set_of } X)$
using *cSup_subset_mono[of {f (upper X)} f' set_of X]*
by (*meson assms(1) assms(2) bdd_above_mono bdd_above_set_of cSup_upper*
imageI fundamental_theorem_of_interval_analysis upper_in_interval)

lemma *lower_F_f*:
fixes *F::<real interval \Rightarrow real interval>*
assumes *inclusion_isotonic F*
and *F is_interval_extension_of f*
shows $\text{lower } (F\ X) \leq f\ (\text{lower } X)$
by (*simp add: assms(1) assms(2) inclusion_isotonic_extension_lower_bound*)


```

lemma upper_F_f:
  fixes F::‹real interval  $\Rightarrow$  real interval›
  assumes inclusion_isotonic F
  and F is_interval_extension_of f
  shows  $f(\text{upper } X) \leq \text{upper}(F X)$ 
  by (simp add: assms(1) assms(2) inclusion_isotonic_extension_upper_bound)

```

```

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)  $\leq$  upper (F a)›
  using inclusion_isotonic_extension_lower_bound[of F f, simplified assms, simplified]
  inclusion_isotonic_extension_upper_bound[of F f, simplified assms, simplified]
  le_left_mono lower_in_interval upper_in_interval
  by (metis adjacent)

```

6.4 Division

```

context interval_division_inverse
begin
lemma inclusion_isotonic_on_inverse[simp]:
  ‹inclusion_isotonic_on  $(\lambda x . \neg 0 \in_i x) ((\text{inverse}::('a \text{ interval} \Rightarrow 'a \text{ interval})))$ ›
  using inverse_non_zero_def
  unfolding inclusion_isotonic_on_def less_eq_interval_def
  by (smt (z3) dual_order.trans in_bounds in_intervall lower_le_upper mk_interval_lower
  mk_interval_upper upper_leq_lower_div)

lemma inclusion_isotonic_on_division[simp]:
  ‹inclusion_isotonic_binary_on  $(\lambda x . \neg 0 \in_i x) (\lambda x y . \text{divide } x y)$ ›
  using inclusion_isotonicM_times divide_non_zero_def inclusion_isotonic_on_inverse
  unfolding o_def inclusion_isotonic_binary_on_def
  inclusion_isotonic_on_def inclusion_isotonic_binary_def
  by metis

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 $w(F X) \leq L w X$ for every $X \subseteq X$. Hence the width of $F X$ approaches 0: 'a at least linearly with the width of X .

definition *lipschitzl_on* :: 'a:: {zero,minus,preorder,times} \Rightarrow 'a interval set \Rightarrow ('a interval \Rightarrow 'a interval) \Rightarrow bool
where *lipschitzl_on* $C U F \longleftrightarrow (o \leq C \wedge (\forall x \in U. width (F x) \leq C * width x))$

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

bundle *lipschitzl_syntax* **begin**

notation *lipschitzl_on* ($_$ —*lipschitzl'_on* [1000])

end

bundle *no_lipschitzl_syntax* **begin**

no_notation *lipschitzl_on* ($_$ —*lipschitzl'_on* [1000])

end

unbundle *lipschitzl_syntax*

lemma *lipschitzl_onI*: C —*lipschitzl_on* $U F$

if $\bigwedge x. x \in U \implies width (F x) \leq C * width x$ $o \leq C$

using that by (auto simp: *lipschitzl_on_def*)

lemma *lipschitzl_onD*:

$width (F x) \leq C * width x$

if C —*lipschitzl_on* $U F$ $x \in U$

using that by (auto simp: *lipschitzl_on_def*)

lemma *lipschitzl_on_nonneg*:

$o \leq C$ **if** C —*lipschitzl_on* $U F$

using that by (auto simp: *lipschitzl_on_def*)

lemma *lipschitz_on_normD*:

$norm (width (F x)) \leq C * norm (width x)$

if C —*lipschitzl_on* $U F$ $x \in U$

using *lipschitzl_onD*[OF that]

by (simp add: width_def dist_norm)

lemma lipschitzL_on_mono: L -lipschitzL_on D ($F :: 'a :: \{\text{linordered_ring}\}$ interval $\Rightarrow 'a$ interval)
 if M -lipschitzL_on E F $M \leq L$ $D \subseteq E$
using that diff_ge_o_iff_ge lower_le_upper
 order_trans[OF_mult_right_mono]
unfolding lipschitzL_on_def width_def
by (metis (no_types, lifting) order_trans subsetD width_def)

lemmas lipschitzL_on_subset
 = lipschitzL_on_mono[OF__order_refl]
 and lipschitzL_on_le = lipschitzL_on_mono[OF_order_refl]

lemma lipschitzL_on_lel:
 C -lipschitzL_on U F
 if $\bigwedge x. x \in U \Rightarrow \text{width}(F x) \leq C * \text{width } x$
 $0 \leq C$
 for $F :: 'a :: \{\text{minus,preorder,times,zero}\}$ interval $\Rightarrow 'a$ interval
proof (rule lipschitzL_onI)
fix x **assume** $x \in U$
consider upper $x \leq$ lower x | lower $x \leq$ upper x
by simp
then show $\text{width}(F x) \leq C * \text{width } x$
proof cases
case 1
then have $\text{width}(F x) \leq C * \text{width } x$
by (auto intro!: that x)
then show ?thesis
by (simp add: dist_commute)
qed (auto intro!: that x)
qed fact

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 lipschitzL_on_compose:
fixes $U :: \langle 'a :: \{\text{linordered_ring}\}$ interval set \rangle
assumes f : C -lipschitzL_on U F and g : D -lipschitzL_on $(F'U)$ G
shows $(D * C)$ -lipschitzL_on U $(G \circ F)$
proof (rule lipschitzL_onI)
show $D * C \geq 0$ **using** lipschitzL_on_nonneg[OF f] lipschitzL_on_nonneg[OF g] **by** simp
fix x **assume** $H: x \in U$
have $\text{width}(G(F x)) \leq D * C * \text{width } x$
using lipschitzL_onD[OF g] H **assms**
unfolding width_def lipschitzL_on_def **apply** simp
by (smt (verit, ccfv_SIG) mult.assoc mult_left_mono order_trans)
then show $\text{width}((G \circ F) x) \leq D * C * \text{width } x$
unfolding comp_def **by** simp
qed

The definition $\llbracket ?C$ -lipschitzL_on $?U$ $?F$; $?D$ -lipschitzL_on $(?F' ?U)$ $?G \rrbracket \Longrightarrow (?D * ?C)$ -lipschitzL_on $?U$ $(?G \circ ?F)$ refers to lemma 6.3 in [4]

lemma *lipschitzl_on_compose2*:
fixes $U :: \langle 'a :: \{\text{linordered_ring}\} \text{ interval set} \rangle$
assumes $F: C \text{--lipschitzl_on } U$ **and** $G: D \text{--lipschitzl_on } (F'U) G$
shows $(D * C) \text{--lipschitzl_on } U (\lambda x. G (F x))$
using *lipschitzl_on_compose F G unfolding o_def by blast*

lemma *lipschitzl_on_cong*:
 $C \text{--lipschitzl_on } U G \iff D \text{--lipschitzl_on } V F$
if $C = D \ U = V \ \bigwedge x. x \in V \implies G x = F x$
using that by *(auto simp: lipschitzl_on_def)*

lemma *lipschitzl_on_transform*:
 $C \text{--lipschitzl_on } U G$
if $C \text{--lipschitzl_on } U F$
 $\bigwedge x. x \in U \implies G x = F x$
using that
unfolding *lipschitzl_on_def width_def*
by simp

lemma *lipschitz_on_empty_iff*: $C \text{--lipschitzl_on } \{\} f \iff C \geq 0$
by *(auto simp: lipschitzl_on_def)*

lemma *lipschitz_on_id*: $(1::\text{real}) \text{--lipschitzl_on } U (\lambda x. x)$
by *(auto simp: lipschitzl_on_def)*

lemma *lipschitz_on_constant*:
assumes $\langle \text{lower } c = \text{upper } c \rangle$
shows $(0::\text{real}) \text{--lipschitzl_on } U (\lambda x. c)$
using *assms by (auto simp: lipschitzl_on_def width_def)*

lemma *lipschitzl_on_mult*:
fixes $X :: 'a :: \{\text{linordered_idom}\}$
assumes *lipschitzl_on C U f*
and $1 \leq X$
shows *lipschitzl_on (X*C) U f*
using *assms interval_width_positive lower_le_upper mult_le_cancel_right1*
unfolding *lipschitzl_on_def width_def*
by *(smt (verit) diff_ge_o_iff_ge linorder_not_le mult.assoc order_trans)*

7.1.2 Interval bounds on reals

lemma *bounded_image_real*:
fixes $X :: \text{real interval}$ **and** $f :: \text{real} \Rightarrow \text{real}$
assumes $\forall x \in \{\text{lower } X.. \text{upper } X\}. f (\text{lower } X) - L * (\text{upper } X - \text{lower } X) \leq f x \wedge f x \leq f (\text{lower } X) + L * (\text{upper } X - \text{lower } X)$
shows $\exists x e. \forall y \in f' \{\text{lower } X.. \text{upper } X\}. \text{dist } x y \leq e$
proof –
let $?x = f (\text{lower } X)$
let $?e = L * (\text{upper } X - \text{lower } X)$
have $\forall y \in f' \{\text{lower } X.. \text{upper } X\}. \text{dist } ?x y \leq ?e$
proof
fix y **assume** $y \in f' \{\text{lower } X.. \text{upper } X\}$
then obtain x **where** $x \in \{\text{lower } X.. \text{upper } X\}$ **and** $y = f x$ **by** *auto*
then have $?x - ?e \leq y \wedge y \leq ?x + ?e$ **using** *assms by auto*

```

then show  $\text{dist } ?x \ y \leq ?e$ 
  unfolding dist_real_def by force
qed
then show ?thesis by auto
qed

```

```

lemma lipschitz_bounded_image_real:
  fixes  $X :: \text{real set}$  and  $f :: \text{real} \Rightarrow \text{real}$ 
  assumes bounded X and L-lipschitz_on X f
  shows bounded (f' X)
  using assms(1) assms(2) bounded_uniformly_continuous_image lipschitz_on_uniformly_continuous by blast

```

```

lemma inf_le_sup_image_real:
  fixes  $X :: \text{real interval}$  and  $f :: \text{real} \Rightarrow \text{real}$ 
  assumes L-lipschitz_on (set_of X) f
  shows  $\text{Inf } (f' \text{ set\_of } X) \leq \text{Sup } (f' \text{ set\_of } X)$ 
proof –
  have bounded (f' set_of X)
    using assms bounded_set_of lipschitz_bounded_image_real by blast
  then have bdd_below (f' set_of X) and bdd_above (f' set_of X)
    using bounded_imp_bdd_below bounded_imp_bdd_above by auto
  then have  $\text{Inf } (f' \text{ set\_of } X) \leq \text{Sup } (f' \text{ set\_of } X)$ 
    by (metis set_of_nonempty cInf_le_cSup empty_is_image)
  then show ?thesis by simp
qed

```

```

lemma sup_image_le_real:
  fixes  $f :: \text{real} \Rightarrow \text{real}$  and  $F :: \text{real interval} \Rightarrow \text{real interval}$  and  $X :: \text{real interval}$ 
  assumes  $f' \text{ set\_of } X \subseteq \text{set\_of } (F X)$ 
  and L-lipschitz_on (set_of X) f
  shows  $\text{Sup } (f' \text{ set\_of } X) \leq \text{Sup } (\text{set\_of } (F X))$ 
proof –
  have ao: bounded (f' set_of X)
    using lipschitz_bounded_image_real[of set_of X] assms bounded_set_of[of X] by simp
  have a1: bdd_above (f' set_of X)
    using assms
    using ao bounded_imp_bdd_above by presburger
  then have a2:  $\forall y \in f' \text{ set\_of } X. y \leq \text{Sup } (f' \text{ set\_of } X)$  using bounded_has_Sup(1) ao
    by blast
  then have a3:  $\text{Sup } (f' \text{ set\_of } X) \leq \text{Sup } (\text{set\_of } (F X))$ 
    using set_of_nonempty assms ao a1 a2 atLeastAtMost_iff closed_real_atLeastAtMost closed_subset_contains_Sup
    empty_is_image set_of_eq sup_set_of
    by metis
  then show ?thesis by simp
qed

```

```

lemma inf_image_le_real:
  fixes  $f :: \text{real} \Rightarrow \text{real}$  and  $F :: \text{real interval} \Rightarrow \text{real interval}$  and  $X :: \text{real interval}$ 
  assumes  $f' \text{ set\_of } X \subseteq \text{set\_of } (F X)$ 
  and L-lipschitz_on (set_of X) f
  shows  $\text{Inf } (\text{set\_of } (F X)) \leq \text{Inf } (f' (\text{set\_of } X))$ 
proof –
  have ao: bounded (f' set_of X)
    using lipschitz_bounded_image_real[of set_of X] assms bounded_set_of[of X] by simp

```

```

have a1: bdd_above (f' set_of X)
  using assms
  by (metis atLeastAtMost_iff bdd_above.unfold set_of_eq subset_eq)
then have a2:  $\forall y \in f' \text{ set\_of } X. y \geq \text{Inf } (f' \text{ set\_of } X)$ 
  using bounded_has_Inf(1) a0
  by blast
then have a3:  $\text{Inf } (\text{set\_of } (F X)) \leq \text{Inf } (f' (\text{set\_of } X))$  using assms
  by (metis set_of_nonempty a0 bounded_imp_bdd_below closed_real_atLeastAtMost closed_subset_contains_Inf
empty_is_image in_bounds inf_set_of set_of_eq)
then show ?thesis by simp
qed

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 (subtraction) 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:  $A \leq B \implies D \leq C \implies A - C \leq B - D$   
begin end
```

```
instance nat::minus_mono
```

```
  by(standard, simp)
```

```
instance int::minus_mono
```

```
  by(standard, simp)
```

```
instance real::minus_mono
```

```
  by(standard, simp)
```

```
instance integer::minus_mono
```

```
  by(standard, simp)
```

```
instance ereal::minus_mono
```

```
  by(standard, simp add: ereal_minus_mono)
```

8.1.2 Infrastructure for Lifting Interval Operations to Lists of Intervals

```
definition un_op_interval_list:: $\langle 'a \text{ interval} \Rightarrow 'a \text{ interval} \rangle$   
   $\Rightarrow 'a \text{ interval list} \Rightarrow 'a \text{ interval list}$ 
```

```
  where
```

```
   $\langle \text{un\_op\_interval\_list } op \text{ } xs = \text{map } op \text{ } xs \rangle$ 
```

```
definition bin_op_interval_list:: $\langle 'a \text{ interval} \Rightarrow 'a \text{ interval} \Rightarrow 'a \text{ interval} \rangle$   
   $\Rightarrow 'a \text{ interval list} \Rightarrow 'a \text{ interval list} \Rightarrow 'a \text{ interval list}$ 
```

```
  where
```

```
   $\langle \text{bin\_op\_interval\_list } op \text{ } xs \text{ } ys = \text{concat } (\text{map } (\lambda x . \text{map } (op \text{ } x) \text{ } xs) \text{ } ys) \rangle$ 
```

```
lemma bin_op_interval_list_non_empty:  $\langle xs \neq [] \wedge ys \neq [] \rangle = (\text{bin\_op\_interval\_list } op \text{ } xs \text{ } ys \neq [])$ 
```

```
  unfolding bin_op_interval_list_def
```

```
  by(auto simp add: ex_in_conv)
```

```

lemma bin_op_interval_list_empty: ⟨xs = [] ∨ ys = []⟩ = (bin_op_interval_list op xs ys = [])⟩
unfolding bin_op_interval_list_def
by(auto simp add: ex_in_conv)

```

```

lemma bin_op_interval_unroll: ⟨bin_op_interval_list op (xs) (y#ys) = (map (op y) xs)@(bin_op_interval_list op xs ys)⟩
unfolding bin_op_interval_list_def by(simp)

```

```

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)⟩
unfolding bin_op_interval_list_def
proof(induction xs ys rule:list_induct2')
case 1
then show ?case by simp
next
case (2 x xs)
then show ?case by simp
next
case (3 y ys)
then show ?case by simp
next
case (4 x xs y ys)
then show ?case
by(auto simp add: op_commute, blast)
qed

```

```

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))))⟩
unfolding bin_op_interval_list_def
proof(induction xs ys rule:list_induct2')
case 1
then show ?case by simp
next
case (2 x xs)
then show ?case by simp
next
case (3 y ys)
then show ?case by simp
next
case (4 x xs y ys)
then show ?case
proof(induction zs)
case Nil
then show ?case by simp
next
case (Cons a zs)
then show ?case
apply(auto simp add: 4 list.set_map op_assoc)[1]
subgoal
by (meson imagel)
subgoal

```

```

  by blast
subgoal
  by (meson UnionI imagel insertI1)
subgoal
  by (meson UN_I imagel insertCI)
done
qed
qed

```

Lifting Unary Minus, Addition, and Multiplication

```

definition <iList_uminus = un_op_interval_list (λ x. - x)>

```

```

definition <iList_plus = bin_op_interval_list (+)>

```

```

definition <iList_times = bin_op_interval_list (*)>

```

```

lemma iList_plus_lower:

```

```

  assumes <A ≠ []> and <B ≠ []>

```

```

  shows <lower (hd (iList_plus A B)) = lower (hd A) + lower (hd B)>

```

```

proof(insert assms, induction B)

```

```

  case Nil

```

```

  then show ?case by simp

```

```

next

```

```

  case (Cons a B)

```

```

  then show ?case

```

```

    unfolding iList_plus_def

```

```

    apply (simp add: hd_map bin_op_interval_unroll[of (+) A a B])

```

```

    using add.commute by blast

```

```

qed

```

```

lemma iList_plus_upper:

```

```

  assumes <A ≠ []> and <B ≠ []>

```

```

  shows <upper (hd (iList_plus A B)) = upper (hd A) + upper (hd B)>

```

```

proof(insert assms, induction B)

```

```

  case Nil

```

```

  then show ?case by simp

```

```

next

```

```

  case (Cons a B)

```

```

  then show ?case

```

```

    unfolding iList_plus_def

```

```

    apply (simp add: hd_map bin_op_interval_unroll[of (+) A a B])

```

```

    using add.commute by blast

```

```

qed

```

```

lemma iList_plus_unroll:

```

```

  <iList_plus ys (x # xs) = map ((+) x) ys @ iList_plus ys xs>

```

```

by (simp add: iList_plus_def bin_op_interval_unroll[of (+) ys x xs])

```

```

lemma a ≠ [] ⇒ (iList_plus [Interval (0, 0)] a) = (a::'a::{ordered_ab_group_add,zero} interval list)

```

```

proof(induction a rule:list_nonempty_induct)

```

```

  case (single x)

```

```

  then show ?case

```

```

    by (metis add.right_neutral append.right_neutral

```

```

        bin_op_interval_list_non_empty iList_plus_def iList_plus_unroll list.map(2) zero_interval_def)

```

```

next

```

```

case (cons x xs)
then show ?case
by (metis add.right_neutral append_Cons append_Nil iList_plus_unroll list.map(1) list.map(2) zero_interval_def)
qed

```

```

lemma iList_plus_commute:
  <set (iList_plus xs ys) = set (iList_plus ys xs)>
using bin_op_interval_list_commute[of (+) xs ys] add.commute
unfolding iList_plus_def
by auto

```

```

lemma iList_plus_assoc:
  <set (iList_plus xs (iList_plus ys zs)) = set (iList_plus (iList_plus xs ys) zs)>
using bin_op_interval_list_assoc[of (+) xs ys zs] add.assoc
unfolding iList_plus_def
by auto

```

```

lemma remdups_append1:
  remdups (remdups xs @ ys) = remdups (xs @ ys)
by(induction xs) auto

```

```

lemma bin_op_interval_remdups_left:
  <remdups (bin_op_interval_list op (remdups x) y) = remdups (bin_op_interval_list op x y)>
proof(induction y)
case Nil
then show ?case
by (simp add: bin_op_interval_list_def)
next
case (Cons y' ys')
then show ?case
by (metis (no_types, opaque_lifting) bin_op_interval_list_non_empty bin_op_interval_unroll
  remdups_append1 remdups_append2 remdups_map_remdups)
qed

```

```

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(induction b)
case Nil
then show ?case
by (metis bin_op_interval_list_empty iList_plus_def)
next
case (Cons a b)
then show ?case
by (metis bin_op_interval_remdups_left iList_plus_def list.disc1 remdups.simps(1))
qed

```

```

lemma interval_add_eq: <a + b = Interval(lower a + lower b, upper a + upper b)>
apply(subst interval_eq_iff[of a+b Interval (lower a + lower b, upper a + upper b)])
using upper_plus[of a b] lower_plus[of a b]
by (simp add: add_mono)

```

```

lemma iList_plus_lower_upper_eq:
  <iList_plus = bin_op_interval_list (λ a b. Interval(lower a + lower b, upper a + upper b))>
  unfolding iList_plus_def using interval_add_eq
  by metis

```

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))>
  by (metis interval_eq le_add_diff_inverse lower_le_upper width_def)

```

```

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(insert assms, unfold sorted_wrt_lower_def cmp_lower_width_def, induct xs ys rule:list_induct2')
  case 1
  then show ?case by simp
next
  case (2 x xs)
  then show ?case by simp
next
  case (3 y ys)
  then show ?case by simp
next
  case (4 x xs y ys)
  then show ?case
    using interval_lower_width_eq
    apply (auto)[1]
    subgoal by (smt (verit) insert_iff interval_lower_width_eq order.asym order_le_imp_less_or_eq)
    subgoal by (smt (verit, ccfv_SIG) antisym insert1 insert_eq_iff interval_lower_width_eq not_less_iff_gr_or_eq)
    done
qed

```

```

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>

```

```
using lower_le_upper unfolding cmp_non_overlapping_def
by(metis dual_order.trans)
```

```
lemma cmp_non_overlapping_upper: <cmp_non_overlapping x y  $\implies$  upper x  $\leq$  upper y>
using lower_le_upper unfolding cmp_non_overlapping_def
by(metis dual_order.trans)
```

```
definition <non_overlapping_sorted = sorted_wrt cmp_non_overlapping>
definition <contiguous xs = ( $\forall$  i < length xs - 1 . upper (xs ! i) = lower (xs ! (i + 1)))>
```

```
lemma non_overlapping_sorted_empty: <non_overlapping_sorted []>
by (simp add: non_overlapping_sorted_def cmp_non_overlapping_def)
```

```
lemma non_overlapping_sorted_unroll:
  assumes xs  $\neq$  [] shows non_overlapping_sorted (x # xs) = (upper x  $\leq$  lower (hd xs)  $\wedge$  non_overlapping_sorted xs)
proof(cases xs)
  case Nil
  then show ?thesis using assms by(simp)
next
  case (Cons y ys)
  then show ?thesis
  apply(simp add: non_overlapping_sorted_def cmp_non_overlapping_def cmp_lower_width_def)
  using lower_le_upper
  by (metis dual_order.trans)
qed
```

```
lemma contiguous_non_overlapping: <contiguous (is::'a::{preorder} interval list)  $\implies$  non_overlapping_sorted is>
proof(induction isrule:induct_list012)
  case 1
  then show ?case
  unfolding contiguous_def non_overlapping_sorted_def cmp_non_overlapping_def
  using lower_le_upper
  by simp
next
  case (2 x)
  then show ?case
  unfolding contiguous_def non_overlapping_sorted_def cmp_non_overlapping_def
  using lower_le_upper
  by simp
next
  case (3 x y zs)
  then show ?case
  apply(subst non_overlapping_sorted_unroll[of (y#zs) x])
  subgoal by(simp)
  subgoal apply(simp add: 3) using 3
  unfolding contiguous_def non_overlapping_sorted_def cmp_non_overlapping_def
  using lower_le_upper
  by fastforce
  done
qed
```

```
definition <cmp_non_adjacent = ( $\lambda$  x y. upper x < lower y)>
```

lemma *cmp_non_adjacent_lower*: $\langle \text{cmp_non_adjacent } x \ y \implies \text{lower } x < \text{lower } y \rangle$
using *lower_le_upper* **unfolding** *cmp_non_adjacent_def*
by(*metis dual_order.strict_trans2*)

lemma *cmp_non_adjacent_upper*: $\langle \text{cmp_non_adjacent } x \ y \implies \text{upper } x < \text{upper } y \rangle$
using *lower_le_upper* **unfolding** *cmp_non_adjacent_def*
by(*metis dual_order.strict_trans1*)

definition $\langle \text{non_adjacent_sorted_wrt_lower} = \text{sorted_wrt } \text{cmp_non_adjacent} \rangle$

lemma *non_adjacent_sorted_wrt_lower_unroll*:
assumes $xs \neq []$
shows $\text{non_adjacent_sorted_wrt_lower } (x \# \ xs) =$
 $((\text{upper } x < \text{lower } (\text{hd } \ xs)) \wedge \text{non_adjacent_sorted_wrt_lower } \ xs)$
proof(*cases xs*)
case *Nil*
then show ?thesis **using** *assms* **by**(*simp*)
next
case (*Cons y ys*)
then show ?thesis
apply(*simp add:non_adjacent_sorted_wrt_lower_def cmp_non_adjacent_def*)
by (*meson cmp_non_adjacent_def cmp_non_adjacent_lower dual_order.strict_trans*)
qed

lemma *non_adjacent_implies_non_overlapping*:
assumes $\langle \text{non_adjacent_sorted_wrt_lower } \ is \rangle$ **shows** $\langle \text{non_overlapping_sorted } \ is \rangle$
proof(*insert assms, induction is*)
case *Nil*
then show ?case
unfolding *non_overlapping_sorted_def cmp_non_overlapping_def*
 $\text{non_adjacent_sorted_wrt_lower_def } \text{cmp_non_adjacent_def}$
by(*simp*)
next
case (*Cons a is*)
then show ?case
unfolding *non_overlapping_sorted_def cmp_non_overlapping_def*
 $\text{non_adjacent_sorted_wrt_lower_def } \text{cmp_non_adjacent_def}$
using *order.strict_implies_order* **by** *auto*
qed

lemma *non_overlapping_implies_sorted_wrt_lower*:
assumes $\langle \text{non_overlapping_sorted } (\text{is}::'a::\{\text{minus_mono}\} \ \text{interval } \text{list}) \rangle$
shows $\langle \text{sorted_wrt_lower } \ is \rangle$
proof(*insert assms, induction is*)
case *Nil*
then show ?case **unfolding** *sorted_wrt_lower_def* **by** *simp*
next
case (*Cons a is*)
then show ?case
unfolding *non_overlapping_sorted_def sorted_wrt_lower_def*
 $\text{cmp_lower_width_def } \text{cmp_non_overlapping_def}$
by (*simp, metis (no_types, lifting) cmp_non_adjacent_def cmp_non_adjacent_lower minus_mono*
 $\text{dual_order.strict_iff_order } \text{lower_le_upper } \text{width_def}$)

qed

```
lemma non_overlapping_implies_sorted_wrt_upper:
  assumes <non_overlapping_sorted is>
  shows <sorted_wrt_upper is>
proof(insert assms, induction is rule:induct_list012)
  case 1
  then show ?case by(simp add: leD sorted_wrt_upper_def)
next
  case (2 x)
  then show ?case by(simp add: leD sorted_wrt_upper_def)
next
  case (3 x y zs)
  then show ?case
  by(auto simp add: cmp_non_overlapping_upper non_overlapping_sorted_def sorted_wrt_upper_def leD)[1]
qed
```

```
lemma non_adjacent_implies_sorted_wrt_lower:
  assumes <non_adjacent_sorted_wrt_lower is>
  shows <sorted_wrt_lower is>
proof(insert assms, induction is)
  case Nil
  then show ?case
  unfolding sorted_wrt_lower_def
  by simp
next
  case (Cons a is)
  then show ?case
  unfolding sorted_wrt_lower_def non_adjacent_sorted_wrt_lower_def
    cmp_non_adjacent_def cmp_lower_width_def width_def
  by(simp split:if_splits, metis dual_order.strict_trans2 lower_le_upper order_less_irrefl)
qed
```

```
lemma non_adjacent_implies_distinct:
  assumes <non_adjacent_sorted_wrt_lower is>
  shows <distinct is>
proof(insert assms, induction is)
  case Nil
  then show ?case
  unfolding sorted_wrt_lower_def
  by simp
next
  case (Cons a is)
  then show ?case
  unfolding sorted_wrt_lower_def non_adjacent_sorted_wrt_lower_def
    cmp_non_adjacent_def cmp_lower_width_def width_def
  by(simp split:if_splits, metis dual_order.strict_trans2 lower_le_upper order_less_irrefl)
qed
```

```
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 $xs \neq []$

shows $sorted_wrt_lower (x \# xs) = ((if\ lower\ x \neq\ lower\ (hd\ xs)$
 $\quad then\ lower\ x < lower\ (hd\ xs)$
 $\quad else\ width\ x \leq width\ (hd\ xs)) \wedge sorted_wrt_lower (xs))$

proof(cases xs)

case Nil

then show ?thesis **using** assms **by** (simp)

next

case (Cons y ys)

then show ?thesis

apply (simp add: sorted_wrt_lower_def cmp_lower_width_def)

by (smt (verit) dual_order.irrefl dual_order.strict_trans2 dual_order.trans order.strict_implies_order)

qed

lemma sorted_wrt_upper_unroll:

assumes $xs \neq []$

shows $sorted_wrt_upper (x \# xs) = ((upper\ x \leq upper\ (hd\ xs)) \wedge sorted_wrt_upper (xs))$

proof(cases xs)

case Nil

then show ?thesis **using** assms **by** (simp)

next

case (Cons y ys)

then show ?thesis

apply (simp add: sorted_wrt_upper_def cmp_lower_width_def)

using dual_order.trans **by** blast

qed

lemma sorted_wrt_lower_tail: $sorted_wrt_lower (x \# xs) \implies sorted_wrt_lower (xs)$

unfolding sorted_wrt_lower_def

by simp

lemma sorted_wrt_lower_tail': $sorted_wrt_lower (x \# y \# ys) \implies sorted_wrt_lower (x \# ys)$

unfolding sorted_wrt_lower_def **by** simp

lemma iList_plus_leq_B:

assumes sorted_wrt_lower A **and** sorted_wrt_lower B **and** $1 < length\ B$

shows $hd (map\ lower (iList_plus\ A\ B)) \leq hd (map\ lower (iList_plus\ A\ (tl\ B)))$

proof(insert assms, induction B)

case Nil

then show ?case **by** simp

next

case (Cons b bs) **note** * = this

then show ?case

proof(cases bs)

case Nil

then show ?thesis

using * **by** simp

```

next
  case (Cons b' bs^) note ** = this
  then have a: <hd (map lower (iList_plus A (b#bs))) ≤ hd (map lower (iList_plus A bs))>
  proof(induction bs)
    case Nil
    then show ?case by simp
  next
  case (Cons b'' bs'')
  then show ?case
  using * **
  by (smt (z3) add.commute add_right_mono dual_order.eq_iff hd_append iList_plus_lower
    iList_plus_unroll list.map_disc_iff list.map_sel(1) list.sel(1) list.simps(3)
    map_append order_less_imp_le sorted_wrt_lower_unroll)
qed
then show ?thesis
using * ** a by simp
qed
qed

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(insert assms, induction A)
    case Nil
    then show ?case by simp
  next
  case (Cons a as) note * = this
  then show ?case
  proof(cases as)
    case Nil
    then show ?thesis
    using * by simp
  next
  case (Cons a' as') note ** = this
  then have a: <hd (map lower (iList_plus (a#as) B)) ≤ hd (map lower (iList_plus as B))>
  proof(induction as)
    case Nil
    then show ?case by simp
  next
  case (Cons a'' as'')
  then show ?case
  using * **
  by (smt (verit, best) add_right_mono bin_op_interval_list_empty dual_order.eq_iff
    iList_plus_def iList_plus_lower list.map_sel(1) list.sel(1) list.simps(3)
    order_less_imp_le sorted_wrt_lower_unroll)
qed
then show ?thesis
using * ** a by simp
qed
qed

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  $\langle xs \neq [] \rangle$ 
  shows  $\langle \text{merge\_overlapping\_intervals\_sorted\_wrt\_lower } xs \rangle \neq []$ 
  using assms
  by(induction xs rule:merge_overlapping_intervals_sorted_wrt_lower.induct, simp_all)

lemma merge_overlapping_intervals_sorted_hd_lower:
  assumes  $\langle xs \neq [] \rangle$ 
  shows  $\text{lower } (\text{hd } (\text{merge\_overlapping\_intervals\_sorted\_wrt\_lower } (xs))) = \text{lower } (\text{hd } xs)$ 
  proof(insert assms, induction xs rule:merge_overlapping_intervals_sorted_wrt_lower.induct)
  case 1
  then show ?case by(simp)
  next
  case (2 x)
  then show ?case by(simp)
  next
  case (3 x y ys)
  then show ?case
  using 3.IH(2) list.sel(1) lower_le_upper max.coboundedI2 mk_interval_lower
  by (metis list.discI max.commute merge_overlapping_intervals_sorted_wrt_lower.simps(3))
qed

lemma merge_overlapping_intervals_sorted_hd_upper:
  assumes  $\langle xs \neq [] \rangle$ 
  shows  $\text{upper } (\text{hd } xs) \leq \text{upper } (\text{hd } (\text{merge\_overlapping\_intervals\_sorted\_wrt\_lower } xs))$ 
  proof(insert assms(1), induction xs rule:merge_overlapping_intervals_sorted_wrt_lower.induct)
  case 1
  then show ?case by(simp)
  next
  case (2 x)
  then show ?case by(simp)
  next
  case (3 x y ys)
  then show ?case
  proof(cases upper x  $\leq$  lower y)
  case True note non_overlapping = this
  then show ?thesis by simp
  next
  case False note overlapping_or_included = this
  then show ?thesis proof(cases upper x  $\leq$  upper y)
  case True note overlapping = this
  then show ?thesis
  proof(cases lower x  $\leq$  upper y)
  case True
  then show ?thesis
  using 3.IH(2)[simplified overlapping_or_included]
  apply(simp add: overlapping_or_included overlapping True)
  using overlapping by simp
  next
  case False
  then show ?thesis
  using overlapping_or_included overlapping False
  by (metis lower_le_upper max.coboundedI1 max_def)
qed

```

```

next
case False note included = this
then show ?thesis
proof(cases lower x ≤ upper y)
case True
then show ?thesis
using 3.IH(2)[simplified overlapping_or_included]
using included by simp
next
case False
then show ?thesis
using overlapping_or_included included False
lower_le_upper[of x] lower_le_upper[of y]
using 3.IH(2) by fastforce
qed
qed
qed
qed

lemma Interval_id[simp]: ⟨Interval (lower x, upper x) = x ⟩
by (simp add: interval_eq_iff)

lemma mk_interval_id[simp]: ⟨(mk_interval (lower x, upper x)) = x⟩
using lower_le_upper[of x] unfolding mk_interval' by (auto)

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(insert assms, induction xs rule:merge_overlapping_intervals_sorted_wrt_lower.induct)
case 1
then show ?case by (simp)
next
case (2 x)
then show ?case by (simp)
next
case (3 x y ys)
then show ?case
proof(cases upper x ≤ lower y)
case True note non_overlapping = this
then show ?thesis by simp
next
case False note overlapping_or_included = this
then show ?thesis proof(cases upper x ≤ upper y)
case True note overlapping = this
then show ?thesis
proof(cases lower x ≤ upper y)
case True
then show ?thesis
using 3.IH(2)[simplified overlapping_or_included]
apply (simp add: width_def overlapping_or_included overlapping True)
using True diff_right_mono list.sel(1) list.simps(3)
merge_overlapping_intervals_sorted_hd_lower merge_overlapping_intervals_sorted_hd_upper
mk_interval_lower mk_interval_upper order.trans overlapping minus_mono

```

```

    by (smt (verit, best) dual_order.refl)
  next
  case False
  then show ?thesis
  using overlapping_or_included overlapping False
  by (metis lower_le_upper max.coboundedl1 max_def)
qed
next
case False note included = this
then show ?thesis
proof(cases lower x ≤ upper y)
  case True
  then show ?thesis
  using 3.IH(2)[simplified overlapping_or_included] included
  by (metis list.discr list.sel(1) max_def
      merge_overlapping_intervals_sorted_wrt_lower.simps(3) mk_interval_id
      overlapping_or_included)
next
case False
then show ?thesis
using overlapping_or_included included False
  lower_le_upper[of x] lower_le_upper[of y]
  using 3.IH(2)[simplified overlapping_or_included]
  by (metis list.discr list.sel(1) max_def merge_overlapping_intervals_sorted_wrt_lower.simps(3) mk_interval_id)
qed
qed
qed
qed

```

```

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(insert assms, induction xs rule:merge_overlapping_intervals_sorted_wrt_lower.induct)
case 1
then show ?case
  unfolding sorted_wrt_lower_def
  by(simp)
next
case (2 x)
then show ?case
  unfolding sorted_wrt_lower_def
  by(simp)
next
case (3 x y ys)
then show ?case
proof(cases upper x ≤ lower y)
  case True note non_overlapping = this
  then show ?thesis
  by (smt (verit) 3.IH(1) 3.prem1 dual_order.trans list.discr merge_overlapping_intervals_sorted_hd_lower
      merge_overlapping_intervals_sorted_hd_width merge_overlapping_intervals_sorted_wrt_lower.simps(3)
      merge_overlapping_intervals_sorted_wrt_lower_non_nil sorted_wrt_lower_unroll)
next

```

```

case False note overlapping_or_included = this
then show ?thesis proof(cases upper x ≤ upper y)
  case True note overlapping = this
  then show ?thesis
  proof(cases lower x ≤ upper y)
    case True
    then show ?thesis
    using 3.IH(2)[simplified overlapping_or_included]
    by (smt (verit, ccfv_threshold) 3.prem1 list.sel(1) max.absorb2
      merge_overlapping_intervals_sorted_wrt_lower.simps(3)
      mk_interval_id mk_interval_lower overlapping_or_included sorted_wrt1
      sorted_wrt_lower_def sorted_wrt_lower_tail' sorted_wrt_lower_unroll)
  next
  case False
  then show ?thesis
  using overlapping_or_included overlapping False
  using lower_le_upper order_trans by blast
  qed
next
case False note included = this
then show ?thesis
proof(cases lower x ≤ upper y)
  case True
  then show ?thesis
  using 3.IH(2)[simplified overlapping_or_included] included
    3.prem1 overlapping_or_included sorted_wrt_lower_tail'
  by (metis max_def merge_overlapping_intervals_sorted_wrt_lower.simps(3) mk_interval_id)
  next
  case False
  then show ?thesis
  using overlapping_or_included included False
    lower_le_upper[of x] lower_le_upper[of y]
  using 3.IH(2)[simplified overlapping_or_included]
    3.prem1 sorted_wrt_lower_tail'
  by (metis max_def merge_overlapping_intervals_sorted_wrt_lower.simps(3) mk_interval_id)
  qed
qed
qed
qed

```

```

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(insert assms, induction xs rule:merge_overlapping_intervals_sorted_wrt_lower.induct)
  case 1
  then show ?case unfolding non_overlapping_sorted_def
    by (metis merge_overlapping_intervals_sorted_wrt_lower.simps(1) sorted_wrt.simps(1))
  next
  case (2 x)
  then show ?case unfolding non_overlapping_sorted_def
    by (metis merge_overlapping_intervals_sorted_wrt_lower.simps(2) sorted_wrt1)
  next
  case (3 x y ys)
  then show ?case

```

```

proof(cases upper x ≤ lower y)
case True note non_overlapping = this
then show ?thesis
  using 3dual_order.trans list.disc1 merge_overlapping_intervals_sorted_hd_lower
    merge_overlapping_intervals_sorted_hd_width merge_overlapping_intervals_sorted_wrt_lower.simps
    merge_overlapping_intervals_sorted_wrt_lower_non_nil non_overlapping_sorted_unroll
  by (smt (verit, ccfv_threshold) list.sel(1) sorted_wrt_lower_tail)
next
case False note overlapping_or_included = this
then show ?thesis proof(cases upper x ≤ upper y)
  case True note overlapping = this
  then show ?thesis
  proof(cases lower x ≤ upper y)
  case True
  then show ?thesis
  using 3.IH(2)[simplified overlapping_or_included]
  by (smt (verit) 3.prem1 list.disc1 list.sel(1)
    max.absorb2 merge_overlapping_intervals_sorted_wrt_lower.simps(3)
    mk_interval_id mk_interval_lower overlapping_or_included sorted_wrt_lower_tail'
    sorted_wrt_lower_unroll)
  next
  case False
  then show ?thesis
  using overlapping_or_included overlapping False
  using lower_le_upper order_trans by blast
  qed
next
case False note included = this
then show ?thesis
proof(cases lower x ≤ upper y)
  case True
  then show ?thesis
  using 3.IH(2)[simplified overlapping_or_included] included
  using 3.prem1 overlapping_or_included sorted_wrt_lower_tail'
  by (metis max.orderE merge_overlapping_intervals_sorted_wrt_lower.simps(3) mk_interval_id nle_le)
  next
  case False
  then show ?thesis
  using overlapping_or_included included False
    lower_le_upper[of x] lower_le_upper[of y]
  using 3.IH(2)[simplified overlapping_or_included]
  using 3.prem1 sorted_wrt_lower_tail'
  by (metis max.orderE merge_overlapping_intervals_sorted_wrt_lower.simps(3) mk_interval_id nle_le)
  qed
qed
qed
qed

```

```

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) ≠ []>
using assms by(induction xs rule:merge_adjacent_intervals_sorted_wrt_lower.induct, simp_all)
```

```
lemma merge_adjacent_intervals_sorted_wrt_lower_non_nil':
shows <(merge_adjacent_intervals_sorted_wrt_lower (x#xs)) ≠ []>
by(simp add: merge_adjacent_intervals_sorted_wrt_lower_non_nil)
```

```
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)>
using assms
proof(induction xs rule:merge_adjacent_intervals_sorted_wrt_lower.induct)
case 1
then show ?case
by (metis merge_adjacent_intervals_sorted_wrt_lower.simps(1))
next
case (2 x)
then show ?case by simp
next
case (3 x y ys)
then show ?case
proof(cases (sorted_wrt_lower (mk_interval (lower x, max (upper y) (upper x)) # ys)))
case True
then show ?thesis
by (simp, metis 3.IH(2) list.sel(1) lower_le_upper max.coboundedI2 mk_interval_lower)
next
case False
then show ?thesis
apply(simp, subst 3 (2))
apply (smt (verit) 3 dual_order.strict_trans leD list.discI list.sel(1) lower_le_upper max.coboundedI2
mk_interval_lower sorted_wrt1 sorted_wrt_lower_def sorted_wrt_lower_unroll)
apply (smt (verit) 3.prem1 dual_order.trans interval_eqI list.discI list.sel(1)
lower_le_upper max.strict_order_iff max_def mk_interval_lower mk_interval_upper
sorted_wrt1 sorted_wrt_lower_def sorted_wrt_lower_unroll)
by (simp add: 3 max.coboundedI2 sorted_wrt_lower_def)
qed
qed
```

```
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)>
apply(induction xs rule:merge_adjacent_intervals_sorted_wrt_lower.induct)
by(auto simp add: max.coboundedI2 mk_interval' sorted_wrt_lower_def image_def)
```

```
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>
using sorted_wrt_lower_distinct_lists_eq[of xs ys, simplified assms, simplified]
by(auto)

```

```

lemma merge_adjacent_intervals_sorted_wrt_lower_lower_upper:
  assumes sorted_wrt_lower xs
  shows  $x \in \text{set } (\text{merge\_adjacent\_intervals\_sorted\_wrt\_lower } xs) \implies \exists l \in \text{set } xs. \exists u \in \text{set } xs. \text{lower } l = \text{lower } x \wedge \text{upper } u = \text{upper } x$ 
proof(insert assms, induction xs rule:merge_adjacent_intervals_sorted_wrt_lower.induct)
  case 1
  then show ?case
  by simp
next
  case (2 x)
  then show ?case
  by simp
next
  case (3 x y ys)
  then show ?case
  proof(cases ys)
  case Nil
  then show ?thesis
  using 3
  by (simp, smt (z3) empty_iff_empty_set list.sel(1) list.sel(3) list.set_cases
    max_def merge_adjacent_intervals_sorted_wrt_lower.simps(2)
    merge_adjacent_intervals_sorted_wrt_lower.simps(3)
    merge_adjacent_intervals_sorted_wrt_lower_sorted_lower_lower_hd mk_interval_lower
    mk_interval_upper)
  next
  case (Cons a list)
  then show ?thesis
  apply(auto simp add: mk_interval' max_def)[1]
  subgoal
  using 3
  by (smt (z3) insert_iff interval_eq_iff list.sel(1) list.simps(15) list.simps(3) lower_le_upper
    max.cobounded1 max_def_raw merge_adjacent_intervals_sorted_wrt_lower.simps(3)
    mk_interval_lower mk_interval_upper sorted_wrt_lower_tail' sorted_wrt_lower_unroll)
  subgoal
  using 3
  by (smt (verit) dual_order.trans list.discr list.inject list.sel(1) lower_le_upper max_def
    merge_adjacent_intervals_sorted_wrt_lower.elims mk_interval_id mk_interval_lower
    mk_interval_upper order.asym set_ConsD sorted_wrt_lower_tail' sorted_wrt_lower_unroll)
  done
qed
qed

```

```

primrec interval_insert_sort_lower_width :: ('a::{linorder,minus}) interval  $\Rightarrow$  'a interval list  $\Rightarrow$  '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 >
  by(induction xs, auto)
```

```
lemma interval_insert_sort_lower_width_nonempty:
  <interval_insert_sort_lower_width x xs ≠ [] >
  by(induction xs, auto)
```

```
lemma interval_insert_sort_wrt_lower:
  <sorted_wrt_lower xs ⇒ sorted_wrt_lower (interval_insert_sort_lower_width x xs)>
proof(induction xs)
  case Nil
  then show ?case by(simp add: sorted_wrt_lower_def cmp_lower_width_def)
next
  case (Cons a xs)
  then show ?case
  apply(auto)[1]
  subgoal
  by (metis cmp_lower_width_def list.discr list.sel(1) sorted_wrt_lower_unroll)
  subgoal
  apply(subst sorted_wrt_lower_unroll)
  apply(simp add: interval_insert_sort_lower_width_nonempty)
  apply(simp add: cmp_lower_width_def sorted_wrt_lower_def split:if_split)
  using interval_insert_sort_lower_width.simps(1)[of a]
  interval_insert_sort_lower_width.simps(2)
  list.sel(1) list.set_cases list.set_sel(1)
  by (smt (verit) interval_insert_sort_lower_width.simps(1) less_le_not_le nle_le)
  done
qed
```

```
lemma interval_isort_elements: set (interval_insert_sort_lower_width x xs) = {x} ∪ set xs
proof(induction xs)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  then show ?case by(auto)
qed
```

```
lemma foldr_isort_elements: set (foldr interval_insert_sort_lower_width xs []) = set xs
proof(induction xs)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  then show ?case
  using interval_isort_elements by(auto)
qed
```

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

lemma *interval_sort_lower_width_length*: $\langle \text{length } (\text{interval_sort_lower_width } xs) = (\text{length } xs) \rangle$

proof(*induction* xs)

case Nil

then show ?case **unfolding** *interval_sort_lower_width_def* **by** *simp*

next

case (Cons a xs)

then show ?case

unfolding *interval_sort_lower_width_def*

by (*simp add: interval_insert_sort_lower_width_length*)

qed

lemma *interval_sort_lower_width_sorted*: $\langle \text{sorted_wrt_lower } (\text{interval_sort_lower_width } xs) \rangle$

proof(*induction* xs)

case Nil

then show ?case

unfolding *interval_sort_lower_width_def* *sorted_wrt_lower_def*

by *simp*

next

case (Cons a xs)

then show ?case

unfolding *interval_sort_lower_width_def*

using *interval_insert_sort_wrt_lower*

by *auto*

qed

lemma *interval_sort_lower_width_set_eq*:

$\langle \text{set } (\text{interval_sort_lower_width } x) = \text{set } x \rangle$

by (*simp add: foldr_isort_elements interval_sort_lower_width_def*)

lemma *interval_sort_lower_width_remdups*:

$\langle \text{remdups } (\text{interval_sort_lower_width } (\text{remdups } xs)) = \text{interval_sort_lower_width } (\text{remdups } xs) \rangle$

unfolding *interval_sort_lower_width_def*

proof(*induction* xs)

case Nil

then show ?case **by** *simp*

next

case (Cons a xs)

then show ?case

by (*metis (no_types, opaque_lifting) foldr_isort_elements interval_sort_lower_width_def*

interval_sort_lower_width_length length_remdups_card_conv length_remdups_eq set_remdups)

qed

lemma *interval_sort_lower_width_distinct*:

assumes $\langle \text{distinct } xs \rangle$ **shows**

$\langle \text{distinct } (\text{interval_sort_lower_width } (\text{remdups } xs)) \rangle$

using *interval_sort_lower_width_remdups* **by** *auto*

lemma *foldr_interval_insert_sort_lower_width_distinct*:

assumes $\langle \text{distinct } zs \rangle$

shows $\langle \text{distinct } (\text{foldr } \text{interval_insert_sort_lower_width } zs \ []) \rangle$

```

proof(insert assms, induction zs)
  case Nil
  then show ?case
  by simp
next
  case (Cons a zs)
  then show ?case
  by (simp, metis (no_types, opaque_lifting) Cons.prem distinct_remdups_id foldr.simps(2)
    interval_sort_lower_width_def interval_sort_lower_width_distinct o_apply)
qed

```

```

lemma non_overlapping_sorted_remdups:
  non_overlapping_sorted xs  $\implies$  non_overlapping_sorted (remdups xs)
proof(induction xs)
  case Nil
  then show ?case
  by(simp)
next
  case (Cons a xs)
  then show ?case
  unfolding non_overlapping_sorted_def
  by(auto)
qed

```

```

lemma insert_in_lower_width:  $x \in \text{set } (\text{interval\_insert\_sort\_lower\_width } a \text{ list}) = (x = a \vee x \in \text{set list})$ 
proof(induction list)
  case Nil
  then show ?case by(simp)
next
  case (Cons a list)
  then show ?case
  unfolding interval_insert_sort_lower_width_def cmp_lower_width_def
  by auto
qed

```

```

lemma remdups_set_eq:
  assumes <set xs = set ys>
  shows <set (remdups xs) = set (remdups ys)>
  using assms by simp

```

```

lemma remdups_lower_hd:
  assumes  $xs \neq []$  and sorted_wrt_lower xs
  shows  $(\text{lower } \circ \text{hd}) (\text{remdups } xs) = (\text{lower } \circ \text{hd}) xs$ 
  using assms proof(induction xs rule:list_nonempty_induct)
  case (single x)
  then show ?case by(simp add:interval_sort_lower_width_def)
next
  case (cons x xs)
  then show ?case
  by (simp, metis cmp_lower_width_def list.collapse order.asym set_ConsD sorted_wrt.simps(2)
    sorted_wrt_lower_def sorted_wrt_lower_unroll)
qed

```

8.1.4 Various Notions of Validity of Sorted Lists of Intervals

Validity Tests

definition $\langle \text{valid_mInterval_ovl } is = (\text{sorted_wrt_lower } is \wedge \text{distinct } is \wedge is \neq []) \rangle$

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 $\langle \text{valid_mInterval_adj } is = (\text{non_overlapping_sorted } is \wedge \text{distinct } is \wedge is \neq []) \rangle$`

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

lemma `valid_adj_imp_ovl: $\langle \text{valid_mInterval_adj } x \Longrightarrow \text{valid_mInterval_ovl } x \rangle$`
`by (simp add: non_overlapping_implies_sorted_wrt_lower valid_mInterval_adj_def valid_mInterval_ovl_def)`

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)]`
`apply (simp add: valid_mInterval_adj_def non_overlapping_sorted_def cmp_non_overlapping_def)`
`by (smt (verit) lower_bounds)`

definition $\langle \text{valid_mInterval_non_ovl } is = (\text{valid_mInterval_ovl } is \wedge \text{non_adjacent_sorted_wrt_lower } is) \rangle$

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: $\langle \text{valid_mInterval_non_ovl } x \Longrightarrow \text{valid_mInterval_ovl } x \rangle$`
`using valid_mInterval_non_ovl_def by blast`
lemma `valid_non_ovl_imp_adj: $\langle \text{valid_mInterval_non_ovl } x \Longrightarrow \text{valid_mInterval_adj } x \rangle$`
`by (simp add: non_adjacent_implies_non_overlapping valid_mInterval_adj_def valid_mInterval_non_ovl_def valid_mInterval_ovl_def)`

lemma `valid_mInterval_non_ovl_sorted: $\text{valid_mInterval_non_ovl } xs \Longrightarrow \text{sorted_wrt_lower } xs$`
`by (metis (no_types, lifting) valid_mInterval_non_ovl_def valid_mInterval_ovl_def)`

lemma `valid_mInterval_non_ovl_unroll:`
 `$\langle \text{ys} \neq [] \Longrightarrow \text{valid_mInterval_non_ovl } (y \# \text{ys}) \Longrightarrow \text{valid_mInterval_non_ovl } \text{ys} \rangle$`
`unfolding valid_mInterval_non_ovl_def valid_mInterval_ovl_def non_adjacent_sorted_wrt_lower_def`
`by (auto simp add: sorted_wrt_lower_tail)`

lemma `valid_mInterval_non_ovl_eq1:`
`assumes $\langle \text{valid_mInterval_non_ovl } xs \rangle$`
`and $\langle \text{valid_mInterval_non_ovl } ys \rangle$`
`and $\langle \text{set } xs = \text{set } ys \rangle$`
`shows $\langle xs = ys \rangle$`
proof(`insert assms, induction xs ys rule: list_induct2'`)
`case 1`
`then show ?case`
`by (simp)`
`next`
`case (2 x xs)`
`then show ?case`
`by simp`
`next`

```

case (3 y ys)
then show ?case
  by simp
next
case (4 x xs y ys)
then show ?case
proof(cases xs)
  case Nil note xsNil = this
  then show ?thesis
  proof(cases ys)
    case Nil
    then show ?thesis
    using xsNil 4 by simp
  next
  case (Cons a list)
  then show ?thesis
  using xsNil 4
  by (simp add: valid_mInterval_non_ovl_def valid_mInterval_ovl_def)
qed
next
case (Cons x' xs') note xCons = this
then show ?thesis
proof(cases ys)
  case Nil
  then show ?thesis
  using xCons 4
  by (simp add: valid_mInterval_non_ovl_def valid_mInterval_ovl_def)
next
case (Cons y' ys')
then show ?thesis
  using xCons 4 valid_mInterval_non_ovl_unroll[of xs x] valid_mInterval_non_ovl_unroll[of ys y]
  unfolding cmp_non_adjacent_def non_adjacent_sorted_wrt_lower_def valid_mInterval_non_ovl_def
    valid_mInterval_ovl_def sorted_wrt_lower_def cmp_lower_width_def
  by (metis (no_types, lifting) 4.prem1 4.prem2 cmp_non_adjacent_lower insert_eq_iff
    list.distinct(1) list.set_intros(1) list.simps(15) non_adjacent_sorted_wrt_lower_def
    order.asym set_ConsD sorted_wrt.simps(2) valid_mInterval_non_ovl_def)
qed
qed
qed

```

Constructors

Overlapping Intervals **definition** $\langle mk_mInterval_ovl = remdups\ o\ interval_sort_lower_width \rangle$

```

lemma mk_mInterval_ovl_non_empty:  $\langle is \neq [] \implies (mk\_mInterval\_ovl\ is) \neq [] \rangle$ 
unfolding mk_mInterval_ovl_def o_def interval_sort_lower_width_def
proof(induction is rule: list_nonempty_induct)
case (single x)
then show ?case by simp
next
case (cons x xs)
then show ?case
  apply(simp)
  by (metis (no_types, lifting) interval_insert_sort_lower_width.simps(2) list.collapse list.simps(3))

```

qed

lemma *mk_mInterval_ovl_empty*[simp]:
mk_mInterval_ovl [] = []
by(simp add: *mk_mInterval_ovl_def interval_sort_lower_width_def*)

lemma *mk_mInterval_ovl_distinct*: <distinct (*mk_mInterval_ovl is*)>
unfolding *mk_mInterval_ovl_def* **by** simp

lemma *sorted_wrt_lower_remdups*:
sorted_wrt_lower xs \implies *sorted_wrt_lower (remdups xs)*
proof(induction *xs*)
 case Nil
 then show ?case
 by(simp)
 next
 case (Cons *a xs*)
 then show ?case
 unfolding *sorted_wrt_lower_def*
 by(auto)
qed

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(induction *xs*)
 case Nil
 then show ?case
 by (metis *interval_sort_lower_width_remdups remdups.simps(1)*)
 next
 case (Cons *a xs*)
 then show ?case
 by (metis (mono_tags, lifting) *distinct_remdups interval_sort_lower_width_remdups interval_sort_lower_width_set_eq interval_sort_lower_width_sorted set_remdups sorted_wrt_lower_distinct_lists_eq sorted_wrt_lower_remdups*)
qed

lemma *mk_mInterval_ovl_sorted*: <*sorted_wrt_lower (mk_mInterval_ovl is)*>
proof(induction *is*)
 case Nil
 then show ?case
 unfolding *mk_mInterval_ovl_def sorted_wrt_lower_def interval_sort_lower_width_def*
 by simp
 next
 case (Cons *a is*)
 then show ?case
 using *interval_sort_lower_width_sorted sorted_wrt_lower_remdups*
 unfolding *mk_mInterval_ovl_def sorted_wrt_lower_def interval_sort_lower_width_def o_def*
 by blast
qed

theorem *mk_mInterval_ovl_valid*: <*is* \neq [] \implies *valid_mInterval_ovl (mk_mInterval_ovl is)*>
unfolding *valid_mInterval_ovl_def*

by (simp add: mk_mInterval_ovl_distinct mk_mInterval_ovl_non_empty mk_mInterval_ovl_sorted)

lemma valid_mk_mInterval_ovl_id:

assumes <valid_mInterval_ovl xs>

shows <mk_mInterval_ovl xs = xs>

proof(insert assms, induction xs)

case Nil

then show ?case

by(simp add: valid_mInterval_ovl_def)

next

case (Cons a xs)

then show ?case

apply(simp add: mk_mInterval_ovl_def interval_sort_lower_width_def valid_mInterval_ovl_def
sorted_wrt_lower_def cmp_lower_width_def)

by (metis (no_types, opaque_lifting) cmp_lower_width_def distinct_singleton foldr_isort_elements
insertI1 interval_insert_sort_lower_width.simps(1) interval_insert_sort_lower_width.simps(2)
interval_sort_lower_width_def list.distinct(1) list.simps(15) min_list.cases remdups.simps(2)
remdups_id_iff_distinct)

qed

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 >

by (metis (no_types, lifting) assms comp_def interval_sort_lower_width_set_eq mk_mInterval_ovl_def
mk_mInterval_ovl_def mk_mInterval_ovl_distinct mk_mInterval_ovl_sorted set_remdups
sorted_wrt_lower_distinct_lists_eq)

Adjacent Intervals **definition** mk_mInterval_adj :: ('a::{minus, linorder, linorder}) interval list \Rightarrow '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))>

using interval_sort_lower_width_sorted sorted_wrt_lower_remdups mk_mInterval_ovl_sorted[of is]
merge_overlapping_intervals_sorted_wrt_lower_sorted_lower[of (mk_mInterval_ovl is)]
merge_overlapping_intervals_sorted_sorted_non_non_overlapping non_overlapping_sorted_remdups

unfolding mk_mInterval_adj_def o_def

by blast

lemma mk_mInterval_adj_sorted: <sorted_wrt_lower (mk_mInterval_adj (is::'a::minus_mono interval list))>

using interval_sort_lower_width_sorted sorted_wrt_lower_remdups mk_mInterval_ovl_sorted[of is]
merge_overlapping_intervals_sorted_wrt_lower_sorted_lower[of (mk_mInterval_ovl is)]

unfolding mk_mInterval_adj_def o_def

by auto

lemma mk_mInterval_adj_non_empty: <is \neq [] \implies (mk_mInterval_adj is) \neq []>

unfolding mk_mInterval_adj_def o_def

using mk_mInterval_ovl_non_empty merge_overlapping_intervals_sorted_wrt_lower_non_nil

by (metis remdups_eq_nil_right_iff)

lemma mk_mInterval_adj_empty[simp]:

mk_mInterval_adj [] = []

by(simp add: mk_mInterval_adj_def)

lemma mk_mInterval_adj_distinct: <distinct (mk_mInterval_adj is)>

unfolding `mk_mInterval_adj_def` **by** `simp`

theorem `mk_mInterval_adj_valid`: $\langle is \neq [] \implies valid_mInterval_adj (mk_mInterval_adj\ is) \rangle$

unfolding `valid_mInterval_adj_def`

using `mk_mInterval_adj_non_overlapping_sorted` `mk_mInterval_adj_distinct` `mk_mInterval_adj_non_empty`
by `auto`

lemma `valid_mk_mInterval_adj_id`:

assumes $\langle valid_mInterval_adj\ xs \rangle$

shows $\langle mk_mInterval_adj\ xs = xs \rangle$

proof(`insert` `assms`, `induction` `xs`)

case `Nil`

then show `?case`

by(`simp` `add`: `valid_mInterval_adj_def`)

next

case (`Cons` `a` `xs`)

then show `?case`

using `valid_mk_mInterval_ovl_id`[`of` `a#xs`, `simplified` `valid_adj_imp_ovl`[`of` `a#xs`, `simplified` `Cons`, `simplified`], `simplified`]

`valid_mk_mInterval_ovl_id`[`of` `xs`, `simplified` `valid_adj_imp_ovl`[`of` `xs`, `simplified` `Cons`, `simplified`], `simplified`]

unfolding `valid_mInterval_adj_def` `mk_mInterval_adj_def` `o_def`

apply(`simp` `add`: `mk_mInterval_ovl_def` `interval_sort_lower_width_def` `valid_mInterval_ovl_def`

`sorted_wrt_lower_def` `cmp_lower_width_def` `non_overlapping_sorted_def` `cmp_non_overlapping_def`)

by (`smt` (`verit`) `One_nat_def` `add.commute` `foldr_interval_insert_sort_lower_width_distinct` `foldr_isort_elements`

`interval_insert_sort_lower_width_simps(2)` `interval_insert_sort_lower_width_length` `length_remdups_card_conv`

`length_remdups_eq` `list.sel(1)` `list.sel(3)` `list.set_intros(1)` `list.size(4)`

`merge_overlapping_intervals_sorted_wrt_lower.elims`

`merge_overlapping_intervals_sorted_wrt_lower_simps(2)` `remdups_simps(2)` `remdups_id_iff_distinct` `set_remdups`)

qed

lemma `mk_mInterval_adj_eq`:

assumes $\langle set\ xs = set\ (ys::\{minus_mono,\ linordered_field\}\ interval\ list) \rangle$

shows $\langle mk_mInterval_adj\ xs = mk_mInterval_adj\ ys \rangle$

by (`metis` (`no_types`, `lifting`) `assms` `comp_def` `interval_sort_lower_width_set_eq` `mk_mInterval_adj_def`

`mk_mInterval_ovl_def` `mk_mInterval_ovl_distinct` `mk_mInterval_ovl_sorted` `set_remdups`

`sorted_wrt_lower_distinct_lists_eq`)

lemma `mk_mInterval_ovl_id`:

`mk_mInterval_ovl` (`mk_mInterval_ovl` `x`) = `mk_mInterval_ovl` `x`

unfolding `mk_mInterval_ovl_def` `o_def`

by (`metis` `comp_def` `mk_mInterval_ovl_def` `mk_mInterval_ovl_empty` `mk_mInterval_ovl_valid`

`valid_mk_mInterval_ovl_id`)

value `valid_mInterval_adj` (`mk_mInterval_adj` (`[lvl` (`1::int`) `2`, `lvl` `1` `3`, `lvl` `1` `1]`))

value `valid_mInterval_adj` (`mk_mInterval_adj` (`[lvl` (`1::int`) `1`, `lvl` `1` `2`, `lvl` `2` `3]`))

Non-Overlapping Intervals definition $\langle mk_mInterval_non_ovl = remdups\ o\ merge_adjacent_intervals_sorted_wrt_lower\ o\ mk_mInterval_ovl \rangle$

lemma `mk_mInterval_non_ovl_distinct`:

`distinct` (`mk_mInterval_non_ovl` `is`)

by (`simp` `add`: `mk_mInterval_non_ovl_def`)

lemma *mk_mInterval_non_ovl_non_empty*:
 $is \neq [] \implies mk_mInterval_non_ovl\ is \neq []$
by (*simp* *add*: *merge_adjacent_intervals_sorted_wrt_lower_non_nil* *mk_mInterval_ovl_non_empty*
mk_mInterval_non_ovl_def)

lemma *mk_mInterval_non_ovl_empty*[*simp*]:
 $mk_mInterval_non_ovl\ [] = []$
by (*simp* *add*: *merge_adjacent_intervals_sorted_wrt_lower_non_nil* *mk_mInterval_non_ovl_def*)

lemma *mk_mInterval_non_ovl_eq*:
assumes $\langle set\ xs = set\ (ys::'a::\{minus_mono,\ linordered_field\})\ interval\ list \rangle$
shows $\langle mk_mInterval_non_ovl\ xs = mk_mInterval_non_ovl\ ys \rangle$
unfolding *mk_mInterval_non_ovl_def* *o_def*
by (*metis* (*no_types*, *opaque_lifting*) *assms* *comp_apply* *foldr_isort_elements*
interval_sort_lower_width_def *mk_mInterval_ovl_def* *mk_mInterval_ovl_distinct*
mk_mInterval_ovl_sorted *set_remdups* *sorted_wrt_lower_distinct_lists_eq*)

lemma *sorted_wrt_lower_merge_adjacent_intervals_sorted_wrt_lower*:
 $sorted_wrt_lower\ xs \implies sorted_wrt_lower\ (merge_adjacent_intervals_sorted_wrt_lower\ xs)$
proof(*induction* *xs* *rule*:*merge_adjacent_intervals_sorted_wrt_lower.induct*)
case 1
then show ?*case* **by** *simp*
next
case (2 *x*)
then show ?*case* **by** *simp*
next
case (3 *x* *y* *ys*)
then show ?*case*
proof(*cases* *upper* *x* < *lower* *y*)
case *True*
then show ?*thesis*
using 3
by (*smt* (*verit*, *del_insts*) *leD* *list.discl* *list.sel*(1) *lower_le_upper*
merge_adjacent_intervals_sorted_wrt_lower.simps(3)
merge_adjacent_intervals_sorted_wrt_lower_non_nil
merge_adjacent_intervals_sorted_wrt_lower_sorted_lower_lower_hd
sorted_wrt_lower_unroll)
next
case *False*
then show ?*thesis*
using 3
by (*smt* (*verit*, *del_insts*) *list.distinct*(1) *list.sel*(1) *max.absorb3* *max.absorb2*
merge_adjacent_intervals_sorted_wrt_lower.simps(3) *mk_interval_id* *mk_interval_lower*
not_le_imp_less *sorted_wrt_lower_tail'* *sorted_wrt_lower_unroll*)
qed
qed

lemma *mk_mInterval_non_ovl_sorted_wrt_lower*:
 $is \neq [] \implies sorted_wrt_lower\ (mk_mInterval_non_ovl\ (is::int\ interval\ list))$
unfolding *mk_mInterval_non_ovl_def* *o_def*
using *mk_mInterval_ovl_sorted* *sorted_wrt_lower_merge_adjacent_intervals_sorted_wrt_lower*

sorted_wrt_lower_remdups **by** blast

lemma valid_ovl_mkInterval_non_ovl: is \neq [] \implies valid_mInterval_ovl (mk_mInterval_non_ovl is)

unfolding valid_mInterval_ovl_def

by (simp add: merge_adjacent_intervals_sorted_wrt_lower_non_nil mk_mInterval_non_ovl_def
mk_mInterval_ovl_non_empty mk_mInterval_ovl_sorted)

sorted_wrt_lower_merge_adjacent_intervals_sorted_wrt_lower
sorted_wrt_lower_remdups)

lemma non_adj_sorted_mkInterval_non_ovl:

sorted_wrt_lower xs

\implies non_adjacent_sorted_wrt_lower (merge_adjacent_intervals_sorted_wrt_lower xs)

proof(induction xs rule:merge_adjacent_intervals_sorted_wrt_lower.induct)

case 1

then show ?case

unfolding non_adjacent_sorted_wrt_lower_def **by**(simp)

next

case (2 x)

then show ?case

unfolding non_adjacent_sorted_wrt_lower_def **by**(simp)

next

case (3 x y ys)

then show ?case

proof(cases upper x < lower y)

case True

then show ?thesis

apply(simp)

apply(subst non_adjacent_sorted_wrt_lower_unroll)

subgoal by (simp add: merge_adjacent_intervals_sorted_wrt_lower_non_nil)

subgoal using 3 True

by (metis list.sel(1) merge_adjacent_intervals_sorted_wrt_lower_sorted_lower_lower_hd sorted_wrt_lower_tail)

done

next

case False

then show ?thesis

proof(cases ys)

case Nil

then show ?thesis

unfolding non_adjacent_sorted_wrt_lower_def cmp_non_adjacent_def

by(simp add: max_def 3 False split:if_splits)

next

case (Cons a list)

then have a: sorted_wrt_lower (mk_interval (lower x, max (upper y) (upper x)) # ys)

proof(cases upper y \leq upper x)

case True **note** * = this

then show ?thesis

proof (cases upper y = upper x)

case True

then show ?thesis

using False True **apply**(simp)

using 3 * False sorted_wrt_lower_unroll[of y#ys x, simplified]

sorted_wrt_lower_tail[of y ys]

sorted_wrt_lower_tail'[of x y ys]

by auto

```

next
case False
then show ?thesis
using False True apply(simp)
using 3 * False sorted_wrt_lower_unroll[of y#ys x, simplified]
sorted_wrt_lower_tail[of y ys]
sorted_wrt_lower_tail'[of x y ys]
by auto
qed
next
case False
then show ?thesis
using Cons False apply(simp)
using 3 False sorted_wrt_lower_unroll[of y#ys x, simplified]
sorted_wrt_lower_tail[of y ys]
sorted_wrt_lower_tail'[of x y ys]
apply(simp)
by (smt (verit, ccfv_threshold) less_le_not_le list.distinct(1) list.sel(1) max.absorb1
max.assoc max.cobounded2 mk_interval_id mk_interval_lower sorted_wrt_lower_unroll)
qed
then show ?thesis
using False 3[simplified a False , simplified]
by simp
qed
qed
qed

```

lemma *bin_op_mInterval_commute:*

```

assumes op_commute: <math>\langle \bigwedge x y. op\ x\ y = op\ y\ x \rangle
shows <math>\langle mk\_mInterval\_non\_ovl\ (bin\_op\_interval\_list\ op\ x\ y) = mk\_mInterval\_non\_ovl\ (bin\_op\_interval\_list\ op\ y\ x) \rangle
using bin\_op\_interval\_list\_commute\ mk\_mInterval\_non\_ovl\_eq
by (metis op\_commute)

```

lemma *iList_plus_mInterval_ovl_assoc:*

```

<math>\langle 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) \rangle
by (meson iList\_plus\_assoc\ mk\_mInterval\_ovl\_eq)

```

lemma *iList_plus_mInterval_adj_commute:*

```

<math>\langle mk\_mInterval\_adj\ (iList\_plus\ x\ y) = mk\_mInterval\_adj\ (iList\_plus\ y\ (x::'a::\{minus\_mono,\ linordered\_field\}\ interval\ list)) \rangle
by (meson iList\_plus\_commute\ mk\_mInterval\_adj\_eq)

```

lemma *iList_plus_mInterval_non_ovl_assoc:*

```

<math>\langle 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) \rangle
by (meson iList\_plus\_assoc\ mk\_mInterval\_non\_ovl\_eq)

```

lemma *iList_plus_mInterval_non_ovl_commute:*

```

<math>\langle mk\_mInterval\_non\_ovl\ (iList\_plus\ x\ y) = mk\_mInterval\_non\_ovl\ (iList\_plus\ y\ (x::'a::\{minus\_mono,\ linordered\_field\}\ interval\ list)) \rangle

```

by (meson iList_plus_commute mk_mInterval_non_ovl_eq)

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)⟩

by (meson iList_plus_assoc mk_mInterval_non_ovl_eq)

lemma sorted_wrt_lower_mk_mInterval_non_ovl: sorted_wrt_lower (mk_mInterval_non_ovl xs)

unfolding mk_mInterval_non_ovl_def

by (simp add: mk_mInterval_ovl_sorted sorted_wrt_lower_merge_adjacent_intervals_sorted_wrt_lower sorted_wrt_lower_remdups)

theorem mk_mInterval_non_ovl_valid: ⟨sorted_wrt_lower is \implies is \neq [] \implies valid_mInterval_non_ovl (mk_mInterval_non_ovl is)⟩

using non_adj_sorted_mkInterval_non_ovl[of is] valid_ovl_mkInterval_non_ovl[of is]

unfolding valid_mInterval_non_ovl_def mk_mInterval_non_ovl_def o_def

by (simp add: distinct_remdups_id mk_mInterval_ovl_sorted non_adj_sorted_mkInterval_non_ovl non_adjacent_implies_distinct)

lemma valid_mk_mInterval_non_ovl_id:

assumes ⟨valid_mInterval_non_ovl xs⟩

shows ⟨mk_mInterval_non_ovl xs = xs⟩

proof(insert assms, induction xs)

case Nil

then show ?case

by(simp add: valid_mInterval_non_ovl_def valid_mInterval_ovl_def)

next

case (Cons a xs)

then show ?case

using valid_mk_mInterval_ovl_id[of a#xs, simplified] valid_non_ovl_imp_ovl[of a#xs, simplified] Cons, simplified], simplified]

valid_mk_mInterval_ovl_id[of xs, simplified] valid_non_ovl_imp_ovl[of xs, simplified] Cons, simplified], simplified]

unfolding valid_mInterval_non_ovl_def mk_mInterval_non_ovl_def o_def

apply(simp)

apply(simp add: mk_mInterval_ovl_def interval_sort_lower_width_def valid_mInterval_ovl_def

sorted_wrt_lower_def cmp_lower_width_def non_overlapping_sorted_def cmp_non_overlapping_def)

by (smt (verit) Cons.prem1 list.sel(1) list.sel(3) merge_adjacent_intervals_sorted_wrt_lower.elims

merge_adjacent_intervals_sorted_wrt_lower.simps(2) non_adj_sorted_mkInterval_non_ovl

non_adjacent_implies_distinct

non_adjacent_sorted_wrt_lower_def non_adjacent_sorted_wrt_lower_unroll remdups_id_iff_distinct

sorted_wrt_lower.simps(2)

valid_mInterval_non_ovl_sorted)

qed

lemma mk_mInterval_non_ovl_single:

mk_mInterval_non_ovl [x] = [x]

by (simp add: non_adjacent_implies_sorted_wrt_lower non_adjacent_sorted_wrt_lower_def

valid_mInterval_non_ovl_def valid_mInterval_ovl_def valid_mk_mInterval_non_ovl_id)

lemma mk_mInterval_non_ovl_id:

mk_mInterval_non_ovl (mk_mInterval_non_ovl x) = mk_mInterval_non_ovl x

unfolding mk_mInterval_non_ovl_def o_def

by (metis (no_types, opaque_lifting) comp_apply distinct_remdups_id mk_mInterval_non_ovl_def mk_mInterval_non_ovl_empty)

*mk_mInterval_ovl_sorted non_adj_sorted_mkInterval_non_ovl non_adjacent_implies_distinct
valid_mInterval_non_ovl_def valid_mk_mInterval_non_ovl_id valid_ovl_mkInterval_non_ovl*

value *valid_mInterval_non_ovl* (*mk_mInterval_non_ovl* ([*lvl* (1::int) 2, *lvl* 1 3, *lvl* 1 1]))
value *valid_mInterval_non_ovl* (*mk_mInterval_non_ovl* ([*lvl* (1::int) 1, *lvl* 1 2, *lvl* 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* $XS = \text{foldr } (\lambda x a. \text{set_of } x \cup a) \text{ } XS \ \{\}$ >

lemma *set_of_interval_list_nonempty*:
assumes *non_empty*: $\langle XS \neq ([]::\text{real interval list}) \rangle$
shows $\langle \text{set_of_interval_list } XS \neq \{\} \rangle$
using *assms* **proof**(*induction* XS *rule*:*list_nonempty_induct*)
case (*single* x)
then show ?*case* **by**(*simp* *add*:*set_of_interval_list_def* *set_of_eq*)
next
case (*cons* x xs)
then show ?*case*
by(*simp* *add*:*set_of_interval_list_def*)
qed

lemma *set_of_interval_list_bdd_below*:
assumes *non_empty*: $\langle XS \neq ([]::\text{real interval list}) \rangle$
shows $\langle \text{bdd_below } (\text{set_of_interval_list } XS) \rangle$
using *assms* **proof**(*induction* XS *rule*:*list_nonempty_induct*)
case (*single* x)
then show ?*case* **by**(*simp* *add*:*set_of_interval_list_def* *set_of_eq*)
next
case (*cons* x xs)
then show ?*case*
by(*simp* *add*:*set_of_eq* *set_of_interval_list_def* *sorted_wrt_lower_def*)
qed

lemma *set_of_interval_list_bdd_above*:
assumes *non_empty*: $\langle XS \neq ([]::\text{real interval list}) \rangle$
shows $\langle \text{bdd_above } (\text{set_of_interval_list } XS) \rangle$
using *assms* **proof**(*induction* XS *rule*:*list_nonempty_induct*)
case (*single* x)
then show ?*case* **by**(*simp* *add*:*set_of_interval_list_def* *set_of_eq*)
next
case (*cons* x xs)
then show ?*case*
by(*simp* *add*:*set_of_eq* *set_of_interval_list_def* *contiguous_def*)

qed

```
lemma inf_set_of_interval_list_lower:
  assumes non_empty:  $\langle XS \neq [] :: \text{real interval list} \rangle$ 
  and sorted:  $\langle \text{sorted\_wrt\_lower } XS \rangle$ 
  shows  $\langle \text{Inf } (\text{set\_of\_interval\_list } XS) = \text{lower } (\text{hd } XS) \rangle$ 
  using assms proof (induction XS rule: list_nonempty_induct)
  case (single x)
  then show ?case by (simp add: set_of_interval_list_def set_of_eq)
next
case (cons x xs)
then show ?case
  apply (simp add: set_of_interval_list_def)
  apply (subst cInf_union_distrib)
  subgoal by simp
  subgoal by (simp add: set_of_eq)
  subgoal by (fold set_of_interval_list_def, simp add: set_of_interval_list_nonempty)
  subgoal
    apply (fold set_of_interval_list_def)
    using sorted_wrt_lower_unroll [of xs x, simplified cons, simplified]
      set_of_interval_list_bdd_below
    by (simp)
  subgoal
    using sorted_wrt_lower_unroll [of xs x, simplified cons, simplified]
    by (metis inf.orderE interval_bounds_real(2) lower_le_upper nle_le order_less_le set_of_eq)
  done
qed
```

```
lemma contiguous_sorted_wrt_upper:
  assumes contiguous (xs: real interval list)
  shows sorted_wrt_upper xs
  unfolding sorted_wrt_upper_def
  using assms unfolding contiguous_def lower_le_upper
  by (metis assms contiguous_non_overlapping non_overlapping_implies_sorted_wrt_upper sorted_wrt_upper_def)
```

```
lemma contiguous_sorted_wrt_lower:
  assumes contiguous (XS: real interval list)
  shows sorted_wrt_lower XS
  unfolding sorted_wrt_lower_def
  using assms contiguous_non_overlapping [of XS] non_overlapping_implies_sorted_wrt_lower
  sorted_wrt_lower_def by metis
```

```
lemma max_last_sorted_wrt_upper:
  assumes  $XS \neq []$  sorted_wrt_upper (XS: 'a: {linorder} interval list)
  shows  $\text{Max } (\text{set } (\text{map upper } XS)) = \text{upper } (\text{last } XS)$ 
  using assms
proof (induction XS rule: list_nonempty_induct)
  case (single x)
  then show ?case by simp
next
case (cons x xs)
then have a0: sorted_wrt  $(\lambda x y. \text{upper } x \leq \text{upper } y)$  (x # xs) by (simp add: sorted_wrt_upper_def)
then have a1:  $\text{Max } (\text{set } (\text{map upper } (x \# xs))) = \text{upper } (\text{last } (x \# xs))$  using cons unfolding sorted_wrt_upper_def by
```

simp

then show ?case **using** *ao a1 cons unfolding sorted_wrt_upper_def by simp*
qed

lemma *min_hd_sorted_wrt_lower*:

assumes $XS \neq []$ *sorted_wrt_lower* ($XS:: 'a::\{\text{linorder, minus, preorder}\}$ interval list)

shows $\text{Min}(\text{set}(\text{map lower } XS)) = \text{lower}(\text{hd } XS)$

using *assms*

proof (*induction XS rule: list_nonempty_induct*)

case (*single x*)

then show ?case **by** *simp*

next

case (*cons x xs*)

then have *ao*: *sorted_wrt* ($\lambda x y. \text{if lower } x = \text{lower } y \text{ then width } x \leq \text{width } y \text{ else lower } x < \text{lower } y$) ($x \# xs$)

unfolding *sorted_wrt_lower_def cmp_lower_width_def by simp*

then have *a1*: $\text{Min}(\text{set}(\text{map lower} (x \# xs))) = \text{lower}(\text{hd} (x \# xs))$ **using** *cons unfolding sorted_wrt_lower_def cmp_lower_width_def*

by (*smt (verit, del_insts) arg_min_list.simps(2) cons.IH cons.prem1 f_arg_min_list_f image_set list.distinct(1)*)

list.exhaust_sel list.sel(1) order_less_imp_le sorted_wrt_lower_unroll)

then show ?case **using** *ao a1 cons unfolding sorted_wrt_upper_def by simp*

qed

lemma *lower_isort*:

assumes $\langle xs \neq [] \rangle$ **and** $\langle (\text{lower } \circ \text{hd}) xs = \text{Min}(\text{lower } \langle \text{set } xs \rangle) \rangle$

shows $\langle (\text{lower } \circ \text{hd}) (\text{interval_sort_lower_width } xs) = (\text{lower } \circ \text{hd}) xs \rangle$

proof(*insert assms, induction xs rule: list_nonempty_induct*)

case (*single x*)

then show ?case **by**(*simp add: interval_sort_lower_width_def*)

next

case (*cons x xs*)

then show ?case

apply(*simp add: interval_sort_lower_width_def*)

by (*metis (no_types, opaque_lifting) comp_eq_dest_lhs cons.prem1 foldr.simps(2) foldr_isort_elements interval_insert_sort_lower_width_nonempty interval_sort_lower_width_def interval_sort_lower_width_sorted list.sel(1) list.set_map min_hd_sorted_wrt_lower*)

qed

lemma *min_sort*:

$\text{Min}(\text{set}(\text{map lower}(\text{foldr interval_insert_sort_lower_width } xs []))) = \text{Min}(\text{set}(\text{map lower } xs))$

using *foldr_isort_elements*

by (*metis list.set_map*)

lemma *mk_mInterval_lower*:

assumes $xs \neq []$

shows $\text{Min}(\text{set}(\text{map lower}(\text{mk_mInterval_non_ovl } xs))) = \text{Min}(\text{set}(\text{map lower } xs))$

proof (*induction xs*)

case *Nil*

then show ?case

unfolding *mk_mInterval_non_ovl_def mk_mInterval_ovl_def interval_sort_lower_width_def*

by *simp*

next

case (*Cons a xs*)

have *a*: $\text{Min}(\text{set}(\text{map lower}(\text{foldr interval_insert_sort_lower_width } xs []))) = \text{Min}(\text{set}(\text{map lower } xs))$

using *foldr_isort_elements*


```

by (metis list.set_map)
then show ?case
unfolding mk_mInterval_non_ovl_def mk_mInterval_ovl_def o_def
using Cons
by (smt (verit, ccfv_SIG) foldr_isort_elements interval_sort_lower_width_def interval_sort_lower_width_sorted
  list.discl list.set_map merge_adjacent_intervals_sorted_wrt_lower_non_nil set_empty
  merge_adjacent_intervals_sorted_wrt_lower_sorted_lower_lower_hd min_hd_sorted_wrt_lower
  set_remdups sorted_wrt_lower_merge_adjacent_intervals_sorted_wrt_lower sorted_wrt_lower_remdups)
qed

lemma sup_set_of_interval_list_upper:
  assumes non_empty:  $\langle XS \neq [] :: \text{real interval list} \rangle$ 
  and sorted:  $\langle \text{sorted\_wrt\_upper } XS \rangle$ 
shows  $\langle \text{Sup (set\_of\_interval\_list } XS) = \text{upper (last } XS) \rangle$ 
using assms proof(induction XS rule:list_nonempty_induct)
  case (single x)
  then show ?case by(simp add:set_of_interval_list_def set_of_eq)
next
case (cons x xs)
have Max (set (map upper (x # xs))) = upper (last (x # xs)) using max_last_sorted_wrt_upper assms cons by blast
then show ?case
  apply(simp add: set_of_interval_list_def)
  apply(subst cSup_union_distrib)
  apply(fold set_of_interval_list_def)
  subgoal by simp
  subgoal using bdd_above_set_of by metis
  subgoal using cons.hyps set_of_interval_list_nonempty by presburger
  subgoal using cons.hyps set_of_interval_list_bdd_above by presburger
  subgoal
  proof —
    assume a1: Max (insert (upper x) (upper 'set xs)) = upper (if xs = [] then x else last xs)
    have f2:  $\forall i. \text{Sup (set\_of } i) = \text{upper } i :: \text{real}$ 
      by (simp add: set_of_eq)
    have [] = xs  $\vee$  sorted_wrt_upper xs
      by (metis cons.prem sorted_wrt_upper_unroll)
    then show ?thesis
      using f2 a1 cons.IH sup_real_def by force
  qed
done
qed

lemma compact_set_of_interval_list:
   $\langle \text{compact (set\_of\_interval\_list (XS :: ('a :: \{\text{preorder, ordered\_euclidean\_space, topological\_space}\} \text{ interval list}))) \rangle$ 
proof(induction XS)
  case Nil
  then show ?case by(simp add:set_of_interval_list_def)
next
case (Cons a XS)
then show ?case
  by(simp add:set_of_interval_list_def, subst compact_Un, simp_all add: compact_set_of Cons set_of_eq)
qed

lemma lower_le_upper_aux:  $\langle xs \neq [] \implies \text{non\_overlapping\_sorted } xs \implies \text{lower (hd } xs) \leq \text{upper (last } xs) \rangle$ 
proof(induction xs rule:induct_list012)

```

```

case 1
then show ?case by(simp)
next
case (2 x)
then show ?case by(simp)
next
case (3 x y zs)
then show ?case
  proof(cases zs)
    case Nil
      then show ?thesis
        using 3 dual_order.trans
        unfolding non_overlapping_sorted_def cmp_non_overlapping_def
        by (simp, smt (verit, best) dual_order.trans lower_le_upper)
    next
      case (Cons a list)
        then show ?thesis
          using 3 dual_order.trans
          unfolding non_overlapping_sorted_def cmp_non_overlapping_def
          by (simp, smt (verit, best) dual_order.trans lower_le_upper)
  qed
qed

```

lemma contiguous_lower_le_upper:
assumes non_empty: $\langle XS \neq ([] :: \text{real interval list}) \rangle$
and contiguous: $\langle \text{contiguous } XS \rangle$
shows $\langle (\text{lower } (\text{hd } XS)) \leq (\text{upper } (\text{last } XS)) \rangle$
by (simp add: contiguous contiguous_non_overlapping lower_le_upper_aux non_empty)

lemma diameter_Sup_Inf:
assumes $\langle \text{compact } X \rangle \langle X \neq \{\} \rangle$
shows $\langle \text{diameter } X \leq \text{Sup } X - \text{Inf } X \rangle$
using assms diameter_compact_attained[of X]
by (metis bounded_imp_bdd_above bounded_imp_bdd_below compact_imp_bounded sup_inf_dist_bounded)

lemma diameter_width_compact:
assumes $\langle \text{compact } X \rangle \langle \text{bdd_below } X \rangle \langle \text{bdd_above } X \rangle \langle X \neq \{\} \rangle$
shows $\langle \text{diameter } X = \text{Sup } X - \text{Inf } X \rangle$
using assms diameter_Sup_Inf[of X, simplified assms, simplified]
 closed_contains_Inf[of X, simplified assms, simplified]
 closed_contains_Sup[of X, simplified assms, simplified]
by (smt (verit, best) compact_imp_bounded compact_imp_closed diameter_bounded_bound dist_real_def)

lemma diameter_contiguous:
assumes non_empty: $\langle XS \neq ([] :: \text{real interval list}) \rangle$
and contiguous: $\langle \text{contiguous } XS \rangle$
shows $\langle \text{diameter } (\text{set_of_interval_list } XS) = \text{dist } (\text{lower } (\text{hd } XS)) (\text{upper } (\text{last } XS)) \rangle$
apply(subst diameter_width_compact)
subgoal
by (simp add: compact_set_of_interval_list contiguous non_empty)
subgoal
by (simp add: $\langle \text{compact } (\text{set_of_interval_list } XS) \rangle$ bounded_imp_bdd_below compact_imp_bounded)
subgoal

```

  by (simp add: contiguous set_of_interval_list_bdd_above non_empty)
subgoal
  by (simp add: set_of_interval_list_nonempty non_empty)
subgoal
  using sup_set_of_interval_list_upper[of XS, simplified assms, simplified]
    inf_set_of_interval_list_lower[of XS, simplified assms, simplified]
    contiguous_lower_le_upper[of XS, simplified assms, simplified]
  unfolding dist_real_def abs_real_def
    by (simp add: contiguous contiguous_non_overlapping contiguous_sorted_wrt_upper
non_overlapping_implies_sorted_wrt_lower)
done

```

```

lemma interval_list_union_contiguous_lower:
  assumes non_empty: <XS ≠ []>
  and sorted: <sorted_wrt_lower XS>
  shows <lower (interval_list_union XS) = lower (hd XS)>
  using assms proof(induction XS rule:interval_list_union.induct)
  case 1
  then show ?case by simp
next
  case (2 l)
  then show ?case by simp
next
  case (3 l v va)
  then show ?case
  unfolding sorted_wrt_lower_def cmp_lower_width_def
  apply (simp)
  by (metis inf.idem inf.strict_order_iff)
qed

```

```

lemma interval_list_union_contiguous_upper:
  assumes non_empty: <XS ≠ []>
  and sorted: <sorted_wrt_upper XS>
  shows <upper (interval_list_union XS) = upper (last XS)>
  using assms proof(induction XS rule:interval_list_union.induct)
  case 1
  then show ?case by simp
next
  case (2 l)
  then show ?case by simp
next
  case (3 l v va)
  then show ?case
  unfolding sorted_wrt_upper_def cmp_lower_width_def
  apply (simp)
  by (metis last_in_set sup.absorb2)
qed

```

```

lemma interval_list_union_contiguous:
  assumes non_empty: <XS ≠ []>
  and sorted_lower: <sorted_wrt_lower XS>
  and sorted_upper: <sorted_wrt_upper XS>

```

shows $\langle \text{interval_list_union } XS = \text{Interval } (\text{lower } (\text{hd } XS), \text{upper } (\text{last } XS)) \rangle$
by (metis Interval_id assms interval_list_union_contiguous_lower interval_list_union_contiguous_upper non_empty)

lemma contiguous_bounds_lower:

assumes non_empty: $\langle XS \neq [] \rangle$
and contiguous: $\langle \text{contiguous } (XS::\text{real interval list}) \rangle$
shows $\text{lower } (\text{hd } XS) = \text{Min } (\text{set } (\text{map } \text{lower } XS))$
using min_hd_sorted_wrt_lower[of XS, simplified assms contiguous_sorted_wrt_lower[of XS, simplified assms]]
by auto[1]

lemma contiguous_bounds_upper:

assumes non_empty: $\langle XS \neq [] \rangle$
and contiguous: $\langle \text{contiguous } (XS::\text{real interval list}) \rangle$
shows $\text{upper } (\text{last } XS) = \text{Max } (\text{set } (\text{map } \text{upper } XS))$
using max_last_sorted_wrt_upper[of XS, simplified assms contiguous_sorted_wrt_upper[of XS, simplified assms]]
by auto[1]

lemma set_of_interval_list_contiguous:

assumes non_empty: $\langle XS \neq ([]::\text{real interval list}) \rangle$
and contiguous: $\langle \text{contiguous } XS \rangle$
shows $\langle \text{set_of_interval_list } XS = \{ \text{lower } (\text{hd } XS) .. \text{upper } (\text{last } XS) \} \rangle$
using assms
proof(induction XS rule:list_nonempty_induct)
case (single x)
then show ?case
using set_of_eq
unfolding contiguous_def set_of_interval_list_def **by** auto[1]

next

case (cons x xs)
then show ?case
using set_of_eq
 $\text{sup_set_of_interval_list_upper[of } (x \# xs), \text{simplified cons contiguous_sorted_wrt_upper[of } (x \# xs), \text{simplified}$
 $\text{assms}], \text{simplified}]$
 $\text{inf_set_of_interval_list_lower[of } (x \# xs), \text{simplified cons contiguous_sorted_wrt_lower[of } (x \# xs), \text{simplified}$
 $\text{assms}], \text{simplified}]$
unfolding set_of_interval_list_def contiguous_def set_of_eq
apply(auto)[1]
subgoal
using cons.premis contiguous_non_overlapping lower_le_upper_aux non_overlapping_sorted_unroll
by fastforce
subgoal
by (smt (verit, ccfv_threshold) Suc_less_eq Suc_pred atLeastAtMost_iff length_greater_o_conv
 $\text{list.sel}(1) \text{ lower_le_upper neq_Nil_conv nth_Cons_o nth_Cons_Suc}$)
subgoal
using less_diff_conv **by** force
subgoal
by (smt (verit, del_insts) One_nat_def Suc_eq_plus1 atLeastAtMost_iff hd_conv_nth
 $\text{length_greater_o_conv less_diff_conv nth_Cons_o nth_Cons_Suc}$)
done
qed

lemma set_of_interval_list_set_eq_interval_list_union_contiguous:

assumes non_empty: $\langle XS \neq ([]::\text{real interval list}) \rangle$
and contiguous: $\langle \text{contiguous } XS \rangle$

shows $\langle \text{set_of_interval_list } XS = \text{set_of } (\text{interval_list_union } XS) \rangle$
using $\text{interval_list_union_contiguous}[\text{of } XS, \text{simplified assms, simplified}]$
 $\text{set_of_interval_list_contiguous}[\text{of } XS, \text{simplified assms, simplified}]$
 $\text{contiguous_non_overlapping}[\text{of } XS, \text{simplified assms, simplified}]$
 $\text{non_overlapping_implies_sorted_wrt_upper}[\text{of } XS]$
 $\text{non_overlapping_implies_sorted_wrt_lower}[\text{of } XS]$
 $\text{interval_list_union_contiguous_lower}[\text{of } XS]$
 $\text{interval_list_union_contiguous_upper}[\text{of } XS]$
apply $(\text{simp add: set_of_eq})$
by (metis non_empty)

lemma $m\text{Interval_ovl_lower_hd_min}$:
 $\langle \text{valid_mInterval_ovl } x \implies \text{Min } (\text{set } (\text{map } \text{lower } x)) = (\text{lower } o \text{ hd } x) \rangle$
proof $(\text{induction } x \text{ rule: induct_list012})$
case 1
then show ?case **unfolding** $\text{valid_mInterval_ovl_def}$ **by** simp
next
case (2 x)
then show ?case **unfolding** $\text{valid_mInterval_ovl_def}$ **by** simp
next
case (3 x y zs)
then show ?case
apply $(\text{simp add: valid_mInterval_ovl_def non_overlapping_sorted_def cmp_non_overlapping_def image_def})$
by $(\text{metis } (\text{no_types, lifting}) \text{list.sel(1) list.simps(3) min.absorb3 min_def sorted_wrt_lower_unroll})$
qed

lemma $m\text{Interval_adj_lower_hd_min}$:
 $\langle \text{valid_mInterval_adj } x \implies \text{Min } (\text{set } (\text{map } \text{lower } x)) = (\text{lower } o \text{ hd } x) \rangle$
proof $(\text{induction } x \text{ rule: induct_list012})$
case 1
then show ?case **unfolding** $\text{valid_mInterval_adj_def}$ **by** simp
next
case (2 x)
then show ?case **by** simp
next
case (3 x y zs)
then show ?case
apply $(\text{simp add: valid_mInterval_adj_def non_overlapping_sorted_def cmp_non_overlapping_def image_def})$
by $(\text{metis lower_le_upper min.absorb1 min.bounded_iff})$
qed

lemma $m\text{Interval_non_ovl_lower_hd_min}$:
 $\langle \text{valid_mInterval_non_ovl } x \implies \text{Min } (\text{set } (\text{map } \text{lower } x)) = (\text{lower } o \text{ hd } x) \rangle$
unfolding $\text{valid_mInterval_non_ovl_def}$
using $m\text{Interval_ovl_lower_hd_min}$
by (auto)

lemma $m\text{Interval_ovl_lower_last_max}$:
 $\langle \text{valid_mInterval_ovl } x \implies (\text{Max } (\text{set } (\text{map } \text{lower } x))) = (\text{lower } o \text{ last } x) \rangle$
proof $(\text{induction } x \text{ rule: induct_list012})$
case 1
then show ?case **unfolding** $\text{valid_mInterval_ovl_def}$ **by** simp

```

next
  case (2 x)
  then show ?case unfolding valid_mInterval_ovl_def by simp
next
  case (3 x y zs)
  then show ?case
  by(auto simp add: sorted_wrt_lower_def cmp_lower_width_def valid_mInterval_ovl_def
    non_overlapping_sorted_def max_def cmp_non_overlapping_def image_def
    split: if_splits)
qed

lemma mInterval_adj_upper_hd_min:
  ⟨ valid_mInterval_adj x ⟹ Min (set (map upper x)) = (upper o hd) x ⟩
proof(induction x rule:induct_list012)
  case 1
  then show ?case unfolding valid_mInterval_adj_def by simp
next
  case (2 x)
  then show ?case unfolding valid_mInterval_adj_def by simp
next
  case (3 x y zs)
  then show ?case
  apply(simp add: sorted_wrt_lower_def cmp_lower_width_def valid_mInterval_adj_def
    non_overlapping_sorted_def max_def cmp_non_overlapping_def image_def
    split: if_splits)
  by (metis (no_types, opaque_lifting) dual_order.trans lower_le_upper min.absorb2 min commute)
qed

lemma mInterval_adj_upper_last_max:
  ⟨ valid_mInterval_adj x ⟹ Max (set (map upper x)) = (upper o last) x ⟩
proof(induction x rule:induct_list012)
  case 1
  then show ?case unfolding valid_mInterval_adj_def by simp
next
  case (2 x)
  then show ?case unfolding valid_mInterval_adj_def by simp
next
  case (3 x y zs)
  then show ?case
  apply(simp add: sorted_wrt_lower_def cmp_lower_width_def valid_mInterval_adj_def
    non_overlapping_sorted_def max_def cmp_non_overlapping_def image_def
    split: if_splits)
  subgoal using le_left_mono lower_le_upper by blast
  subgoal using last_in_set lower_le_upper order_trans by blast
  done
qed

lemma set_of_subeq_aux:
  ⟨ (⋃ x ∈ set is. {lower x..upper x}) ⊆ {Min (lower ' (set is)) .. Max (upper ' (set is))} ⟩
proof(induction is)
  case Nil
  then show ?case by simp
next

```

```

case (Cons a is)
then show ?case
  apply(auto)[1]
  apply (meson List.finite_set Min.coboundedI finite_imageI finite_insert imageI insertCI order.trans)
  by (meson List.finite_set Max.coboundedI finite_imageI finite_insert imageI insert_iff order_trans)+
qed

```

```

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
  using assms proof(induction xs rule:list_nonempty_induct)
    case (single x)
    then show ?case by(simp add:interval_sort_lower_width_def)
  next
    case (cons x xs)
    then show ?case
      apply(simp)
      by (metis list.sel(1) merge_adjacent_intervals_sorted_wrt_lower_sorted_lower_lower_hd)
  qed

```

```

lemma sorted_wrt_lower_hd_min:
  ⟨x ≠ [] ⟹ sorted_wrt_lower x ⟹ Min (set (map lower x)) = (lower ∘ hd) x⟩
proof(induction x rule:induct_list012)
  case 1
  then show ?case by simp
next
  case (2 x)
  then show ?case by simp
next
  case (3 x y zs)
  then show ?case
    apply(simp add: valid_mInterval_ovl_def non_overlapping_sorted_def cmp_non_overlapping_def image_def)
    by (metis (no_types, lifting) list.sel(1) list.simps(3) min.absorb3 min_def sorted_wrt_lower_unroll)
qed

```

```

lemma lower_hd_min_over_mk_mInterval_non_ovl:
  xs ≠ [] ⟹ (lower ∘ hd) xs = Min (lower ` (set xs)) ⟹ (lower ∘ hd) (mk_mInterval_non_ovl xs) = (lower ∘ hd) xs
  unfolding mk_mInterval_non_ovl_def mk_mInterval_ovl_def
    by (metis list.set_map mInterval_ovl_lower_hd_min mk_mInterval_lower mk_mInterval_non_ovl_def
mk_mInterval_ovl_def valid_ovl_mkInterval_non_ovl)

```

```

theorem valid_mInterval_non_ovl_empty: valid_mInterval_non_ovl x ⟹ x ≠ []
  unfolding valid_mInterval_non_ovl_def valid_mInterval_ovl_def
  by simp

```

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 \Rightarrow nat \Rightarrow 'a interval list **where**
uniform_subdivision A n = map (λi . let $i' = \text{of_nat } i$ in
 mk_interval (lower A + (upper A - lower A) * $i' / \text{of_nat } n$,
 lower A + (upper A - lower A) * ($i' + 1$) / $\text{of_nat } n$)) [0.. n]

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

definition *overlapping_ordered* :: 'a::{preorder} interval list \Rightarrow bool **where**
overlapping_ordered xs = ($\forall i$. $i < \text{length } xs - 1 \longrightarrow \text{lower } (xs ! (i + 1)) \leq \text{upper } (xs ! i)$)

definition *overlapping_non_zero_width* :: 'a::{preorder, minus, zero, ord} interval list \Rightarrow bool **where**
overlapping_non_zero_width xs = ($\forall i < \text{length } xs - 1$. $\exists e$. $e \in_i (xs ! (i + 1)) \wedge e \in_i (xs ! i) \wedge 0 < \text{width } (xs ! (i + 1)) \wedge 0 < \text{width } (xs ! i)$)

definition *overlapping* :: 'a::{preorder} interval list \Rightarrow bool **where**
overlapping xs = ($\forall i < \text{length } xs - 1$. $\exists e$. $e \in_i (xs ! (i + 1)) \wedge e \in_i (xs ! i)$)

definition *check_is_uniform_subdivision* :: 'a::linordered_field interval \Rightarrow 'a interval list \Rightarrow bool **where**
check_is_uniform_subdivision A xs = (let $n = \text{length } xs$ in
 if $n = 0$ then True
 else
 let $d = \text{width } A / \text{of_nat } n$ in
 list_all (λx . $\text{width } x = d$) xs \wedge
 contiguous xs \wedge
 lower (hd xs) = lower A \wedge
 upper (last xs) = upper A)

lemma *non_empty_subdivision*:

assumes $0 < n$

shows *uniform_subdivision* A n $\neq []$

unfolding *uniform_subdivision_def* **using** *assms* **by** *simp*

lemma *uniform_subdivision_id*: *uniform_subdivision* X 1 = [X]

unfolding *uniform_subdivision_def* **by** *simp*

lemma *subdivision_length_n*:
assumes $0 < n$
shows $\text{length}(\text{uniform_subdivision } A \ n) = n$
using *assms*
proof(*induction n rule:nat_induct_non_zero*)
case 1
then show ?*case unfolding uniform_subdivision_def by simp*
next
case (*Suc n*)
then show ?*case unfolding uniform_subdivision_def by simp*
qed

lemma *contiguous_uniform_subdivision*: $\text{contiguous}(\text{uniform_subdivision } A \ n)$
proof –
have *a0*: $\forall i < \text{length}(\text{uniform_subdivision } A \ n) - 1.$
 $\text{upper}(\text{uniform_subdivision } A \ n \ ! \ i) = \text{lower } A + (\text{upper } A - \text{lower } A) * \text{of_nat } (i + 1) / \text{of_nat } n$
by (*simp add: uniform_subdivision_def divide_right_mono mult_left_mono*)
have *a1*: $\forall i < \text{length}(\text{uniform_subdivision } A \ n) - 1.$
 $\text{lower } A + (\text{upper } A - \text{lower } A) * \text{of_nat } (i + 1) / \text{of_nat } n = \text{lower}(\text{uniform_subdivision } A \ n \ ! \ (i + 1))$
by (*simp add: uniform_subdivision_def divide_right_mono mult_left_mono*)
have *a2*: $\forall i < \text{length}(\text{uniform_subdivision } A \ n) - 1.$
 $\text{upper}(\text{uniform_subdivision } A \ n \ ! \ i) = \text{lower}(\text{uniform_subdivision } A \ n \ ! \ (i + 1))$
using *a0 a1 by simp*
have *a3*: $\text{contiguous}(\text{uniform_subdivision } A \ n) =$
 $(\forall i < \text{length}(\text{uniform_subdivision } A \ n) - 1.$
 $\text{upper}(\text{uniform_subdivision } A \ n \ ! \ i) = \text{lower}(\text{uniform_subdivision } A \ n \ ! \ (i + 1)))$
unfolding *contiguous_def by simp*
show ?*thesis using a0 a1 a2 a3 by simp*
qed

lemma *overlapping_ordered_uniform_subdivision*:
assumes $0 < n$
shows $\text{overlapping_ordered}(\text{uniform_subdivision } A \ n)$
proof –
have *a0*: $\forall i < \text{length}(\text{uniform_subdivision } A \ n) - 1.$
 $\text{upper}(\text{uniform_subdivision } A \ n \ ! \ i) \geq \text{lower } A + (\text{upper } A - \text{lower } A) * \text{of_nat } (i + 1) / \text{of_nat } n$
using *assms*
by (*simp add: uniform_subdivision_def divide_right_mono mult_left_mono*)
have *a1*: $\forall i < \text{length}(\text{uniform_subdivision } A \ n) - 1.$
 $\text{lower } A + (\text{upper } A - \text{lower } A) * \text{of_nat } (i + 1) / \text{of_nat } n \geq \text{lower}(\text{uniform_subdivision } A \ n \ ! \ (i + 1))$
using *assms*
by (*simp add: uniform_subdivision_def divide_right_mono mult_left_mono*)
have *a2*: $\forall i < \text{length}(\text{uniform_subdivision } A \ n) - 1.$
 $\text{upper}(\text{uniform_subdivision } A \ n \ ! \ i) \geq \text{lower}(\text{uniform_subdivision } A \ n \ ! \ (i + 1))$
using *a0 a1 by force*
have *a3*: $\text{overlapping_ordered}(\text{uniform_subdivision } A \ n) =$
 $(\forall i < \text{length}(\text{uniform_subdivision } A \ n) - 1.$
 $\text{upper}(\text{uniform_subdivision } A \ n \ ! \ i) \geq \text{lower}(\text{uniform_subdivision } A \ n \ ! \ (i + 1)))$
unfolding *overlapping_ordered_def by simp*
show ?*thesis using a0 a1 a2 a3 by simp*
qed

lemma overlapping_uniform_subdivision:
assumes $o < N$
shows $\text{overlapping } (\text{uniform_subdivision } X \ N)$
using *assms*
proof –
let $?n = \text{length } (\text{uniform_subdivision } X \ N) - 1$
have $a0: \forall i < ?n. \text{lower } (\text{uniform_subdivision } X \ N! (i + 1)) = \text{upper } (\text{uniform_subdivision } X \ N! i)$
using *assms contiguous_uniform_subdivision unfolding contiguous_def by metis*
have $a1: \forall i < ?n. \text{upper } (\text{uniform_subdivision } X \ N! i) \in_i \text{uniform_subdivision } X \ N! i$
 $\wedge \text{upper } (\text{uniform_subdivision } X \ N! i) \in_i (\text{uniform_subdivision } X \ N! (i + 1))$
using *ao in_intervall lower_le_upper order.refl*
by *metis*
have $a2: \forall i < ?n. \exists e. e \in_i \text{uniform_subdivision } X \ N! (i + 1) \wedge e \in_i \text{uniform_subdivision } X \ N! i$
using *a1 by auto[1]*
show *?thesis using a2 unfolding overlapping_def by simp*
qed

lemma hd_lower_uniform_subdivision:
assumes $o < n$
shows $\text{lower } (\text{hd } (\text{uniform_subdivision } A \ n)) = \text{lower } A$
proof –
have $\text{lower } (\text{hd } (\text{uniform_subdivision } A \ n)) = \text{lower } A + (\text{upper } A - \text{lower } A) * \text{of_nat } o / \text{of_nat } n$
using *assms*
by (*simp add: uniform_subdivision_def mk_interval' hd_map*)
also have $\dots = \text{lower } A$
by *simp*
finally show *?thesis .*
qed

lemma last_upper_uniform_subdivision:
assumes $o < n$
shows $\text{upper } (\text{last } (\text{uniform_subdivision } A \ n)) = \text{upper } A$
proof –
have $\text{upper } (\text{last } (\text{uniform_subdivision } A \ n)) = \text{lower } A + (\text{upper } A - \text{lower } A) * \text{of_nat } n / \text{of_nat } n$
using *assms*
apply (*auto simp add: uniform_subdivision_def mk_interval' last_map Let_def*)[1]
using *One_nat_def Suc_pred' add.commute le_add_diff_inverse lower_le_upper nonzero_mult_div_cancel_right of_nat_o_less_iff of_nat_Suc order_less_irrefl apply metis*
by (*simp add: divide_right_mono mult_left_mono*)
also have $\dots = \text{upper } A$
using *assms by simp*
finally show *?thesis .*
qed

lemma uniform_subdivisions_width_single:
fixes $A :: 'a::\text{linordered_field interval}$
shows $\langle \text{width } (\text{Interval } (\text{lower } A + (\text{upper } A - \text{lower } A) * x / \text{of_nat } n, \text{lower } A + (\text{upper } A - \text{lower } A) * (x + 1) / \text{of_nat } n)) = \text{width } A / \text{of_nat } n \rangle$
proof –
have $\text{lower } A \leq \text{upper } A$ **using** *lower_le_upper by simp*
then have $\text{leq: } \text{lower } A + (\text{upper } A - \text{lower } A) * x / \text{of_nat } n \leq \text{lower } A + (\text{upper } A - \text{lower } A) * (x + 1) / \text{of_nat } n$
by (*simp add: divide_le_cancel linorder_not_less mult_le_cancel_left*)
have $U: \langle \text{upper } (\text{Interval } (\text{lower } A + (\text{upper } A - \text{lower } A) * x / \text{of_nat } n,$

$$\text{lower } A + (\text{upper } A - \text{lower } A) * (x + 1) / \text{of_nat } n) =$$

$$\text{lower } A + (\text{upper } A - \text{lower } A) * (x + 1) / \text{of_nat } n$$
using upper_bounds leq **by** blast
 have L: $\langle \text{lower } (\text{Interval } (\text{lower } A + (\text{upper } A - \text{lower } A) * x / \text{of_nat } n,$

$$\text{lower } A + (\text{upper } A - \text{lower } A) * (x + 1) / \text{of_nat } n)) =$$

$$\text{lower } A + (\text{upper } A - \text{lower } A) * x / \text{of_nat } n \rangle$$
using lower_bounds leq **by** blast
 then show ?thesis **using** U L add_diff_cancel_left add_diff_cancel_left' diff_divide_distrib
 mult.comm_neutral vector_space_over_itself.scale_right_diff_distrib **unfolding** width_def
 by metis

qed

lemma uniform_subdivisions_width:

assumes $0 < n$

shows $\langle \forall A. A \in \text{set } (\text{uniform_subdivision } X \ n) \longrightarrow \text{width } A = \text{width } X / \text{of_nat } n \rangle$

apply (simp add: uniform_subdivision_def mk_interval' o_def image_def width_def Let_def split: if_split)

apply auto[1]

apply (metis add_diff_cancel_left' diff_divide_distrib mult.right_neutral right_diff_distrib)

using assms uniform_subdivisions_width_single[simplified width_def]

by (simp add: divide_right_mono mult_left_mono)

lemma uniform_subdivision_sum_width:

assumes $0 < n$

shows $\langle \text{sum_list } (\text{map } \text{width } (\text{uniform_subdivision } X \ n)) = \text{width } X \rangle$

proof –

have $\langle \forall a. a \in \text{set } (\text{uniform_subdivision } X \ n) \longrightarrow \text{width } a = \text{width } X / \text{of_nat } n \rangle$

using uniform_subdivisions_width **using** assms **by** blast

then have width: $\forall a. a \in \text{set } (\text{map } \text{width } (\text{uniform_subdivision } X \ n)) \longrightarrow a = \text{width } X / \text{of_nat } n$

unfolding width_def **by** auto[1]

then have width_list: $\text{list_all } (\lambda a. a = \text{width } X / \text{of_nat } n) (\text{map } \text{width } (\text{uniform_subdivision } X \ n))$

unfolding width_def **using** list_all_iff **by** blast

then have length: $\text{length } (\text{map } \text{width } (\text{uniform_subdivision } X \ n)) = n$

unfolding uniform_subdivision_def **by** simp

then have sum_list (map width (uniform_subdivision X n)) = (width X / of_nat n) * of_nat n

using width_list **by** (metis list.map_ident_strong mult_of_nat_commute sum_list_triv width)

then show ?thesis **by** (simp add: assms)

qed

lemma uniform_subdivisions_distinct:

assumes $0 < n \ 0 < \text{width } A$

shows $\text{distinct } (\text{uniform_subdivision } A \ n)$

proof –

have $\forall i < n. \forall j < n. i \neq j \longrightarrow (\text{uniform_subdivision } A \ n) ! i \neq (\text{uniform_subdivision } A \ n) ! j$

proof –

have f1: $\forall i < n. \forall j < n. i \neq j \longrightarrow (\text{upper } A - \text{lower } A) * \text{of_nat } i / \text{of_nat } n \neq (\text{upper } A - \text{lower } A) * \text{of_nat } j / \text{of_nat } n$

using assms(1) assms(2) divide_cancel_right less_numeral_extra(3) mult_cancel_left of_nat_eq_o_iff of_nat_eq_iff width_def

by metis

have f2: $\forall i < n. \forall j < n. i \neq j \longrightarrow \text{lower } A + (\text{upper } A - \text{lower } A) * \text{of_nat } (i + 1) / \text{of_nat } n \neq \text{lower } A + (\text{upper } A - \text{lower } A) * \text{of_nat } (j + 1) / \text{of_nat } n$

using assms(1) assms(2) **unfolding** width_def **by** simp

have f3: $\forall i < n. \text{lower } ((\text{uniform_subdivision } A \ n) ! i) = \text{lower } A + (\text{upper } A - \text{lower } A) * \text{of_nat } i / \text{of_nat } n$

using assms **by** (simp add: uniform_subdivision_def divide_right_mono mult_left_mono Let_def)

have f5: $\forall i < n - 1. (\text{upper } ((\text{uniform_subdivision } A \ n) ! i)) = \text{lower } ((\text{uniform_subdivision } A \ n) ! \text{Suc } i)$

```

    using assms by (simp add: uniform_subdivision_def divide_right_mono mult_left_mono)
  have f6:  $\forall j < n - 1. (\text{upper } ((\text{uniform\_subdivision } A \ n) \ ! \ j)) = \text{lower } ((\text{uniform\_subdivision } A \ n) \ ! \ \text{Suc } j)$ 
    using assms(1) f5 contiguous_uniform_subdivision unfolding contiguous_def subdivision_length_n by blast
  have  $\forall i < n. \forall j < n. i \neq j \longrightarrow \text{lower } ((\text{uniform\_subdivision } A \ n) \ ! \ i) \neq \text{lower } ((\text{uniform\_subdivision } A \ n) \ ! \ j)$ 
    using f1 f2 f3 by auto[1]
  then show ?thesis by metis
qed
then show ?thesis using assms(1) distinct_conv_nth subdivision_length_n by metis
qed

```

lemma *uniform_subdivisions_non_overlapping*:

```

  assumes  $0 < n$ 
  shows non_overlapping_sorted (uniform_subdivision A n)
proof -
  have  $\forall i < n. \forall j < n. i < j \longrightarrow \text{upper } ((\text{uniform\_subdivision } A \ n) \ ! \ i) \leq \text{lower } ((\text{uniform\_subdivision } A \ n) \ ! \ j)$ 
  proof -
    have fo:  $\langle 0 \leq \text{width } A \rangle$  unfolding width_def lower_le_upper by simp
    have f2:  $\forall i < n. \text{upper } ((\text{uniform\_subdivision } A \ n) \ ! \ i) = \text{lower } A + (\text{upper } A - \text{lower } A) * \text{of\_nat } (i + 1) / \text{of\_nat } n$ 
      using assms by (simp add: uniform_subdivision_def divide_right_mono mult_left_mono Let_def)
    have f3:  $\forall j < n. \text{lower } ((\text{uniform\_subdivision } A \ n) \ ! \ j) = \text{lower } A + (\text{upper } A - \text{lower } A) * \text{of\_nat } j / \text{of\_nat } n$ 
      using assms by (simp add: uniform_subdivision_def divide_right_mono mult_left_mono Let_def)
    have f4:  $\forall i < n. \forall j < n. i < j \longrightarrow \text{lower } A + (\text{upper } A - \text{lower } A) * \text{of\_nat } (i + 1) / \text{of\_nat } n \leq \text{lower } A + (\text{upper } A - \text{lower } A) * \text{of\_nat } j / \text{of\_nat } n$ 
      using assms divide_right_mono mult_left_mono Suc_eq_plus1 Suc_le1 add_le_cancel_left
        interval_width_positive of_nat_o_le_iff of_nat_le_iff width_def
      by metis
    have  $\forall i < n. \forall j < n. i < j \longrightarrow \text{upper } ((\text{uniform\_subdivision } A \ n) \ ! \ i) \leq \text{lower } ((\text{uniform\_subdivision } A \ n) \ ! \ j)$ 
      using fo f2 f3 f4 by simp
    then show ?thesis by auto[1]
  proof -
    qed
  then show ?thesis
    unfolding non_overlapping_sorted_def cmp_non_overlapping_def
    by (simp add: assms(1) sorted_wrt_iff_nth_less subdivision_length_n)
  qed

```

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

lemma *uniform_subdivisions_valid_ainterval*:

```

  assumes  $0 < n \ 0 < \text{width } A$ 
  shows valid_mInterval_adj (uniform_subdivision A n)
  using assms
  unfolding valid_mInterval_adj_def
  apply safe
  subgoal using uniform_subdivisions_non_overlapping by blast
  subgoal using uniform_subdivisions_distinct by blast
  subgoal using non_empty_subdivision by blast
  done

```

lemma *uniform_subdivisions_valid*:

```

  assumes  $0 < n$ 
  shows check_is_uniform_subdivision A (uniform_subdivision A n)
  unfolding check_is_uniform_subdivision_def Let_def
  apply (simp split: if_split)
  apply safe
  subgoal using assms uniform_subdivisions_width subdivision_length_n

```

```

by (metis (mono_tags, lifting) Ball_set)
subgoal using assms contiguous_uniform_subdivision by blast
subgoal using assms hd_lower_uniform_subdivision by blast
subgoal using assms last_upper_uniform_subdivision by blast
done

```

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 < $\forall x \in X. \forall x' \in X. \text{dist } x x' \leq \text{width\_set } X$ >
using assms sup_inf_dist_bounded
unfolding width_set_def
by (simp)

```

```

lemma width_inclusion_isotonic_approx:

```

```

fixes F :: <real interval  $\Rightarrow$  real interval>
assumes inclusion_isotonic F F is_interval_extension_of f
shows < $0 \leq \text{width}(F X) - \text{width\_set}(f' \text{ set\_of } X)$ >
by (smt (verit, del_insts) assms(1) assms(2) inclusion_isotonic_inf
inclusion_isotonic_sup width_def width_set_def)

```

```

lemma diameter_width:

```

```

assumes < $a \leq b$ >
shows <diameter {a..b} = width_set {a..b}>
by (simp add: assms linorder_not_less diameter_Sup_Inf width_set_def)

```

```

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 < $x \in (f' S) \implies y \in (f' S) \implies \text{dist } x y \leq \text{diameter}(f' S)$ >
using lipschitz_on_uniformly_continuous[of C S f, simplified assms]
bounded_uniformly_continuous_image[of S f, simplified assms]
diameter_bounded_bound[of f' S x y]

```

by simp

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)|>`

by (`simp add: width_set_def set_of_eq`)

lemma `width_set_dist`:

fixes `f` :: real \Rightarrow real

shows `width_set (set_of X) = (dist (lower X) (upper X))`

by(`simp add:set_of_eq width_set_def dist_real_def`)

lemma `diameter_of`: fixes `X` :: real interval

shows `diameter_upper_lower`: `<diameter (set_of X) = |(lower X) - (upper X)|>`

by (`simp add: linorder_not_less set_of_eq`)

lemma `diameter_dist`:

fixes `X` :: real interval

shows `diameter (set_of X) = (dist (lower X) (upper X))`

unfolding `set_of_eq dist_real_def abs_real_def`

using `lower_le_upper[of X] diameter_closed_interval[of lower X upper X]`

by `argo`

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)>`

using `assms atLeastAtMost_iff bdd_below.unfold bounded_imp_bdd_below image_def`

`lipschitz_bounded_image_real set_of_eq set_of_nonempty`

by `simp`

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)>`

using `assms atLeastAtMost_iff bdd_above.unfold bounded_imp_bdd_above image_def`

`lipschitz_bounded_image_real set_of_eq set_of_nonempty`

by `simp`

lemma `diameter_image_dist`:

fixes `f`::`<real \Rightarrow real>`

assumes `<continuous_on (set_of X) f>`

shows $\langle \exists x \in \text{set_of } X. \exists x' \in \text{set_of } X. \text{diameter } (f' \text{ set_of } X) = \text{dist } (f x) (f x') \rangle$
using *assms compact_continuous_image*[of *set_of X f*, *simplified assms compact_set_of*[of *X*]]
lower_le_upper[of *X*] *diameter_closed_interval*[of *f (lower X) f (upper X)*, *symmetric*]
diameter_compact_attained[of *f' set_of X*] *set_f_nonempty*[of *f X*]
by *fastforce*

lemma *excess_width_inf_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 $\langle \text{dist } (\text{Inf } (f' \text{ set_of } X)) (\text{lower } (F X)) \leq \text{excess_width_diameter } F f X \rangle$
unfolding *dist_real_def abs_real_def excess_width_diameter_def width_def*
using *inclusion_isotonic_inf*[of *F f X*, *simplified assms*]
inclusion_isotonic_sup[of *F f X*, *simplified assms*]
diameter_Sup_Inf[of *f' set_of X*, *simplified assms lipschitz_on_continuous_on*[of *C (set_of X) f*]
compact_img_set_of[of *X f*], *simplified*]
by *simp*

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 $\langle \text{dist } (\text{Inf } (f' \text{ set_of } X)) (\text{lower } (F X)) \leq \text{excess_width } F f X \rangle$
unfolding *dist_real_def abs_real_def excess_width_def width_def*
using *inclusion_isotonic_inf*[of *F f X*, *simplified assms*]
inclusion_isotonic_sup[of *F f X*, *simplified assms*]
by (*simp add: width_set_def*)

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 $\langle \text{dist } (\text{Sup } (f' \text{ set_of } X)) (\text{upper } (F X)) \leq \text{excess_width } F f X \rangle$
unfolding *dist_real_def abs_real_def excess_width_diameter_def width_def*
using *inclusion_isotonic_inf*[of *F f X*, *simplified assms*]
inclusion_isotonic_sup[of *F f X*, *simplified assms*]
diameter_Sup_Inf[of *f' set_of X*, *simplified assms lipschitz_on_continuous_on*[of *C (set_of X) f*]
compact_img_set_of[of *X f*], *simplified*]
by (*simp add: excess_width_def width_def width_set_def*)

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 $\langle \text{dist } (\text{Sup } (f' \text{ set_of } X)) (\text{upper } (F X)) \leq \text{excess_width } F f X \rangle$
unfolding *dist_real_def abs_real_def excess_width_def width_def*
using *inclusion_isotonic_inf*[of *F f X*, *simplified assms*]
inclusion_isotonic_sup[of *F f X*, *simplified assms*]
by (*simp add: width_set_def*)

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 $\text{lower } E \leq (o::'a) \wedge (o::'a) \leq \text{upper } E$ 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  $\exists E. o \in_i E \wedge Y = X + E \wedge width\ Y = width\ X + width\ E$ 
proof –
have  $c \leq a \leq b \leq d$ 
proof –
  have  $leq: lower\ Y \leq lower\ X \wedge upper\ X \leq upper\ Y$  using  $assms(1)$  unfolding  $less\_eq\_interval\_def$  by  $simp$ 
  have  $X\_bounds: lower\ X = a\ upper\ X = b$  unfolding  $X\_def$  by  $(simp\ add: x\_valid\ bounds\_of\_interval\_eq\_lower\_upper) +$ 
  have  $Y\_bounds: lower\ Y = c\ upper\ Y = d$  unfolding  $Y\_def$  by  $(simp\ add: y\_valid\ bounds\_of\_interval\_eq\_lower\_upper) +$ 
  show  $c \leq a \leq b \leq d$  using  $assms(1)$   $X\_bounds\ Y\_bounds$  unfolding  $less\_eq\_interval\_def$  by  $simp\_all$ 
qed
define  $e$  where  $e = c - a$ 
define  $f$  where  $f = d - b$ 
define  $E$  where  $E = Interval(e, f)$ 
have  $o \in_i E$  unfolding  $E\_def\ e\_def\ f\_def$  using  $\langle c \leq a \rangle \langle b \leq d \rangle\ set\_of\_subset\_iff$ 
by  $(smt\ (verit,\ ccfv\_SIG)\ diff\_ge\_o\_iff\_ge\ in\_interval1\ le\_iff\_diff\_le\_o\ lower\_bounds\ order.trans\ upper\_bounds)$ 
have  $Y = X + E$ 
proof –
  have  $X\_bounds: lower\ X = a\ upper\ X = b$ 
  unfolding  $X\_def$  by  $(simp\ add: x\_valid\ bounds\_of\_interval\_eq\_lower\_upper) +$ 
  have  $Y\_bounds: lower\ Y = c\ upper\ Y = d$ 
  unfolding  $Y\_def$  by  $(simp\ add: y\_valid\ bounds\_of\_interval\_eq\_lower\_upper) +$ 
  have  $E\_bound\_1: lower\ E = c - a$ 
  unfolding  $E\_def\ e\_def\ f\_def$ 
  using  $\langle b \leq d \rangle \langle c \leq a \rangle\ diff\_left\_mono\ diff\_self\ lower\_bounds\ order.trans$ 
by  $metis$ 
  have  $E\_bound\_2: upper\ E = d - b$ 
  unfolding  $E\_def\ e\_def\ f\_def$ 
  using  $\langle b \leq d \rangle \langle c \leq a \rangle\ E\_bound\_1 \langle o \in_i E \rangle\ diff\_ge\_o\_iff\_ge\ dual\_order.trans\ in\_bounds\ upper\_bounds$ 
by  $metis$ 
  have  $add: Interval(a, b) + Interval(c - a, d - b) = Interval(c, d)$ 
  using  $X\_bounds\ Y\_bounds\ E\_bound\_1\ E\_bound\_2$ 
by  $(simp\ add: E\_def\ X\_def\ Y\_def\ e\_def\ f\_def\ interval\_eq1)$ 
  show  $?thesis$  unfolding  $E\_def\ X\_def\ Y\_def\ e\_def\ f\_def$  using  $add$  by  $simp$ 
qed
have  $width\ Y = width\ X + width\ E$  unfolding  $width\_def$  using  $E\_def\ X\_def\ Y\_def \langle Y = X + E \rangle\ e\_def\ f\_def$  by  $force$ 
from  $\langle o \in_i E \rangle \langle Y = X + E \rangle \langle width\ Y = width\ X + width\ E \rangle$  show  $?thesis$  by  $auto[1]$ 
qed

```

lemma $excess_width_incl:$

```

fixes  $F :: real\ interval \Rightarrow real\ interval$  and  $X :: real\ interval$ 
assumes  $int: \langle F\ is\_interval\_extension\_of\ f \rangle$ 
and  $iso: inclusion\_isotonic\ F$ 
and  $L-lipschitz\_on\ (set\_of\ X)\ f$ 
shows  $\langle \exists E. FX = Interval(Inf\ (f'\ set\_of\ X), Sup\ (f'\ set\_of\ X)) + E \rangle$ 
proof –
  have  $ao: \langle f'\ (set\_of\ X) \neq \{ \} \rangle$  using  $in\_interval1$  by  $fastforce$ 
  have  $a1: Inf\ (f'\ set\_of\ X) \leq Sup\ (f'\ set\_of\ X)$ 
  using  $assms\ ao\ inf\_le\_sup\_image\_real$  by  $simp$ 
  have  $a2: \langle f'\ (set\_of\ X) \subseteq set\_of\ (FX) \rangle$ 
  using  $assms\ fundamental\_theorem\_of\_interval\_analysis$  by  $simp$ 
  have  $max: \langle Sup\ (f'\ (set\_of\ X)) \leq Sup\ (set\_of\ (FX)) \rangle$ 
  using  $assms(3)\ ao\ a2\ sup\_image\_le\_real[of\ f\ X\ F]$  by  $blast$ 

```

```

have max_interval: Sup (f' (set_of X)) ≤ upper (F X)
  using max_sup_set_of by metis
have min: <Inf (set_of (F X)) ≤ Inf (f' (set_of X))>
  using assms(3) a0 a2 inf_image_le_real[of f X F] by blast
have min_interval: lower (F X) ≤ Inf (f' (set_of X))
  using min_inf_set_of by metis
have lower_min: lower (Interval (Inf (f' set_of X), Sup (f' set_of X))) = Inf (f' set_of X)
  using lower_bounds a1 by simp
have upper_max: upper (Interval (Inf (f' set_of X), Sup (f' set_of X))) = Sup (f' set_of X)
  using upper_bounds a1 by simp
have a3: Interval(Inf (f' set_of X), Sup (f' set_of X)) ≤ F X
  using min_interval max_interval lower_min upper_max unfolding less_eq_interval_def by simp
have a4: ∃ E . Interval(Inf (f' set_of X), Sup (f' set_of X)) + E = F X using a3
  by (metis (no_types, opaque_lifting) bounds_of_interval_eq_lower_upper
    bounds_of_interval_inverse interval_subset_width lower_le_upper)
then show ?thesis by metis
qed

```

```

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: <∃ 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)) ≤ F X>
  proof –
  have lhs: lower (F X) ≤ lower (Interval(Inf (f' set_of X), Sup (f' set_of X)))
    using assms(1,2,3) inf_image_le_real inf_le_sup_image_real inf_set_of
      fundamental_theorem_of_interval_analysis lower_bounds
    by metis
  have rhs: upper (Interval(Inf (f' set_of X), Sup (f' set_of X))) ≤ upper (F X)
    using assms(1,2,3) inf_le_sup_image_real sup_image_le_real sup_set_of
      fundamental_theorem_of_interval_analysis upper_bounds
    by metis
  show ?thesis using lhs rhs unfolding less_eq_interval_def by simp
qed

```

```

lemma each_subdivision_width_order:
  fixes X :: 'a::{linordered_field,lattice,metric_space} interval
  assumes inclusion_isotonic F lipschitzl_on C U F F is_interval_extension_of f
  and set (uniform_subdivision X N) ⊆ U o < N Xs ∈ set (uniform_subdivision X N)
  shows width(F Xs) ≤ C * width (X) / of_nat N
  proof –
  have a0: ∀ Xs ∈ set (uniform_subdivision X N). width (F Xs) ≤ C * width Xs
    using assms(2) assms(4) lipschitzl_onD by blast
  have a1: ∀ Xs ∈ set (uniform_subdivision X N). width(F Xs) ≤ C * width (X) / of_nat N
    using a0 assms(5) uniform_subdivisions_width[of N X] by simp
  show ?thesis using a1 assms by simp
qed

```

```

lemma each_subdivision_excess_width_order:
  fixes X :: real interval
  assumes inclusion_isotonic F lipschitzl_on C U F F is_interval_extension_of f
  and set (uniform_subdivision X N) ⊆ U o < N

```

```

and L—lipschitz_on (set_of (interval_list_union (uniform_subdivision X N))) f
shows  $\forall Xs \in \text{set } (\text{uniform\_subdivision } X \ N) . \text{excess\_width } F \ f \ Xs \leq C * \text{width } (X) / \text{of\_nat } N$ 
proof —
have a0:  $\forall Xs \in \text{set } (\text{uniform\_subdivision } X \ N) . \text{width } (F \ Xs) \leq C * \text{width } Xs$ 
  using assms lipschitzl_onD by blast
have a1:  $\forall Xs \in \text{set } (\text{uniform\_subdivision } X \ N) . \text{width } (F \ Xs) \leq C * \text{width } (X) / \text{of\_nat } N$ 
  using a0 assms uniform_subdivisions_width[of N X] by simp
have a2:  $\forall Xs \in \text{set } (\text{uniform\_subdivision } X \ N) . \text{excess\_width } F \ f \ Xs \leq C * \text{width } (X) / \text{of\_nat } N$ 
proof —
have b0:  $\text{set } (\text{uniform\_subdivision } X \ N) \neq \{\}$ 
  using assms non_empty_subdivision by simp
have b1:  $\forall Xs \in \text{set } (\text{uniform\_subdivision } X \ N) . 0 \leq \text{width\_set } (f' \ \text{set\_of } Xs)$ 
proof —
have c0:  $\forall Xs \in \text{set } (\text{uniform\_subdivision } X \ N) . \text{set\_of } Xs \subseteq \text{set\_of } (\text{interval\_list\_union } (\text{uniform\_subdivision } X \ N))$ 
  using assms interval_list_union_correct in_set_conv_nth non_empty_subdivision
  by metis
then have c1:  $\forall Xs \in \text{set } (\text{uniform\_subdivision } X \ N) . \text{set\_of } Xs \neq \{\}$  using in_interval by fastforce
then have c2:  $\forall Xs \in \text{set } (\text{uniform\_subdivision } X \ N) . \text{Inf } (f' \ \text{set\_of } Xs) \leq \text{Sup } (f' \ \text{set\_of } Xs)$ 
  using assms c0 c1 inf_le_sup_image_real lipschitz_on_subset
  by (metis inf_le_sup_image_real)
then have c3:  $\forall Xs \in \text{set } (\text{uniform\_subdivision } X \ N) . 0 \leq \text{width\_set } (f' \ \text{set\_of } Xs)$ 
  by (simp add: width_set_def)
then show ?thesis by simp
qed
have b2:  $\forall Xs \in \text{set } (\text{uniform\_subdivision } X \ N) . \text{width } (F \ Xs) - \text{width\_set } (f' \ \text{set\_of } Xs) \leq \text{width } (F \ Xs)$ 
  using b0 b1 assms inf_set_of_lower_le_upper sup_set_of_inf_le_sup_image_real image_is_empty
  by (simp add: width_set_def)
then show ?thesis
  using a1 unfolding excess_width_def width_set_def by fastforce
qed
show ?thesis using assms a0 a1 a2 by simp
qed

```

The theorem $\llbracket \text{inclusion_isotonic } ?F; ?C\text{--lipschitzl_on } ?U \ ?F; ?F \text{ is_interval_extension_of } ?f; \text{set } (\text{uniform_subdivision } ?X \ ?N) \subseteq ?U; 0 < ?N; ?Xs \in \text{set } (\text{uniform_subdivision } ?X \ ?N) \rrbracket \implies \text{width } (?F \ ?Xs) \leq ?C * \text{width } ?X / \text{of_nat } ?N$ refers to Theorem 6.1 in [4].

```

lemma sup_interval_max:
fixes X Y :: 'a::{linordered_ring, lattice} interval
shows  $\text{sup } X \ Y = \text{Interval}(\text{min } (\text{lower } X) (\text{lower } Y), \text{max } (\text{upper } X) (\text{upper } Y))$ 
using Interval.lower_sup Interval.upper_sup Interval_id inf_real_def sup_max
by (metis inf_min)

```

```

lemma interval_inf_sup_lower:  $\text{inf } (\text{lower } l_1) (\text{lower } l_2) = \text{lower } (\text{sup } l_1 \ l_2)$ 
unfolding sup_interval_def
by (metis (mono_tags, lifting) Interval.lower_sup sup_interval_def)

```

```

lemma interval_sup_sup_upper:  $\text{sup } (\text{upper } l_1) (\text{upper } l_2) = \text{upper } (\text{sup } l_1 \ l_2)$ 
unfolding sup_interval_def
by (metis (mono_tags, lifting) Interval.upper_sup sup_interval_def)

```

```

lemma interval_union_lower:
assumes contiguous Xs Xs  $\neq \{\}$ 
shows  $\text{lower } (\text{interval\_list\_union } Xs) = \text{lower } (Xs!0)$ 
using assms

```

```

proof(induction Xs rule:interval_list_union.induct)
  case 1
  then show ?case by simp
next
  case (2 I)
  then show ?case by simp
next
  case (3 I v va)
  have ao: contiguous (v # va) using 3 unfolding contiguous_def by auto
  have a1: (v # va) ≠ [] by simp
  then show ?case
    unfolding sorted_wrt_lower_def 3
  proof –
  have bo: lower ((I # v # va) ! o) = lower I by simp
  have b1: lower I ≤ lower v
    using 3.prem1 a1
    unfolding contiguous_def
  by (metis Suc_eq_plus1 add_diff_cancel_left' length_Cons less_Suc_eq_o_disj
    lower_le_upper nth_Cons_o nth_Cons_Suc plus_1_eq_Suc)
  show lower (interval_list_union (I # v # va)) = lower ((I # v # va) ! o)
    using bo b1 3(2) interval_list_union.simps(3) 3.IH Interval.lower_sup ao a1 inf.orderE
      nth_Cons_o
    unfolding sorted_wrt_lower_def
    by metis
  qed
qed

lemma interval_union_upper:
  assumes contiguous Xs Xs ≠ []
  shows upper (interval_list_union Xs) = upper (last Xs)
  using assms
proof(induction Xs rule:interval_list_union.induct)
  case 1
  then show ?case by simp
next
  case (2 I)
  then show ?case by simp
next
  case (3 I v va)
  have ao: contiguous (v # va) using 3 unfolding contiguous_def by auto
  have a1: (v # va) ≠ [] by simp
  then show ?case
  proof –
  have bo: upper ((I # v # va) ! (length (I # v # va) - 1)) = upper (last (I # v # va))
    using last_conv_nth by fastforce
  have b1: upper I ≤ upper v using 3.prem1 unfolding contiguous_def by auto
  show upper (interval_list_union (I # v # va)) = upper (last (I # v # va))
    using bo b1 interval_list_union.simps(3) 3.IH ao a1 interval_list_union.simps(3)
      Interval.upper_sup last.simps
    by (metis (no_types, opaque_lifting) dual_order.trans min_list.cases sup_absorb2 sup_get1)
  qed
qed

lemma union_set:

```

```

assumes  $0 < n$ 
shows  $\text{interval\_list\_union } (\text{uniform\_subdivision } X \ n) = X$ 
using assms
proof (induction  $n$  rule:nat_induct_non_zero)
  case 1
  then show ?case using uniform_subdivision_id interval_list_union.simps(2) by metis
next
  case (Suc  $n$ )
  then show ?case
  proof (induction uniform_subdivision  $X$  (Suc  $n$ ) rule:interval_list_union.induct)
    case 1
    then show ?case by (metis less_Suc_eq non_empty_subdivision)
    next
    case (2  $l$ )
    then show ?case using One_nat_def interval_list_union.simps(2) subdivision_length_n uniform_subdivision_id
    zero_less_Suc
    by metis
    next
    case (3  $l$   $v$   $va$ )
    then have  $a_0$ :  $\text{lower } l = \text{lower } X$ 
    using hd_lower_uniform_subdivision assms
    by (metis list.collapse list.simps(3) nth_Cons_0 zero_less_Suc)
    then have  $a_1$ :  $\text{upper } (\text{last } (l \# v \# va)) = \text{upper } X$ 
    using last_upper_uniform_subdivision[of Suc n X] assms 3.hyps(2) by simp
    then have  $a_2$ :  $\text{contiguous } (l \# v \# va)$ 
    using contiguous_uniform_subdivision assms zero_less_Suc 3.hyps(2) by metis
    then have  $a_3$ :  $\text{overlapping\_ordered } (l \# v \# va)$ 
    using overlapping_uniform_subdivision assms zero_less_Suc 3.hyps(2)
    by (simp add: overlapping_ordered_uniform_subdivision)
    then have  $a_4$ :  $\forall i < \text{length } (l \# v \# va) - 1. \text{upper } ((l \# v \# va) ! i) = \text{lower } ((l \# v \# va) ! (i + 1))$ 
    using  $a_0$   $a_2$  unfolding contiguous_def by simp
    have  $a_5$ :  $\forall i < \text{length } (l \# v \# va) - 1. \text{lower } ((l \# v \# va) ! i) \leq \text{lower } ((l \# v \# va) ! (i + 1))$ 
    using  $a_0$   $a_2$  contiguous_def lower_le_upper by metis
    have  $a_6$ :  $\forall i < \text{length } (l \# v \# va) - 1. \text{upper } ((l \# v \# va) ! i) \leq \text{upper } ((l \# v \# va) ! (i + 1))$ 
    using  $a_0$   $a_2$  contiguous_def lower_le_upper by metis
    then show ?case
    proof —
    have  $\text{lower } (\text{interval\_list\_union } (l \# v \# va)) = \text{lower } X$ 
    proof —
    have  $c_0$ :  $\text{lower } (\text{interval\_list\_union } (l \# v \# va)) = \text{lower } (\text{sup } l \ (\text{interval\_list\_union } (v \# va)))$ 
    using interval_list_union.simps(3) by metis
    have  $c_1$ :  $\text{lower } (\text{sup } l \ (\text{interval\_list\_union } (v \# va))) = \min (\text{lower } l) (\text{lower } (\text{interval\_list\_union } (v \# va)))$ 
    using inf_real_def sup_interval_def sup_interval_max sup_real_def
    by (simp add: inf_min)
    have  $c_2$ :  $\min (\text{lower } l) (\text{lower } (\text{interval\_list\_union } (v \# va))) = \text{lower } l$ 
    using 3 hd_lower_uniform_subdivision[of Suc n X] a_0
    contiguous_uniform_subdivision[of X, simplified contiguous_def 3(2)[symmetric]]
    interval_union_lower[of (l # v # va)]
    by (metis a_2 c_0 c_1 list.discr nth_Cons_0)
    show ?thesis using  $a_0$   $c_0$   $c_1$   $c_2$  by simp
    qed
    moreover have  $\text{upper } (\text{interval\_list\_union } (l \# v \# va)) = \text{upper } X$ 
    proof —
    have  $c_0$ :  $\text{upper } (\text{interval\_list\_union } (l \# v \# va)) = \text{upper } (\text{sup } l \ (\text{interval\_list\_union } (v \# va)))$ 

```

```

using interval_list_union.simps(3) by metis
have c1: upper (sup I (interval_list_union (v # va))) = max (upper I) (upper (interval_list_union (v # va)))
using inf_real_def sup_interval_def sup_interval_max sup_real_def
by (simp add: sup_max)
have c2: max (upper I) (upper (interval_list_union (v # va))) = upper (last (I # v # va))
using contiguous_uniform_subdivision[of X, simplified contiguous_def 3(2)[symmetric]]
interval_union_upper[of (I # v # va)]
by (metis a2 co c1 list.disc1)
show ?thesis using co c1 c2 3.hyps(2) last_upper_uniform_subdivision by auto[1]
qed
ultimately show ?thesis
using interval_eq1 3 by auto[1]
qed
qed
qed

```

lemma sum_list_less:

```

assumes list_all ( $\lambda n. n \leq (y::real)$ ) xs
shows sum_list xs  $\leq y * \text{length xs}$ 
proof –
have  $\forall x \in \text{set xs}. x \leq y$ 
using assms list_all_iff by metis
hence sum_list xs  $\leq \text{sum\_list (replicate (length xs) y)}$ 
using sum_list_mono
by (simp add: order_less_imp_le sum_list_mono2)
also have ... =  $y * \text{length xs}$ 
by (simp add: sum_list_replicate)
finally show ?thesis by simp
qed

```

lemma in_bounds2:

```

fixes X Y :: 'a::{\linordered_ring} interval
shows  $x \in_i X \wedge x \in_i Y \implies$ 
 $(\text{lower } Y \leq \text{lower } X \wedge \text{upper } Y \leq \text{upper } X \wedge \text{lower } X \leq \text{upper } Y \wedge \text{lower } Y \leq \text{upper } X) \vee$ 
 $(\text{lower } X \leq \text{lower } Y \wedge \text{upper } X \leq \text{upper } Y \wedge \text{lower } X \leq \text{upper } Y \wedge \text{lower } Y \leq \text{upper } X) \vee$ 
 $(\text{lower } Y \leq \text{lower } X \wedge \text{upper } X \leq \text{upper } Y \wedge \text{lower } Y \leq \text{lower } X \wedge \text{lower } Y \leq \text{upper } X) \vee$ 
 $(\text{lower } X \leq \text{lower } Y \wedge \text{upper } Y \leq \text{upper } X \wedge \text{lower } X \leq \text{upper } Y \wedge \text{lower } Y \leq \text{upper } X)$ 
apply(clarify)
by (metis (full_types) in_bounds nle_le order.trans)

```

lemma overlapping_width_sum:

```

fixes X Y :: 'a::{\linordered_ring, lattice} interval
assumes overlapping [X,Y]
shows width (sup X Y)  $\leq \text{width } X + \text{width } Y$ 
proof –
have a0:  $\forall i < \text{length } [X, Y] - 1. \exists e. e \in_i [X, Y] \wedge (i + 1) \wedge e \in_i [X, Y] \wedge ! i$ 
using assms unfolding overlapping_def by blast
have a1:  $(\text{lower } Y \leq \text{lower } X \wedge \text{upper } Y \leq \text{upper } X \wedge \text{lower } X \leq \text{upper } Y \wedge \text{lower } Y \leq \text{upper } X) \vee$ 
 $(\text{lower } X \leq \text{lower } Y \wedge \text{upper } X \leq \text{upper } Y \wedge \text{lower } X \leq \text{upper } Y \wedge \text{lower } Y \leq \text{upper } X) \vee$ 
 $(\text{lower } Y \leq \text{lower } X \wedge \text{upper } X \leq \text{upper } Y \wedge \text{lower } Y \leq \text{lower } X \wedge \text{lower } Y \leq \text{upper } X) \vee$ 
 $(\text{lower } X \leq \text{lower } Y \wedge \text{upper } Y \leq \text{upper } X \wedge \text{lower } X \leq \text{upper } Y \wedge \text{lower } Y \leq \text{upper } X)$ 
using assms in_bounds2[of _ X Y] unfolding overlapping_def by auto
have a2:  $(\text{lower } Y \leq \text{lower } X \wedge \text{upper } Y \leq \text{upper } X \wedge \text{lower } X \leq \text{upper } Y \wedge \text{lower } Y \leq \text{upper } X) \implies \text{width (sup X Y)} \leq$ 
width X + width Y

```

```

  by (simp add: inf_min width_def)
  have a3: (lower X ≤ lower Y ∧ upper X ≤ upper Y ∧ lower X ≤ upper Y ∧ lower Y ≤ upper X) ⇒ width (sup X Y) ≤
width X + width Y
  by (simp add: inf_min width_def)
  have a4: (lower Y ≤ lower X ∧ upper X ≤ upper Y ∧ lower Y ≤ lower X ∧ lower Y ≤ upper X) ⇒ width (sup X Y) ≤
width X + width Y
  by (simp add: inf_min sup_max width_def)
  have a5: (lower X ≤ lower Y ∧ upper Y ≤ upper X ∧ lower X ≤ upper Y ∧ lower Y ≤ upper X) ⇒ width (sup X Y) ≤
width X + width Y
  by (metis interval_width_positive le_add_same_cancel1 less_eq_interval_def sup.order_iff)
  show ?thesis using assms a0 a1 a2 a3 a4 a5 by blast
qed

```

lemma *interval_list_union_width*:

```

fixes xs :: 'a::{linordered_ring, lattice} interval list
assumes overlapping xs xs ≠ []
shows overlapping xs ⇒ width (interval_list_union xs) ≤ sum_list (map width xs)
using assms unfolding contiguous_def width_def
proof (induction xs rule: interval_list_union.induct)
  case 1
  then show ?case by simp
next
  case (2 l)
  then show ?case by simp
next
  case (3 l1 l2 l3)
  then show ?case unfolding overlapping_def
proof —
  have a0: width (interval_list_union (l1 # l2 # l3)) = width (sup l1 (interval_list_union (l2 # l3)))
  using interval_list_union.simps by simp
  have a1: ... ≤ width l1 + width (interval_list_union (l2 # l3))
  proof —
  have b0: overlapping [l1, interval_list_union (l2 # l3)]
  using 3.prem1 list.simps(3) list.size(3) list.size(4) interval_list_union_correct
  unfolding overlapping_def
  by (smt (verit, ccfv_threshold) One_nat_def Suc_eq_plus1 diff_add_inverse2
length_greater_o_conv less_one nth_Cons' subsetD)
  show ?thesis
  using overlapping_width_sum[of l1 (interval_list_union (l2 # l3)), simplified b0]
  by blast
qed
  have a2: ... ≤ width l1 + sum_list (map width (l2 # l3))
  proof —
  have b0: (width l1 + width (interval_list_union (l2 # l3)) ≤ width l1 + sum_list (map width (l2 # l3))) =
(width (interval_list_union (l2 # l3)) ≤ sum_list (map width (l2 # l3))) by simp
  have b1: overlapping (l2 # l3) using 3.prem2 unfolding overlapping_def by force
  have b2: l2 # l3 ≠ [] by simp
  have b3: width (interval_list_union (l2 # l3)) ≤ sum_list (map width (l2 # l3))
  using 3.IH b1 b2 unfolding width_def by simp
  show ?thesis using b0 b3 by simp
qed
  have a3: ... = sum_list (map width (l1 # l2 # l3)) by auto[1]
  show ?thesis using a0 a1 a2 a3 map_eq_conv width_def
  by (smt (verit, ccfv_SIG) dual_order.trans)

```

qed
qed

lemma *map_non_zero_width*:

fixes $U :: 'a :: \{\text{linordered_idom}\}$ *interval set*
assumes C —*lipschitzl_on U F inclusion_isotonic F set xs* $\subseteq U$
shows $\forall x \in \text{set } xs. 0 \leq \text{width } x \longrightarrow 0 \leq \text{width } (F x)$

proof

fix x

assume $ao: x \in \text{set } xs$

show $0 \leq \text{width } x \longrightarrow 0 \leq \text{width } (F x)$

proof

assume $a1: 0 \leq \text{width } x$

have $a2: \text{width } (F x) \leq C * \text{width } x$ **using** $\text{assms}(1,3)$ ao **unfolding** *lipschitzl_on_def* **by** *blast*

have $a4: \text{width } (F x) \leq C * \text{width } x + 1$ **using** $\text{assms}(1,3)$ ao **unfolding** *lipschitzl_on_def* **by** *fastforce*

have $0 \leq C * \text{width } x$ **using** $\text{assms}(1)$ $a1$ **unfolding** *lipschitzl_on_def* **by** *simp*

show $0 \leq \text{width } (F x)$ **using** $\text{assms}(1)$ ao *interval_width_positive* **unfolding** *lipschitzl_on_def interval_width_positive* **by**

blast

qed

qed

lemma *inclusion_isotonic_preserves_overlapping*:

assumes *inclusion_isotonic F xs* $\neq []$ *F is_interval_extension_of f*

shows *contiguous xs* \implies *overlapping (map F xs)*

proof (*induct xs rule: induct_list012*)

case 1

then show *?case*

unfolding *overlapping_def* **by** *simp*

next

case (2 x)

then show *?case*

unfolding *overlapping_def* **by** *simp*

next

case (3 x y zs)

then show *?case*

proof —

have $ao: \forall i < \text{length } (\text{map } F (x \# y \# zs)) - 1. \exists e. e \in_i \text{map } F (x \# y \# zs) ! (i + 1) \wedge e \in_i \text{map } F (x \# y \# zs) ! i$

proof —

have $bo: \text{length } (\text{map } F (x \# y \# zs)) \geq 1$ **using** 3.prem1 **by** *simp*

have $b2: \forall i < \text{length } (x \# y \# zs) - 1. \text{upper } ((x \# y \# zs) ! i) = \text{lower } ((x \# y \# zs) ! (i + 1))$

using 3.prem2 **unfolding** *contiguous_def* **by** *auto[1]*

have $b3: \forall i < \text{length } (x \# y \# zs) - 1. \text{upper } ((x \# y \# zs) ! i) \leq \text{upper } ((x \# y \# zs) ! (i + 1))$

using 3.prem3 **unfolding** *contiguous_def* **by** *auto[1]*

have $b4: \forall i < \text{length } (x \# y \# zs) - 1. \text{lower } ((x \# y \# zs) ! i) \leq \text{lower } ((x \# y \# zs) ! (i + 1))$

using 3.prem4 *lower_le_upper* $b2$ **unfolding** *contiguous_def* **by** *metis*

have $b5: \forall i < \text{length } (x \# y \# zs) - 1. f (\text{upper } ((x \# y \# zs) ! i)) = f (\text{lower } ((x \# y \# zs) ! (i + 1)))$

using $b2$ **by** *simp*

have $b6: \forall i < \text{length } (x \# y \# zs) - 1. f (\text{upper } ((x \# y \# zs) ! i)) \in_i F ((x \# y \# zs) ! i)$

using $\text{assms } b2 b4$ *in_intervall interval_of_in_eq*

unfolding *is_interval_extension_of_def inclusion_isotonic_def*

by (*metis lower_interval_of lower_le_upper upper_interval_of*)

have $b7: \forall i < \text{length } (x \# y \# zs) - 1. f (\text{upper } ((x \# y \# zs) ! i)) \in_i F ((x \# y \# zs) ! (i + 1))$

using $\text{assms } b2 b3$ *in_intervall interval_of_in_eq*


```

unfolding is_interval_extension_of_def inclusion_isotonic_def
by (metis lower_interval_of lower_le_upper upper_interval_of)
have b8:  $\forall i < \text{length } (x \# y \# zs) - 1. f (\text{upper } ((x \# y \# zs) ! i)) \in_i F ((x \# y \# zs) ! (i+1)) \wedge f (\text{upper } ((x \# y \# zs) ! i)) \in_i F ((x \# y \# zs) ! i)$ 
using b6 b7 by blast
show ?thesis using b8 apply simp
by (metis One_nat_def Suc_eq_plus1 le_simps(2) list.map(2) list.size(4) nth_map order_less_le)
qed
show ?thesis using ao unfolding overlapping_def by simp
qed
qed

```

```

lemma bounded_image_of:
fixes f ::  $\langle \text{real} \Rightarrow \text{real} \rangle$ 
assumes  $\langle L\text{-lipschitz\_on } (\text{set\_of } X) f \rangle$ 
shows  $\langle \text{bounded } (f' \text{ set\_of } X) \rangle$ 
using lipschitz_bounded_image_real[of set_of X L f] assms
using bounded_set_of set_of_nonempty by blast

```

```

lemma dist_le_diameter:
fixes f ::  $\langle \text{real} \Rightarrow \text{real} \rangle$ 
assumes  $\langle C\text{-lipschitz\_on } (\text{set\_of } X) f \rangle$ 
shows  $\langle \text{dist } (f (\text{upper } X)) (f (\text{lower } X)) \leq \text{diameter } (f' \text{ set\_of } X) \rangle$ 
using diameter_bounded_bound[of f' set_of X f (upper X) f (lower X), simplified]
    bounded_image_of[of C X f, simplified assms]
by simp

```

```

lemma excess_width_inf_sup:
fixes X :: real interval and f ::  $\langle \text{real} \Rightarrow \text{real} \rangle$ 
assumes  $\langle \text{continuous\_on } (\text{set\_of } X) f \rangle$ 
shows  $\text{Inf } (f' \text{ set\_of } X) - \text{lower } (F X) + \text{upper } (F X) - \text{Sup } (f' \text{ set\_of } X) \leq \text{excess\_width } F f X$ 
using diameter_Sup_Inf[of f' set_of X, simplified compact_img_set_of set_f_nonempty assms]
unfolding excess_width_def width_def width_set_def set_of_eq
by simp

```

```

lemma excess_width_lower_bound:
fixes X :: real interval
assumes inclusion_isotonic F F is_interval_extension_of f  $\langle \text{continuous\_on } (\text{set\_of } X) f \rangle$ 
shows  $\text{Inf } (f' \text{ set\_of } X) - \text{lower } (F X) \leq \text{excess\_width } F f X$ 
proof –
have ao:  $0 \leq \text{upper } (F X) - \text{Sup } (f' \text{ set\_of } X)$ 
using assms(1) assms(2) diff_ge_o_iff_ge inclusion_isotonic_sup
by metis
show ?thesis using ao excess_width_inf_sup assms
by (smt (verit, ccfv_threshold))
qed

```

```

lemma excess_width_upper_bound:
fixes X :: real interval
assumes inclusion_isotonic F F is_interval_extension_of f  $\langle \text{continuous\_on } (\text{set\_of } X) f \rangle$ 
shows  $\text{upper } (F X) - \text{Sup } (f' \text{ set\_of } X) \leq \text{excess\_width } F f X$ 
proof –
have ao:  $0 \leq \text{Inf } (f' \{ \text{lower } X .. \text{upper } X \}) - \text{lower } (F X)$ 

```

```

using assms(1) assms(2) diff_ge_o_iff_ge inclusion_isotonic_sup set_of_eq
inclusion_isotonic_inf
by metis
show ?thesis using ao excess_width_inf_sup assms unfolding set_of_eq
by (smt (verit, ccfv_threshold))
qed

```

lemma lipschitz_excess_width_lower_bound:

```

fixes X :: real interval
assumes inclusion_isotonic F lipschitzL_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  $\text{Inf } (f' \text{ set_of } X) - \text{lower } (F X) \leq C * \text{width } X$ 
proof—
have ao: excess_width F f X  $\leq C * \text{width } X$ 
using each_subdivision_excess_width_order[of F C U f X N L, simplified assms]
assms(4) assms(5) assms(6) div_by_1 insert_iff less_numeral_extra(1) list.set(2) of_nat_1
uniform_subdivision_id
by metis
have a1:  $\text{Inf } (f' \text{ set_of } X) - \text{lower } (F X) \leq \text{excess\_width } F f X$ 
using excess_width_lower_bound[of F f X, simplified assms]
by (metis assms(5) assms(6) lipschitz_on_continuous_on union_set zero_less_one)
show ?thesis using ao a1 by simp
qed

```

lemma lipschitz_excess_width_upper_bound:

```

fixes X :: real interval
assumes inclusion_isotonic F lipschitzL_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  $\text{upper } (F X) - \text{Sup } (f' \text{ set_of } X) \leq C * \text{width } X$ 
proof—
have ao: excess_width F f X  $\leq C * \text{width } X$ 
using each_subdivision_excess_width_order[of F C U f X N L, simplified assms]
assms(4) assms(5) assms(6) div_by_1 insert_iff less_numeral_extra(1) list.set(2) of_nat_1
uniform_subdivision_id
by metis
have a1:  $\text{upper } (F X) - \text{Sup } (f' \text{ set_of } X) \leq \text{excess\_width } F f X$ 
using excess_width_upper_bound[of F f X, simplified assms]
by (metis assms(5) assms(6) lipschitz_on_continuous_on union_set zero_less_one)
show ?thesis using ao a1 by simp
qed

```

lemma excess_width_bound_inf:

```

fixes X :: real interval
assumes excess_width_bound:  $\langle \text{excess\_width } F f X \leq k \rangle$ 
and inclusion_isotonic:  $\langle \text{inclusion\_isotonic } F \rangle$ 
and interval_extension:  $\langle F \text{ is\_interval\_extension\_of } f \rangle$ 
shows  $\langle \text{Inf } (f' \text{ set_of } X) - k \leq \text{lower } (F X) \rangle$ 
using excess_width_bound[simplified excess_width_def width_def width_set_def]
inclusion_isotonic interval_extension
by (smt (verit, best) inclusion_isotonic_sup)

```

lemma excess_width_bound_sup:

```

fixes X :: real interval
assumes excess_width_bound:  $\langle \text{excess\_width } F f X \leq k \rangle$ 
and inclusion_isotonic:  $\langle \text{inclusion\_isotonic } F \rangle$ 
and interval_extension:  $\langle F \text{ is\_interval\_extension\_of } f \rangle$ 
shows  $\langle \text{upper } (F X) \leq \text{Sup } (f' \text{ set\_of } X) + k \rangle$ 
using excess_width_bound[simplified excess_width_def width_def width_set_def]
      inclusion_isotonic interval_extension
by (smt (verit, best) inclusion_isotonic_inf)

```

```

lemma set_of_interval_list_subset_inf_sup:
  assumes non_empty:  $\langle XS \neq [] \rangle$ 
shows  $\langle \text{set\_of\_interval\_list } XS \subseteq \{ \text{Min}(\text{set } (\text{map lower } XS)) .. \text{Max}(\text{set } (\text{map upper } XS)) \} \rangle$ 
using assms unfolding set_of_interval_list_def
proof(induction XS rule:list_nonempty_induct)
  case (single x)
  then show ?case by(simp add:set_of_eq)
next
  case (cons x xs)
  then show ?case
    apply(simp add:set_of_eq)
    by fastforce
qed

```

```

lemma lower_bound_F_inf:
  assumes non_empty:  $\langle XS \neq [] \rangle$ 
and inclusion_isotonic:  $\langle \text{inclusion\_isotonic } F \rangle$ 
and interval_extension:  $\langle F \text{ is\_interval\_extension\_of } f \rangle$ 
and sorted_lower:  $\langle \text{sorted\_wrt\_lower } XS \rangle$ 
and lipschitz:  $\langle 0 \leq C \rangle \langle C - \text{lipschitz\_on } ((\text{set\_of\_interval\_list } XS)) f \rangle$ 
and excess_width_bounded:  $\langle \text{Max } (\text{set } ((\text{map } (\text{excess\_width } F f)) XS)) \leq k \rangle$ 
shows  $\langle (\text{Inf } (f' (\text{set\_of\_interval\_list } XS))) - k \leq \text{Inf } (\text{set\_of\_interval\_list } (\text{map } F XS)) \rangle$ 
using assms proof(induction XS rule:list_nonempty_induct)
  case (single x)
  then show ?case
    using excess_width_bound_inf[of F f x k]
    unfolding set_of_interval_list_def
    by (simp add: inf_set_of)
next
  case (cons x xs)
  then show ?case
    using excess_width_bound_inf[of F f x]
    unfolding set_of_interval_list_def
    apply(simp)
    apply(fold set_of_interval_list_def)
    apply(subst image_Un)
    apply(subst cInf_union_distrib)
    subgoal
      by simp
    subgoal
      apply(simp add: set_of_eq)
      by (meson bounded_imp_bdd_below compact_lcc compact_continuous_image compact_imp_bounded
        lipschitz_on_continuous_on lipschitz_on_subset sup_ge1)
    subgoal
      unfolding set_of_interval_list_def

```

```

by (metis image_is_empty set_of_interval_list_def set_of_interval_list_nonempty)
subgoal
using compact_set_of_interval_list[of XS]
compact_imp_bounded[of (set_of_interval_list XS)]
by (meson bounded_imp_bdd_below compact_continuous_image compact_imp_bounded
compact_set_of_interval_list lipschitz_on_continuous_on lipschitz_on_mono sup_ge2)
subgoal
apply(subst cInf_union_distrib)
subgoal
by simp
subgoal
by (meson bdd_below_set_of)
subgoal
by(simp add: set_of_interval_list_nonempty)
subgoal
using set_of_interval_list_bdd_below by simp
subgoal
proof –
assume a1: xs ≠ []
assume a2: [sorted_wrt_lower xs; C–lipschitz_on (set_of_interval_list xs) f] ⇒ Inf (f' set_of_interval_list xs) – k
≤ Inf (set_of_interval_list (map F xs))
assume a3: sorted_wrt_lower (x # xs)
assume a4: C–lipschitz_on (set_of x ∪ set_of_interval_list xs) f
assume a5: excess_width F f x ≤ k ∧ (∀ a ∈ set xs. excess_width F f a ≤ k)
assume a6: ∧ k. excess_width F f x ≤ k ⇒ Inf (f' set_of x) – k ≤ lower (F x)
have f7: sorted_wrt_lower xs
using a3 a1 sorted_wrt_lower_unroll by blast
have f8: Inf (set_of (F x)) = lower (F x)
using inf_set_of by blast
have f9: C–lipschitz_on (set_of_interval_list xs) f
using a4 lipschitz_on_subset by blast
have f10: inf (Inf (f' set_of x)) (Inf (f' set_of_interval_list xs)) ≤ Inf (f' set_of x)
using inf_le1 by blast
have f11: inf (Inf (f' set_of x)) (Inf (f' set_of_interval_list xs)) ≤ Inf (f' set_of_interval_list xs)
using inf_le2 by blast
have Inf (f' set_of x) – excess_width F f x ≤ lower (F x)
using a6 by blast
have inf (Inf (f' set_of x)) (Inf (f' set_of_interval_list xs)) – k ≤ Inf (set_of (F x))
∧ inf (Inf (f' set_of x)) (Inf (f' set_of_interval_list xs)) – k ≤ Inf (set_of_interval_list (map F xs))
using f11 f10 f9 f8 f7 a5 a2
using <Inf (f' set_of x) – excess_width F f x ≤ lower (F x)> by linarith
then show ?thesis by simp
qed
done
done
qed

```

```

lemma upper_bound_F_sup:
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 ⟨Sup (set_of_interval_list (map F XS)) ≤ (Sup (f' (set_of_interval_list XS))) + k⟩
using assms proof(induction XS rule:list_nonempty_induct)
case (single x)
then show ?case
  using excess_width_bound_sup[of F f x k]
  unfolding set_of_interval_list_def
  by (simp add: sup_set_of)
next
case (cons x xs)
then show ?case
  using excess_width_bound_inf[of F f x]
  unfolding set_of_interval_list_def
  apply(simp)
  apply(fold set_of_interval_list_def)
  apply(subst image_Un)
  apply(subst cSup_union_distrib)
  subgoal
    by simp
  subgoal
    by(simp add: set_of_eq)
  subgoal
    by(simp add: set_of_interval_list_nonempty)
  subgoal
    using set_of_interval_list_bdd_above by simp
  subgoal
    apply(subst cSup_union_distrib)
    subgoal
      by simp
    subgoal
      by (meson bdd_above_mono bdd_above_set_of fundamental_theorem_of_interval_analysis)
    subgoal
      by(simp add: set_of_interval_list_nonempty)
    subgoal
      using set_of_interval_list_bdd_above
      by (meson bounded_imp_bdd_above compact_continuous_image compact_imp_bounded
        compact_set_of_interval_list lipschitz_on_continuous_on lipschitz_on_subset sup_ge2)
    subgoal
      apply(auto)[1]
    subgoal
      by (verit) excess_width_def inclusion_isotonic_inf sup_ge1 sup_set_of width_def width_set_def)
    subgoal
      by (smt (verit, ccfv_threshold) lipschitz_on_subset sorted_wrt_upper_unroll sup_ge2)
    done
  done
done
qed

```

```

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)

```

```

using assms proof(induction XS rule:list_nonempty_induct)
case (single x)
then show ?case
  apply(simp add: set_of_interval_list_def set_of_eq)
  by (metis inclusion_isotonic_inf set_of_eq)
next
case (cons x xs)
then show ?case
  apply(simp add: set_of_interval_list_def set_of_eq)
  apply(subst image_Un)
  apply(subst clnf_union_distrib)
  subgoal
    by simp
  subgoal
    by simp
  subgoal
    proof —
      assume  $xs \neq []$ 
      then have set_of_interval_list (map F xs)  $\neq \{\}$ 
      by (simp add: set_of_interval_list_nonempty)
      then show ?thesis
      by (simp add: set_of_eq set_of_interval_list_def)
    qed
  subgoal
    proof —
      have  $\forall is. \text{bdd\_below} (\text{set\_of\_interval\_list } is :: \text{real set})$ 
      by (simp add: bounded_imp_bdd_below compact_imp_bounded compact_set_of_interval_list)
      then show ?thesis
      by (simp add: set_of_eq set_of_interval_list_def)
    qed
  subgoal
    apply(subst clnf_union_distrib)
    subgoal
      by simp
    subgoal
      by (meson bounded_imp_bdd_below compact_lcc compact_continuous_image compact_imp_bounded
        lipschitz_on_continuous_on lipschitz_on_subset sup_ge1)
    subgoal
      proof —
        assume  $xs \neq []$ 
        then have set_of_interval_list xs  $\neq \{\}$ 
        by (simp add: set_of_interval_list_nonempty)
        then show ?thesis
        by (simp add: set_of_eq set_of_interval_list_def)
      qed
    subgoal
      proof —
        assume  $C\text{-lipschitz\_on} (\{\text{lower } x.. \text{upper } x\} \cup \text{foldr } (\lambda x. (\cup) \{\text{lower } x.. \text{upper } x\}) \text{ xs } \{\}) f$ 
        then have  $C\text{-lipschitz\_on} (\{\text{lower } x.. \text{upper } x\} \cup \text{set\_of\_interval\_list } \text{xs}) f$ 
        by (simp add: set_of_eq set_of_interval_list_def)
        then have bounded (f ' set_of_interval_list xs)
        by (metis compact_imp_bounded compact_set_of_interval_list lipschitz_bounded_image_real lipschitz_on_subset
          sup_ge2)
        then show ?thesis

```

```

    by (simp add: bounded_imp_bdd_below set_of_eq set_of_interval_list_def)
  qed
  subgoal
    apply (simp, rule conjI)
  subgoal
    using inclusion_isotonic_inf [of F f x, simplified assms set_of_eq, simplified]
    by (smt (verit, ccfv_threshold) inclusion_isotonic_inf le_infl1 set_of_eq)
  subgoal
    proof -
      assume a1: xs ≠ []
      assume a2: [sorted_wrt_upper xs; C-lipschitz_on (foldr (λx. (∪) {lower x..upper x}) xs { }) f] ⇒ Inf (foldr (λx.
(∪) {lower x..upper x}) (map F xs) { }) ≤ Inf (f' foldr (λx. (∪) {lower x..upper x}) xs { })
      assume a3: sorted_wrt_upper (x # xs)
      assume C-lipschitz_on ({lower x..upper x} ∪ foldr (λx. (∪) {lower x..upper x}) xs { }) f
      then have C-lipschitz_on (foldr (λi. (∪) {lower i..upper i}) xs { }) f
        using lipschitz_on_mono by blast
      then show ?thesis
        using a3 a2 a1 by (meson le_infl2 sorted_wrt_upper_unroll)
    qed
  done
done
done
qed

```

lemma *Sup_interval_list_approx*: **assumes** *non_empty*: $\langle XS \neq ([] :: \text{real interval list}) \rangle$

```

and inclusion_isotonic:  $\langle \text{inclusion\_isotonic } F \rangle$ 
and interval_extension:  $\langle F \text{ is\_interval\_extension\_of } f \rangle$ 
and sorted_lower:  $\langle \text{sorted\_wrt\_lower } XS \rangle$ 
and lipschitz:  $\langle 0 \leq C \rangle \langle C\text{-lipschitz\_on } ((\text{set\_of\_interval\_list } XS)) f \rangle$ 
and excess_width_bounded:  $\langle (\text{Max } (\text{set } ((\text{map } (\text{excess\_width } F f)) XS))) \leq k \rangle$ 
shows  $\text{Sup } (f' \text{ set\_of\_interval\_list } XS) \leq \text{Sup } (\text{set\_of\_interval\_list } (\text{map } F XS))$ 
using assms proof (induction XS rule: list_nonempty_induct)
case (single x)
then show ?case
  apply (simp add: set_of_interval_list_def set_of_eq)
  by (metis inclusion_isotonic_sup set_of_eq)

```

next

```

case (cons x xs)
then show ?case
  apply (simp add: set_of_interval_list_def set_of_eq)
  apply (subst image_Un)
  apply (subst cSup_union_distrib)
  subgoal
    by simp
  subgoal
    by (meson bounded_imp_bdd_above compact_lcc compact_continuous_image compact_imp_bounded lips-
chitz_on_continuous_on lipschitz_on_subset sup_ge1)
  subgoal
    proof -
      assume a1: xs ≠ []
      have f2:  $\forall R Ra \text{ is\_isa } f fa. ((R :: \text{real set}) \neq Ra \vee \text{is } \neq \text{isa} \vee (\exists R i. (i :: \text{real interval}) \in \text{set } \text{is} \wedge f i R \neq fa i R)) \vee \text{foldr } f \text{ is}$ 
 $R = \text{foldr } fa \text{ isa } Ra$ 
      by (meson foldr_cong)
      obtain ii :: (real interval ⇒ real set ⇒ real set) ⇒ (real interval ⇒ real set ⇒ real set) ⇒ real interval list ⇒ real

```

```

interval and RR :: (real interval  $\Rightarrow$  real set  $\Rightarrow$  real set)  $\Rightarrow$  (real interval  $\Rightarrow$  real set  $\Rightarrow$  real set)  $\Rightarrow$  real interval list  $\Rightarrow$  real
set where
   $\forall x_0 x_1 x_3. (\exists v_6 v_7. v_7 \in \text{set } x_3 \wedge x_1 v_7 v_6 \neq x_0 v_7 v_6) = (ii \ x_0 \ x_1 \ x_3 \in \text{set } x_3 \wedge x_1 \ (ii \ x_0 \ x_1 \ x_3) \ (RR \ x_0 \ x_1 \ x_3) \neq x_0 \ (ii \ x_0 \ x_1 \ x_3) \ (RR \ x_0 \ x_1 \ x_3))$ 
  by moura
  then have  $\forall R \ Ra \ is \ is_a \ f \ fa. (R \neq Ra \vee is \neq is_a \vee ii \ fa \ f \ is \in \text{set } is \wedge f \ (ii \ fa \ f \ is) \ (RR \ fa \ f \ is) \neq fa \ (ii \ fa \ f \ is) \ (RR \ fa \ f \ is))$ 
 $\vee \text{foldr } f \ is \ R = \text{foldr } fa \ is_a \ Ra$ 
  using f2 by presburger
  then show ?thesis
  using a1 by (smt (z3) image_is_empty set_of_eq set_of_interval_list_def set_of_interval_list_nonempty)
qed
subgoal
proof -
  assume C—lipschitz_on ( $\{lower \ x..upper \ x\} \cup \text{foldr } (\lambda x. (\cup) \{lower \ x..upper \ x\}) \ xs \ \{\}) \ f$ 
  then have C—lipschitz_on ( $\{lower \ x..upper \ x\} \cup \text{set\_of\_interval\_list } xs$ ) f
  by (simp add: set_of_eq set_of_interval_list_def)
  then have bounded (f' set_of_interval_list xs)
  by (metis (no_types) compact_imp_bounded compact_set_of_interval_list lipschitz_bounded_image_real lipschitz_on_subset sup_ge2)
  then show ?thesis
  by (simp add: bounded_imp_bdd_above set_of_eq set_of_interval_list_def)
qed
subgoal
apply (subst cSup_union_distrib)
subgoal
by simp
subgoal
by (meson bdd_above_lcc)
subgoal
proof -
  assume xs  $\neq []$ 
  then have set_of_interval_list (map F xs)  $\neq \{\}$ 
  by (simp add: set_of_interval_list_nonempty)
  then show ?thesis
  by (simp add: set_of_eq set_of_interval_list_def)
qed
subgoal
proof -
  have  $\forall is. \text{bdd\_above } (\text{set\_of\_interval\_list } is::\text{real set})$ 
  by (simp add: bounded_imp_bdd_above compact_imp_bounded compact_set_of_interval_list)
  then show ?thesis
  by (simp add: set_of_eq set_of_interval_list_def)
qed
subgoal
apply (simp, rule conjI)
subgoal
using inclusion_isotonic_sup[of F f x, simplified assms set_of_eq, simplified] le_supI1
by auto
subgoal
proof -
  assume a1: xs  $\neq []$ 
  assume a2:  $\llbracket \text{sorted\_wrt\_lower } xs; C\text{---lipschitz\_on } (\text{foldr } (\lambda x. (\cup) \{lower \ x..upper \ x\}) \ xs \ \{\}) \ f \rrbracket \implies \text{Sup } (f' \text{foldr } (\lambda x. (\cup) \{lower \ x..upper \ x\}) \ xs \ \{\}) \leq \text{Sup } (\text{foldr } (\lambda x. (\cup) \{lower \ x..upper \ x\}) \ (\text{map } F \ xs) \ \{\})$ 
  assume a3: sorted_wrt_lower (x # xs)

```



```

assume a4: C—lipschitz_on ({lower x..upper x} ∪ foldr (λx. (∪) {lower x..upper x}) xs {}) f
have f5: sorted_wrt_lower xs
using a3 a1 sorted_wrt_lower_unroll by blast
have f6: Sup (foldr (λi. (∪) {lower i..upper i}) (map F xs) {}) ≤ sup (upper (F x)) (Sup (foldr (λi. (∪) {lower
i..upper i}) (map F xs) {}))
using sup_ge2 by blast
have C—lipschitz_on (foldr (λi. (∪) {lower i..upper i}) xs {}) f
using a4 lipschitz_on_mono by blast
then show ?thesis
using f6 f5 a2 by linarith
qed
done
done
done
qed

```

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>

```

subgoal

unfolding width_set_def

```

using lipschitz_on_nonneg[of C ((set_of_interval_list XS)) f, simplified assms, simplified]
  lower_bound_F_inf[of XS F f C k, simplified assms, simplified]
  upper_bound_F_sup[of XS F f C k, simplified assms, simplified]

```

by(simp)

subgoal

unfolding width_set_def

```

using lipschitz_on_nonneg[of C ((set_of_interval_list XS)) f, simplified assms, simplified]
  Inf_interval_list_approx[of XS F f C k, simplified assms, simplified]
  Sup_interval_list_approx[of XS F f C k, simplified assms, simplified]

```

by simp

done

lemma max_subdivision_excess_width_order:

```

fixes X :: real interval
assumes inclusion_isotonic F lipschitz_l_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(cases (set (map (excess_width F f) (uniform_subdivision X N))) = {})
case True
then show ?thesis
using non_empty_subdivision[of N X, simplified assms, simplified]
by simp
next
case False

```

```

then show ?thesis
apply (subst set_map)
using each_subdivision_excess_width_order[of F C U f X N L, simplified assms, simplified]
      set_of_interval_list_set_eq_interval_list_union_contiguous[of (uniform_subdivision X N), simplified contiguous_uniform_subdivision[of X N] non_empty_subdivision[of N X, simplified assms, simplified], simplified]
      using assms(6) by auto[1]
qed

```

lemma set_of_interval_list_set_eq_interval_list_union_contiguous:

```

assumes non_empty:  $\langle XS \neq ([] :: \text{real interval list}) \rangle$ 
and contiguous:  $\langle \text{contiguous } XS \rangle$ 
shows  $\langle \text{set_of\_interval\_list } XS = \text{set\_of } (\text{interval\_list\_union } XS) \rangle$ 
using interval_list_union_contiguous[of XS, simplified assms, simplified]
      set_of_interval_list_contiguous[of XS, simplified assms, simplified]
      contiguous_non_overlapping[of XS, simplified assms, simplified]
      non_overlapping_implies_sorted_wrt_upper[of XS]
      non_overlapping_implies_sorted_wrt_lower[of XS]
      interval_list_union_contiguous_lower[of XS]
      interval_list_union_contiguous_upper[of XS]
using set_of_eq non_empty by metis

```

lemma width_eq_wdith_set:

```

fixes X ::  $\langle ('a :: \{\text{conditionally\_complete\_lattice, minus, preorder}\}) \text{ interval} \rangle$ 
shows  $\langle \text{width } X = \text{width\_set } (\text{set\_of } X) \rangle$ 
unfolding width_def set_of_eq width_set_def by (simp)

```

lemma width_zero_lower_upper:

```

fixes X :: real interval
assumes  $\langle \text{width } X = 0 \rangle$ 
shows  $\langle \text{lower } X = \text{upper } X \rangle$ 
using assms width_def[of X]
by simp

```

lemma width_zero_mk_interval:

```

fixes X :: real interval
assumes  $\langle \text{width } X = 0 \rangle$ 
shows  $\langle \exists x. X = \text{mk\_interval}(x,x) \rangle$ 
using assms width_def[of X]
unfolding mk_interval'
by (auto, metis Interval_id)

```

lemma width_zero_interval_of:

```

fixes X :: real interval
assumes  $\langle \text{width } X = 0 \rangle$ 
shows  $\langle \exists x. X = \text{interval\_of } x \rangle$ 
using assms width_def[of X]
by (metis eq_iff_diff_eq_0 interval_eqI lower_interval_of upper_interval_of)

```

lemma width_zero_interval_extension:

```

fixes F :: real interval  $\Rightarrow$  real interval
assumes  $\langle F \text{ is\_interval\_extension\_of } f \rangle$ 
and  $\langle \text{width } X = 0 \rangle$ 
shows  $\langle \text{width } (F X) = 0 \rangle$ 

```

```

using assms width_zero_interval_of[of X, simplified assms, simplified]
unfolding is_interval_extension_of_def
by (metis add_o add_diff_cancel lower_interval_of upper_interval_of width_def)

```

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 -dimensional interval vectors then F is Lipschitz in Y

```

lemma interval_extension_bounded:
  fixes  $F :: \text{real interval} \Rightarrow \text{real interval}$ 
  assumes  $\langle F \text{ is\_interval\_extension\_of } f \rangle$ 
  and  $\langle \text{width } (F X) \rangle / \langle \text{width } X \rangle \leq L$ 
shows  $\text{width } (F X) \leq L * \text{width } X$ 
proof(cases width X = 0)
  case True
  then show ?thesis
    using width_zero_interval_extension[of F f X, simplified True assms, simplified]
    by auto
  next
  case False
  then show ?thesis
    using assms interval_width_positive[of X]
    by (metis mult.commute mult_right_mono nonzero_mult_div_cancel_left times_divide_eq_right)
qed

```

```

lemma lipschitz_on_implies_lipschitzl_on:
  fixes  $F :: \text{real interval} \Rightarrow \text{real interval}$ 
  assumes  $\langle F \text{ is\_interval\_extension\_of } f \rangle$ 
  and  $\langle C\text{-lipschitz\_on } X f \rangle$ 
  and  $\langle \bigcup (\text{set\_of } 'Y) \subseteq X \rangle$ 
  and  $\langle 0 \leq L \rangle$ 
  and  $\langle \forall y \in Y. (\text{width } (F y)) / (\text{width } y) \leq L \rangle$ 
shows  $L\text{-lipschitzl\_on } Y F$ 
unfolding lipschitzl_on_def
using assms interval_extension_bounded
by(auto)

```

```

lemma lipschitz_on_implies_lipschitzl_on2:
  fixes  $f :: \text{real} \Rightarrow \text{real}$ 
  assumes  $\langle S \neq [] \rangle$  and  $\langle 0 \leq C \rangle$ 
  and  $\langle F \text{ is\_interval\_extension\_of } f \rangle$ 
  and  $\langle 0 \leq L \rangle$ 
  and  $\langle \forall y \in (\text{set } S). (\text{width } (F y)) / (\text{width } y) \leq L \rangle$ 
  and  $\langle C\text{-lipschitz\_on } (\text{set\_of } (\text{interval\_list\_union } (S))) f \rangle$ 
shows  $\langle L\text{-lipschitzl\_on } (\text{set } (S)) F \rangle$ 
apply(rule lipschitzl_on1)
subgoal using assms interval_extension_bounded by blast
subgoal using assms by blast
done

```

```

lemma width_img_Max:
  assumes  $\langle \text{finite } S \rangle$ 

```

```

shows  $\langle \forall x \in S. \text{width } (F x) \leq \text{Max } (\text{width } 'F ' S) \rangle$ 
using assms by auto
lemma width_Min:
assumes  $\langle \text{finite } S \rangle$ 
shows  $\langle \forall x \in S. \text{Min } (\text{width } ' S) \leq \text{width } x \rangle$ 
using assms by auto

```

```

lemma lipschitzl_on_le_interval:
fixes  $F :: \langle \text{real interval} \Rightarrow \text{real interval} \rangle$ 
assumes  $\text{inc\_isotonic\_}F: \langle \text{inclusion\_isotonic } F \rangle$ 
  and  $\text{lipschitzl\_}F: \langle C - \text{lipschitzl\_on } \{X\} F \rangle$ 
  and  $\text{interval\_inc}: \langle x \leq X \rangle$ 
shows  $\langle \text{width } (F x) \leq C * \text{width } X \rangle$ 
using assms
unfolding inclusion_isotonic_def lipschitzl_on_def
  width_def less_eq_interval_def
by fastforce

```

```

lemma lipschitzl_on_le_lipschitzl_on:
fixes  $F :: \langle \text{real interval} \Rightarrow \text{real interval} \rangle$ 
assumes  $\text{inc\_isotonic\_}F: \langle \text{inclusion\_isotonic } F \rangle$ 
  and  $\text{lipschitzl\_}F: \langle C - \text{lipschitzl\_on } \{X\} F \rangle$ 
  and  $\text{interval\_inc}: \langle x \leq X \rangle$ 
  and  $\text{interval\_ext}: \langle F \text{ is\_interval\_extension\_of } f \rangle$ 
shows  $\langle \exists L. L - \text{lipschitzl\_on } \{x\} F \rangle$ 
apply(rule exI[of  $C * (\text{width } X) / (\text{width } x)$ ])
apply(subst lipschitzl_onI, simp_all add: assms)
subgoal
apply(rule conjI)
subgoal
using width_zero_interval_extension[of  $F f x$ , simplified assms, simplified]
by simp
subgoal
using inc_isotonic_F interval_inc lipschitzl_F lipschitzl_on_le_interval by blast
done
using lipschitzl_on_le_interval[of  $F C X x$ , simplified assms , simplified]
subgoal
using lipschitzl_on_nonneg assms
by (metis divide_nonneg_nonneg interval_width_positive mult_nonneg_nonneg)
done

```

```

lemma uniform_subdivision_le:
fixes  $X :: \langle \text{real interval} \rangle$ 
assumes  $\langle N > 0 \rangle$ 
shows  $\langle \forall x \in \text{set } (\text{uniform\_subdivision } X N). x \leq X \rangle$ 
using last_upper_uniform_subdivision[of  $N X$ , simplified assms, simplified]
  hd_lower_uniform_subdivision[of  $N X$ , simplified assms, simplified]
  non_overlapping_implies_sorted_wrt_upper[of (uniform_subdivision  $X N$ ), simplified uni-
form_subdivisions_non_overlapping assms, simplified]
  non_overlapping_implies_sorted_wrt_lower[of (uniform_subdivision  $X N$ ), simplified uni-
form_subdivisions_non_overlapping assms, simplified]
unfolding sorted_wrt_upper_def sorted_wrt_lower_def cmp_lower_width_def less_eq_interval_def
by (metis (no_types, lifting) assms in_set_conv_nth interval_list_union_correct non_empty_subdivision
set_of_subset_iff union_set)

```

```

lemma lipschitzl_on_uniform_subdivision:
  fixes F :: ‹real interval ⇒ real interval›
  assumes inc_isotonic_F: ‹inclusion_isotonic F›
    and lipschitzl_F: ‹C–lipschitzl_on ({X}) F›
    and ‹N>0›
  shows ‹∀ x∈(set (uniform_subdivision X N)). width (F x) ≤ C * width X›
using lipschitzl_on_le_interval[of F C X, simplified assms, simplified]
  uniform_subdivision_le[of N X, simplified assms, simplified ]
by simp

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 –
  have x * y ≤ x * z using assms by simp
  hence (x * y) / (y * z) ≤ (x * z) / (y * z)
    using assms
    by (simp add: frac_le)
  thus ?thesis using assms by auto[1]
qed

lemma each_subdivision_width_order':
  fixes X :: real interval
  assumes F is_interval_extension_of f
    and 0 < N
    and Xs ∈ set (uniform_subdivision X N)
  shows ∃ L. width(F Xs) ≤ L * width (X) / of_nat N
by (metis assms less_numeral_extra(3) mult_eq_o_iff nonzero_divide_eq_eq
  of_nat_le_o_iff order_refl uniform_subdivisions_width width_zero_interval_extension)

lemma uniform_subdivision_min_nonzero:
  assumes ‹N > 0›
    and ‹width X > 0›
  shows ‹0 < Min (width 'set (uniform_subdivision X N))›
using assms unfolding image_set uniform_subdivision_def Let_def width_def
by (simp, simp add: order_le_less)

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)›
using assms
proof(induction N)
  case 0
  then show ?case
    by simp
next
  case (Suc N)
  then show ?case
    using width_zero_lower_upper[of X, simplified assms, simplified]
    unfolding uniform_subdivision_def

```

by (simp, metis append.right_neutral map_replicate_trivial mk_interval_id replicate_app_Cons_same)
qed

lemma set_of_interval_list_zero_width:

fixes $X :: \langle \text{real interval} \rangle$

assumes positive_N: $\langle 0 < N \rangle$

and zero_width_X: $\langle 0 = \text{width } X \rangle$

shows $\langle \text{set_of_interval_list } (\text{uniform_subdivision } X \ N) = \{\text{lower } X .. \text{upper } X\} \rangle$

proof(insert assms, simp add: uniform_subdivision_width_zero_replicate_eq[*of* $N \ X$, simplified assms, simplified, symmetric], induction N)

case 0

then show ?case

by (simp)

next

case (Suc N)

then show ?case

unfolding set_of_interval_list_def set_of_eq

by (simp, fastforce)

qed

lemma width_zero_subdivision: $\text{width } X = (0 :: \text{real}) \implies N > 0 \implies \text{set } (\text{uniform_subdivision } X \ N) = \{X\}$

using width_zero_lower_upper[*of* X]

unfolding uniform_subdivision_def Let_def mk_interval'

apply (auto simp add: image_def)[1]

apply (metis Interval_id)

apply (metis Interval_id)

done

lemma lipschitz_on_implies_lipschitzl_on_pre:

fixes $f :: \langle \text{real} \Rightarrow \text{real} \rangle$

and $F :: \langle \text{real interval} \Rightarrow \text{real interval} \rangle$

assumes $\langle \text{finite } S \rangle$

and $\langle 0 < \text{Min } (\text{width } 'S) \rangle$

shows $\langle \text{let } \text{max_F_width} = \text{Max } (\text{width } ' (F ' S));$

$\text{min_f_width} = \text{Min } (\text{width } ' S)$

$\text{in } \forall x \in S. \text{width } (F x) \leq (\text{max_F_width} / \text{min_f_width}) * \text{width } x \rangle$

unfolding Let_def

by (simp, smt (verit) assms division_leq_pos interval_width_positive mult_eq_o_iff nonzero_mult_div_cancel_right width_Min width_img_Max zero_le_divide_iff)

lemma lipschitz_on_implies_lipschitzl_on':

fixes $f :: \langle \text{real} \Rightarrow \text{real} \rangle$

and $F :: \langle \text{real interval} \Rightarrow \text{real interval} \rangle$

assumes non_empty: $\langle S \neq \{\} \rangle$

and finite: $\langle \text{finite } S \rangle$

and non_zero_width: $\langle 0 < \text{Min } (\text{width } 'S) \rangle$

and interval_ext_F: $\langle F \text{ is_interval_extension_of } f \rangle$

shows $\langle \exists L. L \text{--lipschitzl_on } S \ F \rangle$

unfolding lipschitzl_on_def

apply (rule exI[*of* $_ (\text{Max } (\text{width } ' (F ' S)) / (\text{Min } (\text{width } ' S)))$])

apply (rule conjI)

subgoal

apply (rule divide_nonneg_nonneg)

subgoal

```

using assms interval_width_positive
by (metis (mono_tags, lifting) Max_in finite_imageE image_is_empty)
subgoal
  using assms interval_width_positive
  by(auto)[1]
done
subgoal
  by (meson assms lipschitz_on_implies_lipschitzl_on_pre)
done

```

lemma natural_extension_transfer_lipschitz:

```

assumes positive_N: <0 < N>
and inc_isotonic_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—lipschitzl_on (set (uniform_subdivision X N)) F>
proof(cases o < width X)
case True
then show ?thesis
apply(subst lipschitzl_onl)
subgoal
  using assms
  each_subdivision_width_order'[of F f N X ]
  each_subdivision_width_order[of F C set (uniform_subdivision X N) f X N ]
  uniform_subdivision_le[of N X ]
  unfolding lipschitz_on_def is_natural_interval_extension_of_def
    inclusion_isotonic_def
  dist_real_def abs_real_def width_def mk_interval'
  apply(auto split:if_splits)[1]
  by (smt (z3) Interval_id interval_of.abs_eq interval_of_in_eq less_eq_interval_def lower_bounds lower_in_interval upper_bounds)
subgoal
  using assms lipschitz_on_nonneg by auto
subgoal by simp
done
next
case False
  have width_zero: <width X = 0>
  using False
  by (meson interval_width_positive linorder_not_le nle_le)
have us_X: <set (uniform_subdivision X N) = {X}>
  using width_zero width_zero_subdivision
  using positive_N by blast
then show ?thesis
  using width_zero
  apply(simp add:assms)
  apply(subst lipschitzl_onl)
  subgoal using assms
  each_subdivision_width_order'[of F f N X ]
  each_subdivision_width_order[of F C set (uniform_subdivision X N) f X N ]
  uniform_subdivision_le[of N X ]
  unfolding lipschitz_on_def is_natural_interval_extension_of_def
    inclusion_isotonic_def

```

```

dist_real_def abs_real_def width_def mk_interval'
apply (auto split:if_splits)[1]
by (meson interval_ext_F natural_interval_extension_implies_interval_extension)
subgoal
  using assms lipschitz_on_nonneg by auto
subgoal by simp
done
qed

```

```

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 –
fix L
have subset: Xs ≤ X
using assms(3) uniform_subdivision_le[of N X]
by (metis (no_types, lifting) bot_nat_o_extremum_strict
  bot_nat_o_not_eq_extremum in_set_conv_nth length_map list.size(3) map_nth
  uniform_subdivision_def)
show ?thesis
by (meson assms(1) interval_set_leq_eq lipschitz_on_subset subset)
qed

```

```

lemma lipschitz_on_lipschitz_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 –
have subset: ∀ Xs ∈ set(uniform_subdivision (X::real interval) N) . Xs ≤ X
using assms uniform_subdivision_le[of N X] by blast
show ∃ L . ∀ Xs ∈ set (uniform_subdivision X N). L ← lipschitz_on (set_of Xs) f
proof –
obtain L where L: L = C using lipschitz_f by auto
have L ← lipschitz_on (set_of Xs) f if Xs ∈ set (uniform_subdivision X N) for Xs
proof –
show ?thesis using lipschitz_on_division_lipschitz_on[of C X f N Xs, simplified assms, simplified] L
by (meson interval_set_leq_eq lipschitz_f lipschitz_on_subset subset that)
qed
thus ?thesis by auto
qed
qed

```

```

lemma lipschitz_on_lipschitz_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 –
have subset: ∀ Xs ∈ set(uniform_subdivision (X::real interval) N) . Xs ≤ X
using assms uniform_subdivision_le[of N X] by blast

```



```

show  $\exists L. \forall N > 0. \forall Xs \in \text{set}(\text{uniform\_subdivision } X N). L \text{--lipschitz\_on}(\text{set\_of } Xs) f$ 
proof –
  obtain  $L$  where  $L: L = C$  using lipschitz_f by auto
  have  $L \text{--lipschitz\_on}(\text{set\_of } Xs) f$  if  $Xs \in \text{set}(\text{uniform\_subdivision } X N)$  for  $Xs$ 
  proof –
    show ?thesis using lipschitz_on_division_lipschitz_on [of C X f N Xs, simplified assms, simplified]  $L$ 
    by (meson interval_set_leq_eq lipschitz_f lipschitz_on_subset subset that)
  qed
  thus ?thesis
  by (meson interval_set_leq_eq lipschitz_f lipschitz_on_subset uniform_subdivision_le)
qed
qed

lemma lipschitzl_on_division_lipschitzl_on:
  assumes lipschitz_f: C--lipschitzl_on (set(uniform_subdivision X N)) F
  and non_empty: uniform_subdivision X N  $\neq$  []
  and subdivision: Xs  $\in$  set(uniform_subdivision (X::real interval) N)
  shows  $\exists L. L \text{--lipschitzl\_on} \{Xs\} F$ 
proof –
  fix  $L$ 
  have subset: Xs  $\leq$  X
  using assms(3) uniform_subdivision_le [of N X]
  by (smt (verit, del_insts) gr_zero1 list.map_disc_iff list.size(3) map_nth non_empty uniform_subdivision_def)
  show ?thesis
  by (meson empty_subset1 insert_subset lipschitzl_on_le lipschitz_f subdivision)
qed

lemma lipschitzl_on_lipschitzl_on_subdivisions:
  fixes  $X :: \text{real interval}$ 
  assumes lipschitz_f: C--lipschitzl_on (set(uniform_subdivision X N)) F
  and non_zero: 0 < N
  shows  $\exists L. \forall Xs \in \text{set}(\text{uniform\_subdivision } X N). L \text{--lipschitzl\_on} \{Xs\} F$ 
proof –
  have subset:  $\forall Xs \in \text{set}(\text{uniform\_subdivision } (X::\text{real interval}) N). Xs \leq X$ 
  using assms uniform_subdivision_le [of N X] by blast
  show  $\exists L. \forall Xs \in \text{set}(\text{uniform\_subdivision } X N). L \text{--lipschitzl\_on} \{Xs\} F$ 
proof –
  obtain  $L$  where  $L: L = C$  using lipschitz_f by auto
  have  $L \text{--lipschitzl\_on} \{Xs\} F$  if  $Xs \in \text{set}(\text{uniform\_subdivision } X N)$  for  $Xs$ 
  proof –
    show ?thesis using assms
    by (simp add: L lipschitzl_on_def that)
  qed
  thus ?thesis by auto
qed
qed

```

9.4 Lipschitz Convergence

```

lemma isotonic_lipschitz_refinement':
  assumes positive_N: <0 < N>
  and inc_isotonic_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 (cases o < width X)
case True
have us_eq_set_of_X: <(set_of_interval_list (uniform_subdivision X N)) = set_of X>
by (simp add: Lipschitz_Subdivisions_Refinements.set_of_interval_list_set_eq_interval_list_union_contiguous
contiguous_uniform_subdivision non_empty_subdivision positive_N union_set)
have lipschitz_f': <C—lipschitz_on (set_of_interval_list (uniform_subdivision X N)) f>
by (simp add: lipschitz_f us_eq_set_of_X)
have us_nonempty: <uniform_subdivision X N ≠ []>
by (simp add: assms(1) non_empty_subdivision)
have us_nonempty_set: <set (uniform_subdivision X N) ≠ {}>
by (simp add: us_nonempty)
have us_finite: <finite (set (uniform_subdivision X N))>
by simp
have sorted_lower: <sorted_wrt_lower (uniform_subdivision X N)>
by (simp add: contiguous_sorted_wrt_lower contiguous_uniform_subdivision)
have sorted_upper: <sorted_wrt_upper (uniform_subdivision X N)>
by (simp add: contiguous_sorted_wrt_upper contiguous_uniform_subdivision)
have lipschitzl: <∃ L. L—lipschitzl_on (set (uniform_subdivision X N)) F>
using lipschitz_on_implies_lipschitzl_on'[of set (uniform_subdivision X N) F f, simplified
us_nonempty_set us_finite interval_ext_F]
uniform_subdivision_min_nonzero[of N X, simplified positive_N True, simplified]
by(simp)
have set_of_interval_eq: <(set_of_interval_list (uniform_subdivision X N)) = (set_of (interval_list_union
(uniform_subdivision X N)))>
by (simp add: Lipschitz_Subdivisions_Refinements.set_of_interval_list_set_eq_interval_list_union_contiguous
contiguous_uniform_subdivision us_nonempty)
have width_bounded: <∃ L. Max (excess_width F f' set (uniform_subdivision X N)) ≤ 2 * (L * width X / real N)>
using
each_subdivision_excess_width_order[of F _ (set (uniform_subdivision X N)) f X N C,
simplified inc_isotonic_F interval_ext_F positive_N lipschitz_f'[simplified set_of_interval_eq], simplified]
max_subdivision_excess_width_order[of F _ (set (uniform_subdivision X N)) f X N C,
simplified inc_isotonic_F interval_ext_F positive_N lipschitz_f', simplified]
proof —
assume a1:  $\bigwedge C. C—lipschitzl_on (set (uniform_subdivision X N)) F \implies$ 
Max (excess_width F f' set (uniform_subdivision X N)) ≤ C * width X / real N
have  $\forall r \ ra \ rb. (r::real) / rb * ra = r * ra / rb$ 
by fastforce
then show ?thesis
using a1 by (metis  $\exists L. L—lipschitzl_on (set (uniform_subdivision X N)) F$ 
divide_numeral_1 le_divide_eq_numeral1(1) mult_numeral_1 order_antisym_conv order_refl)
qed
have width_limit:<∃ 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)>
using width_bounded
map_inclusion_isotonic_excess_width_bounded(1)[of uniform_subdivision X N F f C,
simplified us_nonempty interval_ext_F inc_isotonic_F sorted_lower sorted_upper lipschitz_f', simplified]
by (metis (no_types, lifting) us_eq_set_of_X inc_isotonic_F interval_ext_F lipschitzl lipschitz_f' list.set_map
max_subdivision_excess_width_order order.refl positive_N)
then show ?thesis
by simp

```

```

next
case False
have width_zero: ⟨width X = 0⟩
using False
by (meson interval_width_positive linorder_not_le nle_le)
have us_nonempty: ⟨uniform_subdivision X N ≠ []⟩
by (simp add: assms(1) non_empty_subdivision)
have set_of_interval_eq: ⟨(set_of (interval_list_union (uniform_subdivision X N))) = (set_of_interval_list
(uniform_subdivision X N))⟩
by (simp add: Lipschitz_Subdivisions_Refinements.set_of_interval_list_set_eq_interval_list_union_contiguous
contiguous_uniform_subdivision us_nonempty)
have w_zero_1: ⟨width_set (set_of_interval_list (map F (uniform_subdivision X N))) = 0⟩
by (metis (full_types) Lipschitz_Subdivisions_Refinements.set_of_interval_list_set_eq_interval_list_union_contiguous
contiguous_uniform_subdivision interval_ext_F map_replicate non_empty_subdivision positive_N
uniform_subdivision_width_zero_replicate_eq union_set width_eq_wdith_set width_zero
width_zero_interval_extension)
have w_zero_2: ⟨width_set (f' (set_of X)) = 0⟩
using set_of_interval_list_zero_width[of N X, simplified positive_N width_zero, simplified]
width_zero_width_zero_lower_upper[of X, simplified width_zero, simplified]
by (metis diff_ge_0_iff_ge inc_isotonic_F inf_le_sup_image_real interval_ext_F lipschitz_f_nle_le
width_inclusion_isotonic_approx width_set_def width_zero_interval_extension)
then show ?thesis
using width_zero w_zero_1 w_zero_2
by simp
qed

```

lemma isotonic_lipschitz_refinement1:

```

assumes positive_N: ⟨0 < N⟩
and inc_isotonic_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—lipschitzl_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 (cases 0 < width X)
case True
have us_eq_set_of_X: ⟨(set_of_interval_list (uniform_subdivision X N)) = set_of X⟩
by (simp add: Lipschitz_Subdivisions_Refinements.set_of_interval_list_set_eq_interval_list_union_contiguous
contiguous_uniform_subdivision non_empty_subdivision positive_N union_set)
have lipschitz_f': ⟨L—lipschitz_on (set_of_interval_list (uniform_subdivision X N)) f⟩
by (simp add: lipschitz_f us_eq_set_of_X)
have us_nonempty: ⟨uniform_subdivision X N ≠ []⟩
by (simp add: assms(1) non_empty_subdivision)
have us_nonempty_set: ⟨set (uniform_subdivision X N) ≠ {}⟩
by (simp add: us_nonempty)
have us_finite: ⟨finite (set (uniform_subdivision X N))⟩
by simp
have sorted_lower: ⟨sorted_wrt_lower (uniform_subdivision X N)⟩
by (simp add: contiguous_sorted_wrt_lower contiguous_uniform_subdivision)
have sorted_upper: ⟨sorted_wrt_upper (uniform_subdivision X N)⟩
by (simp add: contiguous_sorted_wrt_upper contiguous_uniform_subdivision)
have lipschitzl: ⟨C—lipschitzl_on (set (uniform_subdivision X N)) F⟩
using lipschitz_F by blast

```

```

have set_of_interval_eq: ⟨(set_of_interval_list (uniform_subdivision X N)) = (set_of (interval_list_union
(uniform_subdivision X N)))⟩
by (simp add: Lipschitz_Subdivisions_Refinements.set_of_interval_list_set_eq_interval_list_union_contiguous
contiguous_uniform_subdivision us_nonempty)
have width_bounded: ⟨Max (excess_width F f' set (uniform_subdivision X N)) ≤ 2 * (C * width X / real N)⟩
using
  each_subdivision_excess_width_order[of F C (set (uniform_subdivision X N)) f X N _,
  simplified inc_isotonic_F interval_ext_F positive_N lipschitz_f'[simplified set_of_interval_eq], simplified]
  max_subdivision_excess_width_order[of F C (set (uniform_subdivision X N)) f X N _,
  simplified inc_isotonic_F interval_ext_F positive_N lipschitz_f', simplified]
by (smt (verit) divide_nonneg_nonneg interval_width_positive lipschitzl_on_def lipschitz_F lipschitz_f'
of_nat_o_less_iff positive_N split_mult_pos_le)
have width_limit:⟨width_set (set_of_interval_list (map F (uniform_subdivision X N)))
– width_set (f' (set_of X))
≤ 2 * (C * width X / real N)⟩
using width_bounded
by (metis assms(3) inc_isotonic_F lipschitz_F lipschitz_f' map_inclusion_isotonic_excess_width_bounded(1)
max_subdivision_excess_width_order order_le_less positive_N sorted_lower sorted_upper us_eq_set_of_X us_nonempty)

then show ?thesis
by simp
next
case False
have width_zero: ⟨width X = 0⟩
using False
by (meson interval_width_positive linorder_not_le nle_le)
have us_nonempty: ⟨uniform_subdivision X N ≠ []⟩
by (simp add: assms(1) non_empty_subdivision)
have set_of_interval_eq: ⟨(set_of (interval_list_union (uniform_subdivision X N))) = (set_of_interval_list
(uniform_subdivision X N))⟩
by (simp add: Lipschitz_Subdivisions_Refinements.set_of_interval_list_set_eq_interval_list_union_contiguous
contiguous_uniform_subdivision us_nonempty)
have w_zero_1: ⟨width_set (set_of_interval_list (map F (uniform_subdivision X N))) = 0⟩
by (metis (full_types) Lipschitz_Subdivisions_Refinements.set_of_interval_list_set_eq_interval_list_union_contiguous
contiguous_uniform_subdivision interval_ext_F map_replicate non_empty_subdivision positive_N
uniform_subdivision_width_zero_replicate_eq union_set width_eq_width_set width_zero
width_zero_interval_extension)
have w_zero_2: ⟨width_set (f' (set_of X)) = 0⟩
using set_of_interval_list_zero_width[of N X, simplified positive_N width_zero, simplified]
width_zero width_zero_lower_upper[of X, simplified width_zero, simplified]
by (simp add: set_of_eq width_set_def)
then show ?thesis
using width_zero w_zero_1 w_zero_2
by simp
qed

lemma isotonic_lipschitz_refinement:
assumes positive_N: ⟨0 < N⟩
and inc_isotonic_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–lipschitzl_on (set (uniform_subdivision X N)) F⟩
shows ⟨excess_width_set (refinement_set F N) f X ≤ 2 * (C * width X / real N)⟩

```

```
using isotonic_lipschitz_refinement![of N F f L X C, simplified assms, simplified]  
unfolding excess_width_set_def refinement_set_def by simp
```

```
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* \Rightarrow '*a interval*

where *inverse_interval* *a* = (
 if ($\neg o \in_i a$) *then* *mk_interval* ($1 / (\text{upper } a), 1 / (\text{lower } a)$)
 else if *lower* *a* = *o* \wedge *o* < *upper* *a* *then* *mk_interval* ($1 / \text{upper } a, \infty$)
 else if *lower* *a* < *o* \wedge *o* < *upper* *a* *then* *mk_interval* ($-\infty, \infty$)
 else if *lower* *a* < *upper* *a* \wedge *upper* *a* = *o* *then* *mk_interval* ($-\infty, 1 / \text{lower } a$)
 else *undefined*
)

definition *divide_interval* :: '*a interval* \Rightarrow '*a interval* \Rightarrow '*a interval*

where *divide_interval* *a b* = *inverse* *b* * *a*

instance ..

end

interpretation *interval_division_inverse* *divide* *inverse*

apply(*unfold_locales*)

subgoal

by (*simp add: inverse_interval_def*)

subgoal

by(*simp add: divide_interval_def*)

done

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
unfolding valid_mInterval_ovl_def
apply(intro exI[where x=[Interval (l,l) ] for l])
by(auto simp add: sorted_wrt_lower_def non_overlapping_sorted_def)
```

setup_lifting type_definition_minterval_ovl

lift_definition mlower_ovl::('a::{minus_mono}) minterval_ovl \Rightarrow 'a is <lower o hd> .

lift_definition mupper_ovl::('a::{minus_mono}) minterval_ovl \Rightarrow 'a is <upper o last> .

lift_definition mlist_ovl::('a::{minus_mono}) minterval_ovl \Rightarrow 'a interval list is <id> .

12.1.2 Equality and Orderings

lemma minterval_ovl_eq_iff: $a = b \iff mlist_ovl\ a = mlist_ovl\ b$
by transfer auto

lemma ainterval_eqI: $mlist_ovl\ a = mlist_ovl\ b \implies a = b$
by (auto simp: minterval_ovl_eq_iff)

lemma minterval_ovl_imp_upper_lower_eq :
 $a = b \implies mlower_ovl\ a = mlower_ovl\ b \wedge mupper_ovl\ a = mupper_ovl\ b$

by *transfer auto*

lemma *valid_mInterval_ovl_lower_le_upper*:

valid_mInterval_ovl i \implies (*lower* \circ *hd*) *i* \leq (*upper* \circ *last*) *i*

proof(*induction i*)

case *Nil*

then show ?*case*

unfolding *valid_mInterval_ovl_def o_def sorted_wrt_lower_def cmp_lower_width_def*

by *simp*

next

case (*Cons a i*)

then show ?*case*

unfolding *valid_mInterval_ovl_def o_def sorted_wrt_lower_def cmp_lower_width_def*

by (*metis* (*no_types*, *lifting*) *Cons.IH Cons.premis comp_apply distinct.simps(2)*)

in_bounds last.simps list.sel(1) lower_in_interval order_less_imp_le order_less_le_trans sorted_wrt_lower_unroll valid_mInterval_ovl_def)

qed

lemma *mLower_non_ovl_le_mUpper_non_ovl*[*simp*]: *mLower_ovl i* \leq *mUpper_ovl i*

by(*transfer*, *metis valid_mInterval_ovl_lower_le_upper*)

lift_definition *set_of_ovl* :: '*a*::{*minus_mono*} *minterval_ovl* \Rightarrow '*a* *set*

is λ *is*. $\bigcup_{x \in \text{set } is} \{ \text{lower } x.. \text{upper } x \}$.

lemma *not_in_ovl_eq*:

$\langle \neg e \in \text{set_of_ovl } xs \rangle = (\forall x \in \text{set } (m\text{list_ovl } xs). \neg e \in \text{set_of } x)$

proof(*induction* (*mList_ovl xs*))

case *Nil*

then show ?*case*

by (*metis UN_iff empty_iff empty_set mList_ovl.rep_eq set_of_ovl.rep_eq*)

next

case (*Cons a x*)

then show ?*case*

by (*simp add: mList_ovl.rep_eq set_of_eq set_of_ovl.rep_eq*)

qed

lemma *in_ovl_eq*:

$\langle e \in \text{set_of_ovl } xs \rangle = (\exists x \in \text{set } (m\text{list_ovl } xs). e \in \text{set_of } x)$

proof(*induction* (*mList_ovl xs*))

case *Nil*

then show ?*case*

by (*metis UN_iff empty_iff empty_set mList_ovl.rep_eq set_of_ovl.rep_eq*)

next

case (*Cons a x*)

then show ?*case*

by (*simp add: mList_ovl.rep_eq set_of_eq set_of_ovl.rep_eq*)

qed

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*

by (*auto simp add: valid_mInterval_ovl_def*)

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

lemma *mInterval_ovl'_split_asm*:
 $P (mInterval_ovl' is) \longleftrightarrow$
 $\neg((\exists ivl. valid_mInterval_ovl is \wedge mlist_ovl ivl = is \wedge \neg P (Some ivl)) \vee (\neg valid_mInterval_ovl is \wedge \neg P None))$
unfolding *mInterval_ovl'_split*
by *auto*

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

lemma *mInterval'_eq_Some*: $mInterval_ovl' is = Some i \implies mlist_ovl i = is$
by (*simp split: mInterval_ovl'_splits*)

end

lemma *set_of_ovl_non_zero_list_all*:
 $\langle 0 \notin set_of_ovl xs \implies \forall x \in set (mlist_ovl xs). \neg 0 \in_i x \rangle$

proof(*induction mlist_ovl xs*)

case *Nil*

then show *?case*

by *simp*

next

case (*Cons a x*)

then show *?case*

using *in_ovl_eq by blast*

qed

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

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

instance proof qed (*simp add: equal_minterval_ovl_def minterval_ovl_eq_iff*)
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 proof qed
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 $\lambda a b. [Interval (inf (lower (hd a)) (lower (hd b))), sup (upper (last a)) (upper (last b))]$

by(*auto simp: valid_mInterval_ovl_def sorted_wrt_lower_def non_adjacent_sorted_wrt_lower_def non_overlapping_sorted_def le_inf1 le_sup1 valid_mInterval_non_ovl_def mupper_ovl_def mlower_ovl_def*)

instance

by(*standard*)

end

instantiation *minterval_ovl* :: (*{lattice, minus_mono}*) *preorder*

begin

instance

apply(*standard*)

subgoal

using *less_minterval_ovl_def* **by** *auto*

subgoal

by (*simp add: less_eq_minterval_ovl_def*)

subgoal

by (*meson less_eq_minterval_ovl_def order.trans*)

done

end

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

unfolding *valid_mInterval_ovl_def valid_mInterval_ovl_def non_adjacent_sorted_wrt_lower_def cmp_non_adjacent_def sorted_wrt_lower_def*

by *simp*

lemma *mlower_ovl_minterval_ovl_of*[*simp*]: *mlower_ovl* (*minterval_ovl_of* *a*) = *a*

by *transfer auto*

lemma *mupper_ovl_minterval_ovl_of*[*simp*]: *mupper_ovl* (*minterval_ovl_of* *a*) = *a*

by *transfer auto*

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* (*o*, *o*)]

by (*simp add: mk_mInterval_ovl_valid*)

lemma *mlower_ovl_zero*[*simp*]: *mlower_ovl* *o* = *o*

by(*transfer, simp add: mk_mInterval_ovl_def interval_sort_lower_width_def*)

lemma *mupper_ovl_zero*[*simp*]: *mupper_ovl* *o* = *o*

by(*transfer, simp add: mk_mInterval_ovl_def interval_sort_lower_width_def*)

instance proof qed

end

instantiation *minterval_ovl* :: (*{minus_mono, one}*) *one*

begin

lift_definition *one_minterval_ovl*::'a *minterval_ovl* is *mk_mInterval_ovl* [*Interval* (1, 1)]
by (*simp add: mk_mInterval_ovl_valid*)

lemma *mlower_ovl_one*[*simp*]: *mlower_ovl* 1 = 1
by(*transfer, simp add: mk_mInterval_ovl_def interval_sort_lower_width_def*)

lemma *mupper_ovl_one*[*simp*]: *mupper_ovl* 1 = 1
by(*transfer, simp add: mk_mInterval_ovl_def interval_sort_lower_width_def*)

instance proof qed
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* \Rightarrow 'a *minterval_ovl* \Rightarrow 'a *minterval_ovl*
is λ a b . *mk_mInterval_ovl* (*iList_plus* a b)
by (*metis bin_op_interval_list_non_empty iList_plus_def mk_mInterval_ovl_valid valid_mInterval_ovl_def*)

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))
by (*metis (no_types, lifting) assms(1) assms(2) bin_op_interval_list_empty iList_plus_lower_upper_eq mk_mInterval_ovl_id mk_mInterval_ovl_valid*)

lift_definition *plus_minterval_non_ovl*::'a *minterval_ovl* \Rightarrow 'a *minterval_ovl* \Rightarrow 'a *minterval_ovl*
is λ a b . *mk_mInterval_ovl* (*iList_plus* a b)
by (*simp add: valid_mk_interval_iList_plus*)

lemma *interval_plus_com*:
 $\langle a + b = b + a \rangle$ **for** *a*::'a::(*{minus_mono, ordered_ab_semigroup_add, linordered_field}*) *minterval_ovl*
apply(*transfer*)
using *plus_minterval_ovl_def*
by (*metis (no_types, opaque_lifting) foldr_isort_elements iList_plus_commute interval_sort_lower_width_def mk_mInterval_ovl_def mk_mInterval_ovl_distinct mk_mInterval_ovl_sorted_o_def set_remdups sorted_wrt_lower_distinct_lists_eq*)

instance proof qed
end

12.1.5 Unary Minus

lemma *a*: (*x*::'a::(*ordered_ab_group_add interval*) \neq *y* \implies $-x \neq -y$
apply(*simp add: uminus_interval_def*)
by (*smt (z3) Pair_inject bounds_of_interval_inverse case_prod_Pair_iden case_prod_unfold neg_equal_iff_equal uminus_interval.rep_eq*)

lemma *b*: *distinct* (*is*::'a::(*ordered_ab_group_add interval list*) \implies *distinct* (*map* (λ i. $-i$) *is*)
proof(*induction is*)
case *Nil*
then show ?*case by simp*
next

```

case (Cons a is)
then show ?case using a by force
qed

```

```

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

```

```

lift_definition uminus_minterval_ovl :: 'a minterval_ovl  $\Rightarrow$  'a minterval_ovl
  is  $\lambda$  is . mk_mInterval_ovl (rev (map ( $\lambda$  i.  $-i$ ) is))
  by (metis list.map_disc_iff mk_mInterval_ovl_valid rev_is_Nil_conv valid_mInterval_ovl_def)
instance ..
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  $\Rightarrow$  'a minterval_ovl  $\Rightarrow$  'a minterval_ovl
  where minus_minterval_ovl a b = a + - b

```

```

instance ..
end

```

12.1.7 Multiplication

```

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

```

```

lift_definition times_minterval_ovl :: 'a minterval_ovl  $\Rightarrow$  'a minterval_ovl  $\Rightarrow$  'a minterval_ovl
  is  $\lambda$  a b . mk_mInterval_ovl (iList_times a b)
  by (metis bin_op_interval_list_non_empty iList_times_def mk_mInterval_ovl_empty
    mk_mInterval_ovl_valid)

```

```

instance ..
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  $\Rightarrow$  'a minterval_ovl
  and divide :: <'a:: {linordered_field, zero, minus, minus_mono, real_normed_algebra, linear_continuum_topology}>
  minterval_ovl  $\Rightarrow$  'a minterval_ovl  $\Rightarrow$  'a minterval_ovl
  assumes inverse_left:  $\langle \neg 0 \in \text{set\_of\_ovl } x \implies 1 \leq (\text{inverse } x) * x \rangle$ 
  and divide:  $\langle \neg 0 \in \text{set\_of\_ovl } y \implies x \leq (\text{divide } x y) * y \rangle$ 
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  $\Rightarrow$  '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* ((_/ ∈_{no}_) [51, 51] 50)
where *in_minterval_ovl* x X ≡ x ∈ set_of_ovl X

lemma *in_minterval_ovl_to_minterval_ovl*[intro!]: a ∈_{no} minterval_ovl_of a
by (metis (mono_tags, lifting) UN_iff list.set_intros(1) lower_in_interval lower_point_interval
minterval_ovl_of.rep_eq set_of_eq set_of_ovl.rep_eq)

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

lemma *set_of_one_ovl*[simp]: set_of_ovl (1::'a::{one, order, minus_mono} minterval_ovl) = {1}
apply (auto simp: set_of_ovl_def)[1]
subgoal
by (simp add: interval_sort_lower_width_set_eq mk_mInterval_ovl_def one_minterval_ovl.rep_eq)
subgoal
by (simp add: interval_sort_lower_width_set_eq mk_mInterval_ovl_def one_minterval_ovl.rep_eq)
done

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
unfolding *valid_mInterval_non_ovl_def*
apply(intro exI[where x=[Interval (l,l)] for l])
by(auto simp add: valid_mInterval_ovl_def sorted_wrt_lower_def non_adjacent_sorted_wrt_lower_def)

setup_lifting type_definition_minterval_non_ovl

lift_definition mlower_non_ovl::('a::{minus_mono}) minterval_non_ovl \Rightarrow 'a is <lower o hd> .

lift_definition mupper_non_ovl::('a::{minus_mono}) minterval_non_ovl \Rightarrow 'a is <upper o last> .

lift_definition mlist_non_ovl::('a::{minus_mono}) minterval_non_ovl \Rightarrow 'a interval list is <id> .

12.2.2 Equality and Orderings

lemma minterval_non_ovl_eq_iff: $a = b \iff mlist_non_ovl\ a = mlist_non_ovl\ b$
by transfer auto

lemma ainterval_eq!: $mlist_non_ovl\ a = mlist_non_ovl\ b \implies a = b$
by (auto simp: minterval_non_ovl_eq_iff)

lemma minterval_non_ovl_imp_upper_lower_eq :
 $a = b \implies mlower_non_ovl\ a = mlower_non_ovl\ b \wedge mupper_non_ovl\ a = mupper_non_ovl\ b$
by transfer auto

lemma mlower_non_ovl_le_mupper_non_ovl[simp]: $mlower_non_ovl\ i \leq mupper_non_ovl\ i$
by (transfer, metis comp_def hd_Nil_eq_last lower_le_upper lower_le_upper_aux
non_adjacent_implies_non_overlapping valid_mInterval_non_ovl_def)

lift_definition set_of_non_ovl :: 'a::{minus_mono} minterval_non_ovl \Rightarrow 'a set
is λ is. $\bigcup x \in \text{set is. } \{ \text{lower } x .. \text{upper } x \}$.

lemma set_non_ovl_of_subset: $\text{set_of_non_ovl}\ (x::'a::\text{minus_mono}\ \text{minterval_non_ovl}) \subseteq \{ \text{mlower_non_ovl}\ x .. \text{mupper_non_ovl}\ x \}$
apply (transfer, simp)
using set_of_subeq_aux
mInterval_ovl_lower_hd_min[symmetric, simplified o_def]
mInterval_adj_upper_last_max[symmetric, simplified o_def]
valid_non_ovl_imp_adj valid_non_ovl_imp_ovl
list.set_map
by (metis (mono_tags, lifting))

lemma not_in_non_ovl_eq:
 $\langle \neg e \in \text{set_of_non_ovl}\ xs \rangle = \langle \forall x \in \text{set}\ (mlist_non_ovl\ xs). \neg e \in \text{set_of}\ x \rangle$
proof(induction (mlist_non_ovl xs))
case Nil
then show ?case
by (metis UN_iff empty_iff empty_set mlist_non_ovl.rep_eq set_of_non_ovl.rep_eq)
next
case (Cons a x)
then show ?case
by (simp add: mlist_non_ovl.rep_eq set_of_eq set_of_non_ovl.rep_eq)
qed

lemma in_non_ovl_eq:
 $\langle e \in \text{set_of_non_ovl}\ xs \rangle = \langle \exists x \in \text{set}\ (mlist_non_ovl\ xs). e \in \text{set_of}\ x \rangle$
proof(induction (mlist_non_ovl xs))
case Nil

```

then show ?case
  by (metis UN_iff empty_iff empty_set mlist_non_ovl.rep_eq set_of_non_ovl.rep_eq)
next
case (Cons a x)
then show ?case
  by (simp add: mlist_non_ovl.rep_eq set_of_eq set_of_non_ovl.rep_eq)
qed

```

```

lemma set_of_non_ovl_non_zero_list_all:
  <0 ∉ set_of_non_ovl xs ⟹ ∀ x ∈ set (mlist_non_ovl xs). ¬ 0 ∈i x>
proof(induction mlist_non_ovl xs)
case Nil
then show ?case
  by simp
next
case (Cons a x)
then show ?case
  using in_non_ovl_eq by blast
qed

```

```

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
by (auto simp add: valid_mInterval_non_ovl_def)

```

```

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)
by transfer auto

```

```

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))
unfolding mInterval_non_ovl'_split
by auto

```

```

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
  by (simp split: mInterval_non_ovl'_splits)

```

```

end

```

```

instantiation minterval_non_ovl :: ({minus_mono}) equal
begin

```

```

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

```

```

instance proof qed (simp add: equal_minterval_non_ovl_def minterval_non_ovl_eq_iff)
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 \longleftrightarrow *mlower_non_ovl* b \leq *mlower_non_ovl* a \wedge *mupper_non_ovl* a \leq *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 \leq y \wedge \neg y \leq x)

instance proof qed

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)))]

by(*auto simp: valid_mInterval_ovl_def sorted_wrt_lower_def non_adjacent_sorted_wrt_lower_def non_overlapping_sorted_def le_inf1 le_sup1 valid_mInterval_non_ovl_def mupper_non_ovl_def mlower_non_ovl_def*)

lemma *mlower_non_ovl_sup[simp]*: *mlower_non_ovl* (*sup* A B) = *inf* (*mlower_non_ovl* A) (*mlower_non_ovl* B)

apply(*transfer*)

by (*metis comp_apply le_supE le_sup1 list.sel(1) lower_bounds lower_le_upper_aux sup_inf_absorb valid_mInterval_adj_def valid_non_ovl_imp_adj*)

lemma *mupper_non_ovl_sup[simp]*: *mupper_non_ovl* (*sup* A B) = *sup* (*mupper_non_ovl* A) (*mupper_non_ovl* B)

apply(*transfer*)

by (*metis (no_types, lifting) comp_def inf_sup_absorb last.simps le_inf1 le_inf_iff lower_le_upper_aux upper_bounds valid_mInterval_adj_def valid_non_ovl_imp_adj*)

instance

by(*standard*)

end

instantiation *minterval_non_ovl* :: (*{lattice,minus_mono}*) preorder

begin

instance

apply(*standard*)

subgoal

using *less_minterval_non_ovl_def* by *auto*

subgoal

by (*simp add: less_eq_minterval_non_ovl_def*)

subgoal

by (*meson less_eq_minterval_non_ovl_def order.trans*)

done

end

lemma *set_of_minterval_non_ovl_union*: *set_of_non_ovl* A \cup *set_of_non_ovl* B \subseteq *set_of_non_ovl* (*sup* A B)

for A: 'a::(*{lattice, minus_mono}*) *minterval_non_ovl*

apply(*transfer, simp*)

using *set_of_subeq_aux*

mInterval_ovl_lower_hd_min[*symmetric, simplified o_def*]

mInterval_adj_upper_last_max[*symmetric, simplified o_def*]

```

    valid_non_ovl_imp_adj valid_non_ovl_imp_ovl
    list.set_map
by (smt (verit) le_sup_iff lower_bounds lower_le_upper_aux lower_sup set_of_eq set_of_subset_iff
      sup commute sup commute sup.order_iff sup_ge1 sup_ge1 sup_inf_absorb upper_bounds
      valid_mInterval_adj_def)

lemma minterval_non_ovl_union_commute: sup A B = sup B A for A :: 'a :: {minus_mono, lattice} minterval_non_ovl
apply (auto simp add: minterval_non_ovl_eq_iff inf commute sup commute)[1]
by (simp add: mlist_non_ovl.rep_eq inf commute sup_minterval_non_ovl.rep_eq sup commute)

lemma minterval_non_ovl_union_mono1: set_of_non_ovl a  $\subseteq$  set_of_non_ovl (sup a A)
for A :: 'a :: {minus_mono, lattice} minterval_non_ovl
apply (transfer, simp)
using set_of_subseq_aux
    mInterval_ovl_lower_hd_min[symmetric, simplified o_def]
    mInterval_adj_upper_last_max[symmetric, simplified o_def]
    valid_non_ovl_imp_adj valid_non_ovl_imp_ovl
    list.set_map
by (smt (verit, del_insts) inf.absorb_iff2 inf_le1 le_inf1 lower_bounds lower_le_upper_aux
      set_of_eq set_of_subset_iff sup_ge1 upper_bounds valid_mInterval_adj_def)

lemma minterval_non_ovl_union_mono2: set_of_non_ovl A  $\subseteq$  set_of_non_ovl (sup a A) for A :: 'a :: {lattice, minus_mono}
minterval_non_ovl
apply (transfer, simp)
using set_of_subseq_aux
    mInterval_ovl_lower_hd_min[symmetric, simplified o_def]
    mInterval_adj_upper_last_max[symmetric, simplified o_def]
    valid_non_ovl_imp_adj valid_non_ovl_imp_ovl
    list.set_map
by (smt (verit, del_insts) inf.absorb_iff2 le_sup_iff lower_bounds lower_le_upper_aux nle_le
      set_of_eq set_of_subset_iff sup_inf_absorb upper_bounds valid_mInterval_adj_def)

lift_definition minterval_non_ovl_of :: 'a :: {minus_mono}  $\Rightarrow$  'a minterval_non_ovl is  $\lambda x. [Interval(x, x)]$ 
unfolding valid_mInterval_non_ovl_def valid_mInterval_ovl_def non_adjacent_sorted_wrt_lower_def
    cmp_non_adjacent_def sorted_wrt_lower_def
by simp

lemma mlower_non_ovl_minterval_non_ovl_of[simp]: mlower_non_ovl (minterval_non_ovl_of a) = a
by transfer auto

lemma mupper_non_ovl_minterval_non_ovl_of[simp]: mupper_non_ovl (minterval_non_ovl_of a) = a
by transfer auto

definition width_non_ovl :: 'a :: {minus_mono} minterval_non_ovl  $\Rightarrow$  '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 (o, o)]
by (simp add: mk_mInterval_non_ovl_valid sorted_wrt_lower_def)

```

lemma *mlower_non_ovl_zero*[simp]: $mlower_non_ovl\ 0 = 0$
by(transfer, simp add: mk_mInterval_non_ovl_def mk_mInterval_ovl_def interval_sort_lower_width_def)

lemma *mupper_non_ovl_zero*[simp]: $mupper_non_ovl\ 0 = 0$
by(transfer, simp add: mk_mInterval_non_ovl_def mk_mInterval_ovl_def interval_sort_lower_width_def)

instance proof qed
end

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

lift_definition *one_mininterval_non_ovl*:: 'a *mininterval_non_ovl* is *mk_mInterval_non_ovl* [Interval (1, 1)]
by (metis list.simps(3) mk_mInterval_non_ovl_single mk_mInterval_non_ovl_valid
sorted_wrt_lower_mk_mInterval_non_ovl)

lemma *mlower_non_ovl_one*[simp]: $mlower_non_ovl\ 1 = 1$
by(transfer, simp add: mk_mInterval_non_ovl_def mk_mInterval_ovl_def interval_sort_lower_width_def)

lemma *mupper_non_ovl_one*[simp]: $mupper_non_ovl\ 1 = 1$
by(transfer, simp add: mk_mInterval_non_ovl_def mk_mInterval_ovl_def interval_sort_lower_width_def)

instance proof qed
end

12.2.4 Addition

instantiation *mininterval_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))*
by (metis (no_types, lifting) assms(1) assms(2) bin_op_interval_list_empty iList_plus_lower_upper_eq
mk_mInterval_non_ovl_id mk_mInterval_non_ovl_non_empty mk_mInterval_non_ovl_valid
sorted_wrt_lower_mk_mInterval_non_ovl)

lift_definition *plus_mininterval_non_ovl*:: 'a *mininterval_non_ovl* \Rightarrow 'a *mininterval_non_ovl* \Rightarrow 'a *mininterval_non_ovl*
is $\lambda a\ b. mk_mInterval_non_ovl\ (iList_plus\ a\ b)$
by (simp add: valid_mk_interval_iList_plus)

lemma *interval_plus_com*:
 $\langle a + b = b + a \rangle$ **for** $a :: 'a :: \{minus_mono, ordered_ab_semigroup_add, linordered_field\}$ *mininterval_non_ovl*
apply(transfer)
using *iList_plus_mInterval_non_ovl_commute plus_mininterval_non_ovl_def*
by auto

instance proof qed

end

12.2.5 Unary Minus

lemma *a*: $(x :: 'a :: ordered_ab_group_add\ interval) \neq y \implies -x \neq -y$

```

apply(simp add:uminus_interval_def)
by (smt (z3) Pair_inject bounds_of_interval_inverse case_prod_Pair_iden case_prod_unfold neg_equal_iff_equal uminus_interval.rep_eq)

```

lemma *b: distinct (is::'a::ordered_ab_group_add interval list) \implies distinct (map ($\lambda i. -i$) is)*

proof(induction is)

case Nil

then show ?case **by** simp

next

case (Cons a is)

then show ?case **using** a **by** force

qed

instantiation *mininterval_non_ovl :: ({minus_mono, ordered_ab_group_add}) uminus*

begin

lift_definition *uminus_mininterval_non_ovl::'a mininterval_non_ovl \Rightarrow 'a mininterval_non_ovl*

is $\lambda is . mk_mInterval_non_ovl (rev (map (\lambda i. -i) is))$

by (metis (no_types, lifting) distinct_remdups_id list.map_disc_iff mk_mInterval_non_ovl_def

mk_mInterval_ovl_sorted non_adj_sorted_mkInterval_non_ovl non_adjacent_implies_distinct

o_def rev_is_Nil_conv valid_mInterval_non_ovl_def valid_mInterval_ovl_def

valid_ovl_mkInterval_non_ovl)

instance ..

end

12.2.6 Subtraction

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

begin

definition *minus_mininterval_non_ovl::'a mininterval_non_ovl \Rightarrow 'a mininterval_non_ovl \Rightarrow 'a mininterval_non_ovl*

where minus_mininterval_non_ovl a b = a + - b

instance ..

end

12.2.7 Multiplication

instantiation *mininterval_non_ovl :: ({minus_mono, linordered_field}) times*

begin

lift_definition *times_mininterval_non_ovl::'a mininterval_non_ovl \Rightarrow 'a mininterval_non_ovl \Rightarrow 'a mininterval_non_ovl*

is $\lambda a b . mk_mInterval_non_ovl (iList_times a b)$

by (metis (no_types, lifting) bin_op_interval_list_empty distinct_remdups_id iList_times_def

mk_mInterval_non_ovl_def mk_mInterval_ovl_sorted non_adj_sorted_mkInterval_non_ovl

non_adjacent_implies_distinct o_def valid_mInterval_non_ovl_def valid_mInterval_ovl_def

valid_ovl_mkInterval_non_ovl)

instance ..

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 ((_/ ∈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 ∈no minterval_non_ovl of a
  by (simp add: minterval_non_ovl_of.rep_eq set_of_non_ovl_def)
```

```
instance minterval_non_ovl :: ({one, preorder, minus_mono, linordered_semiring}) power
proof qed
```

```
lemma set_of_one_non_ovl[simp]: set_of_non_ovl (1::'a::{one, minus_mono, order} minterval_non_ovl) = {1}
  apply (transfer)
  by (auto simp: mk_mInterval_non_ovl_def mk_mInterval_ovl_def interval_sort_lower_width_def set_of_non_ovl_def)
```

```
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
unfolding valid_mInterval_adj_def
apply(intro ex1[where x=[Interval (l,l) ] for l])
by(auto simp add: sorted_wrt_lower_def non_overlapping_sorted_def)

```

```

setup_lifting type_definition_minterval_adj

```

```

lift_definition mlower_adj::('a::{minus_mono}) minterval_adj  $\Rightarrow$  'a is <lower o hd> .
lift_definition mupper_adj::('a::{minus_mono}) minterval_adj  $\Rightarrow$  'a is <upper o last> .
lift_definition mlist_adj::('a::{minus_mono}) minterval_adj  $\Rightarrow$  'a interval list is <id> .

```

12.3.2 Equality and Orderings

```

lemma minterval_adj_eq_iff: a = b  $\iff$  mlist_adj a = mlist_adj b
by transfer auto

```

```

lemma ainterval_eq!: mlist_adj a = mlist_adj b  $\implies$  a = b
by (auto simp: minterval_adj_eq_iff)

```

```

lemma minterval_adj_imp_upper_lower_eq :
  a = b  $\implies$  mlower_adj a = mlower_adj b  $\wedge$  mupper_adj a = mupper_adj b
by transfer auto

```

```

lemma mlower_adj_le_mupper_adj[simp]: mlower_adj i  $\leq$  mupper_adj i
by (transfer, metis comp_def lower_le_upper_aux valid_mInterval_adj_def)

```

```

lift_definition set_of_adj :: 'a::{minus_mono} minterval_adj  $\Rightarrow$  'a set
is  $\lambda$  is.  $\bigcup_{x \in \text{set } is} \{ \text{lower } x .. \text{upper } x \}$  .

```

```

lemma set_adj_of_subset: set_of_adj (x::'a::{minus_mono} minterval_adj)  $\subseteq$  {mlower_adj x .. mupper_adj x}
apply(transfer, simp)
using set_of_subeq_aux
  mInterval_ovl_lower_hd_min[symmetric, simplified o_def]
  mInterval_adj_upper_last_max[symmetric, simplified o_def]
  valid_adj_imp_ovl
  list.set_map
by (smt (verit, best))

```

```

lemma not_in_adj_eq:
   $\langle \neg e \in \text{set\_of\_adj } xs = (\forall x \in \text{set } (mlist\_adj \text{ } xs). \neg e \in \text{set\_of } x) \rangle$ 
proof(induction (mlist_adj xs))
case Nil
then show ?case
by (metis UN_iff empty_iff empty_set mlist_adj.rep_eq set_of_adj.rep_eq)
next
case (Cons a x)
then show ?case
by (simp add: mlist_adj.rep_eq set_of_eq set_of_adj.rep_eq)
qed

```

lemma *in_adj_eq*:
 $\langle e \in \text{set_of_adj } xs = (\exists x \in \text{set } (m\text{list_adj } xs). e \in \text{set_of } x) \rangle$
proof(*induction* (*mlist_adj* *xs*))
case *Nil*
then show ?*case*
by (*metis UN_iff empty_iff empty_set mlist_adj.rep_eq set_of_adj.rep_eq*)
next
case (*Cons a x*)
then show ?*case*
by (*simp add: mlist_adj.rep_eq set_of_eq set_of_adj.rep_eq*)
qed

lemma *set_of_adj_non_zero_list_all*:
 $\langle 0 \notin \text{set_of_adj } xs \implies \forall x \in \text{set } (m\text{list_adj } xs). \neg 0 \in_i x \rangle$
proof(*induction* *mlist_adj* *xs*)
case *Nil*
then show ?*case*
by *simp*
next
case (*Cons a x*)
then show ?*case*
using *in_adj_eq* **by** *blast*
qed

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

lift_definition (*code_dt*) *mInterval_adj'*::'*a*::*minus_mono* *interval list* \implies '*a* *minterval_adj* *option*
is λ *is*. *if* *valid_mInterval_adj* *is* *then* *Some is* *else* *None*
by (*auto simp add: valid_mInterval_adj_def*)

lemma *mInterval_adj'_split*:
 $P (m\text{Interval_adj}' \text{ is}) \longleftrightarrow$
 $(\forall \text{ivl. } \text{valid_mInterval_adj } \text{is} \longrightarrow m\text{list_adj } \text{ivl} = \text{is} \longrightarrow P (\text{Some } \text{ivl})) \wedge (\neg \text{valid_mInterval_adj } \text{is} \longrightarrow P \text{ None})$
by *transfer auto*

lemma *mInterval_adj'_split_asm*:
 $P (m\text{Interval_adj}' \text{ is}) \longleftrightarrow$
 $\neg((\exists \text{ivl. } \text{valid_mInterval_adj } \text{is} \wedge m\text{list_adj } \text{ivl} = \text{is} \wedge \neg P (\text{Some } \text{ivl})) \vee (\neg \text{valid_mInterval_adj } \text{is} \wedge \neg P \text{ None}))$
unfolding *mInterval_adj'_split*
by *auto*

lemmas *mInterval_adj'_splits* = *mInterval_adj'_split mInterval_adj'_split_asm*

lemma *mInterval'_eq_Some*: *mInterval_adj' is* = *Some i* \implies *mlist_adj i* = *is*
by (*simp split: mInterval_adj'_splits*)

end

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

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

instance proof qed (simp add: equal_minterval_adj_def minterval_adj_eq_iff)
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 proof qed
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))]
by (auto simp: valid_minterval_ovl_def sorted_wrt_lower_def non_adjacent_sorted_wrt_lower_def
non_overlapping_sorted_def le_inf1 le_sup1 valid_minterval_adj_def mupper_adj_def
mlower_adj_def)

lemma mlower_adj_sup[simp]: mlower_adj (sup A B) = inf (mlower_adj A) (mlower_adj B)
apply (transfer)
by (metis comp_apply le_supE le_sup1 list.sel(1) lower_bounds lower_le_upper_aux sup_inf_absorb
valid_minterval_adj_def)

lemma mupper_adj_sup[simp]: mupper_adj (sup A B) = sup (mupper_adj A) (mupper_adj B)
apply (transfer)
by (metis (no_types, lifting) comp_def inf_sup_absorb last.simps le_inf1 le_inf_iff lower_le_upper_aux
upper_bounds valid_minterval_adj_def)

instance
by (standard)
end

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

instance
apply (standard)
subgoal
using less_minterval_adj_def **by** auto
subgoal
by (simp add: less_eq_minterval_adj_def)
subgoal
by (meson less_eq_minterval_adj_def order.trans)
done
end

lemma set_of_minterval_adj_union: set_of_adj A \cup set_of_adj B \subseteq set_of_adj (sup A B)
for A: 'a::($\{lattice, minus_mono\}$) minterval_adj
apply (transfer, simp)
using set_of_subeq_aux

```

mInterval_ovl_lower_hd_min[symmetric, simplified o_def]
mInterval_adj_upper_last_max[symmetric, simplified o_def]
valid_adj_imp_ovl
list.set_map
by (smt (verit) le_sup_iff lower_bounds lower_le_upper_aux lower_sup set_of_eq set_of_subset_iff
sup commute sup commute sup.order_iff sup_ge1 sup_ge1 sup_inf_absorb upper_bounds
valid_mInterval_adj_def)

lemma minterval_adj_union_commute: sup A B = sup B A for A :: 'a:: {minus_mono, lattice} minterval_adj
apply (auto simp add: minterval_adj_eq_iff inf commute sup commute) [1]
by (simp add: mlist_adj.rep_eq inf_commute sup_minterval_adj.rep_eq sup_commute)

lemma minterval_adj_union_mono1: set_of_adj a  $\subseteq$  set_of_adj (sup a A)
for A :: 'a:: {minus_mono, lattice} minterval_adj
apply (transfer, simp)
using set_of_subeq_aux
mInterval_ovl_lower_hd_min[symmetric, simplified o_def]
mInterval_adj_upper_last_max[symmetric, simplified o_def]
valid_adj_imp_ovl
list.set_map
by (smt (verit, del_insts) inf_absorb_iff2 inf_le1 le_inf1 lower_bounds lower_le_upper_aux
set_of_eq set_of_subset_iff sup_ge1 upper_bounds valid_mInterval_adj_def)

lemma minterval_adj_union_mono2: set_of_adj A  $\subseteq$  set_of_adj (sup a A) for A :: 'a:: {lattice, minus_mono} minterval_adj
apply (transfer, simp)
using set_of_subeq_aux
mInterval_ovl_lower_hd_min[symmetric, simplified o_def]
mInterval_adj_upper_last_max[symmetric, simplified o_def]
valid_adj_imp_ovl
list.set_map
by (smt (verit, del_insts) inf_absorb_iff2 le_sup_iff lower_bounds lower_le_upper_aux nle_le
set_of_eq set_of_subset_iff sup_inf_absorb upper_bounds valid_mInterval_adj_def)

lift_definition minterval_adj_of :: 'a:: {minus_mono}  $\Rightarrow$  'a minterval_adj is  $\lambda x. [Interval(x, x)]$ 
unfolding valid_mInterval_adj_def valid_mInterval_ovl_def non_adjacent_sorted_wrt_lower_def
cmp_non_adjacent_def sorted_wrt_lower_def non_overlapping_sorted_def
by simp

lemma mlower_adj_minterval_adj_of [simp]: mlower_adj (minterval_adj_of a) = a
by transfer auto

lemma mupper_adj_minterval_adj_of [simp]: mupper_adj (minterval_adj_of a) = a
by transfer auto

definition width_adj :: 'a:: {minus_mono} minterval_adj  $\Rightarrow$  '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 (o, o)]
by (simp add: mk_mInterval_adj_valid)

```

lemma *mlower_adj_zero*[simp]: *mlower_adj* 0 = 0
by(transfer, simp add: mk_mInterval_adj_def mk_mInterval_ovl_def interval_sort_lower_width_def)

lemma *mupper_adj_zero*[simp]: *mupper_adj* 0 = 0
by(transfer, simp add: mk_mInterval_adj_def mk_mInterval_ovl_def interval_sort_lower_width_def)

instance proof qed
end

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

lift_definition *one_minterval_adj*:: '*a minterval_adj* is *mk_mInterval_adj* [Interval (1, 1)]
by (simp add: mk_mInterval_adj_valid)

lemma *mlower_adj_one*[simp]: *mlower_adj* 1 = 1
by(transfer, simp add: mk_mInterval_adj_def mk_mInterval_ovl_def interval_sort_lower_width_def)

lemma *mupper_adj_one*[simp]: *mupper_adj* 1 = 1
by(transfer, simp add: mk_mInterval_adj_def mk_mInterval_ovl_def interval_sort_lower_width_def)

instance proof qed
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* \Rightarrow '*a minterval_adj* \Rightarrow '*a minterval_adj*
is $\lambda a b . mk_mInterval_adj (iList_plus a b)$
by (metis bin_op_interval_list_empty iList_plus_def mk_mInterval_adj_valid valid_mInterval_adj_def)

instance proof qed

lemma *interval_plus_com*:
 $\langle a + b = b + a \rangle$ for *a*:: '*a minterval_adj*
apply(transfer)
using *iList_plus_mInterval_adj_commute plus_minterval_adj_def*
by(auto)

end

12.3.5 Unary Minus

lemma *a*: (*x*:: '*a*::*ordered_ab_group_add interval*) $\neq y \implies -x \neq -y$
apply(simp add: *uminus_interval_def*)
by (smt (z3) Pair_inject bounds_of_interval_inverse case_prod_Pair_iden case_prod_unfold neg_equal_iff_equal uminus_interval.rep_eq)

lemma *b*: *distinct* (*is*:: '*a*::*ordered_ab_group_add interval list*) \implies *distinct* (*map* ($\lambda i . -i$) *is*)
proof(induction *is*)
case Nil

```

then show ?case by simp
next
case (Cons a is)
then show ?case using a by force
qed

```

```

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))
by (metis (no_types, opaque_lifting) list.map_disc_iff mk_mInterval_non_ovl_id mk_mInterval_non_ovl_non_empty
  mk_mInterval_non_ovl_valid rev_is_Nil_conv sorted_wrt_lower_mk_mInterval_non_ovl
  valid_non_ovl_imp_adj)

```

```

instance ..
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 ..
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)
by (metis bin_op_interval_list_empty iList_times_def mk_mInterval_non_ovl_id
  mk_mInterval_non_ovl_non_empty mk_mInterval_non_ovl_valid sorted_wrt_lower_mk_mInterval_non_ovl
  valid_non_ovl_imp_adj)

```

```

instance ..
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 ((_/ ∈adj _) [51, 51] 50)
  where in_minterval_adj x X ≡ x ∈ set_of_adj X
```

```
lemma in_minterval_adj_to_minterval_adj[intro!]: a ∈adj minterval_adj_of a
  by (metis (mono_tags, lifting) UN_iff list.set_intros(1) lower_in_interval lower_point_interval
  minterval_adj_of.rep_eq set_of_eq set_of_adj.rep_eq)
```

```
instance minterval_adj :: ({one, preorder, minus_mono, linordered_semiring}) power
proof qed
```

```
lemma set_of_one_adj[simp]: set_of_adj (1::'a::{one, minus_mono, order} minterval_adj) = {1}
  apply (transfer)
  by (auto simp: mk_mInterval_adj_def mk_mInterval_ovl_def interval_sort_lower_width_def set_of_adj_def)
```

```
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,infimum,uminus}) interval  $\Rightarrow$  ('a interval) list
  where inverse_interval a = (
    if ( $\neg o \in_i a$ ) then [mk_interval (1 / (upper a), 1 / (lower a))]
    else if lower a = o  $\wedge$  o < upper a then [mk_interval (1 / upper a,  $\infty$ )]
    else if lower a < o  $\wedge$  o < upper a then [mk_interval ( $-\infty$ , 1 / lower a), mk_interval (1 / upper a,  $\infty$ )]
    else if lower a < upper a  $\wedge$  upper a = o then [mk_interval( $-\infty$ , 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 <mininterval_ovl_inverse x = mInterval_ovl (mk_mInterval_ovl(minverse (mList_ovl x)))>
definition <mininterval_ovl_divide x y = (mininterval_ovl_inverse y) * x>

```

```

lemma set_of_ovl_non_zero_map_inverse:
  assumes <o  $\notin$  set_of_ovl xs>
  shows <concat (map inverse_interval (mList_ovl xs)) = map ( $\lambda i.$  mk_interval (1 / upper i, 1 / lower i)) (mList_ovl xs)>
proof(insert assms, induction mList_ovl xs)
case Nil
then show ?case
  by simp

```

```

next
case (Cons a x)
then show ?case
  using set_of_ovl_non_zero_list_all[of xs, simplified Cons, simplified]
  by (metis (no_types, lifting) concat_map_singleton inverse_interval_def map_eq_conv)
qed

```

```

interpretation minterval_ovl_division_inverse minterval_ovl_divide minterval_ovl_inverse
apply(unfold_locales)
subgoal
  using set_of_ovl_non_zero_map_inverse
  unfolding minterval_ovl_inverse_def minverse_def o_def un_op_interval_list_def
  by fastforce
subgoal by(metis minterval_ovl_divide_def)
done

```

end

13.2.2 Non Overlapping Multi-Intervals (\exists Extended_Multi_Interval_Division_Non_Overlapping)

theory

Extended_Multi_Interval_Division_Non_Overlapping

imports

Extended_Multi_Interval_Division_Core

begin

definition \langle minterval_non_ovl_inverse $x =$ mInterval_non_ovl (mk_mInterval_non_ovl(minverse (mlist_non_ovl x))) \rangle

definition \langle minterval_non_ovl_divide $x y =$ (minterval_non_ovl_inverse y) * x \rangle

lemma set_of_non_ovl_non_zero_map_inverse:

assumes $\langle 0 \notin$ set_of_non_ovl xs \rangle

shows \langle concat (map inverse_interval (mlist_non_ovl xs)) = map ($\lambda i.$ mk_interval (1 / upper i, 1 / lower i)) (mlist_non_ovl xs) \rangle

proof(insert assms, induction mlist_non_ovl xs)

case Nil

then show ?case

by simp

next

case (Cons a x)

then show ?case

using set_of_non_ovl_non_zero_list_all[of xs, simplified Cons, simplified]

by (metis (no_types, lifting) concat_map_singleton inverse_interval_def map_eq_conv)

qed

interpretation minterval_non_ovl_division_inverse minterval_non_ovl_divide minterval_non_ovl_inverse

apply(unfold_locales)

subgoal

using set_of_non_ovl_non_zero_map_inverse

unfolding minterval_non_ovl_inverse_def minverse_def o_def un_op_interval_list_def

by fastforce

subgoal by(metis minterval_non_ovl_divide_def)

done

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 $\langle \text{minterval_adj_inverse } x = \text{mInterval_adj } (\text{mk_mInterval_adj } (\text{minverse } (\text{mlist_adj } x))) \rangle$

definition $\langle \text{minterval_adj_divide } x y = (\text{minterval_adj_inverse } y) * x \rangle$

lemma *set_of_adj_non_zero_map_inverse*:

assumes $\langle 0 \notin \text{set_of_adj } xs \rangle$

shows $\langle \text{concat } (\text{map } \text{inverse_interval } (\text{mlist_adj } xs)) = \text{map } (\lambda i. \text{mk_interval } (1 / \text{upper } i, 1 / \text{lower } i)) (\text{mlist_adj } xs) \rangle$

proof(*insert assms, induction mlist_adj xs*)

case *Nil*

then show *?case*

by *simp*

next

case (*Cons a x*)

then show *?case*

using *set_of_adj_non_zero_list_all*[*of xs, simplified Cons, simplified*]

by (*metis* (*no_types, lifting*) *concat_map_singleton inverse_interval_def map_eq_conv*)

qed

interpretation *minterval_adj_division_inverse minterval_adj_divide minterval_adj_inverse*

apply(*unfold_locales*)

subgoal

using *set_of_adj_non_zero_map_inverse*

unfolding *minterval_adj_inverse_def minverse_def o_def un_op_interval_list_def*

by *fastforce*

subgoal by(*metis minterval_adj_divide_def*)

done

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.