

Light-Weight Containers

Andreas Lochbihler

February 6, 2026

Abstract

This development provides a framework for container types like sets and maps such that generated code implements these containers with different (efficient) data structures. Thanks to type classes and refinement during code generation, this light-weight approach can seamlessly replace Isabelle's default setup for code generation. Heuristics automatically pick one of the available data structures depending on the type of elements to be stored, but users can also choose on their own. The extensible design permits to add more implementations at any time.

To support arbitrary nesting of sets, we define a linear order on sets based on a linear order of the elements and provide efficient implementations. It even allows to compare complements with non-complements.

Contents

1	Introduction	7
2	An executable linear order on sets	9
2.1	Auxiliary definitions	9
2.2	Definitions to prove equations about the cardinality of data types	12
2.2.1	Specialised <i>range</i> constants	12
2.2.2	Cardinality primitives for polymorphic HOL types	15
2.3	Shortcut fusion for lists	16
2.3.1	The type of generators for finite lists	17
2.3.2	Generators for ' <i>a list</i> '	20
2.3.3	Destroying lists	28
2.4	List fusion for lexicographic order	31
2.4.1	Setup for list fusion	32
2.5	Every partial order can be extended to a total order	33
2.6	An executable linear order on sets	41
2.6.1	Definition of the linear order	41
2.6.2	Implementation based on sorted lists	54
2.6.3	Implementation of proper intervals for sets	62
2.6.4	Proper intervals for HOL types	86
2.6.5	List fusion for the order and proper intervals on ' <i>a set</i> '	90
2.6.6	Drop notation	94
2.6.7	Introduction	95
3	Light-weight containers	97
3.1	A linear order for code generation	97
3.1.1	Optional comparators	97
3.1.2	Generator for the <i>ccompare</i> -class	98
3.1.3	Instantiations for HOL types	99
3.1.4	Proper intervals	100
3.2	Instantiate <i>proper-interval</i> of for ' <i>a list</i> '	105
3.3	A type class for optional equality testing	107
3.3.1	Generator for the <i>ceq</i> -class	108

3.3.2	Type class instances for HOL types	108
3.4	A type class for optional enumerations	111
3.4.1	Definition	111
3.4.2	Generator for the <i>cenum</i> -class	112
3.4.3	Instantiations	112
3.5	Locales to abstract over HOL equality	117
3.6	More on red-black trees	118
3.6.1	More lemmas	118
3.6.2	Build the cross product of two RBTs	118
3.6.3	Build an RBT where keys are paired with themselves	121
3.6.4	Folding and quantifiers over RBTs	122
3.6.5	List fusion for RBTs	123
3.7	Mappings implemented by red-black trees	126
3.7.1	Type definition	126
3.7.2	Operations	127
3.7.3	Properties	129
3.8	Additional operations for associative lists	133
3.8.1	Operations on the raw type	133
3.8.2	Operations on the abstract type $(\text{'a}, \text{'b})$ <i>alist</i>	135
3.9	Sets implemented by distinct lists	137
3.9.1	Operations on the raw type with parametrised equality	137
3.9.2	The type of distinct lists	139
3.9.3	Operations	140
3.9.4	Properties	141
3.10	Sets implemented by red-black trees	146
3.10.1	Type and operations	148
3.10.2	Primitive operations	148
3.10.3	Properties	149
3.11	Sets implemented as Closures	153
3.12	Different implementations of sets	154
3.12.1	Auxiliary functions	154
3.12.2	Delete code equation with set as constructor	159
3.12.3	Set implementations	160
3.12.4	Set operations	160
3.12.5	Type class instantiations	197
3.12.6	Generator for the <i>set-impl</i> -class	199
3.12.7	Pretty printing for sets	200
3.13	Different implementations of maps	202
3.13.1	Map implementations	202
3.13.2	Map operations	202
3.13.3	Type classes	206
3.13.4	Generator for the <i>mapping-impl</i> -class	207
3.14	Infrastructure for operation identification	209
3.15	Compatibility with Regular-Sets	213

4	User guide	215
4.1	Characteristics	215
4.2	Getting started	216
4.3	New types as elements	217
4.3.1	Equality testing	217
4.3.2	Ordering	219
4.3.3	Heuristics for picking an implementation	220
4.3.4	Set comprehensions	221
4.3.5	Nested sets	222
4.4	New implementations for containers	224
4.4.1	Model and verify the data structure	225
4.4.2	Generalise the data structure	225
4.4.3	Hide the invariants of the data structure	226
4.4.4	Connecting to the container	227
4.5	Changing the configuration	231
4.6	New containers types	231
4.7	Troubleshooting	231
4.7.1	Nesting of mappings	232
4.7.2	Wellsortedness errors	232
4.7.3	Exception raised at run-time	232
4.7.4	LC slows down my code	233

Chapter 1

Introduction

This development focuses on generating efficient code for container types like sets and maps. It falls into two parts: First, we define linear order on sets (Ch. 2) that is efficiently executable given a linear order on the elements. Second, we define an extensible framework LC (for light-weight containers) that supports multiple (efficient) implementations of container types (Ch. 3) in generated code. Both parts heavily exploit type classes and the refinement features of the code generator [2]. This way, we are able to implement the HOL types for sets and maps directly, as the name light-weight containers (LC) emphasises.

In comparison with the Isabelle Collections Framework (ICF) [4, 3], the style of refinement is the major difference. In the ICF, the container types are replaced with the types of the data structures inside the logic. Typically, the user has to define his operations that involve maps and sets a second time such that they work on the concrete data structures; then, she has to prove that both definitions agree. With LC, the refinement happens inside the code generator. Hence, the formalisation can stick with the types *'a set* and *('a,'b) mapping* and there is no need to duplicate definitions or prove refinement. The drawback is that with LC, we can only implement operations that can be fully specified on the abstract container type. In particular, the internal representation of the implementations may not affect the result of the operations. For example, it is not possible to pick non-deterministically an element from a set or fold a set with a non-commutative operation, i.e., the result depends on the order of visiting the elements.

For more documentation and introductory material refer to the userguide (Chapter 4) and the ITP-2013 paper [5].

```
theory Containers-Auxiliary imports  
  HOL-Library.Monad-Syntax  
begin
```


Chapter 2

An executable linear order on sets

2.1 Auxiliary definitions

lemma *insert-bind-set*: $\text{insert } a \ A \ggg f = f \ a \cup (A \ggg f)$
by(*auto simp add: Set.bind-def*)

lemma *set-bind-iff*:
 $\text{set } (\text{List.bind } xs \ f) = \text{Set.bind } (\text{set } xs) (\text{set } \circ f)$
by(*induct xs*)(*simp-all add: insert-bind-set*)

lemma *set-bind-conv-fold*: $\text{set } xs \ggg f = \text{fold } ((\cup) \circ f) \ xs \ \{\}$
by(*induct xs rule: rev-induct*)(*simp-all add: insert-bind-set*)

lemma *card-gt-1D*:
assumes $\text{card } A > 1$
shows $\exists x \ y. x \in A \wedge y \in A \wedge x \neq y$
proof(*rule ccontr*)
from *assms* **have** $A \neq \{\}$ **by** *auto*
then obtain x **where** $x \in A$ **by** *auto*
moreover
assume $\neg ?thesis$
ultimately have $A = \{x\}$ **by** *auto*
with *assms* **show** *False* **by** *simp*
qed

lemma *card-eq-1-iff*: $\text{card } A = 1 \longleftrightarrow (\exists x. A = \{x\})$
proof

assume *card*: $\text{card } A = 1$
hence [*simp*]: *finite* A **using** *card-gt-0-iff*[*of* A] **by** *simp*
have $A = \{\text{THE } x. x \in A\}$
proof(*intro equalityI subsetI*)
fix x
assume $x: x \in A$

```

hence (THE x. x ∈ A) = x
proof(rule the-equality)
  fix x'
  assume x': x' ∈ A
  show x' = x
  proof(rule ccontr)
    assume neg: x' ≠ x
    from x x' have eq: A = insert x (insert x' (A - {x, x'})) by auto
    have card A = 2 + card (A - {x, x'}) using neg by (subst eq)(simp)
    with card show False by simp
  qed
qed
thus x ∈ {THE x. x ∈ A} by simp
next
fix x
assume x ∈ {THE x. x ∈ A}
hence x: x = (THE x. x ∈ A) by simp
from card have A ≠ {} by auto
then obtain x' where x': x' ∈ A by blast
thus x ∈ A unfolding x
proof(rule theI)
  fix x
  assume x: x ∈ A
  show x = x'
  proof(rule ccontr)
    assume neg: x ≠ x'
    from x x' have eq: A = insert x (insert x' (A - {x, x'})) by auto
    have card A = 2 + card (A - {x, x'}) using neg by (subst eq)(simp)
    with card show False by simp
  qed
qed
qed
thus ∃ x. A = {x} ..
qed auto

```

lemma *card-eq-Suc-0-ex1*: $\text{card } A = \text{Suc } 0 \longleftrightarrow (\exists! x. x \in A)$
by(auto simp only: One-nat-def[symmetric] card-eq-1-iff)

context *linorder* **begin**

lemma *sorted-last*: $\llbracket \text{sorted } xs; x \in \text{set } xs \rrbracket \implies x \leq \text{last } xs$
by(cases xs rule: rev-cases)(auto simp add: sorted-append)

end

lemma *empty-filter-conv*: $\llbracket \rrbracket = \text{filter } P \text{ } xs \longleftrightarrow (\forall x \in \text{set } xs. \neg P \ x)$
by(auto dest: sym simp add: filter-empty-conv)

definition $ID :: 'a \Rightarrow 'a$ **where** $ID = id$

lemma $ID\text{-code}$ [$code$, $code\text{-unfold}$]: $ID = (\lambda x. x)$
by($simp$ add : $ID\text{-def}$ $id\text{-def}$)

lemma $ID\text{-Some}$: $ID (Some\ x) = Some\ x$
by($simp$ add : $ID\text{-def}$)

lemma $ID\text{-None}$: $ID\ None = None$
by($simp$ add : $ID\text{-def}$)

lexicographic order on pairs

context

fixes $leq\text{-}a :: 'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\langle \sqsubseteq_a \rangle$ 50)
and $less\text{-}a :: 'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\langle \sqsubset_a \rangle$ 50)
and $leq\text{-}b :: 'b \Rightarrow 'b \Rightarrow bool$ (**infix** $\langle \sqsubseteq_b \rangle$ 50)
and $less\text{-}b :: 'b \Rightarrow 'b \Rightarrow bool$ (**infix** $\langle \sqsubset_b \rangle$ 50)

begin

definition $less\text{-}eq\text{-}prod :: ('a \times 'b) \Rightarrow ('a \times 'b) \Rightarrow bool$ (**infix** $\langle \sqsubseteq \rangle$ 50)
where $less\text{-}eq\text{-}prod = (\lambda(x1, x2) (y1, y2). x1 \sqsubseteq_a y1 \vee x1 \sqsubset_a y1 \wedge x2 \sqsubseteq_b y2)$

definition $less\text{-}prod :: ('a \times 'b) \Rightarrow ('a \times 'b) \Rightarrow bool$ (**infix** $\langle \sqsubset \rangle$ 50)
where $less\text{-}prod = (\lambda(x1, x2) (y1, y2). x1 \sqsubset_a y1 \vee x1 \sqsubseteq_a y1 \wedge x2 \sqsubset_b y2)$

lemma $less\text{-}eq\text{-}prod\text{-}simps$ [$simp$]:
 $(x1, x2) \sqsubseteq (y1, y2) \longleftrightarrow x1 \sqsubseteq_a y1 \vee x1 \sqsubset_a y1 \wedge x2 \sqsubseteq_b y2$
by($simp$ add : $less\text{-}eq\text{-}prod\text{-}def$)

lemma $less\text{-}prod\text{-}simps$ [$simp$]:
 $(x1, x2) \sqsubset (y1, y2) \longleftrightarrow x1 \sqsubset_a y1 \vee x1 \sqsubseteq_a y1 \wedge x2 \sqsubset_b y2$
by($simp$ add : $less\text{-}prod\text{-}def$)

end

context

fixes $leq\text{-}a :: 'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\langle \sqsubseteq_a \rangle$ 50)
and $less\text{-}a :: 'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\langle \sqsubset_a \rangle$ 50)
and $leq\text{-}b :: 'b \Rightarrow 'b \Rightarrow bool$ (**infix** $\langle \sqsubseteq_b \rangle$ 50)
and $less\text{-}b :: 'b \Rightarrow 'b \Rightarrow bool$ (**infix** $\langle \sqsubset_b \rangle$ 50)
assumes $lin\text{-}a$: $class.linorder\ leq\text{-}a\ less\text{-}a$
and $lin\text{-}b$: $class.linorder\ leq\text{-}b\ less\text{-}b$

begin

abbreviation ($input$) $less\text{-}eq\text{-}prod' :: ('a \times 'b) \Rightarrow ('a \times 'b) \Rightarrow bool$ (**infix** $\langle \sqsubseteq \rangle$ 50)
where $less\text{-}eq\text{-}prod' \equiv less\text{-}eq\text{-}prod\ leq\text{-}a\ less\text{-}a\ leq\text{-}b$

abbreviation ($input$) $less\text{-}prod' :: ('a \times 'b) \Rightarrow ('a \times 'b) \Rightarrow bool$ (**infix** $\langle \sqsubset \rangle$ 50)
where $less\text{-}prod' \equiv less\text{-}prod\ leq\text{-}a\ less\text{-}a\ less\text{-}b$

```

lemma linorder-prod:
  class.linorder ( $\sqsubseteq$ ) ( $\sqsubset$ )
proof –
  interpret a: linorder ( $\sqsubseteq_a$ ) ( $\sqsubset_a$ ) by(fact lin-a)
  interpret b: linorder ( $\sqsubseteq_b$ ) ( $\sqsubset_b$ ) by(fact lin-b)
  show ?thesis by unfold-locales auto
qed

end

hide-const less-eq-prod' less-prod'

```

```

end

```

```

theory Card-Datatype
imports HOL-Library.Cardinality
begin

```

2.2 Definitions to prove equations about the cardinality of data types

2.2.1 Specialised *range* constants

```

definition rangeIt :: 'a  $\Rightarrow$  ('a  $\Rightarrow$  'a)  $\Rightarrow$  'a set
where rangeIt x f = range ( $\lambda n. (f \hat{\sim} n) x$ )

```

```

definition rangeC :: ('a  $\Rightarrow$  'b) set  $\Rightarrow$  'b set
where rangeC F = ( $\bigcup f \in F. \text{range } f$ )

```

```

lemma infinite-rangeIt:
  assumes inj: inj f
  and x:  $\forall y. x \neq f y$ 
  shows  $\neg \text{finite } (\text{rangeIt } x f)$ 
proof –
  have inj ( $\lambda n. (f \hat{\sim} n) x$ )
  proof(rule injI)
    fix n m
    assume  $(f \hat{\sim} n) x = (f \hat{\sim} m) x$ 
    thus  $n = m$ 
  proof(induct n arbitrary: m)
    case 0
    thus ?case using x by(cases m)(auto intro: sym)
  next
    case (Suc n)
    thus ?case using x by(cases m)(auto intro: sym dest: injD[OF inj])
  qed
qed

```

2.2. DEFINITIONS TO PROVE EQUATIONS ABOUT THE CARDINALITY OF DATA TYPES 13

```

thus ?thesis
  by(auto simp add: rangeIt-def dest: finite-imageD)
qed

lemma in-rangeC:  $f \in A \implies f x \in \text{rangeC } A$ 
by(auto simp add: rangeC-def)

lemma in-rangeCE: assumes  $y \in \text{rangeC } A$ 
  obtains  $f x$  where  $f \in A \quad y = f x$ 
using assms by(auto simp add: rangeC-def)

lemma in-rangeC-singleton:  $f x \in \text{rangeC } \{f\}$ 
by(auto simp add: rangeC-def)

lemma in-rangeC-singleton-const:  $x \in \text{rangeC } \{\lambda-. x\}$ 
by(rule in-rangeC-singleton)

lemma rangeC-rangeC:  $f \in \text{rangeC } A \implies f x \in \text{rangeC } (\text{rangeC } A)$ 
by(auto simp add: rangeC-def)

lemma rangeC-eq-empty:  $\text{rangeC } A = \{\} \longleftrightarrow A = \{\}$ 
by(auto simp add: rangeC-def)

lemma Ball-rangeC-iff:
   $(\forall x \in \text{rangeC } A. P x) \longleftrightarrow (\forall f \in A. \forall x. P (f x))$ 
by(auto intro: in-rangeC elim: in-rangeCE)

lemma Ball-rangeC-singleton:
   $(\forall x \in \text{rangeC } \{f\}. P x) \longleftrightarrow (\forall x. P (f x))$ 
by(simp add: Ball-rangeC-iff)

lemma Ball-rangeC-rangeC:
   $(\forall x \in \text{rangeC } (\text{rangeC } A). P x) \longleftrightarrow (\forall f \in \text{rangeC } A. \forall x. P (f x))$ 
by(simp add: Ball-rangeC-iff)

lemma finite-rangeC:
  assumes inj:  $\forall f \in A. \text{inj } f$ 
  and disjoint:  $\forall f \in A. \forall g \in A. f \neq g \longrightarrow (\forall x y. f x \neq g y)$ 
  shows  $\text{finite } (\text{rangeC } (A :: ('a \Rightarrow 'b) \text{ set})) \longleftrightarrow \text{finite } A \wedge (A \neq \{\}) \longrightarrow \text{finite}$ 
  (UNIV :: 'a set)
  (is ?lhs  $\longleftrightarrow$  ?rhs)
proof
  assume ?lhs
  thus ?rhs using inj disjoint
proof(induct rangeC A arbitrary: A rule: finite-psubset-induct)
  case (psubset A)
  show ?case
  proof(cases A = \{\})
  case True thus ?thesis by simp

```

```

next
case False
then obtain f A' where A: A = insert f A' and f: f ∈ A   f ∉ A'
  by(fastforce dest: mk-disjoint-insert)
from A have rA: rangeC A = rangeC A' ∪ range f
  by(auto simp add: rangeC-def)

have ¬ range f ⊆ rangeC A'
proof
  assume range f ⊆ rangeC A'
  moreover obtain x where x: x ∈ range f by auto
  ultimately have x ∈ rangeC A' by auto
  then obtain g where g ∈ A'   x ∈ range g by(auto simp add: rangeC-def)
  with ⟨f ∉ A'⟩ A have g ∈ A   f ≠ g by auto
  with ⟨f ∈ A⟩ have ∧x y. f x ≠ g y by(rule psubset.prem[srule-format])
  thus False using x ⟨x ∈ range g⟩ by auto
qed
hence rangeC A' ⊂ rangeC A unfolding rA by auto
hence finite A' ∧ (A' ≠ {}) → finite (UNIV :: 'a set)
  using psubset.prem
  by -(erule psubset.hyps, auto simp add: A)
with A have finite A by simp
moreover with ⟨finite (rangeC A)⟩ A ⟨∀f ∈ A. inj f⟩
have finite (UNIV :: 'a set)
  by(auto simp add: rangeC-def dest: finite-imageD)
ultimately show ?thesis by blast
qed
qed
qed(auto simp add: rangeC-def)

lemma finite-rangeC-singleton-const:
  finite (rangeC {λ-. x})
by(auto simp add: rangeC-def image-def)

lemma card-Un:
  [| finite A; finite B |] ⇒ card (A ∪ B) = card (A) + card (B) - card(A ∩ B)
by(subst card-Un-Int) simp-all

lemma card-rangeC-singleton-const:
  card (rangeC {λ-. f}) = 1
by(simp add: rangeC-def image-def)

lemma card-rangeC:
  assumes inj: ∀f ∈ A. inj f
  and disjoint: ∀f ∈ A. ∀g ∈ A. f ≠ g → (∀x y. f x ≠ g y)
  shows card (rangeC (A :: ('a ⇒ 'b) set)) = CARD('a) * card A
  (is ?lhs = ?rhs)
proof(cases finite (UNIV :: 'a set) ∧ finite A)
  case False

```

```

thus ?thesis using False finite-rangeC[OF assms]
  by(auto simp add: card-eq-0-iff rangeC-eq-empty)
next
  case True
  { fix f
    assume f ∈ A
    hence card (range f) = CARD('a) using inj by(simp add: card-image) }
  thus ?thesis using disjoint True unfolding rangeC-def
    by(subst card-UN-disjoint) auto
qed

```

```

lemma rangeC-Int-rangeC:
   $\llbracket \forall f \in A. \forall g \in B. \forall x y. f x \neq g y \rrbracket \implies \text{rangeC } A \cap \text{rangeC } B = \{\}$ 
by(auto simp add: rangeC-def)

```

```

lemmas rangeC-simps =
  in-rangeC-singleton
  in-rangeC-singleton-const
  rangeC-rangeC
  rangeC-eq-empty
  Ball-rangeC-singleton
  Ball-rangeC-rangeC
  finite-rangeC
  finite-rangeC-singleton-const
  card-rangeC-singleton-const
  card-rangeC
  rangeC-Int-rangeC

```

```

bundle card-datatype =
  rangeC-simps [simp]
  card-Un [simp]
  fun-eq-iff [simp]
  Int-Un-distrib [simp]
  Int-Un-distrib2 [simp]
  card-eq-0-iff [simp]
  imageI [simp] image-eqI [simp del]
  conj-cong [cong]
  infinite-rangeIt [simp]

```

2.2.2 Cardinality primitives for polymorphic HOL types

```

ML <
  structure Card-Simp-Rules = Named-Thms
  (
    val name = @{binding card-simps}
    val description = Simplification rules for cardinality of types
  )
  >
setup <Card-Simp-Rules.setup>

```

definition *card-fun* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$
where *card-fun* *a b* = (if $a \neq 0 \wedge b \neq 0 \vee b = 1$ then $b \hat{^} a$ else 0)

lemma *CARD-fun* [*card-simps*]:
 $\text{CARD}('a \Rightarrow 'b) = \text{card-fun } \text{CARD}('a) \text{ CARD}('b)$
by(*simp add: card-fun card-fun-def*)

definition *card-sum* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$
where *card-sum* *a b* = (if $a = 0 \vee b = 0$ then 0 else $a + b$)

lemma *CARD-sum* [*card-simps*]:
 $\text{CARD}('a + 'b) = \text{card-sum } \text{CARD}('a) \text{ CARD}('b)$
by(*simp add: card-UNIV-sum card-sum-def*)

definition *card-option* :: $\text{nat} \Rightarrow \text{nat}$
where *card-option* *n* = (if $n = 0$ then 0 else *Suc* *n*)

lemma *CARD-option* [*card-simps*]:
 $\text{CARD}('a \text{ option}) = \text{card-option } \text{CARD}('a)$
by(*simp add: card-option-def card-UNIV-option*)

definition *card-prod* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$
where *card-prod* *a b* = $a * b$

lemma *CARD-prod* [*card-simps*]:
 $\text{CARD}('a * 'b) = \text{card-prod } \text{CARD}('a) \text{ CARD}('b)$
by(*simp add: card-prod-def*)

definition *card-list* :: $\text{nat} \Rightarrow \text{nat}$
where *card-list* - = 0

lemma *CARD-list* [*card-simps*]: $\text{CARD}('a \text{ list}) = \text{card-list } \text{CARD}('a)$
by(*simp add: card-list-def infinite-UNIV-listI*)

end

theory *List-Fusion*

imports

Main

begin

2.3 Shortcut fusion for lists

lemma *Option-map-mono* [*partial-function-mono*]:
 $\text{mono-option } f \Longrightarrow \text{mono-option } (\lambda x. \text{map-option } g (f x))$
apply(*rule monotoneI*)
apply(*drule (1) monotoneD*)

apply(*auto simp add: map-option-case flat-ord-def split: option.split*)
done

lemma *list-all2-coinduct* [*consumes 1, case-names Nil Cons, case-conclusion Cons*
hd tl, coinduct pred: list-all2]:

assumes *X: X xs ys*
and *Nil': $\bigwedge xs ys. X xs ys \implies xs = [] \longleftrightarrow ys = []$*
and *Cons': $\bigwedge xs ys. \llbracket X xs ys; xs \neq []; ys \neq [] \rrbracket \implies A (hd xs) (hd ys) \wedge (X (tl xs) (tl ys) \vee list-all2 A (tl xs) (tl ys))$*
shows *list-all2 A xs ys*
using *X*
proof(*induction xs arbitrary: ys*)
case *Nil*
from *Nil'[OF this]* **show** *?case by simp*
next
case (*Cons x xs*)
from *Nil'[OF Cons.premis] Cons'[OF Cons.premis] Cons.IH*
show *?case by(auto simp add: neq-Nil-conv)*
qed

2.3.1 The type of generators for finite lists

type-synonym (*'a, 's*) *raw-generator* = (*'s \Rightarrow bool*) \times (*'s \Rightarrow 'a \times 's*)

inductive-set *terminates-on* :: (*'a, 's*) *raw-generator* \Rightarrow *'s set*

for *g* :: (*'a, 's*) *raw-generator*

where

stop: $\neg fst g s \implies s \in terminates-on g$
| unfold: $\llbracket fst g s; snd (snd g s) \in terminates-on g \rrbracket \implies s \in terminates-on g$

definition *terminates* :: (*'a, 's*) *raw-generator* \Rightarrow *bool*

where *terminates g \longleftrightarrow (terminates-on g = UNIV)*

lemma *terminatesI* [*intro?*]:

($\bigwedge s. s \in terminates-on g \implies terminates g$)
by(*auto simp add: terminates-def*)

lemma *terminatesD*:

terminates g $\implies s \in terminates-on g$
by(*auto simp add: terminates-def*)

lemma *terminates-on-stop*:

terminates-on ($\lambda-. False, next$) = *UNIV*
by(*auto intro: terminates-on.stop*)

lemma *wf-terminates*:

assumes *wf R*
and *step: $\bigwedge s. fst g s \implies (snd (snd g s), s) \in R$*
shows *terminates g*

```

proof(rule terminatesI)
  fix s
  from  $\langle wf\ R \rangle$  show  $s \in terminates\text{-}on\ g$ 
  proof(induction rule: wf-induct[rule-format, consumes 1, case-names wf])
    case (wf s)
    show ?case
    proof(cases fst g s)
      case True
      hence  $(snd\ (snd\ g\ s),\ s) \in R$  by(rule step)
      hence  $snd\ (snd\ g\ s) \in terminates\text{-}on\ g$  by(rule wf.IH)
      with True show ?thesis by(rule terminates-on.unfold)
    qed(rule terminates-on.stop)
  qed
qed

lemma terminates-wfD:
  assumes terminates g
  shows  $wf\ \{(snd\ (snd\ g\ s),\ s) \mid s.\ fst\ g\ s\}$ 
proof(rule wfUNIVI)
  fix thesis s
  assume  $wf\ [rule\text{-}format]: \forall s. (\forall s'. (s', s) \in \{(snd\ (snd\ g\ s),\ s) \mid s.\ fst\ g\ s\} \longrightarrow$ 
thesis s')  $\longrightarrow thesis\ s$ 
  from assms have  $s \in terminates\text{-}on\ g$  by(auto simp add: terminates-def)
  thus thesis s by induct (auto intro: wf)
qed

lemma terminates-wfE:
  assumes terminates g
  obtains R where  $wf\ R \ \bigwedge s.\ fst\ g\ s \implies (snd\ (snd\ g\ s),\ s) \in R$ 
by(rule that)(rule terminates-wfD[OF assms], simp)

context fixes g :: ('a, 's) raw-generator begin

partial-function (option) terminates-within :: 's  $\Rightarrow$  nat option
where
  terminates-within s =
  (let (has-next, next) = g
  in if has-next s then
    map-option ( $\lambda n. n + 1$ ) (terminates-within ( $snd\ (next\ s)$ ))
  else Some 0)

lemma terminates-on-conv-dom-terminates-within:
  terminates-on g = dom terminates-within
proof(safe)
  fix s
  assume  $s \in terminates\text{-}on\ g$ 
  then show  $\exists n. terminates\text{-}within\ s = Some\ n$ 
  by(induct)(subst terminates-within.simps, simp add: split-beta)+
next

```

```

fix  $s\ n$ 
assume  $\text{terminates-within } s = \text{Some } n$ 
then show  $s \in \text{terminates-on } g$ 
  by( $\text{induct rule: terminates-within.raw-induct[rotated 1, consumes 1]}$ )( $\text{auto simp}$ 
 $\text{add: split-beta split: if-split-asm intro: terminates-on.intros}$ )
qed

end

```

```

lemma  $\text{terminates-within-unfold}$ :
   $\text{has-next } s \implies$ 
   $\text{terminates-within } (\text{has-next}, \text{next})\ s = \text{map-option } (\lambda n. n + 1)\ (\text{terminates-within}$ 
   $(\text{has-next}, \text{next})\ (\text{snd } (\text{next } s)))$ 
by( $\text{simp add: terminates-within.simps}$ )

```

```

typedef  $(\ 'a, 's)\ \text{generator} = \{g :: (\ 'a, 's)\ \text{raw-generator. terminates } g\}$ 
morphisms  $\text{generator Generator}$ 
proof
  show  $(\lambda-. \text{False}, \text{undefined}) \in ?\text{generator}$ 
  by( $\text{simp add: terminates-on-stop terminates-def}$ )
qed

```

```

setup-lifting  $\text{type-definition-generator}$ 

```

```

lemma  $\text{terminates-on-generator-eq-UNIV}$ :
   $\text{terminates-on } (\text{generator } g) = \text{UNIV}$ 
by  $\text{transfer(simp add: terminates-def)}$ 

```

```

lemma  $\text{terminates-within-stop}$ :
   $\text{terminates-within } (\lambda-. \text{False}, \text{next})\ s = \text{Some } 0$ 
by( $\text{simp add: terminates-within.simps}$ )

```

```

lemma  $\text{terminates-within-generator-neq-None}$ :
   $\text{terminates-within } (\text{generator } g)\ s \neq \text{None}$ 
by( $\text{transfer}(auto\ \text{simp add: terminates-def terminates-on-conv-dom-terminates-within})$ )

```

```

locale  $\text{list} =$ 
  fixes  $g :: (\ 'a, 's)\ \text{generator}$  begin

```

```

definition  $\text{has-next} :: '\ s \Rightarrow \text{bool}$ 
where  $\text{has-next} = \text{fst } (\text{generator } g)$ 

```

```

definition  $\text{next} :: '\ s \Rightarrow '\ a \times '\ s$ 
where  $\text{next} = \text{snd } (\text{generator } g)$ 

```

```

function  $\text{unfoldr} :: '\ s \Rightarrow '\ a\ \text{list}$ 
where  $\text{unfoldr } s = (\text{if } \text{has-next } s \text{ then let } (a, s') = \text{next } s \text{ in } a \ \# \ \text{unfoldr } s' \text{ else } [])$ 
by  $\text{pat-completeness auto}$ 
termination

```

```

proof –
  have terminates (generator g) using generator[of g] by simp
  thus ?thesis
  by(rule terminates-wfE)(erule termination, metis has-next-def next-def snd-conv)
qed

```

```

declare unfoldr.simps [simp del]

```

```

lemma unfoldr-simps:
  has-next s  $\implies$  unfoldr s = fst (next s) # unfoldr (snd (next s))
   $\neg$  has-next s  $\implies$  unfoldr s = []
by(simp-all add: unfoldr.simps split-beta)

```

```

end

```

```

declare
  list.has-next-def[code]
  list.next-def[code]
  list.unfoldr.simps[code]

```

```

context includes lifting-syntax
begin

```

```

lemma generator-has-next-transfer [transfer-rule]:
  (pcr-generator (=) (=)  $\implies$  (=)) fst list.has-next
by(auto simp add: generator.pcr-cr-eq cr-generator-def list.has-next-def dest: sym)

```

```

lemma generator-next-transfer [transfer-rule]:
  (pcr-generator (=) (=)  $\implies$  (=)) snd list.next
by(auto simp add: generator.pcr-cr-eq cr-generator-def list.next-def)

```

```

end

```

```

lemma unfoldr-eq-Nil-iff [iff]:
  list.unfoldr g s = []  $\longleftrightarrow$   $\neg$  list.has-next g s
by(subst list.unfoldr.simps)(simp add: split-beta)

```

```

lemma Nil-eq-unfoldr-iff [simp]:
  [] = list.unfoldr g s  $\longleftrightarrow$   $\neg$  list.has-next g s
by(auto intro: sym dest: sym)

```

2.3.2 Generators for 'a list

```

primrec list-has-next :: 'a list  $\Rightarrow$  bool
where
  list-has-next []  $\longleftrightarrow$  False
| list-has-next (x # xs)  $\longleftrightarrow$  True

```

```

primrec list-next :: 'a list  $\Rightarrow$  'a  $\times$  'a list

```

where

$list\text{-}next\ (x \#\ xs) = (x, xs)$

lemma *terminates-list-generator*: *terminates* (*list-has-next*, *list-next*)

proof

fix *xs*

show $xs \in \textit{terminates-on}$ (*list-has-next*, *list-next*)

by(*induct xs*)(*auto intro: terminates-on.intros*)

qed

lift-definition *list-generator* :: ('a, 'a list) generator

is (*list-has-next*, *list-next*)

by(*rule terminates-list-generator*)

lemma *has-next-list-generator* [*simp*]:

$list.\textit{has-next}\ list\text{-}generator = list\text{-}has\text{-}next$

by *transfer simp*

lemma *next-list-generator* [*simp*]:

$list.\textit{next}\ list\text{-}generator = list\text{-}next$

by *transfer simp*

lemma *unfoldr-list-generator*:

$list.\textit{unfoldr}\ list\text{-}generator\ xs = xs$

by(*induct xs*)(*simp-all add: list.unfoldr-simps*)

lemma *terminates-replicate-generator*:

$\textit{terminates}\ (\lambda n :: \textit{nat}. 0 < n, \lambda n. (a, n - 1))$

by(*rule wf-terminates*)(*lexicographic-order*)

lift-definition *replicate-generator* :: 'a \Rightarrow ('a, nat) generator

is $\lambda a. (\lambda n. 0 < n, \lambda n. (a, n - 1))$

by(*rule terminates-replicate-generator*)

lemma *has-next-replicate-generator* [*simp*]:

$list.\textit{has-next}\ (\textit{replicate-generator}\ a)\ n \longleftrightarrow 0 < n$

by *transfer simp*

lemma *next-replicate-generator* [*simp*]:

$list.\textit{next}\ (\textit{replicate-generator}\ a)\ n = (a, n - 1)$

by *transfer simp*

lemma *unfoldr-replicate-generator*:

$list.\textit{unfoldr}\ (\textit{replicate-generator}\ a)\ n = \textit{replicate}\ n\ a$

by(*induct n*)(*simp-all add: list.unfoldr-simps*)

context *fixes f* :: 'a \Rightarrow 'b **begin**

lift-definition *map-generator* :: ('a, 's) generator \Rightarrow ('b, 's) generator

is $\lambda(\text{has-next}, \text{next}). (\text{has-next}, \lambda s. \text{let } (a, s') = \text{next } s \text{ in } (f a, s'))$
by $(\text{erule terminates-wfE})(\text{erule wf-terminates}, \text{auto simp add: split-beta})$

lemma *has-next-map-generator* [simp]:
 $\text{list.has-next } (\text{map-generator } g) = \text{list.has-next } g$
by *transfer clarsimp*

lemma *next-map-generator* [simp]:
 $\text{list.next } (\text{map-generator } g) = \text{apfst } f \circ \text{list.next } g$
by *transfer(simp add: fun-eq-iff split-beta apfst-def map-prod-def)*

lemma *unfoldr-map-generator*:
 $\text{list.unfoldr } (\text{map-generator } g) = \text{map } f \circ \text{list.unfoldr } g$
(is ?lhs = ?rhs)
proof *(rule ext)*
fix s
show $?lhs s = ?rhs s$
by *(induct s taking: map-generator g rule: list.unfoldr.induct)*
 $(\text{subst } (1\ 2) \text{ list.unfoldr.simps}, \text{auto simp add: split-beta apfst-def map-prod-def})$
qed

end

context *fixes* $g1 :: ('a, 's1) \text{ raw-generator}$
and $g2 :: ('a, 's2) \text{ raw-generator}$
begin

fun *append-has-next* $:: 's1 \times 's2 + 's2 \Rightarrow \text{bool}$
where
 $\text{append-has-next } (\text{Inl } (s1, s2)) \longleftrightarrow \text{fst } g1\ s1 \vee \text{fst } g2\ s2$
 $|\ \text{append-has-next } (\text{Inr } s2) \longleftrightarrow \text{fst } g2\ s2$

fun *append-next* $:: 's1 \times 's2 + 's2 \Rightarrow 'a \times ('s1 \times 's2 + 's2)$
where
 $\text{append-next } (\text{Inl } (s1, s2)) =$
 $(\text{if } \text{fst } g1\ s1 \text{ then}$
 $\quad \text{let } (x, s1') = \text{snd } g1\ s1 \text{ in } (x, \text{Inl } (s1', s2))$
 $\quad \text{else } \text{append-next } (\text{Inr } s2))$
 $|\ \text{append-next } (\text{Inr } s2) = (\text{let } (x, s2') = \text{snd } g2\ s2 \text{ in } (x, \text{Inr } s2'))$

end

lift-definition *append-generator* $:: ('a, 's1) \text{ generator} \Rightarrow ('a, 's2) \text{ generator} \Rightarrow$
 $('a, 's1 \times 's2 + 's2) \text{ generator}$
is $\lambda g1\ g2. (\text{append-has-next } g1\ g2, \text{append-next } g1\ g2)$
proof *(rule terminatesI, safe)*
fix *has-next1* **and** *next1* $:: 's1 \Rightarrow 'a \times 's1$
and *has-next2* **and** *next2* $:: 's2 \Rightarrow 'a \times 's2$
and s

```

assume  $t1$ : terminates ( $has\_next1$ ,  $next1$ )
and  $t2$ : terminates ( $has\_next2$ ,  $next2$ )
let  $?has\_next = append\_has\_next$  ( $has\_next1$ ,  $next1$ ) ( $has\_next2$ ,  $next2$ )
let  $?next = append\_next$  ( $has\_next1$ ,  $next1$ ) ( $has\_next2$ ,  $next2$ )
note [simp] = split-beta
and [intro] = terminates-on.intros
{ fix  $s2 :: 's2$ 
  from  $t2$  have  $s2 \in terminates\_on$  ( $has\_next2$ ,  $next2$ ) by(rule terminatesD)
  hence  $Inr\ s2 \in terminates\_on$  ( $?has\_next$ ,  $?next$ ) by induct auto }
note  $Inr' = this$ 

show  $s \in terminates\_on$  ( $?has\_next$ ,  $?next$ )
proof(cases s)
  case  $Inr$  thus  $?thesis$  by(simp add: Inr')
next
  case ( $Inl\ s1s2$ )
  moreover obtain  $s1\ s2$  where  $s1s2 = (s1, s2)$  by(cases s1s2)
  ultimately have  $s = Inl\ (s1, s2)$  by simp
  from  $t1$  have  $s1 \in terminates\_on$  ( $has\_next1$ ,  $next1$ ) by(rule terminatesD)
  thus  $?thesis$  unfolding  $s$ 
  proof induct
    case stop thus  $?case$ 
    by(cases has\_next2 s2)(auto simp add: Inr')
  qed auto
qed
qed

```

definition $append_init :: 's1 \Rightarrow 's2 \Rightarrow 's1 \times 's2 + 's2$
where $append_init\ s1\ s2 = Inl\ (s1, s2)$

lemma $has_next_append_generator$ [*simp*]:
 $list.has_next$ ($append_generator\ g1\ g2$) ($Inl\ (s1, s2)$) \longleftrightarrow
 $list.has_next\ g1\ s1 \vee list.has_next\ g2\ s2$
 $list.has_next$ ($append_generator\ g1\ g2$) ($Inr\ s2$) $\longleftrightarrow list.has_next\ g2\ s2$
by(*transfer, simp*) $+$

lemma $next_append_generator$ [*simp*]:
 $list.next$ ($append_generator\ g1\ g2$) ($Inl\ (s1, s2)$) =
(*if* $list.has_next\ g1\ s1$ *then*
 $let\ (x, s1') = list.next\ g1\ s1$ *in* ($x, Inl\ (s1', s2)$)
else $list.next$ ($append_generator\ g1\ g2$) ($Inr\ s2$))
 $list.next$ ($append_generator\ g1\ g2$) ($Inr\ s2$) = $apsnd\ Inr$ ($list.next\ g2\ s2$)
by(*transfer, simp add: apsnd-def map-prod-def*) $+$

lemma $unfoldr_append_generator_Inr$:
 $list.unfoldr$ ($append_generator\ g1\ g2$) ($Inr\ s2$) = $list.unfoldr\ g2\ s2$
by(*induct s2 taking: g2 rule: list.unfoldr.induct*)(*subst (1 2) list.unfoldr.simps,*
auto split: prod.splits)

lemma *unfoldr-append-generator-Inl*:
 $list.unfoldr$ (*append-generator* $g1$ $g2$) (*Inl* ($s1$, $s2$)) =
 $list.unfoldr$ $g1$ $s1$ @ $list.unfoldr$ $g2$ $s2$
apply(*induct* $s1$ *taking*: $g1$ *rule*: $list.unfoldr.induct$)
apply(*subst* (1 2 3) $list.unfoldr.simps$)
apply(*auto split*: $prod.splits$ *simp add*: $apsnd-def$ $map-prod-def$ *unfoldr-append-generator-Inr*)
apply(*simp add*: $list.unfoldr.simps$)
done

lemma *unfoldr-append-generator*:
 $list.unfoldr$ (*append-generator* $g1$ $g2$) (*append-init* $s1$ $s2$) =
 $list.unfoldr$ $g1$ $s1$ @ $list.unfoldr$ $g2$ $s2$
by(*simp add*: *unfoldr-append-generator-Inl* *append-init-def*)

lift-definition *zip-generator* :: ('a, 's1) *generator* \Rightarrow ('b, 's2) *generator* \Rightarrow ('a \times 'b, 's1 \times 's2) *generator*
is λ (*has-next1*, *next1*) (*has-next2*, *next2*).
 $(\lambda$ ($s1$, $s2$). *has-next1* $s1$ \wedge *has-next2* $s2$,
 λ ($s1$, $s2$). *let* (x , $s1'$) = *next1* $s1$; (y , $s2'$) = *next2* $s2$
in ((x , y), ($s1'$, $s2'$)))

proof(*rule terminatesI*, *safe*)
fix *has-next1* *next1* *has-next2* *next2* $s1$ $s2$
assume $t1$: *terminates* (*has-next1*, *next1*)
and $t2$: *terminates* (*has-next2*, *next2*)
have $s1 \in$ *terminates-on* (*has-next1*, *next1*) $s2 \in$ *terminates-on* (*has-next2*, *next2*)
using $t1$ $t2$ **by**(*simp-all add*: *terminatesD*)
thus ($s1$, $s2$) \in *terminates-on* (λ ($s1$, $s2$). *has-next1* $s1$ \wedge *has-next2* $s2$, λ ($s1$, $s2$). *let* (x , $s1'$) = *next1* $s1$; (y , $s2'$) = *next2* $s2$ *in* ((x , y), ($s1'$, $s2'$)))
by(*induct arbitrary*: $s2$)(*auto* 4 3 *elim*: *terminates-on.cases* *intro*: *terminates-on.intros* *simp add*: *split-beta* *Let-def*)
qed

abbreviation (*input*) *zip-init* :: 's1 \Rightarrow 's2 \Rightarrow 's1 \times 's2
where *zip-init* \equiv *Pair*

lemma *has-next-zip-generator* [*simp*]:
 $list.has-next$ (*zip-generator* $g1$ $g2$) ($s1$, $s2$) \longleftrightarrow
 $list.has-next$ $g1$ $s1$ \wedge $list.has-next$ $g2$ $s2$
by *transfer* *clarsimp*

lemma *next-zip-generator* [*simp*]:
 $list.next$ (*zip-generator* $g1$ $g2$) ($s1$, $s2$) =
((*fst* ($list.next$ $g1$ $s1$), *fst* ($list.next$ $g2$ $s2$)),
(*snd* ($list.next$ $g1$ $s1$), *snd* ($list.next$ $g2$ $s2$)))
by *transfer*(*simp add*: *split-beta*)

lemma *unfoldr-zip-generator*:

```

list.unfoldr (zip-generator g1 g2) (zip-init s1 s2) =
  zip (list.unfoldr g1 s1) (list.unfoldr g2 s2)
by(induct (s1, s2) arbitrary: s1 s2 taking: zip-generator g1 g2 rule: list.unfoldr.induct)
  (subst (1 2 3) list.unfoldr.simps, auto 9 2 simp add: split-beta)

```

context fixes *bound* :: nat **begin**

lift-definition *upt-generator* :: (nat, nat) generator
 is ($\lambda n. n < bound, \lambda n. (n, Suc\ n)$)
 by(rule wf-terminates)(relation measure ($\lambda n. bound - n$), auto)

lemma *has-next-upt-generator* [simp]:
 $list.has_next\ upt_generator\ n \longleftrightarrow n < bound$
 by transfer simp

lemma *next-upt-generator* [simp]:
 $list.next\ upt_generator\ n = (n, Suc\ n)$
 by transfer simp

lemma *unfoldr-upt-generator*:
 $list.unfoldr\ upt_generator\ n = [n..<bound]$
 by(induct bound - n arbitrary: n)(simp-all add: list.unfoldr-simps upt-conv-Cons)

end

context fixes *bound* :: int **begin**

lift-definition *upto-generator* :: (int, int) generator
 is ($\lambda n. n \leq bound, \lambda n. (n, n + 1)$)
 by(rule wf-terminates)(relation measure ($\lambda n. nat\ (bound + 1 - n)$), auto)

lemma *has-next-upto-generator* [simp]:
 $list.has_next\ upto_generator\ n \longleftrightarrow n \leq bound$
 by transfer simp

lemma *next-upto-generator* [simp]:
 $list.next\ upto_generator\ n = (n, n + 1)$
 by transfer simp

lemma *unfoldr-upto-generator*:
 $list.unfoldr\ upto_generator\ n = [n..bound]$
 by(induct n taking: upto-generator rule: list.unfoldr.induct)(subst list.unfoldr.simps,
 subst upto.simps, auto)

end

context
 fixes *P* :: 'a \Rightarrow bool
begin

```

context
  fixes  $g :: ('a, 's)$  raw-generator
begin

inductive  $filter\text{-}has\text{-}next :: 's \Rightarrow bool$ 
where
   $\llbracket fst\ g\ s;\ P\ (fst\ (snd\ g\ s)) \rrbracket \Longrightarrow filter\text{-}has\text{-}next\ s$ 
   $\llbracket fst\ g\ s;\ \neg\ P\ (fst\ (snd\ g\ s));\ filter\text{-}has\text{-}next\ (snd\ (snd\ g\ s)) \rrbracket \Longrightarrow filter\text{-}has\text{-}next\ s$ 

partial-function (tailrec)  $filter\text{-}next :: 's \Rightarrow 'a \times 's$ 
where
   $filter\text{-}next\ s = (let\ (x,\ s') = snd\ g\ s\ in\ if\ P\ x\ then\ (x,\ s')\ else\ filter\text{-}next\ s')$ 

end

lift-definition  $filter\text{-}generator :: ('a, 's)$  generator  $\Rightarrow ('a, 's)$  generator
  is  $\lambda g.$  ( $filter\text{-}has\text{-}next\ g,$   $filter\text{-}next\ g$ )
proof(rule wf-terminates)
  fix  $g :: ('a, 's)$  raw-generator and  $s$ 
  let  $?R = \{(snd\ (snd\ g\ s),\ s) \mid s.\ fst\ g\ s\}$ 
  let  $?g = (filter\text{-}has\text{-}next\ g,\ filter\text{-}next\ g)$ 
  assume terminates g
  thus  $wf\ (?R^+)$  by(rule terminates-wfD[THEN wf-trancl])
  assume  $fst\ ?g\ s$ 
  hence  $filter\text{-}has\text{-}next\ g\ s$  by simp
  thus  $(snd\ (snd\ ?g\ s),\ s) \in ?R^+$ 
  by induct(subst filter-next.simps, auto simp add: split-beta filter-next.simps split del: if-split intro: trancl-into-trancl)
qed

lemma has-next-filter-generator:
   $list.has\text{-}next\ (filter\text{-}generator\ g)\ s \longleftrightarrow$ 
   $list.has\text{-}next\ g\ s \wedge (let\ (x,\ s') = list.next\ g\ s\ in\ if\ P\ x\ then\ True\ else\ list.has\text{-}next$ 
   $(filter\text{-}generator\ g)\ s')$ 
apply(transfer)
apply simp
apply(subst filter-has-next.simps)
apply auto
done

lemma next-filter-generator:
   $list.next\ (filter\text{-}generator\ g)\ s =$ 
   $(let\ (x,\ s') = list.next\ g\ s$ 
   $in\ if\ P\ x\ then\ (x,\ s')\ else\ list.next\ (filter\text{-}generator\ g)\ s')$ 
apply transfer
apply simp
apply(subst filter-next.simps)
apply(simp cong: if-cong)

```

done

lemma *has-next-filter-generator-induct* [*consumes 1, case-names find step*]:
assumes *list.has-next (filter-generator g) s*
and find: $\bigwedge s. \llbracket \text{list.has-next } g \ s; P \ (\text{fst} \ (\text{list.next } g \ s)) \rrbracket \implies Q \ s$
and step: $\bigwedge s. \llbracket \text{list.has-next } g \ s; \neg P \ (\text{fst} \ (\text{list.next } g \ s)); Q \ (\text{snd} \ (\text{list.next } g \ s)) \rrbracket \implies Q \ s$
shows $Q \ s$
using *assms* **by** *transfer(auto elim: filter-has-next.induct)*

lemma *filter-generator-empty-conv:*

$\text{list.has-next} \ (\text{filter-generator } g) \ s \longleftrightarrow (\exists x \in \text{set} \ (\text{list.unfoldr } g \ s). P \ x) \ (\text{is } ?lhs \longleftrightarrow ?rhs)$

proof

assume *?lhs*

thus *?rhs*

proof(*induct rule: has-next-filter-generator-induct*)

case (*find s*)

thus *?case*

by(*cases list.next g s*)(*subst list.unfoldr.simps, auto*)

next

case (*step s*)

thus *?case*

by(*cases list.next g s*)(*subst list.unfoldr.simps, auto*)

qed

next

assume *?rhs*

then obtain *x* **where** $x \in \text{set} \ (\text{list.unfoldr } g \ s) \quad P \ x$ **by** *blast*

thus *?lhs*

proof(*induct xs ≡ list.unfoldr g s arbitrary: s*)

case *Nil* **thus** *?case* **by**(*simp del: Nil-eq-unfoldr-iff*)

next

case (*Cons x' xs*)

from $\langle x' \# xs = \text{list.unfoldr } g \ s \rangle$ [*symmetric, simp*]

have [*simp*]: $\text{fst} \ (\text{list.next } g \ s) = x' \wedge \text{list.has-next } g \ s \wedge \text{list.unfoldr } g \ (\text{snd} \ (\text{list.next } g \ s)) = xs$

by(*subst (asm) list.unfoldr.simps*)(*simp add: split-beta split: if-split-asm*)

from *Cons.hyps(1)*[*of snd (list.next g s)*] $\langle x \in \text{set} \ (\text{list.unfoldr } g \ s) \rangle \langle P \ x \rangle$ **show** *?case*

by(*subst has-next-filter-generator*)(*auto simp add: split-beta*)

qed

qed

lemma *unfoldr-filter-generator:*

$\text{list.unfoldr} \ (\text{filter-generator } g) \ s = \text{filter } P \ (\text{list.unfoldr } g \ s)$

unfolding *list-all2-eq*

proof(*coinduction arbitrary: s*)

case *Nil*

thus *?case* **by**(*simp add: filter-empty-conv filter-generator-empty-conv*)

```

next
  case (Cons s)
  hence list.has-next (filter-generator g) s by simp
  thus ?case
  proof(induction rule: has-next-filter-generator-induct)
    case (find s)
    thus ?case
      apply(subst (1 2 3 5) list.unfoldr.simps)
      apply(subst (1 2) has-next-filter-generator)
      apply(subst next-filter-generator)
      apply(simp add: split-beta)
      apply(rule disjI1 exI conjI refl)+
      apply(subst next-filter-generator)
      apply(simp add: split-beta)
    done
  next
  case (step s)
  from step.hyps
  have list.unfoldr (filter-generator g) s = list.unfoldr (filter-generator g) (snd
(list.next g s))
    apply(subst (1 2) list.unfoldr.simps)
    apply(subst has-next-filter-generator)
    apply(subst next-filter-generator)
    apply(auto simp add: split-beta)
  done
  moreover from step.hyps
  have filter P (list.unfoldr g (snd (list.next g s))) = filter P (list.unfoldr g s)
    by(subst (2) list.unfoldr.simps)(auto simp add: split-beta)
  ultimately show ?case using step.IH by simp
qed
qed
end

```

2.3.3 Destroying lists

definition *hd-fusion* :: ('a, 's) generator \Rightarrow 's \Rightarrow 'a
where *hd-fusion* g s = hd (list.unfoldr g s)

lemma *hd-fusion-code* [code]:

hd-fusion g s = (if list.has-next g s then fst (list.next g s) else undefined)

unfolding *hd-fusion-def*

by(subst list.unfoldr.simps)(simp add: hd-def split-beta)

declare *hd-fusion-def* [symmetric, code-unfold]

definition *fold-fusion* :: ('a, 's) generator \Rightarrow ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 's \Rightarrow 'b \Rightarrow 'b
where *fold-fusion* g f s = fold f (list.unfoldr g s)

lemma *fold-fusion-code* [code]:

fold-fusion $g f s b =$
 (if *list.has-next* $g s$ then
 let $(x, s') = \text{list.next } g s$
 in *fold-fusion* $g f s' (f x b)$
 else b)

unfolding *fold-fusion-def*

by(subst *list.unfoldr.simps*)(simp add: *split-beta*)

declare *fold-fusion-def*[*symmetric, code-unfold*]

definition *gen-length-fusion* :: ($'a, 's$) generator \Rightarrow nat \Rightarrow $'s \Rightarrow$ nat

where *gen-length-fusion* $g n s = n + \text{length } (\text{list.unfoldr } g s)$

lemma *gen-length-fusion-code* [code]:

gen-length-fusion $g n s =$
 (if *list.has-next* $g s$ then *gen-length-fusion* $g (\text{Suc } n) (\text{snd } (\text{list.next } g s))$ else n)

unfolding *gen-length-fusion-def*

by(subst *list.unfoldr.simps*)(simp add: *split-beta*)

definition *length-fusion* :: ($'a, 's$) generator \Rightarrow $'s \Rightarrow$ nat

where *length-fusion* $g s = \text{length } (\text{list.unfoldr } g s)$

lemma *length-fusion-code* [code]:

length-fusion $g = \text{gen-length-fusion } g 0$

by(simp add: *fun-eq-iff length-fusion-def gen-length-fusion-def*)

declare *length-fusion-def*[*symmetric, code-unfold*]

definition *map-fusion* :: ($'a \Rightarrow 'b$) \Rightarrow ($'a, 's$) generator \Rightarrow $'s \Rightarrow$ $'b$ list

where *map-fusion* $f g s = \text{map } f (\text{list.unfoldr } g s)$

lemma *map-fusion-code* [code]:

map-fusion $f g s =$
 (if *list.has-next* $g s$ then
 let $(x, s') = \text{list.next } g s$
 in $f x \# \text{map-fusion } f g s'$
 else [])

unfolding *map-fusion-def*

by(subst *list.unfoldr.simps*)(simp add: *split-beta*)

declare *map-fusion-def*[*symmetric, code-unfold*]

definition *append-fusion* :: ($'a, 's1$) generator \Rightarrow ($'a, 's2$) generator \Rightarrow $'s1 \Rightarrow$ $'s2$

\Rightarrow $'a$ list

where *append-fusion* $g1 g2 s1 s2 = \text{list.unfoldr } g1 s1 @ \text{list.unfoldr } g2 s2$

lemma *append-fusion* [code]:

append-fusion $g1 g2 s1 s2 =$

```

    (if list.has-next g1 s1 then
      let (x, s1') = list.next g1 s1
      in x # append-fusion g1 g2 s1' s2
    else list.unfoldr g2 s2)
unfolding append-fusion-def
by(subst list.unfoldr.simps)(simp add: split-beta)

declare append-fusion-def[symmetric, code-unfold]

definition zip-fusion :: ('a, 's1) generator  $\Rightarrow$  ('b, 's2) generator  $\Rightarrow$  's1  $\Rightarrow$  's2  $\Rightarrow$ 
('a  $\times$  'b) list
where zip-fusion g1 g2 s1 s2 = zip (list.unfoldr g1 s1) (list.unfoldr g2 s2)

lemma zip-fusion-code [code]:
  zip-fusion g1 g2 s1 s2 =
    (if list.has-next g1 s1  $\wedge$  list.has-next g2 s2 then
      let (x, s1') = list.next g1 s1;
          (y, s2') = list.next g2 s2
      in (x, y) # zip-fusion g1 g2 s1' s2'
    else [])
unfolding zip-fusion-def
by(subst (1 2) list.unfoldr.simps)(simp add: split-beta)

declare zip-fusion-def[symmetric, code-unfold]

definition list-all-fusion :: ('a, 's) generator  $\Rightarrow$  ('a  $\Rightarrow$  bool)  $\Rightarrow$  's  $\Rightarrow$  bool
where list-all-fusion g P s = List.list-all P (list.unfoldr g s)

lemma list-all-fusion-code [code]:
  list-all-fusion g P s  $\longleftrightarrow$ 
  (list.has-next g s  $\longrightarrow$ 
    (let (x, s') = list.next g s
     in P x  $\wedge$  list-all-fusion g P s'))
unfolding list-all-fusion-def
by(subst list.unfoldr.simps)(simp add: split-beta)

declare list-all-fusion-def[symmetric, code-unfold]

definition list-all2-fusion :: ('a  $\Rightarrow$  'b  $\Rightarrow$  bool)  $\Rightarrow$  ('a, 's1) generator  $\Rightarrow$  ('b, 's2)
generator  $\Rightarrow$  's1  $\Rightarrow$  's2  $\Rightarrow$  bool
where
  list-all2-fusion P g1 g2 s1 s2 =
    list-all2 P (list.unfoldr g1 s1) (list.unfoldr g2 s2)

lemma list-all2-fusion-code [code]:
  list-all2-fusion P g1 g2 s1 s2 =
    (if list.has-next g1 s1 then
      list.has-next g2 s2  $\wedge$ 
      (let (x, s1') = list.next g1 s1;

```

```

      (y, s2') = list.next g2 s2
    in P x y ∧ list-all2-fusion P g1 g2 s1' s2')
  else ¬ list.has-next g2 s2)
unfolding list-all2-fusion-def
by(subst (1 2) list.unfoldr.simps)(simp add: split-beta)

declare list-all2-fusion-def[symmetric, code-unfold]

definition singleton-list-fusion :: ('a, 'state) generator ⇒ 'state ⇒ bool
where singleton-list-fusion gen state = (case list.unfoldr gen state of [-] ⇒ True |
- ⇒ False)

lemma singleton-list-fusion-code [code]:
  singleton-list-fusion g s ↔
  list.has-next g s ∧ ¬ list.has-next g (snd (list.next g s))
by(auto 4 5 simp add: singleton-list-fusion-def split: list.split-if-split-asm prod.splits
elim: list.unfoldr.elims dest: sym)

end

theory Lexicographic-Order imports
  List-Fusion
  HOL-Library.Char-ord
begin

hide-const (open) List.lexordp

```

2.4 List fusion for lexicographic order

```
context linorder begin
```

```
lemma lexordp-take-index-conv:
```

```

lexordp xs ys ↔
  (length xs < length ys ∧ take (length xs) ys = xs) ∨
  (∃ i < min (length xs) (length ys). take i xs = take i ys ∧ xs ! i < ys ! i)
(is ?lhs = ?rhs)

```

```
proof
```

```
  assume ?lhs thus ?rhs
```

```
  by induct (auto 4 3 del: disjCI intro: disjI2 exI[where x=Suc i for i])
```

```
next
```

```
  assume ?rhs (is ?prefix ∨ ?less) thus ?lhs
```

```
  proof
```

```
    assume ?prefix
```

```
    hence ys = xs @ hd (drop (length xs) ys) # tl (drop (length xs) ys)
```

```
    by (metis append-Nil2 append-take-drop-id less-not-refl list.collapse)
```

```
    thus ?thesis unfolding lexordp-iff by blast
```

```
  next
```

```
    assume ?less
```

```

then obtain  $i$  where  $i < \min (\text{length } xs) (\text{length } ys)$ 
and  $\text{take } i \text{ } xs = \text{take } i \text{ } ys$  and  $\text{nth: } xs ! i < ys ! i$  by blast
hence  $xs = \text{take } i \text{ } xs @ xs ! i \# \text{drop } (\text{Suc } i) \text{ } xs$   $ys = \text{take } i \text{ } xs @ ys ! i \#$ 
 $\text{drop } (\text{Suc } i) \text{ } ys$ 
by  $-(\text{subst } \text{append-take-drop-id}[\text{symmetric, of-} i], \text{simp-all add: Cons-nth-drop-Suc})$ 
with  $\text{nth}$  show ?thesis unfolding lexordp-iff by blast
qed
qed

```

— *lexord* is extension of partial ordering *List.lex*

lemma *lexordp-lex*: $(xs, ys) \in \text{lex } \{(xs, ys). xs < ys\} \longleftrightarrow \text{lexordp } xs \text{ } ys \wedge \text{length } xs = \text{length } ys$

proof(*induct xs arbitrary: ys*)

case *Nil* **thus** *?case* **by** *clarsimp*

next

case *Cons* **thus** *?case* **by**(*cases ys*)(*simp-all, safe, simp*)

qed

end

2.4.1 Setup for list fusion

context *ord* **begin**

definition *lexord-fusion* :: $('a, 's1) \text{ generator} \Rightarrow ('a, 's2) \text{ generator} \Rightarrow 's1 \Rightarrow 's2 \Rightarrow \text{bool}$

where [*code del*]: $\text{lexord-fusion } g1 \text{ } g2 \text{ } s1 \text{ } s2 = \text{lexordp } (\text{list.unfoldr } g1 \text{ } s1) (\text{list.unfoldr } g2 \text{ } s2)$

definition *lexord-eq-fusion* :: $('a, 's1) \text{ generator} \Rightarrow ('a, 's2) \text{ generator} \Rightarrow 's1 \Rightarrow 's2 \Rightarrow \text{bool}$

where [*code del*]: $\text{lexord-eq-fusion } g1 \text{ } g2 \text{ } s1 \text{ } s2 = \text{lexordp-eq } (\text{list.unfoldr } g1 \text{ } s1) (\text{list.unfoldr } g2 \text{ } s2)$

lemma *lexord-fusion-code*:

$\text{lexord-fusion } g1 \text{ } g2 \text{ } s1 \text{ } s2 \longleftrightarrow$

(*if list.has-next g1 s1 then*

if list.has-next g2 s2 then

let $(x, s1') = \text{list.next } g1 \text{ } s1;$

$(y, s2') = \text{list.next } g2 \text{ } s2$

in $x < y \vee \neg y < x \wedge \text{lexord-fusion } g1 \text{ } g2 \text{ } s1' \text{ } s2'$

else False

else list.has-next g2 s2)

unfolding *lexord-fusion-def*

by(*subst (1 2) list.unfoldr.simps*)(*auto split: prod.split-asm*)

lemma *lexord-eq-fusion-code*:

$\text{lexord-eq-fusion } g1 \text{ } g2 \text{ } s1 \text{ } s2 \longleftrightarrow$

(*list.has-next g1 s1* \longrightarrow

2.5. EVERY PARTIAL ORDER CAN BE EXTENDED TO A TOTAL ORDER33

```

    list.has-next g2 s2 ∧
    (let (x, s1') = list.next g1 s1;
        (y, s2') = list.next g2 s2
        in x < y ∨ ¬ y < x ∧ lexord-eq-fusion g1 g2 s1' s2')
unfolding lexord-eq-fusion-def
by(subst (1 2) list.unfoldr.simps)(auto split: prod.split-asm)

end

```

```

lemmas [code] =
    lexord-fusion-code ord.lexord-fusion-code
    lexord-eq-fusion-code ord.lexord-eq-fusion-code

```

```

lemmas [symmetric, code-unfold] =
    lexord-fusion-def ord.lexord-fusion-def
    lexord-eq-fusion-def ord.lexord-eq-fusion-def

```

end

```

theory Extend-Partial-Order
imports Main
begin

```

2.5 Every partial order can be extended to a total order

```

lemma ChainsD:  $\llbracket x \in C; C \in \text{Chains } r; y \in C \rrbracket \implies (x, y) \in r \vee (y, x) \in r$ 
by(simp add: Chains-def)

```

```

lemma Chains-Field:  $\llbracket C \in \text{Chains } r; x \in C \rrbracket \implies x \in \text{Field } r$ 
by(auto simp add: Chains-def Field-def)

```

```

lemma total-onD:
     $\llbracket \text{total-on } A \ r; x \in A; y \in A \rrbracket \implies (x, y) \in r \vee x = y \vee (y, x) \in r$ 
unfolding total-on-def by blast

```

```

lemma linear-order-imp-linorder: linear-order  $\{(A, B). \text{leq } A \ B\} \implies \text{class.linorder}$ 
    leq  $(\lambda x \ y. \text{leq } x \ y \wedge \neg \text{leq } y \ x)$ 
by(unfold-locales)(auto 4 4 simp add: linear-order-on-def partial-order-on-def pre-
    order-on-def dest: refl-onD antisymD transD total-onD)

```

```

lemma (in linorder) linear-order: linear-order  $\{(A, B). A \leq B\}$ 
by(auto simp add: linear-order-on-def partial-order-on-def preorder-on-def total-on-def
    intro: refl-onI antisymI transI)

```

```

definition order-consistent :: ('a × 'a) set  $\implies$  ('a × 'a) set  $\implies$  bool

```

where *order-consistent* $r s \iff (\forall a a'. (a, a') \in r \longrightarrow (a', a) \in s \longrightarrow a = a')$

lemma *order-consistent-sym*:

order-consistent $r s \implies$ *order-consistent* $s r$

by(*auto simp add: order-consistent-def*)

lemma *antisym-order-consistent-self*:

antisym $r \implies$ *order-consistent* $r r$

by(*auto simp add: order-consistent-def dest: antisymD*)

lemma *refl-on-trancl*:

assumes *refl-on* $A r$

shows *refl-on* $A (r^{\widehat{+}})$

proof (*rule refl-onI*)

show $\bigwedge x. x \in A \implies (x, x) \in r^+$

using *assms refl-on-def* **by** *fastforce*

qed

lemma *total-on-refl-on-consistent-into*:

assumes r : *total-on* $A r$ *refl-on* $A r$

and *consist*: *order-consistent* $r s$

and $x: x \in A$ **and** $y: y \in A$ **and** $s: (x, y) \in s$

shows $(x, y) \in r$

proof(*cases* $x = y$)

case *False*

with $r x y$ **have** $(x, y) \in r \vee (y, x) \in r$ **unfolding** *total-on-def* **by** *blast*

thus *?thesis*

proof

assume $(y, x) \in r$

with s *consist* **have** $x = y$ **unfolding** *order-consistent-def* **by** *blast*

with *False* **show** *?thesis* **by** *contradiction*

qed

qed(*blast intro: refl-onD r x y*)

lemma *porder-linorder-tranclpE* [*consumes 5, case-names base step*]:

assumes r : *partial-order-on* $A r$

and s : *linear-order-on* $B s$

and *consist*: *order-consistent* $r s$

and *B-subset-A*: $B \subseteq A$

and *trancl*: $(x, y) \in (r \cup s)^{\widehat{+}}$

obtains $(x, y) \in r$

| $u v$ **where** $(x, u) \in r$ $(u, v) \in s$ $(v, y) \in r$

proof(*atomize-elim*)

from r **have** *refl-on* $A r$ *trans* r **by**(*simp-all add: partial-order-on-def pre-order-on-def*)

from s **have** *refl-on* $B s$ *total-on* $B s$ *trans* s

by(*simp-all add: partial-order-on-def preorder-on-def linear-order-on-def*)

2.5. EVERY PARTIAL ORDER CAN BE EXTENDED TO A TOTAL ORDER 35

```

from tranci show  $(x, y) \in r \vee (\exists u v. (x, u) \in r \wedge (u, v) \in s \wedge (v, y) \in r)$ 
proof(induction)
  case (base y)
  thus ?case
  proof
    assume  $(x, y) \in s$ 
    with  $\langle \text{refl-on } B \ s \rangle$  have  $x \in B \quad y \in B$ 
    unfolding atomize-conj
    using order-on-defs(1)[of  $B \ s$ ] order-on-defs(2)[of  $B \ s$ ] order-on-defs(3)[of
   $B \ s$ ] s
    by blast
    with  $B\text{-subset-}A$  have  $x \in A \quad y \in A$  by blast+
    hence  $(x, x) \in r \quad (y, y) \in r$ 
    using  $\langle \text{refl-on } A \ r \rangle$  by(blast intro: refl-onD)+
    with  $\langle (x, y) \in s \rangle$  show ?thesis by blast
  qed(simp)
next
  case (step y z)
  from  $\langle (y, z) \in r \cup s \rangle$  show ?case
  proof
    assume  $(y, z) \in s$ 
    with  $\langle \text{refl-on } B \ s \rangle$  have  $y \in B \quad z \in B$ 
    unfolding atomize-conj
    using linear-order-on-def[of  $B \ s$ ] order-on-defs(1)[of  $B \ s$ ] partial-order-on-def[of
   $B \ s$ ] s
    by blast
    from step.IH show ?thesis
    proof
      assume  $(x, y) \in r$ 
      moreover from  $\langle z \in B \rangle$   $B\text{-subset-}A$   $\langle \text{refl-on } A \ r \rangle$ 
      have  $(z, z) \in r$  by(blast dest: refl-onD)
      ultimately show ?thesis using  $\langle (y, z) \in s \rangle$  by blast
    next
      assume  $\exists u v. (x, u) \in r \wedge (u, v) \in s \wedge (v, y) \in r$ 
      then obtain  $u \ v$  where  $(x, u) \in r \quad (u, v) \in s \quad (v, y) \in r$  by blast
      from  $\langle (u, v) \in s \rangle$  have  $v \in B$ 
      using linear-order-on-def[of  $B \ s$ ] order-on-defs(1)[of  $B \ s$ ] partial-order-on-def[of
   $B \ s$ ] s
      by blast
      with  $\langle \text{total-on } B \ s \rangle$   $\langle \text{refl-on } B \ s \rangle$  order-consistent-sym[OF consist]
      have  $(v, y) \in s$  using  $\langle y \in B \rangle$   $\langle (v, y) \in r \rangle$ 
      by(rule total-on-refl-on-consistent-into)
      with  $\langle \text{trans } s \rangle$  have  $(v, z) \in s$  using  $\langle (y, z) \in s \rangle$  by(rule transD)
      with  $\langle \text{trans } s \rangle$   $\langle (u, v) \in s \rangle$  have  $(u, z) \in s$  by(rule transD)
      moreover from  $\langle z \in B \rangle$   $B\text{-subset-}A$  have  $z \in A \ ..$ 
      with  $\langle \text{refl-on } A \ r \rangle$  have  $(z, z) \in r$  by(rule refl-onD)
      ultimately show ?thesis using  $\langle (x, u) \in r \rangle$  by blast
    qed
  next

```

```

    assume (y, z) ∈ r
    with step.IH show ?thesis
    by(blast intro: transD[OF ‹trans r›])
  qed
qed
qed

lemma porder-on-consistent-linorder-on-trancl-antisym:
  assumes r: partial-order-on A r
  and s: linear-order-on B s
  and consist: order-consistent r s
  and B-subset-A: B ⊆ A
  shows antisym ((r ∪ s)ˆ+)
proof(rule antisymI)
  fix x y
  let ?rs = (r ∪ s)ˆ+
  assume (x, y) ∈ ?rs (y, x) ∈ ?rs
  from r have antisym r trans r by(simp-all add: partial-order-on-def pre-
order-on-def)
  from s have total-on B s refl-on B s trans s antisym s
    by(simp-all add: partial-order-on-def preorder-on-def linear-order-on-def)

  from r s consist B-subset-A ‹(x, y) ∈ ?rs›
  show x = y
proof(cases rule: porder-linorder-tranclpE)
  case base
  from r s consist B-subset-A ‹(y, x) ∈ ?rs›
  show ?thesis
proof(cases rule: porder-linorder-tranclpE)
  case base
  with ‹antisym r› ‹(x, y) ∈ r› show ?thesis by(rule antisymD)
next
  case (step u v)
  from ‹(v, x) ∈ r› ‹(x, y) ∈ r› ‹(y, u) ∈ r› have (v, u) ∈ r
    by(blast intro: transD[OF ‹trans r›])
  with consist have v = u using ‹(u, v) ∈ s›
    by(simp add: order-consistent-def)
  with ‹(y, u) ∈ r› ‹(v, x) ∈ r› have (y, x) ∈ r
    by(blast intro: transD[OF ‹trans r›])
  with ‹antisym r› ‹(x, y) ∈ r› show ?thesis by(rule antisymD)
qed
next
  case (step u v)
  from r s consist B-subset-A ‹(y, x) ∈ ?rs›
  show ?thesis
proof(cases rule: porder-linorder-tranclpE)
  case base
  from ‹(v, y) ∈ r› ‹(y, x) ∈ r› ‹(x, u) ∈ r› have (v, u) ∈ r
    by(blast intro: transD[OF ‹trans r›])

```

2.5. EVERY PARTIAL ORDER CAN BE EXTENDED TO A TOTAL ORDER³⁷

```

with consist  $\langle (u, v) \in s \rangle$ 
have  $u = v$  by(auto simp add: order-consistent-def)
with  $\langle (v, y) \in r \rangle \langle (x, u) \in r \rangle$  have  $(x, y) \in r$ 
  by(blast intro: transD[OF  $\langle \text{trans } r \rangle$ ])
with  $\langle \text{antisym } r \rangle$  show ?thesis using  $\langle (y, x) \in r \rangle$  by(rule antisymD)
next
  case (step  $u' v'$ )
  note  $r\text{-into-}s = \text{total-on-refl-on-consistent-into}[OF \langle \text{total-on } B \ s \rangle \langle \text{refl-on } B \ s \rangle$ 
order-consistent-sym[OF consist]]
  from  $\langle \text{refl-on } B \ s \rangle \langle (u, v) \in s \rangle \langle (u', v') \in s \rangle$ 
  have  $u \in B \quad v \in B \quad u' \in B \quad v' \in B$ 
  unfolding atomize-conj
  using linear-order-on-def[of  $B \ s$ ] partial-order-on-def[of  $B \ s$ ] preorder-on-def[of
 $B \ s$ ] s
  by blast
  from  $\langle \text{trans } r \rangle \langle (v', x) \in r \rangle \langle (x, u) \in r \rangle$  have  $(v', u) \in r$  by(rule transD)
  with  $\langle v' \in B \rangle \langle u \in B \rangle$  have  $(v', u) \in s$  by(rule r-into-s)
  also note  $\langle (u, v) \in s \rangle$  also (transD[OF  $\langle \text{trans } s \rangle$ ])
  from  $\langle \text{trans } r \rangle \langle (v, y) \in r \rangle \langle (y, u') \in r \rangle$  have  $(v, u') \in r$  by(rule transD)
  with  $\langle v \in B \rangle \langle u' \in B \rangle$  have  $(v, u') \in s$  by(rule r-into-s)
  finally (transD[OF  $\langle \text{trans } s \rangle$ ])
  have  $v' = u'$  using  $\langle (u', v') \in s \rangle$  by(rule antisymD[OF  $\langle \text{antisym } s \rangle$ ])
  moreover with  $\langle (v, u') \in s \rangle \langle (v', u) \in s \rangle$  have  $(v, u) \in s$ 
  by(blast intro: transD[OF  $\langle \text{trans } s \rangle$ ])
  with  $\langle \text{antisym } s \rangle \langle (u, v) \in s \rangle$  have  $u = v$  by(rule antisymD)
  ultimately have  $(x, y) \in r \quad (y, x) \in r$ 
  using  $\langle (x, u) \in r \rangle \langle (v, y) \in r \rangle \langle (y, u') \in r \rangle \langle (v', x) \in r \rangle$ 
  by(blast intro: transD[OF  $\langle \text{trans } r \rangle$ ])+
  with  $\langle \text{antisym } r \rangle$  show ?thesis by(rule antisymD)
qed
qed
qed

```

lemma *porder-on-linorder-on-tranclp-porder-onI*:

```

assumes  $r$ : partial-order-on  $A \ r$ 
and  $s$ : linear-order-on  $B \ s$ 
and consist: order-consistent  $r \ s$ 
and subset:  $B \subseteq A$ 
shows partial-order-on  $A \ ((r \cup s)^\wedge+)$ 
unfolding partial-order-on-def preorder-on-def
proof(intro conjI)
  let  $?rs = r \cup s$ 
  show  $?rs^+ \subseteq A \times A$ 
  by (smt (verit, ccfv-threshold) consist mem-Sigma-iff order-on-defs(1,2)
  porder-linorder-tranclpE r s subrelI subset subset-iff)
from  $r$  have refl-on  $A \ r$  by(simp add: partial-order-on-def preorder-on-def)
moreover from  $s$  have refl-on  $B \ s$ 
  by(simp add: linear-order-on-def partial-order-on-def preorder-on-def)
ultimately have refl-on  $(A \cup B) \ ?rs$  by(rule refl-on-Un)

```

also from *subset* have $A \cup B = A$ by *blast*
 finally show *refl-on* A ($?rs^+$) by(*rule refl-on-trancl*)

show *trans* ($?rs^+$) by(*rule trans-trancl*)

from r *s consist subset* show *antisym* ($?rs^+$)
 by(*rule porder-on-consistent-linorder-on-trancl-antisym*)

qed

lemma *porder-extend-to-linorder*:

assumes r : *partial-order-on* A r

obtains s where *linear-order-on* A s *order-consistent* r s

proof(*atomize-elim*)

define S where $S = \{s. \text{partial-order-on } A \ s \wedge r \subseteq s\}$

from r have *r-in-S*: $r \in S$ unfolding *S-def* by *auto*

have $\exists y \in S. \forall x \in S. y \subseteq x \longrightarrow x = y$

proof(*rule Zorn-Lemma2*[*rule-format*])

fix c

assume $c \in \text{chains } S$

hence $c \subseteq S$ by(*rule chainsD2*)

show $\exists y \in S. \forall x \in c. x \subseteq y$

proof(*cases* $c = \{\}$)

case *True*

with *r-in-S* show *?thesis* by *blast*

next

case *False*

then obtain s where $s \in c$ by *blast*

hence s : *partial-order-on* A s

and *r-in-s*: $r \subseteq s$

using $\langle c \subseteq S \rangle$ unfolding *S-def* by *blast+*

have *partial-order-on* A $(\bigcup c)$

unfolding *partial-order-on-def* *preorder-on-def*

proof(*intro conjI*)

show $\bigcup c \subseteq A \times A$

proof (*rule subsetI*)

fix x

assume $x \in \bigcup c$

then obtain X where $X \in c$ and $x \in X$ by *blast*

from $\langle X \in c \rangle \langle c \subseteq S \rangle$ have $X \in S$..

hence *partial-order-on* A X unfolding *S-def* by *simp*

with $\langle x \in X \rangle$ show $x \in A \times A$

using *partial-order-onD* by *fastforce*

qed

next

show *refl-on* A $(\bigcup c)$

proof(*rule refl-onI*)

2.5. EVERY PARTIAL ORDER CAN BE EXTENDED TO A TOTAL ORDER 39

```

next
  fix x
  assume  $x \in A$ 
  with s have  $(x, x) \in s$  unfolding partial-order-on-def preorder-on-def
    by(blast dest: refl-onD)
  with  $\langle s \in c \rangle$  show  $(x, x) \in \bigcup c$  by(rule UnionI)
qed

show antisym  $(\bigcup c)$ 
proof(rule antisymI)
  fix x y
  assume  $(x, y) \in \bigcup c \quad (y, x) \in \bigcup c$ 
  then obtain X Y where  $X \in c \quad Y \in c \quad (x, y) \in X \quad (y, x) \in Y$  by
blast
  from  $\langle X \in c \rangle \langle Y \in c \rangle \langle c \subseteq S \rangle$  have antisym X    antisym Y
    unfolding S-def by(auto simp add: partial-order-on-def)
  moreover from  $\langle c \in chains S \rangle \langle X \in c \rangle \langle Y \in c \rangle$ 
  have  $X \subseteq Y \vee Y \subseteq X$  by(rule chainsD)
  ultimately show  $x = y$  using  $\langle (x, y) \in X \rangle \langle (y, x) \in Y \rangle$ 
    by(auto dest: antisymD)
qed

show trans  $(\bigcup c)$ 
proof(rule transI)
  fix x y z
  assume  $(x, y) \in \bigcup c \quad (y, z) \in \bigcup c$ 
  then obtain X Y where  $X \in c \quad Y \in c \quad (x, y) \in X \quad (y, z) \in Y$  by
blast
  from  $\langle X \in c \rangle \langle Y \in c \rangle \langle c \subseteq S \rangle$  have trans X    trans Y
    unfolding S-def by(auto simp add: partial-order-on-def preorder-on-def)
  from  $\langle c \in chains S \rangle \langle X \in c \rangle \langle Y \in c \rangle$ 
  have  $X \subseteq Y \vee Y \subseteq X$  by(rule chainsD)
  thus  $(x, z) \in \bigcup c$ 
  proof
    assume  $X \subseteq Y$ 
    with  $\langle trans Y \rangle \langle (x, y) \in X \rangle \langle (y, z) \in Y \rangle$ 
    have  $(x, z) \in Y$  by(blast dest: transD)
    with  $\langle Y \in c \rangle$  show ?thesis by(rule UnionI)
  next
    assume  $Y \subseteq X$ 
    with  $\langle trans X \rangle \langle (x, y) \in X \rangle \langle (y, z) \in Y \rangle$ 
    have  $(x, z) \in X$  by(blast dest: transD)
    with  $\langle X \in c \rangle$  show ?thesis by(rule UnionI)
  qed
qed
qed
moreover
have  $r \subseteq \bigcup c$  using r-in-s  $\langle s \in c \rangle$  by blast
ultimately have  $\bigcup c \in S$  unfolding S-def by simp

```

thus *?thesis* by *blast*
 qed
 qed
 then obtain *s* where $s \in S$ and *y-max*: $\bigwedge t. \llbracket t \in S; s \subseteq t \rrbracket \implies s = t$ by *blast*

 have *partial-order-on A s* using $\langle s \in S \rangle$
 unfolding *S-def* by *simp*
 moreover
 have *r-in-s*: $r \subseteq s$ using $\langle s \in S \rangle$ unfolding *S-def* by *blast*

 have *total-on A s*
 unfolding *total-on-def*
 proof(*intro strip*)
 fix *x y*
 assume $x \in A \quad y \in A \quad x \neq y$
 show $(x, y) \in s \vee (y, x) \in s$
 proof(*rule ccontr*)
 assume $\neg ?thesis$
 hence *xy*: $(x, y) \notin s \quad (y, x) \notin s$ by *simp-all*

 define *s'* where $s' = \{(a, b). a = x \wedge (b = y \vee b = x) \vee a = y \wedge b = y\}$
 let $?s' = (s \cup s')^+$
 note $\langle \text{partial-order-on } A \ s \rangle$
 moreover have *linear-order-on* $\{x, y\}$ *s'* unfolding *s'-def*
 by(*auto simp add: linear-order-on-def partial-order-on-def preorder-on-def*
total-on-def intro: refl-onI transI antisymI)
 moreover have *order-consistent s s'*
 unfolding *s'-def* using *xy* unfolding *order-consistent-def* by *blast*
 moreover have $\{x, y\} \subseteq A$ using $\langle x \in A \rangle \langle y \in A \rangle$ by *blast*
 ultimately have *partial-order-on A ?s'*
 by(*rule porder-on-linorder-on-transclp-porder-onI*)
 moreover have $r \subseteq ?s'$ using *r-in-s* by *auto*
 ultimately have $?s' \in S$ unfolding *S-def* by *simp*
 moreover have $s \subseteq ?s'$ by *auto*
 ultimately have $s = ?s'$ by(*rule y-max*)
 moreover have $(x, y) \in ?s'$ by(*auto simp add: s'-def*)
 ultimately show *False* using $\langle (x, y) \notin s \rangle$ by *simp*
 qed
 qed
 ultimately have *linear-order-on A s* by(*simp add: linear-order-on-def*)
 moreover have *order-consistent r s* unfolding *order-consistent-def*
 proof(*intro strip*)
 fix *a a'*
 assume $(a, a') \in r \quad (a', a) \in s$
 from $\langle (a, a') \in r \rangle$ have $(a, a') \in s$ using *r-in-s* by *blast*
 with $\langle \text{partial-order-on } A \ s \rangle \langle (a', a) \in s \rangle$
 show $a = a'$ unfolding *partial-order-on-def* by(*blast dest: antisymD*)
 qed
 ultimately show $\exists s. \text{linear-order-on } A \ s \wedge \text{order-consistent } r \ s$ by *blast*

qed

end

```

theory Set-Linorder
imports
  Containers-Auxiliary
  Lexicographic-Order
  Extend-Partial-Order
  HOL-Library.Cardinality
begin

```

2.6 An executable linear order on sets

2.6.1 Definition of the linear order

Extending finite and cofinite sets

Partition sets into finite and cofinite sets and distribute the rest arbitrarily such that complement switches between the two.

consts *infinite-complement-partition* :: 'a set set

specification (*infinite-complement-partition*)
finite-complement-partition: $finite\ A :: 'a\ set \implies A \in infinite-complement-partition$
complement-partition: $\neg\ finite\ (UNIV :: 'a\ set)$
 $\implies (A :: 'a\ set) \in infinite-complement-partition \iff \neg\ A \notin infinite-complement-partition$

proof(*cases finite (UNIV :: 'a set)*)

case *False*

define *Field-r* **where** $Field-r = \{\mathcal{P} :: 'a\ set\ set. (\forall C \in \mathcal{P}. \neg C \notin \mathcal{P}) \wedge (\forall A. finite\ A \implies A \in \mathcal{P})\}$

define *r* **where** $r = \{(\mathcal{P}1, \mathcal{P}2). \mathcal{P}1 \subseteq \mathcal{P}2 \wedge \mathcal{P}1 \in Field-r \wedge \mathcal{P}2 \in Field-r\}$

have *in-Field-r* [*simp*]: $\bigwedge \mathcal{P}. \mathcal{P} \in Field-r \iff (\forall C \in \mathcal{P}. \neg C \notin \mathcal{P}) \wedge (\forall A. finite\ A \implies A \in \mathcal{P})$

unfolding *Field-r-def* **by** *simp*

have *in-r* [*simp*]: $\bigwedge \mathcal{P}1\ \mathcal{P}2. (\mathcal{P}1, \mathcal{P}2) \in r \iff \mathcal{P}1 \subseteq \mathcal{P}2 \wedge \mathcal{P}1 \in Field-r \wedge \mathcal{P}2 \in Field-r$

unfolding *r-def* **by** *simp*

have *Field-r* [*simp*]: $Field\ r = Field-r$ **by**(*auto simp add: Field-def Field-r-def*)

have *Partial-order r*

by(*auto simp add: Field-def r-def partial-order-on-def preorder-on-def intro!: refl-onI transI antisymI*)

moreover **have** $\exists \mathcal{B} \in Field\ r. \forall \mathcal{A} \in \mathcal{C}. (\mathcal{A}, \mathcal{B}) \in r$ **if** \mathcal{C} : $\mathcal{C} \in Chains\ r$ **for** \mathcal{C}

proof –

let $?\mathcal{B} = \bigcup \mathcal{C} \cup \{A. finite\ A\}$

have $*$: $?\mathcal{B} \in Field\ r$ **using** *False C*

by *clarsimp(safe, drule (2) ChainsD, auto 4 4 dest: Chains-Field)*

hence $\bigwedge \mathcal{A}. \mathcal{A} \in \mathcal{C} \implies (\mathcal{A}, ?\mathcal{B}) \in r$

```

    using  $\mathfrak{C}$  by(auto simp del: in-Field-r dest: Chains-Field)
    with * show  $\exists \mathcal{B} \in \text{Field } r. \forall \mathcal{A} \in \mathfrak{C}. (\mathcal{A}, \mathcal{B}) \in r$  by blast
  qed
  ultimately have  $\exists \mathcal{P} \in \text{Field } r. \forall \mathcal{A} \in \text{Field } r. (\mathcal{P}, \mathcal{A}) \in r \longrightarrow \mathcal{A} = \mathcal{P}$ 
    by(rule Zorns-po-lemma)
  then obtain  $\mathcal{P}$  where  $\mathcal{P}: \mathcal{P} \in \text{Field } r$ 
    and max:  $\bigwedge \mathcal{A}. [\mathcal{A} \in \text{Field } r; (\mathcal{P}, \mathcal{A}) \in r] \implies \mathcal{A} = \mathcal{P}$  by blast
  have  $\forall A. \text{finite } A \longrightarrow A \in \mathcal{P}$  using  $\mathcal{P}$  by simp
  moreover {
    fix  $C$ 
    have  $C \in \mathcal{P} \vee - C \in \mathcal{P}$ 
    proof(rule ccontr)
      assume  $\neg ?thesis$ 
      hence  $C: C \notin \mathcal{P} \quad - C \notin \mathcal{P}$  by simp-all
      let  $?A = \text{insert } C \mathcal{P}$ 
      have *:  $?A \in \text{Field } r$  using  $C \mathcal{P}$  by auto
      hence  $(\mathcal{P}, ?A) \in r$  using  $\mathcal{P}$  by auto
      with * have  $?A = \mathcal{P}$  by(rule max)
      thus False using  $C$  by auto
    qed
    hence  $C \in \mathcal{P} \longleftrightarrow - C \notin \mathcal{P}$  using  $\mathcal{P}$  by auto }
  ultimately have  $\exists \mathcal{P} :: 'a \text{ set set}. (\forall A. \text{finite } A \longrightarrow A \in \mathcal{P}) \wedge (\forall C. C \in \mathcal{P} \longleftrightarrow - C \notin \mathcal{P})$ 
    by blast
  thus  $?thesis$  by metis
  qed auto

```

lemma not-in-complement-partition:

```

   $\neg \text{finite } (UNIV :: 'a \text{ set})$ 
   $\implies (A :: 'a \text{ set}) \notin \text{infinite-complement-partition} \longleftrightarrow - A \in \text{infinite-complement-partition}$ 
  by(metis complement-partition)

```

lemma not-in-complement-partition-False:

```

   $[(A :: 'a \text{ set}) \in \text{infinite-complement-partition}; \neg \text{finite } (UNIV :: 'a \text{ set})]$ 
   $\implies - A \in \text{infinite-complement-partition} = \text{False}$ 
  by(simp add: not-in-complement-partition)

```

lemma infinite-complement-partition-finite [simp]:

```

   $\text{finite } (UNIV :: 'a \text{ set}) \implies \text{infinite-complement-partition} = (UNIV :: 'a \text{ set set})$ 
  by(auto intro: finite-subset finite-complement-partition)

```

lemma Compl-eq-empty-iff: $- A = \{\} \longleftrightarrow A = UNIV$

by auto

A lexicographic-style order on finite subsets

context *ord* begin

definition set-less-aux :: $'a \text{ set} \Rightarrow 'a \text{ set} \Rightarrow \text{bool}$ (infix $\langle \square'' \rangle$ 50)

where $A \sqsubset' B \iff \text{finite } A \wedge \text{finite } B \wedge (\exists y \in B - A. \forall z \in (A - B) \cup (B - A). y \leq z \wedge (z \leq y \longrightarrow y = z))$

definition *set-less-eq-aux* :: 'a set \Rightarrow 'a set \Rightarrow bool (**infix** $\langle \sqsubseteq' \rangle$ 50)

where $A \sqsubseteq' B \iff A \in \text{infinite-complement-partition} \wedge A = B \vee A \sqsubset' B$

lemma *set-less-aux-irrefl* [*iff*]: $\neg A \sqsubset' A$

by(*auto simp add: set-less-aux-def*)

lemma *set-less-eq-aux-refl* [*iff*]: $A \sqsubseteq' A \iff A \in \text{infinite-complement-partition}$

by(*simp add: set-less-eq-aux-def*)

lemma *set-less-aux-empty* [*simp*]: $\neg A \sqsubset' \{\}$

by(*auto simp add: set-less-aux-def intro: finite-subset finite-complement-partition*)

lemma *set-less-eq-aux-empty* [*simp*]: $A \sqsubseteq' \{\} \iff A = \{\}$

by(*auto simp add: set-less-eq-aux-def finite-complement-partition*)

lemma *set-less-aux-antisym*: $\llbracket A \sqsubset' B; B \sqsubset' A \rrbracket \implies \text{False}$

by(*auto simp add: set-less-aux-def split: if-split-asm*)

lemma *set-less-aux-conv-set-less-eq-aux*:

$A \sqsubset' B \iff A \sqsubseteq' B \wedge \neg B \sqsubseteq' A$

by(*auto simp add: set-less-eq-aux-def dest: set-less-aux-antisym*)

lemma *set-less-eq-aux-antisym*: $\llbracket A \sqsubseteq' B; B \sqsubseteq' A \rrbracket \implies A = B$

by(*auto simp add: set-less-eq-aux-def dest: set-less-aux-antisym*)

lemma *set-less-aux-finiteD*: $A \sqsubset' B \implies \text{finite } A \wedge B \in \text{infinite-complement-partition}$

by(*auto simp add: set-less-aux-def finite-complement-partition*)

lemma *set-less-eq-aux-infinite-complement-partitionD*:

$A \sqsubseteq' B \implies A \in \text{infinite-complement-partition} \wedge B \in \text{infinite-complement-partition}$

by(*auto simp add: set-less-eq-aux-def dest: set-less-aux-finiteD intro: finite-complement-partition*)

lemma *Compl-set-less-aux-Compl*:

$\text{finite } (\text{UNIV} :: 'a \text{ set}) \implies \neg A \sqsubset' - B \iff B \sqsubset' A$

by(*auto simp add: set-less-aux-def finite-subset[OF subset-UNIV]*)

lemma *Compl-set-less-eq-aux-Compl*:

$\text{finite } (\text{UNIV} :: 'a \text{ set}) \implies \neg A \sqsubseteq' - B \iff B \sqsubseteq' A$

by(*auto simp add: set-less-eq-aux-def Compl-set-less-aux-Compl*)

lemma *set-less-aux-insert-same*:

$x \in A \iff x \in B \implies \text{insert } x A \sqsubset' \text{insert } x B \iff A \sqsubset' B$

by(*auto simp add: set-less-aux-def*)

lemma *set-less-eq-aux-insert-same*:

$\llbracket A \in \text{infinite-complement-partition}; \text{insert } x B \in \text{infinite-complement-partition};$

$x \in A \longleftrightarrow x \in B$]
 $\implies \text{insert } x A \sqsubseteq' \text{insert } x B \longleftrightarrow A \sqsubseteq' B$
by(*auto simp add: set-less-eq-aux-def set-less-aux-insert-same*)

end

context *order* **begin**

lemma *set-less-aux-singleton-iff*: $A \sqsubseteq' \{x\} \longleftrightarrow \text{finite } A \wedge (\forall a \in A. x < a)$
by(*auto simp add: set-less-aux-def less-le Bex-def*)

end

context *linorder* **begin**

lemma *wlog-le* [*case-names sym le*]:
assumes $\bigwedge a b. P a b \implies P b a$
and $\bigwedge a b. a \leq b \implies P a b$
shows $P b a$
by (*metis assms linear*)

lemma *empty-set-less-aux* [*simp*]: $\{\} \sqsubseteq' A \longleftrightarrow A \neq \{\} \wedge \text{finite } A$
by(*auto 4 3 simp add: set-less-aux-def intro!: Min-eqI intro: bexI* [**where** $x = \text{Min } A$]
order-trans [**where** $y = \text{Min } A$] *Min-in*)

lemma *empty-set-less-eq-aux* [*simp*]: $\{\} \sqsubseteq' A \longleftrightarrow \text{finite } A$
by(*auto simp add: set-less-eq-aux-def finite-complement-partition*)

lemma *set-less-aux-trans*:
assumes $AB: A \sqsubseteq' B$ **and** $BC: B \sqsubseteq' C$
shows $A \sqsubseteq' C$

proof –

from $AB BC$ **have** $A: \text{finite } A$ **and** $B: \text{finite } B$ **and** $C: \text{finite } C$

by(*auto simp add: set-less-aux-def*)

from $AB A B$ **obtain** ab **where** $ab \in B - A$

and $\text{min}AB: \bigwedge x. \llbracket x \in A; x \notin B \rrbracket \implies ab \leq x \wedge (x \leq ab \longrightarrow ab = x)$

and $\text{min}BA: \bigwedge x. \llbracket x \in B; x \notin A \rrbracket \implies ab \leq x \wedge (x \leq ab \longrightarrow ab = x)$

unfolding *set-less-aux-def* **by** *auto*

from $BC B C$ **obtain** bc **where** $bc \in C - B$

and $\text{min}BC: \bigwedge x. \llbracket x \in B; x \notin C \rrbracket \implies bc \leq x \wedge (x \leq bc \longrightarrow bc = x)$

and $\text{min}CB: \bigwedge x. \llbracket x \in C; x \notin B \rrbracket \implies bc \leq x \wedge (x \leq bc \longrightarrow bc = x)$

unfolding *set-less-aux-def* **by** *auto*

show *?thesis*

proof(*cases* $ab \leq bc$)

case *True*

hence $ab \in C - A$ $ab \notin A - C$

using $ab bc$ **by**(*auto dest: minBC antisym*)

moreover {

fix x

```

assume  $x: x \in (C - A) \cup (A - C)$ 
hence  $ab \leq x$ 
  by(cases  $x \in B$ )(auto dest: minAB minBA minBC minCB intro: or-
der-trans[OF True])
  moreover hence  $ab \neq x \longrightarrow \neg x \leq ab$  using  $ab \leq bc$   $x$ 
    by(cases  $x \in B$ )(auto dest: antisym)
  moreover note calculation }
ultimately show ?thesis using  $A$   $C$  unfolding set-less-aux-def by auto
next
case False
with linear[of  $ab \leq bc$ ] have  $bc \leq ab$  by simp
with  $ab \leq bc$  have  $bc \in C - A \quad bc \notin A - C$  by(auto dest: minAB antisym)
moreover {
  fix  $x$ 
  assume  $x: x \in (C - A) \cup (A - C)$ 
  hence  $bc \leq x$ 
    by(cases  $x \in B$ )(auto dest: minAB minBA minBC minCB intro: or-
der-trans[OF <math>bc \leq ab</math>])
    moreover hence  $bc \neq x \longrightarrow \neg x \leq bc$  using  $ab \leq bc$   $x$ 
      by(cases  $x \in B$ )(auto dest: antisym)
    moreover note calculation }
  ultimately show ?thesis using  $A$   $C$  unfolding set-less-aux-def by auto
qed
qed

```

```

lemma set-less-eq-aux-trans [trans]:
   $\llbracket A \sqsubseteq' B; B \sqsubseteq' C \rrbracket \Longrightarrow A \sqsubseteq' C$ 
by(auto simp add: set-less-eq-aux-def dest: set-less-aux-trans)

```

```

lemma set-less-trans-set-less-eq [trans]:
   $\llbracket A \sqsubset' B; B \sqsubseteq' C \rrbracket \Longrightarrow A \sqsubset' C$ 
by(auto simp add: set-less-eq-aux-def dest: set-less-aux-trans)

```

```

lemma set-less-eq-aux-porder: partial-order-on infinite-complement-partition  $\{(A, B). A \sqsubseteq' B\}$ 
by(auto simp add: preorder-on-def partial-order-on-def intro!: refl-onI transI anti-
symI dest: set-less-eq-aux-infinite-complement-partitionD intro: set-less-eq-aux-antisym
set-less-eq-aux-trans del: equalityI)

```

```

lemma psubset-finite-imp-set-less-aux:
  assumes AsB:  $A \subset B$  and B: finite B
  shows  $A \sqsubset' B$ 

```

proof –

```

from AsB B have A: finite A by(auto intro: finite-subset)
moreover from AsB B have Min  $(B - A) \in B - A$  by – (rule Min-in, auto)
ultimately show ?thesis using B AsB
  by(auto simp add: set-less-aux-def intro!: rev-bexI[where  $x = \text{Min } (B - A)$ ]
Min-eqI dest: Min-ge-iff[THEN iffD1, rotated 2])
qed

```

lemma *subset-finite-imp-set-less-eq-aux*:

$\llbracket A \subseteq B; \text{finite } B \rrbracket \implies A \sqsubseteq' B$

by(cases $A = B$)(auto simp add: set-less-eq-aux-def finite-complement-partition intro: psubset-finite-imp-set-less-aux)

lemma *empty-set-less-aux-finite-iff*:

$\text{finite } A \implies \{\} \sqsubseteq' A \longleftrightarrow A \neq \{\}$

by(auto intro: psubset-finite-imp-set-less-aux)

lemma *set-less-aux-finite-total*:

assumes A : *finite* A **and** B : *finite* B

shows $A \sqsubseteq' B \vee A = B \vee B \sqsubseteq' A$

proof(cases $A \subseteq B \vee B \subseteq A$)

case *True* **thus** ?thesis

using A B psubset-finite-imp-set-less-aux[of A B] psubset-finite-imp-set-less-aux[of B A]

by auto

next

case *False*

hence A' : $\neg A \subseteq B$ **and** B' : $\neg B \subseteq A$ **and** AnB : $A \neq B$ **by** auto

thus ?thesis **using** A B

proof(induct $Min(B - A)$ $Min(A - B)$ arbitrary: A B rule: wlog-le)

case (sym m n)

from sym.hyps[OF refl refl] sym.premis **show** ?case **by** blast

next

case (le A B)

note $A = \langle \text{finite } A \rangle$ **and** $B = \langle \text{finite } B \rangle$

and $A' = \langle \neg A \subseteq B \rangle$ **and** $B' = \langle \neg B \subseteq A \rangle$

{ **fix** z

assume z : $z \in (A - B) \cup (B - A)$

hence $Min(B - A) \leq z \wedge (z \leq Min(A - B)) \longrightarrow Min(B - A) = z$

proof

assume $z \in B - A$

hence $Min(B - A) \leq z$ **by**(simp add: B)

thus ?thesis **by** auto

next

assume $z \in A - B$

hence $Min(A - B) \leq z$ **by**(simp add: A)

with le.hyps **show** ?thesis **by**(auto)

qed }

moreover **have** $Min(B - A) \in B - A$ **by**(rule *Min-in*)(simp-all add: B B')

ultimately **have** $A \sqsubseteq' B$ **using** A B **by**(auto simp add: set-less-aux-def)

thus ?case ..

qed

qed

lemma *set-less-eq-aux-finite-total*:

$\llbracket \text{finite } A; \text{finite } B \rrbracket \implies A \sqsubseteq' B \vee A = B \vee B \sqsubseteq' A$

by(*drule* (1) *set-less-aux-finite-total*)(*auto simp add: set-less-eq-aux-def*)

lemma *set-less-eq-aux-finite-total2*:

$\llbracket \text{finite } A; \text{finite } B \rrbracket \implies A \sqsubseteq' B \vee B \sqsubseteq' A$

by(*drule* (1) *set-less-eq-aux-finite-total*)(*auto simp add: finite-complement-partition*)

lemma *set-less-aux-rec*:

assumes *A*: *finite A* **and** *B*: *finite B*

and *A'*: $A \neq \{\}$ **and** *B'*: $B \neq \{\}$

shows $A \sqsubseteq' B \longleftrightarrow \text{Min } B < \text{Min } A \vee \text{Min } A = \text{Min } B \wedge A - \{\text{Min } A\} \sqsubseteq' B - \{\text{Min } A\}$

proof(*cases* $\text{Min } A = \text{Min } B$)

case *True*

from *A A' B B'* **have** *insert* (*Min A*) *A* = *A* *insert* (*Min B*) *B* = *B*

by(*auto simp add: ex-in-conv[symmetric]* *exI*)

with *True* **show** *?thesis*

by(*subst* (2) *set-less-aux-insert-same[symmetric, where x=Min A]*) *simp-all*

next

case *False*

have $A \sqsubseteq' B \longleftrightarrow \text{Min } B < \text{Min } A$

proof

assume *AB*: $A \sqsubseteq' B$

with *B A* **obtain** *ab* **where** *ab*: $ab \in B - A$

and *AB*: $\bigwedge x. \llbracket x \in A; x \notin B \rrbracket \implies ab \leq x$

by(*auto simp add: set-less-aux-def*)

{ fix *a* **assume** $a \in A$

hence $\text{Min } B \leq a$ **using** *A A' B B' ab*

by(*cases* $a \in B$)(*auto intro: order-trans[where y=ab] dest: AB*) }

hence $\text{Min } B \leq \text{Min } A$ **using** *A A'* **by** *simp*

with *False* **show** $\text{Min } B < \text{Min } A$ **using** *False* **by** *auto*

next

assume $\text{Min } B < \text{Min } A$

hence $\forall z \in A - B \cup (B - A). \text{Min } B \leq z \wedge (z \leq \text{Min } B \longrightarrow \text{Min } B = z)$

using *A B A' B'* **by**(*auto 4 4 intro: Min-in Min-eqI dest: bspec bspec[where x=Min B]*)

moreover **have** $\text{Min } B \notin A$ **using** $\langle \text{Min } B < \text{Min } A \rangle$ **by** (*metis A Min-le not-less*)

ultimately **show** $A \sqsubseteq' B$ **using** *A B A' B'* **by**(*simp add: set-less-aux-def* *bestI[where x=Min B]*)

qed

thus *?thesis* **using** *False* **by** *simp*

qed

lemma *set-less-eq-aux-rec*:

assumes *finite A* *finite B* $A \neq \{\}$ $B \neq \{\}$

shows $A \sqsubseteq' B \longleftrightarrow \text{Min } B < \text{Min } A \vee \text{Min } A = \text{Min } B \wedge A - \{\text{Min } A\} \sqsubseteq' B - \{\text{Min } A\}$

proof(*cases* $A = B$)

case *True* **thus** *?thesis* **using** *assms* **by**(*simp add: finite-complement-partition*)

```

next
  case False
  moreover
  hence  $Min\ A = Min\ B \implies A - \{Min\ A\} \neq B - \{Min\ B\}$ 
    by (metis (lifting) assms Min-in insert-Diff)
  ultimately show ?thesis using set-less-aux-rec[OF assms]
    by(simp add: set-less-eq-aux-def cong: conj-cong)
qed

```

```

lemma set-less-aux-Min-antimono:
   $\llbracket Min\ A < Min\ B; finite\ A; finite\ B; A \neq \{\} \rrbracket \implies B \sqsubset' A$ 
using set-less-aux-rec[of B A]
by(cases B = \{\})(simp-all add: empty-set-less-aux-finite-iff)

```

```

lemma sorted-Cons-Min:  $sorted\ (x \# xs) \implies Min\ (insert\ x\ (set\ xs)) = x$ 
by(auto simp add: intro: Min-eqI)

```

```

lemma set-less-aux-code:
   $\llbracket sorted\ xs; distinct\ xs; sorted\ ys; distinct\ ys \rrbracket$ 
 $\implies set\ xs \sqsubset' set\ ys \longleftrightarrow ord.lexordp\ (>) xs\ ys$ 
  apply(induct xs ys rule: list-induct2')
  apply(simp-all add: empty-set-less-aux-finite-iff sorted-Cons-Min set-less-aux-rec
neq-Nil-conv)
  apply(auto cong: conj-cong)
  done

```

```

lemma set-less-eq-aux-code:
  assumes sorted xs distinct xs sorted ys distinct ys
  shows  $set\ xs \sqsubseteq' set\ ys \longleftrightarrow ord.lexordp-eq\ (>) xs\ ys$ 
  proof -
    have dual: class.linorder  $(\geq) (>)$ 
      by(rule linorder.dual-linorder) unfold-locales
    from assms show ?thesis
    by(auto simp add: set-less-eq-aux-def finite-complement-partition linorder.lexordp-eq-conv-lexord[OF
dual] set-less-aux-code intro: sorted-distinct-set-unique)
  qed

```

end

Extending (\sqsubseteq') to have $\{\}$ as least element

context *ord* begin

```

definition set-less-eq-aux' :: 'a set  $\Rightarrow$  'a set  $\Rightarrow$  bool (infix  $\langle \sqsubseteq'' \rangle$  50)
where  $A \sqsubseteq'' B \longleftrightarrow A \sqsubseteq' B \vee A = \{\} \wedge B \in infinite-complement-partition$ 

```

```

lemma set-less-eq-aux'-refl:
   $A \sqsubseteq'' A \longleftrightarrow A \in infinite-complement-partition$ 
by(auto simp add: set-less-eq-aux'-def)

```

lemma *set-less-eq-aux'-antisym*: $\llbracket A \sqsubseteq'' B; B \sqsubseteq'' A \rrbracket \implies A = B$
by(*auto simp add: set-less-eq-aux'-def intro: set-less-eq-aux-antisym del: equalityI*)

lemma *set-less-eq-aux'-infinite-complement-partitionD*:

$A \sqsubseteq'' B \implies A \in \text{infinite-complement-partition} \wedge B \in \text{infinite-complement-partition}$

by(*auto simp add: set-less-eq-aux'-def intro: finite-complement-partition dest: set-less-eq-aux-infinite-complement-partitionD*)

lemma *empty-set-less-eq-def [simp]*: $\{\} \sqsubseteq'' B \longleftrightarrow B \in \text{infinite-complement-partition}$

by(*auto simp add: set-less-eq-aux'-def dest: set-less-eq-aux-infinite-complement-partitionD*)

end

context *linorder* **begin**

lemma *set-less-eq-aux'-trans*: $\llbracket A \sqsubseteq'' B; B \sqsubseteq'' C \rrbracket \implies A \sqsubseteq'' C$

by(*auto simp add: set-less-eq-aux'-def del: equalityI intro: set-less-eq-aux-trans dest: set-less-eq-aux-infinite-complement-partitionD*)

lemma *set-less-eq-aux'-porder*: *partial-order-on infinite-complement-partition* $\{(A, B). A \sqsubseteq'' B\}$

by(*auto simp add: partial-order-on-def preorder-on-def intro!: refl-onI transI anti-symI dest: set-less-eq-aux'-antisym set-less-eq-aux'-infinite-complement-partitionD simp add: set-less-eq-aux'-refl intro: set-less-eq-aux'-trans*)

end

Extend (\sqsubseteq'') **to a total order on** *infinite-complement-partition*

context *ord* **begin**

definition *set-less-eq-aux''* :: $'a \text{ set} \Rightarrow 'a \text{ set} \Rightarrow \text{bool}$ (**infix** $\langle \sqsubseteq'''' \rangle$ 50)

where *set-less-eq-aux''* =

(*SOME* *sleg*.

$(\text{linear-order-on UNIV } \{(a, b). a \leq b\} \longrightarrow \text{linear-order-on infinite-complement-partition } \{(A, B). \text{sleg } A \ B\}) \wedge \text{order-consistent } \{(A, B). A \sqsubseteq'' B\} \{(A, B). \text{sleg } A \ B\})$)

lemma *set-less-eq-aux''-spec*:

shows *linear-order* $\{(a, b). a \leq b\} \implies \text{linear-order-on infinite-complement-partition } \{(A, B). A \sqsubseteq'' B\}$

(**is** *PROP* *?thesis1*)

and *order-consistent* $\{(A, B). A \sqsubseteq'' B\} \{(A, B). A \sqsubseteq'' B\}$ (**is** *?thesis2*)

proof –

let $?P = \lambda \text{sleg}. (\text{linear-order } \{(a, b). a \leq b\} \longrightarrow \text{linear-order-on infinite-complement-partition } \{(A, B). \text{sleg } A \ B\}) \wedge$

$\text{order-consistent } \{(A, B). A \sqsubseteq'' B\} \{(A, B). \text{sleg } A \ B\}$

have $Ex ?P$

proof(*cases linear-order* $\{(a, b). a \leq b\}$)

case *False*

```

have antisym  $\{(a, b). a \sqsubseteq'' b\}$ 
  by (rule antisymI) (simp add: set-less-eq-aux'-antisym)
then show ?thesis using False
  by (auto intro: antisym-order-consistent-self)
next
case True
hence partial-order-on infinite-complement-partition  $\{(A, B). A \sqsubseteq'' B\}$ 
  by (rule linorder.set-less-eq-aux'-porder[OF linear-order-imp-linorder])
then obtain s where linear-order-on infinite-complement-partition s
  and order-consistent  $\{(A, B). A \sqsubseteq'' B\}$  s by (rule porder-extend-to-linorder)
thus ?thesis by (auto intro: exI[where  $x = \lambda A B. (A, B) \in s$ ])
qed
hence ?P (Eps ?P) by (rule someI-ex)
thus PROP ?thesis1 ?thesis2 by (simp-all add: set-less-eq-aux''-def)
qed

end

context linorder begin

lemma set-less-eq-aux''-linear-order:
  linear-order-on infinite-complement-partition  $\{(A, B). A \sqsubseteq''' B\}$ 
by (rule set-less-eq-aux''-spec)(rule linear-order)

lemma set-less-eq-aux''-refl [iff]:  $A \sqsubseteq''' A \longleftrightarrow A \in \text{infinite-complement-partition}$ 
using set-less-eq-aux''-linear-order
by (auto simp add: linear-order-on-def partial-order-on-def preorder-on-def dest: refl-onD)

lemma set-less-eq-aux'-into-set-less-eq-aux'':
  assumes  $A \sqsubseteq'' B$ 
  shows  $A \sqsubseteq''' B$ 
proof (rule ccontr)
  assume nleq:  $\neg ?thesis$ 
  moreover from assms have A:  $A \in \text{infinite-complement-partition}$ 
    and B:  $B \in \text{infinite-complement-partition}$ 
  by (auto dest: set-less-eq-aux'-infinite-complement-partitionD)
  with set-less-eq-aux''-linear-order have  $A \sqsubseteq''' B \vee A = B \vee B \sqsubseteq''' A$ 
  by (auto simp add: linear-order-on-def dest: total-onD)
  ultimately have  $B \sqsubseteq''' A$  using B by auto
  with assms have  $A = B$  using set-less-eq-aux''-spec(2)
  by (simp add: order-consistent-def)
  with A nleq show False by simp
qed

lemma finite-set-less-eq-aux''-finite:
  assumes finite A and finite B
  shows  $A \sqsubseteq''' B \longleftrightarrow A \sqsubseteq'' B$ 
proof

```

```

assume  $A \sqsubseteq''' B$ 
from assms have  $A \sqsubseteq' B \vee B \sqsubseteq' A$  by(rule set-less-eq-aux-finite-total2)
hence  $A \sqsubseteq'' B \vee B \sqsubseteq'' A$  by(auto simp add: set-less-eq-aux'-def)
thus  $A \sqsubseteq'' B$ 
proof
  assume  $B \sqsubseteq'' A$ 
  hence  $B \sqsubseteq''' A$  by(rule set-less-eq-aux'-into-set-less-eq-aux'')
  with  $\langle A \sqsubseteq''' B \rangle$  set-less-eq-aux''-linear-order have  $A = B$ 
  by(auto simp add: linear-order-on-def partial-order-on-def dest: antisymD)
  thus ?thesis using assms by(simp add: finite-complement-partition set-less-eq-aux'-def)
qed
qed(rule set-less-eq-aux'-into-set-less-eq-aux'')

```

lemma *set-less-eq-aux''-finite:*

```

  finite (UNIV :: 'a set)  $\implies$  set-less-eq-aux'' = set-less-eq-aux
by(auto simp add: fun-eq-iff finite-set-less-eq-aux''-finite set-less-eq-aux'-def finite-subset[OF subset-UNIV])

```

lemma *set-less-eq-aux''-antisym:*

```

   $\llbracket A \sqsubseteq''' B; B \sqsubseteq''' A;
    A \in \text{infinite-complement-partition}; B \in \text{infinite-complement-partition} \rrbracket
    \implies A = B$ 

```

using *set-less-eq-aux''-linear-order*

by(*auto simp add: linear-order-on-def partial-order-on-def dest: antisymD del: equalityI*)

lemma *set-less-eq-aux''-trans:* $\llbracket A \sqsubseteq''' B; B \sqsubseteq''' C \rrbracket \implies A \sqsubseteq''' C$

using *set-less-eq-aux''-linear-order*

by(*auto simp add: linear-order-on-def partial-order-on-def preorder-on-def dest: transD*)

lemma *set-less-eq-aux''-total:*

```

   $\llbracket A \in \text{infinite-complement-partition}; B \in \text{infinite-complement-partition} \rrbracket
    \implies A \sqsubseteq''' B \vee B \sqsubseteq''' A$ 

```

using *set-less-eq-aux''-linear-order*

by(*auto simp add: linear-order-on-def dest: total-onD*)

end

Extend (\sqsubseteq''') **to cofinite sets**

context *ord* **begin**

definition *set-less-eq* :: '*a set* \implies '*a set* \implies bool (**infix** $\langle \sqsubseteq \rangle$ 50)

where

$A \sqsubseteq B \iff$

(*if* $A \in \text{infinite-complement-partition}$ *then* $A \sqsubseteq''' B \vee B \notin \text{infinite-complement-partition}$
else $B \notin \text{infinite-complement-partition} \wedge \neg B \sqsubseteq''' \neg A$)

definition *set-less* :: 'a set \Rightarrow 'a set \Rightarrow bool (**infix** \sqsubset 50)
where $A \sqsubset B \longleftrightarrow A \sqsubseteq B \wedge \neg B \sqsubseteq A$

lemma *set-less-eq-def2*:

$A \sqsubseteq B \longleftrightarrow$

(if finite (UNIV :: 'a set) then $A \sqsubseteq''' B$

else if $A \in$ infinite-complement-partition then $A \sqsubseteq''' B \vee B \notin$ infinite-complement-partition

else $B \notin$ infinite-complement-partition $\wedge \neg B \sqsubseteq''' \neg A$)

by(simp add: set-less-eq-def)

end

context *linorder* **begin**

lemma *set-less-eq-refl* [*iff*]: $A \sqsubseteq A$

by(auto simp add: set-less-eq-def2 not-in-complement-partition)

lemma *set-less-eq-antisym*: $\llbracket A \sqsubseteq B; B \sqsubseteq A \rrbracket \Longrightarrow A = B$

by(auto simp add: set-less-eq-def2 set-less-eq-aux''-finite not-in-complement-partition

not-in-complement-partition-False del: equalityI split: if-split-asm dest: set-less-eq-aux-antisym

set-less-eq-aux''-antisym)

lemma *set-less-eq-trans*: $\llbracket A \sqsubseteq B; B \sqsubseteq C \rrbracket \Longrightarrow A \sqsubseteq C$

by(auto simp add: set-less-eq-def split: if-split-asm intro: set-less-eq-aux''-trans)

lemma *set-less-eq-total*: $A \sqsubseteq B \vee B \sqsubseteq A$

by(auto simp add: set-less-eq-def2 set-less-eq-aux''-finite not-in-complement-partition

not-in-complement-partition-False intro: set-less-eq-aux-finite-total2 finite-subset[OF

subset-UNIV] del: disjCI dest: set-less-eq-aux''-total)

lemma *set-less-eq-linorder*: class.linorder (\sqsubseteq) (\sqsubset)

by(unfold-locales)(auto simp add: set-less-def set-less-eq-antisym set-less-eq-total

intro: set-less-eq-trans)

lemma *set-less-eq-conv-set-less*: $set-less-eq A B \longleftrightarrow A = B \vee set-less A B$

by(auto simp add: set-less-def del: equalityI dest: set-less-eq-antisym)

lemma *Compl-set-less-eq-Compl*: $\neg A \sqsubseteq \neg B \longleftrightarrow B \sqsubseteq A$

by(auto simp add: set-less-eq-def2 not-in-complement-partition-False not-in-complement-partition

set-less-eq-aux''-finite Compl-set-less-eq-aux-Compl)

lemma *Compl-set-less-Compl*: $\neg A \sqsubset \neg B \longleftrightarrow B \sqsubset A$

by(simp add: set-less-def Compl-set-less-eq-Compl)

lemma *set-less-eq-finite-iff*: $\llbracket finite A; finite B \rrbracket \Longrightarrow A \sqsubseteq B \longleftrightarrow A \sqsubseteq' B$

by(auto simp add: set-less-eq-def finite-complement-partition set-less-eq-aux'-def fi-

nite-set-less-eq-aux''-finite)

lemma *set-less-finite-iff*: $\llbracket finite A; finite B \rrbracket \Longrightarrow A \sqsubset B \longleftrightarrow A \sqsubset' B$

by(*simp add: set-less-def set-less-aux-conv-set-less-eq-aux set-less-eq-finite-iff*)

lemma *infinite-set-less-eq-Complement*:

$\llbracket \text{finite } A; \text{finite } B; \neg \text{finite } (UNIV :: 'a \text{ set}) \rrbracket \implies A \sqsubseteq - B$

by(*simp add: set-less-eq-def finite-complement-partition not-in-complement-partition*)

lemma *infinite-set-less-Complement*:

$\llbracket \text{finite } A; \text{finite } B; \neg \text{finite } (UNIV :: 'a \text{ set}) \rrbracket \implies A \sqsubset - B$

by(*auto simp add: set-less-def dest: set-less-eq-antisym intro: infinite-set-less-eq-Complement*)

lemma *infinite-Complement-set-less-eq*:

$\llbracket \text{finite } A; \text{finite } B; \neg \text{finite } (UNIV :: 'a \text{ set}) \rrbracket \implies \neg - A \sqsubseteq B$

using *infinite-set-less-eq-Complement*[of $A B$] *Compl-set-less-eq-Compl*[of $A - B$]

by(*auto dest: set-less-eq-antisym*)

lemma *infinite-Complement-set-less*:

$\llbracket \text{finite } A; \text{finite } B; \neg \text{finite } (UNIV :: 'a \text{ set}) \rrbracket \implies \neg - A \sqsubset B$

using *infinite-Complement-set-less-eq*[of $A B$]

by(*simp add: set-less-def*)

lemma *empty-set-less-eq [iff]*: $\{\} \sqsubseteq A$

by(*auto simp add: set-less-eq-def finite-complement-partition intro: set-less-eq-aux'-into-set-less-eq-aux''*)

lemma *set-less-eq-empty [iff]*: $A \sqsubseteq \{\} \longleftrightarrow A = \{\}$

by (*metis empty-set-less-eq set-less-eq-antisym*)

lemma *empty-set-less-iff [iff]*: $\{\} \sqsubset A \longleftrightarrow A \neq \{\}$

by(*simp add: set-less-def*)

lemma *not-set-less-empty [simp]*: $\neg A \sqsubset \{\}$

by(*simp add: set-less-def*)

lemma *set-less-eq-UNIV [iff]*: $A \sqsubseteq UNIV$

using *Compl-set-less-eq-Compl*[of $- A \ \{\}$] by *simp*

lemma *UNIV-set-less-eq [iff]*: $UNIV \sqsubseteq A \longleftrightarrow A = UNIV$

using *Compl-set-less-eq-Compl*[of $\{\} \ - A$]

by(*simp add: Compl-eq-empty-iff*)

lemma *set-less-UNIV-iff [iff]*: $A \sqsubset UNIV \longleftrightarrow A \neq UNIV$

by(*simp add: set-less-def*)

lemma *not-UNIV-set-less [simp]*: $\neg UNIV \sqsubset A$

by(*simp add: set-less-def*)

end

2.6.2 Implementation based on sorted lists

type-synonym 'a proper-interval = 'a option \Rightarrow 'a option \Rightarrow bool

class proper-introl = ord +
fixes proper-interval :: 'a proper-interval

class proper-interval = proper-introl +
assumes proper-interval-simps:
 proper-interval None None = True
 proper-interval None (Some y) = ($\exists z. z < y$)
 proper-interval (Some x) None = ($\exists z. x < z$)
 proper-interval (Some x) (Some y) = ($\exists z. x < z \wedge z < y$)

context proper-introl **begin**

function set-less-eq-aux-Compl :: 'a option \Rightarrow 'a list \Rightarrow 'a list \Rightarrow bool

where

set-less-eq-aux-Compl ao [] ys \longleftrightarrow True
 | set-less-eq-aux-Compl ao xs [] \longleftrightarrow True
 | set-less-eq-aux-Compl ao (x # xs) (y # ys) \longleftrightarrow
 (if x < y then proper-interval ao (Some x) \vee set-less-eq-aux-Compl (Some x) xs
 (y # ys)
 else if y < x then proper-interval ao (Some y) \vee set-less-eq-aux-Compl (Some y)
 (x # xs) ys
 else proper-interval ao (Some y))

by(pat-completeness) simp-all

termination by(lexicographic-order)

fun Compl-set-less-eq-aux :: 'a option \Rightarrow 'a list \Rightarrow 'a list \Rightarrow bool

where

Compl-set-less-eq-aux ao [] [] \longleftrightarrow \neg proper-interval ao None
 | Compl-set-less-eq-aux ao [] (y # ys) \longleftrightarrow \neg proper-interval ao (Some y) \wedge Compl-set-less-eq-aux
 (Some y) [] ys
 | Compl-set-less-eq-aux ao (x # xs) [] \longleftrightarrow \neg proper-interval ao (Some x) \wedge Compl-set-less-eq-aux
 (Some x) xs []
 | Compl-set-less-eq-aux ao (x # xs) (y # ys) \longleftrightarrow
 (if x < y then \neg proper-interval ao (Some x) \wedge Compl-set-less-eq-aux (Some x)
 xs (y # ys)
 else if y < x then \neg proper-interval ao (Some y) \wedge Compl-set-less-eq-aux (Some
 y) (x # xs) ys
 else \neg proper-interval ao (Some y))

fun set-less-aux-Compl :: 'a option \Rightarrow 'a list \Rightarrow 'a list \Rightarrow bool **where**

set-less-aux-Compl ao [] [] \longleftrightarrow proper-interval ao None
 | set-less-aux-Compl ao [] (y # ys) \longleftrightarrow proper-interval ao (Some y) \vee set-less-aux-Compl
 (Some y) [] ys
 | set-less-aux-Compl ao (x # xs) [] \longleftrightarrow proper-interval ao (Some x) \vee set-less-aux-Compl
 (Some x) xs []
 | set-less-aux-Compl ao (x # xs) (y # ys) \longleftrightarrow

```

    (if  $x < y$  then proper-interval ao (Some  $x$ )  $\vee$  set-less-aux-Compl (Some  $x$ )  $xs$  ( $y$ 
#  $ys$ )
    else if  $y < x$  then proper-interval ao (Some  $y$ )  $\vee$  set-less-aux-Compl (Some  $y$ ) ( $x$ 
#  $xs$ )  $ys$ 
    else proper-interval ao (Some  $y$ ))

```

```

function Compl-set-less-aux :: 'a option  $\Rightarrow$  'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool where
  Compl-set-less-aux ao []  $ys$   $\longleftrightarrow$  False
| Compl-set-less-aux ao  $xs$  []  $\longleftrightarrow$  False
| Compl-set-less-aux ao ( $x$  #  $xs$ ) ( $y$  #  $ys$ )  $\longleftrightarrow$ 
  (if  $x < y$  then  $\neg$  proper-interval ao (Some  $x$ )  $\wedge$  Compl-set-less-aux (Some  $x$ )  $xs$ 
( $y$  #  $ys$ )
  else if  $y < x$  then  $\neg$  proper-interval ao (Some  $y$ )  $\wedge$  Compl-set-less-aux (Some  $y$ )
( $x$  #  $xs$ )  $ys$ 
  else  $\neg$  proper-interval ao (Some  $y$ ))
by pat-completeness simp-all
termination by lexicographic-order

```

end

```

lemmas [code] =
  proper-intrvl.set-less-eq-aux-Compl.simps
  proper-intrvl.set-less-aux-Compl.simps
  proper-intrvl.Compl-set-less-eq-aux.simps
  proper-intrvl.Compl-set-less-aux.simps

```

```

class linorder-proper-interval = linorder + proper-interval
begin

```

```

theorem assumes fin: finite (UNIV :: 'a set)
  and  $xs$ : sorted  $xs$    distinct  $xs$ 
  and  $ys$ : sorted  $ys$    distinct  $ys$ 
  shows set-less-eq-aux-Compl2-conv-set-less-eq-aux-Compl:
     $set\ xs \sqsubseteq' set\ ys \longleftrightarrow set-less-eq-aux-Compl\ None\ xs\ ys$  (is ?Compl2)
  and Compl1-set-less-eq-aux-conv-Compl-set-less-eq-aux:
     $set\ xs \sqsubseteq' set\ ys \longleftrightarrow Compl-set-less-eq-aux\ None\ xs\ ys$  (is ?Compl1)
proof -
  note  $fin'$  [simp] = finite-subset[OF subset-UNIV fin]

```

```

  define above where above = case-option UNIV (Collect o less)
  have above-simps [simp]: above None = UNIV  $\wedge x.$  above (Some  $x$ ) = { $y.$   $x <$ 
 $y$ }
  and above-upclosed:  $\wedge x\ y\ ao.$  [ $x \in above\ ao; x < y$ ]  $\implies y \in above\ ao$ 
  and proper-interval-Some2:  $\wedge x\ ao.$  proper-interval ao (Some  $x$ )  $\longleftrightarrow (\exists z \in above$ 
 $ao.$   $z < x$ )
  and proper-interval-None2:  $\wedge ao.$  proper-interval ao None  $\longleftrightarrow above\ ao \neq \{\}$ 
  by(simp-all add: proper-interval-simps above-def split: option.splits)

```

```

{ fix  $ao\ x\ A\ B$ 

```

```

assume ex: proper-interval ao (Some x)
  and A:  $\forall a \in A. x \leq a$ 
  and B:  $\forall b \in B. x \leq b$ 
from ex obtain z where z-ao:  $z \in \text{above } ao$  and z:  $z < x$ 
  by(auto simp add: proper-interval-Some2)
with A have A-eq:  $A \cap \text{above } ao = A$ 
  by(auto simp add: above-upclosed)
from z-ao z B have B-eq:  $B \cap \text{above } ao = B$ 
  by(auto simp add: above-upclosed)
define w where w = Min (above ao)
with z-ao have  $w \leq z \quad \forall z \in \text{above } ao. w \leq z \quad w \in \text{above } ao$ 
  by(auto simp add: Min-le-iff intro: Min-in)
hence  $A \cap \text{above } ao \sqsubseteq' (-B) \cap \text{above } ao$  (is ?lhs  $\sqsubseteq'$  ?rhs)
  using A B z by(auto simp add: set-less-aux-def intro!: bexI[where x=w])
hence  $A \sqsubseteq' ?rhs$  unfolding A-eq by(simp add: set-less-eq-aux-def)
moreover
from z-ao z A B have  $z \in -A \cap \text{above } ao \quad z \notin B$  by auto
hence neg:  $-A \cap \text{above } ao \neq B \cap \text{above } ao$  by auto
have  $\neg -A \cap \text{above } ao \sqsubseteq' B \cap \text{above } ao$  (is  $\neg ?lhs' \sqsubseteq' ?rhs'$ )
  using A B z z-ao by(force simp add: set-less-aux-def not-less dest: bspec[where
x=z])
with neg have  $\neg ?lhs' \sqsubseteq' B$  unfolding B-eq by(auto simp add: set-less-eq-aux-def)
  moreover note calculation }
note proper-interval-set-less-eqI = this(1)
  and proper-interval-not-set-less-eq-auxI = this(2)

{ fix ao
  assume set xs  $\cup$  set ys  $\subseteq$  above ao
  with xs ys
  have set-less-eq-aux-Compl ao xs ys  $\longleftrightarrow$  set xs  $\sqsubseteq' (- \text{set } ys) \cap \text{above } ao$ 
  proof(induction ao xs ys rule: set-less-eq-aux-Compl.induct)
    case 1 thus ?case by simp
  next
    case 2 thus ?case by(auto intro: subset-finite-imp-set-less-eq-aux)
  next
    case ( $\exists ao$  x xs y ys)
    note ao =  $\langle \text{set } (x \# xs) \cup \text{set } (y \# ys) \subseteq \text{above } ao \rangle$ 
    hence x-ao:  $x \in \text{above } ao$  and y-ao:  $y \in \text{above } ao$  by simp-all
    note yys =  $\langle \text{sorted } (y \# ys) \rangle \langle \text{distinct } (y \# ys) \rangle$ 
    hence ys: sorted ys distinct ys and y-Min:  $\forall y' \in \text{set } ys. y < y'$ 
      by(auto simp add: less-le)
    note xxs =  $\langle \text{sorted } (x \# xs) \rangle \langle \text{distinct } (x \# xs) \rangle$ 
    hence xs: sorted xs distinct xs and x-Min:  $\forall x' \in \text{set } xs. x < x'$ 
      by(auto simp add: less-le)
    let ?lhs = set  $(x \# xs)$  and ?rhs =  $- \text{set } (y \# ys) \cap \text{above } ao$ 
    show ?case
    proof(cases  $x < y$ )
      case True
      show ?thesis

```

```

proof(cases proper-interval ao (Some x))
  case True
    hence ?lhs  $\sqsubseteq'$  ?rhs using x-Min y-Min  $\langle x < y \rangle$ 
      by(auto intro!: proper-interval-set-less-eqI)
    with True show ?thesis using  $\langle x < y \rangle$  by simp
  next
    case False
      have set xs  $\cup$  set (y # ys)  $\subseteq$  above (Some x) using True x-Min y-Min
by auto
    with True xs yys
      have IH: set-less-eq-aux-Compl (Some x) xs (y # ys) =
        (set xs  $\sqsubseteq'$  - set (y # ys)  $\cap$  above (Some x))
        by(rule 3.IH)
      from True y-Min x-ao have x  $\in$  - set (y # ys)  $\cap$  above ao by auto
      hence ?rhs  $\neq$  {} by blast
      moreover have Min ?lhs = x using x-Min x-ao by(auto intro!: Min-eqI)
      moreover have Min ?rhs = x using  $\langle x < y \rangle$  y-Min x-ao False
        by(auto intro!: Min-eqI simp add: proper-interval-Some2)
      moreover have set xs = set xs - {x}
        using ao x-Min by auto
      moreover have - set (y # ys)  $\cap$  above (Some x) = - set (y # ys)  $\cap$ 
        above ao - {x}
        using False x-ao by(auto simp add: proper-interval-Some2 intro:
        above-upclosed)
      ultimately show ?thesis using True False IH
        by(simp del: set-simps)(subst (2) set-less-eq-aux-rec, simp-all add: x-ao)
    qed
  next
    case False
      show ?thesis
      proof(cases y < x)
        case True
          show ?thesis
          proof(cases proper-interval ao (Some y))
            case True
              hence ?lhs  $\sqsubseteq'$  ?rhs using x-Min y-Min  $\langle \neg x < y \rangle$ 
                by(auto intro!: proper-interval-set-less-eqI)
              with True show ?thesis using  $\langle \neg x < y \rangle$  by simp
            next
              case False
                have set (x # xs)  $\cup$  set ys  $\subseteq$  above (Some y)
                  using  $\langle y < x \rangle$  x-Min y-Min by auto
                with  $\langle \neg x < y \rangle$   $\langle y < x \rangle$  xxs ys
                  have IH: set-less-eq-aux-Compl (Some y) (x # xs) ys =
                    (set (x # xs)  $\sqsubseteq'$  - set ys  $\cap$  above (Some y))
                    by(rule 3.IH)
                moreover have - set ys  $\cap$  above (Some y) = ?rhs
                using y-ao False by(auto intro: above-upclosed simp add: proper-interval-Some2)
                ultimately show ?thesis using  $\langle \neg x < y \rangle$  True False by simp
          
```

```

    qed
  next
  case False with  $\langle \neg x < y \rangle$  have  $x = y$  by auto
  { assume proper-interval ao (Some y)
    hence  $?lhs \sqsubseteq' ?rhs$  using x-Min y-Min  $\langle x = y \rangle$ 
      by(auto intro!: proper-interval-set-less-eqI) }
  moreover
  { assume  $?lhs \sqsubseteq' ?rhs$ 
    moreover have  $?lhs \neq ?rhs$ 
    proof
      assume eq:  $?lhs = ?rhs$ 
      have  $x \in ?lhs$  using x-ao by simp
      also note eq also note  $\langle x = y \rangle$ 
      finally show False by simp
    qed
    ultimately obtain z where  $z \in \text{above } ao \quad z < y$  using  $\langle x = y \rangle$  y-ao
      by(fastforce simp add: set-less-eq-aux-def set-less-aux-def not-le dest!:
bspec[where  $x=y$ ])
    hence proper-interval ao (Some y) by(auto simp add: proper-interval-Some2)
  }
  ultimately show ?thesis using  $\langle x = y \rangle \langle \neg x < y \rangle \langle \neg y < x \rangle$  by auto
  qed
  qed
  qed }
  from this[of None] show ?Compl2 by simp

{ fix ao
  assume set xs  $\cup$  set ys  $\subseteq$  above ao
  with xs ys
  have Compl-set-less-eq-aux ao xs ys  $\longleftrightarrow (\neg \text{set } xs) \cap \text{above } ao \sqsubseteq' \text{set } ys$ 
  proof(induction ao xs ys rule: Compl-set-less-eq-aux.induct)
    case 1 thus ?case by(simp add: proper-interval-None2)
  next
  case (2 ao y ys)
  from  $\langle \text{sorted } (y \# ys) \rangle \langle \text{distinct } (y \# ys) \rangle$ 
  have ys: sorted ys distinct ys and y-Min:  $\forall y' \in \text{set } ys. y < y'$ 
    by(auto simp add: less-le)
  show ?case
  proof(cases proper-interval ao (Some y))
    case True
    hence  $\neg - \text{set } [] \cap \text{above } ao \sqsubseteq' \text{set } (y \# ys)$  using y-Min
      by  $-(\text{erule } \text{proper-interval-not-set-less-eq-auxI}, \text{auto})$ 
    thus ?thesis using True by simp
  next
  case False
  note ao =  $\langle \text{set } [] \cup \text{set } (y \# ys) \subseteq \text{above } ao \rangle$ 
  hence y-ao:  $y \in \text{above } ao$  by simp
  from ao y-Min have set  $[] \cup \text{set } ys \subseteq \text{above } (\text{Some } y)$  by auto
  with  $\langle \text{sorted } [] \rangle \langle \text{distinct } [] \rangle$  ys

```

```

    have Compl-set-less-eq-aux (Some y) [] ys  $\longleftrightarrow$   $\neg$  set []  $\cap$  above (Some y)
 $\sqsubseteq'$  set ys
    by(rule 2.IH)
    moreover have above ao  $\neq$  {} using y-ao by auto
    moreover have Min (above ao) = y
    and Min (set (y # ys)) = y
    using y-ao False ao by(auto intro!: Min-eqI simp add: proper-interval-Some2
not-less)
    moreover have above ao - {y} = above (Some y) using False y-ao
    by(auto simp add: proper-interval-Some2 intro: above-upclosed)
    moreover have set ys - {y} = set ys
    using y-Min y-ao by(auto)
    ultimately show ?thesis using False y-ao
    by(simp)(subst (2) set-less-eq-aux-rec, simp-all)
qed
next
case (3 ao x xs)
from <sorted (x # xs)> <distinct (x # xs)>
have xs: sorted xs distinct xs and x-Min:  $\forall x' \in \text{set } xs. x < x'$ 
by(auto simp add: less-le)
show ?case
proof(cases proper-interval ao (Some x))
case True
then obtain z where z  $\in$  above ao z < x by(auto simp add: proper-interval-Some2)
hence z  $\in$   $\neg$  set (x # xs)  $\cap$  above ao using x-Min by auto
thus ?thesis using True by auto
next
case False
note ao = <set (x # xs)  $\cup$  set []  $\subseteq$  above ao>
hence x-ao: x  $\in$  above ao by simp
from ao have set xs  $\cup$  set []  $\subseteq$  above (Some x) using x-Min by auto
with xs <sorted []> <distinct []>
have Compl-set-less-eq-aux (Some x) xs []  $\longleftrightarrow$ 
 $\neg$  set xs  $\cap$  above (Some x)  $\sqsubseteq'$  set []
by(rule 3.IH)
moreover have  $\neg$  set (x # xs)  $\cap$  above ao =  $\neg$  set xs  $\cap$  above (Some x)
using False x-ao by(auto simp add: proper-interval-Some2 intro: above-upclosed)
ultimately show ?thesis using False by simp
qed
next
case (4 ao x xs y ys)
note ao = <set (x # xs)  $\cup$  set (y # ys)  $\subseteq$  above ao>
hence x-ao: x  $\in$  above ao and y-ao: y  $\in$  above ao by simp-all
note xxs = <sorted (x # xs)> <distinct (x # xs)>
hence xs: sorted xs distinct xs and x-Min:  $\forall x' \in \text{set } xs. x < x'$ 
by(auto simp add: less-le)
note yys = <sorted (y # ys)> <distinct (y # ys)>
hence ys: sorted ys distinct ys and y-Min:  $\forall y' \in \text{set } ys. y < y'$ 
by(auto simp add: less-le)

```

```

let ?lhs = - set (x # xs) ∩ above ao and ?rhs = set (y # ys)
show ?case
proof(cases x < y)
  case True
  show ?thesis
  proof(cases proper-interval ao (Some x))
    case True
    hence ¬ ?lhs ⊆' ?rhs using x-Min y-Min ⟨x < y⟩
      by -(erule proper-interval-not-set-less-eq-auxI, auto)
    thus ?thesis using True ⟨x < y⟩ by simp
  next
  case False
  have set xs ∪ set (y # ys) ⊆ above (Some x)
    using ao x-Min y-Min True by auto
  with True xs yys
  have Compl-set-less-eq-aux (Some x) xs (y # ys) ⟷
    - set xs ∩ above (Some x) ⊆' set (y # ys)
    by(rule 4.IH)
  moreover have - set xs ∩ above (Some x) = ?lhs
  using x-ao False by(auto intro: above-upclosed simp add: proper-interval-Some2)
  ultimately show ?thesis using False True by simp
qed
next
case False
show ?thesis
proof(cases y < x)
  case True
  show ?thesis
  proof(cases proper-interval ao (Some y))
    case True
    hence ¬ ?lhs ⊆' ?rhs using x-Min y-Min ⟨y < x⟩
      by -(erule proper-interval-not-set-less-eq-auxI, auto)
    thus ?thesis using True ⟨y < x⟩ ⟨¬ x < y⟩ by simp
  next
  case False
  from ao True x-Min y-Min
  have set (x # xs) ∪ set ys ⊆ above (Some y) by auto
  with ⟨¬ x < y⟩ True xxs ys
  have Compl-set-less-eq-aux (Some y) (x # xs) ys ⟷
    - set (x # xs) ∩ above (Some y) ⊆' set ys
    by(rule 4.IH)
  moreover have y ∈ ?lhs using True x-Min y-ao by auto
  hence ?lhs ≠ {} by auto
  moreover have Min ?lhs = y using True False x-Min y-ao
    by(auto intro!: Min-eqI simp add: not-le not-less proper-interval-Some2)
  moreover have Min ?rhs = y using y-Min y-ao by(auto intro!: Min-eqI)
  moreover have - set (x # xs) ∩ above (Some y) = ?lhs - {y}
  using y-ao False by(auto intro: above-upclosed simp add: proper-interval-Some2)
  moreover have set ys = set ys - {y}

```

```

      using y-ao y-Min by(auto intro: above-upclosed)
      ultimately show ?thesis using True False ⟨¬ x < y⟩ y-ao
      by(simp)(subst (2) set-less-eq-aux-rec, simp-all)
    qed
  next
  case False
  with ⟨¬ x < y⟩ have x = y by auto
  { assume proper-interval ao (Some y)
    hence ¬ ?lhs ⊆' ?rhs using x-Min y-Min ⟨x = y⟩
      by -(erule proper-interval-not-set-less-eq-auxI, auto) }
  moreover
  { assume ¬ ?lhs ⊆' ?rhs
    also have ?rhs = set (y # ys) ∩ above ao
      using ao by auto
    finally obtain z where z ∈ above ao    z < y
      using ⟨x = y⟩ x-ao x-Min[unfolded Ball-def]
      by(fastforce simp add: set-less-eq-aux-def set-less-aux-def simp add:
less-le not-le dest!: bspec[where x=y])
    hence proper-interval ao (Some y)
      by(auto simp add: proper-interval-Some2) }
  ultimately show ?thesis using ⟨x = y⟩ by auto
  qed
  qed
  qed }
  from this[of None] show ?Compl1 by simp
  qed

```

lemma *set-less-aux-Compl-iff:*

set-less-aux-Compl ao xs ys \longleftrightarrow *set-less-eq-aux-Compl ao xs ys* \wedge \neg *Compl-set-less-eq-aux*
ao ys xs
by(*induct ao xs ys rule: set-less-aux-Compl.induct*)(*auto simp add: not-less-iff-gr-or-eq*)

lemma *Compl-set-less-aux-Compl-iff:*

Compl-set-less-aux ao xs ys \longleftrightarrow *Compl-set-less-eq-aux ao xs ys* \wedge \neg *set-less-eq-aux-Compl*
ao ys xs
by(*induct ao xs ys rule: Compl-set-less-aux.induct*)(*auto simp add: not-less-iff-gr-or-eq*)

theorem *assumes* *fin: finite (UNIV :: 'a set)*

and *xs: sorted xs distinct xs*

and *ys: sorted ys distinct ys*

shows *set-less-aux-Compl2-conv-set-less-aux-Compl:*

set xs \sqsubseteq' *set ys* \longleftrightarrow *set-less-aux-Compl None xs ys* (**is** ?*Compl2*)

and *Compl1-set-less-aux-conv-Compl-set-less-aux:*

\neg *set xs* \sqsubseteq' *set ys* \longleftrightarrow *Compl-set-less-aux None xs ys* (**is** ?*Compl1*)

using *assms*

by(*simp-all only: set-less-aux-conv-set-less-eq-aux set-less-aux-Compl-iff Compl-set-less-aux-Compl-iff*
set-less-eq-aux-Compl2-conv-set-less-eq-aux-Compl Compl1-set-less-eq-aux-conv-Compl-set-less-eq-aux)

end

2.6.3 Implementation of proper intervals for sets

definition *length-last* :: 'a list \Rightarrow nat \times 'a
where *length-last* xs = (length xs, last xs)

lemma *length-last-Nil* [code]:
length-last [] = (0, undefined)
by(simp add: *length-last-def last-def*)

lemma *length-last-Cons-code* [symmetric, code]:
fold ($\lambda x (n, -) . (n + 1, x)$) xs (1, x) = *length-last* (x # xs)
by(*induct* xs rule: *rev-induct*)(simp-all add: *length-last-def*)

context *proper-introl* **begin**

fun *exhaustive-above* :: 'a \Rightarrow 'a list \Rightarrow bool **where**
exhaustive-above x [] \longleftrightarrow \neg *proper-interval* (Some x) None
| *exhaustive-above* x (y # ys) \longleftrightarrow \neg *proper-interval* (Some x) (Some y) \wedge *exhaustive-above* y ys

fun *exhaustive* :: 'a list \Rightarrow bool **where**
exhaustive [] = False
| *exhaustive* (x # xs) \longleftrightarrow \neg *proper-interval* None (Some x) \wedge *exhaustive-above* x xs

fun *proper-interval-set-aux* :: 'a list \Rightarrow 'a list \Rightarrow bool
where
proper-interval-set-aux xs [] \longleftrightarrow False
| *proper-interval-set-aux* [] (y # ys) \longleftrightarrow ys \neq [] \vee *proper-interval* (Some y) None
| *proper-interval-set-aux* (x # xs) (y # ys) \longleftrightarrow
(if x < y then False
else if y < x then *proper-interval* (Some y) (Some x) \vee ys \neq [] \vee \neg *exhaustive-above* x xs
else *proper-interval-set-aux* xs ys)

fun *proper-interval-set-Compl-aux* :: 'a option \Rightarrow nat \Rightarrow 'a list \Rightarrow 'a list \Rightarrow bool
where
proper-interval-set-Compl-aux ao n [] [] \longleftrightarrow
CARD('a) > n + 1
| *proper-interval-set-Compl-aux* ao n [] (y # ys) \longleftrightarrow
(let m = CARD('a) - n; (len-y, y') = *length-last* (y # ys)
in m \neq len-y \wedge (m = len-y + 1 \longrightarrow \neg *proper-interval* (Some y') None))
| *proper-interval-set-Compl-aux* ao n (x # xs) [] \longleftrightarrow
(let m = CARD('a) - n; (len-x, x') = *length-last* (x # xs)
in m \neq len-x \wedge (m = len-x + 1 \longrightarrow \neg *proper-interval* (Some x') None))
| *proper-interval-set-Compl-aux* ao n (x # xs) (y # ys) \longleftrightarrow
(if x < y then
proper-interval ao (Some x) \vee
proper-interval-set-Compl-aux (Some x) (n + 1) xs (y # ys)
else if y < x then

```

    proper-interval ao (Some y) ∨
    proper-interval-set-Compl-aux (Some y) (n + 1) (x # xs) ys
  else proper-interval ao (Some x) ∧
    (let m = card (UNIV :: 'a set) - n in m - length ys ≠ 2 ∨ m - length xs ≠
    2))

```

```

fun proper-interval-Compl-set-aux :: 'a option ⇒ 'a list ⇒ 'a list ⇒ bool
where

```

```

  proper-interval-Compl-set-aux ao (x # xs) (y # ys) ←→
  (if x < y then
    ¬ proper-interval ao (Some x) ∧
    proper-interval-Compl-set-aux (Some x) xs (y # ys)
  else if y < x then
    ¬ proper-interval ao (Some y) ∧
    proper-interval-Compl-set-aux (Some y) (x # xs) ys
  else ¬ proper-interval ao (Some x) ∧ (ys = [] → xs ≠ []))
| proper-interval-Compl-set-aux ao - - ←→ False

```

end

```

lemmas [code] =
  proper-intrvl.exhaustive-above.simps
  proper-intrvl.exhaustive.simps
  proper-intrvl.proper-interval-set-aux.simps
  proper-intrvl.proper-interval-set-Compl-aux.simps
  proper-intrvl.proper-interval-Compl-set-aux.simps

```

context linorder-proper-interval **begin**

lemma exhaustive-above-iff:

```

  [| sorted xs; distinct xs; ∀ x' ∈ set xs. x < x' |] ⇒ exhaustive-above x xs ←→ set
  xs = {z. z > x}

```

proof(induction x xs rule: exhaustive-above.induct)

case 1 **thus** ?case **by**(simp add: proper-interval-simps)

next

case (2 x y ys)

from ⟨sorted (y # ys)⟩ ⟨distinct (y # ys)⟩

have ys: sorted ys distinct ys **and** y: ∀ y' ∈ set ys. y < y'

by(auto simp add: less-le)

hence exhaustive-above y ys = (set ys = {z. y < z}) **by**(rule 2.IH)

moreover from ⟨∀ y' ∈ set (y # ys). x < y'⟩ **have** x < y **by** simp

ultimately show ?case **using** y

by(fastforce simp add: proper-interval-simps)

qed

lemma exhaustive-correct:

assumes sorted xs distinct xs

shows exhaustive xs ←→ set xs = UNIV

proof(cases xs)

```

    case Nil thus ?thesis by auto
next
  case Cons
  show ?thesis using assms unfolding Cons exhaustive.simps
    apply(subst exhaustive-above-iff)
    apply(auto simp add: less-le proper-interval-simps not-less intro: order-antisym)
  done
qed

```

```

theorem proper-interval-set-aux:
  assumes fin: finite (UNIV :: 'a set)
  and xs: sorted xs distinct xs
  and ys: sorted ys distinct ys
  shows proper-interval-set-aux xs ys  $\longleftrightarrow$  ( $\exists A. \text{set } xs \sqsubset' A \wedge A \sqsubset' \text{set } ys$ )
proof -
  note [simp] = finite-subset[OF subset-UNIV fin]
  show ?thesis using xs ys
  proof(induction xs ys rule: proper-interval-set-aux.induct)
    case 1 thus ?case by simp
  next
    case (2 y ys)
    hence  $\forall y' \in \text{set } ys. y < y'$  by(auto simp add: less-le)
    thus ?case
      by(cases ys)(auto simp add: proper-interval-simps set-less-aux-singleton-iff
        intro: psubset-finite-imp-set-less-aux)
  next
    case (3 x xs y ys)
    from  $\langle \text{sorted } (x \# xs) \rangle \langle \text{distinct } (x \# xs) \rangle$ 
    have xs: sorted xs distinct xs and x:  $\forall x' \in \text{set } xs. x < x'$ 
      by(auto simp add: less-le)
    from  $\langle \text{sorted } (y \# ys) \rangle \langle \text{distinct } (y \# ys) \rangle$ 
    have ys: sorted ys distinct ys and y:  $\forall y' \in \text{set } ys. y < y'$ 
      by(auto simp add: less-le)
    have Minxxs:  $\text{Min } (\text{set } (x \# xs)) = x$  and xnxs:  $x \notin \text{set } xs$ 
      using x by(auto intro!: Min-eqI)
    have Minyyys:  $\text{Min } (\text{set } (y \# ys)) = y$  and ynys:  $y \notin \text{set } ys$ 
      using y by(auto intro!: Min-eqI)
    show ?case
    proof(cases  $x < y$ )
      case True
      hence  $\text{set } (y \# ys) \sqsubset' \text{set } (x \# xs)$  using Minxxs Minyyys
        by -(rule set-less-aux-Min-antimono, simp-all)
      thus ?thesis using True by(auto dest: set-less-aux-trans set-less-aux-antisym)
    next
      case False
      show ?thesis
      proof(cases  $y < x$ )
        case True
        { assume proper-interval (Some y) (Some x)

```

then obtain z where $z: y < z \quad z < x$ by(*auto simp add: proper-interval-simps*)
 hence $\text{set } (x \# xs) \sqsubset' \{z\}$ using x
 by $-(\text{rule set-less-aux-Min-antimono}, \text{auto})$
 moreover have $\{z\} \sqsubset' \text{set } (y \# ys)$ using $z \ y \ \text{Minyys}$
 by $-(\text{rule set-less-aux-Min-antimono}, \text{auto})$
 ultimately have $\exists A. \text{set } (x \# xs) \sqsubset' A \wedge A \sqsubset' \text{set } (y \# ys)$ by *blast* }
 moreover
 { assume $ys \neq []$
 hence $\{y\} \sqsubset' \text{set } (y \# ys)$ using y
 by $-(\text{rule psubset-finite-imp-set-less-aux}, \text{auto simp add: neq-Nil-conv})$
 moreover have $\text{set } (x \# xs) \sqsubset' \{y\}$ using *False True x*
 by $-(\text{rule set-less-aux-Min-antimono}, \text{auto})$
 ultimately have $\exists A. \text{set } (x \# xs) \sqsubset' A \wedge A \sqsubset' \text{set } (y \# ys)$ by *blast* }
 moreover
 { assume $\neg \text{exhaustive-above } x \ xs$
 then obtain z where $z: z > x \quad z \notin \text{set } xs$ using x
 by(*auto simp add: exhaustive-above-iff[OF xs x]*)
 let $?A = \text{insert } z \ (\text{set } (x \# xs))$
 have $\text{set } (x \# xs) \sqsubset' ?A$ using z
 by $-(\text{rule psubset-finite-imp-set-less-aux}, \text{auto})$
 moreover have $?A \sqsubset' \text{set } (y \# ys)$ using *Minyys False True z x*
 by $-(\text{rule set-less-aux-Min-antimono}, \text{auto})$
 ultimately have $\exists A. \text{set } (x \# xs) \sqsubset' A \wedge A \sqsubset' \text{set } (y \# ys)$ by *blast* }
 moreover
 { fix A
 assume $A: \text{set } (x \# xs) \sqsubset' A$ and $A': A \sqsubset' \{y\}$
 and $pi: \neg \text{proper-interval } (\text{Some } y) \ (\text{Some } x)$
 from A have *nempty: $A \neq \{\}$* by *auto*
 have $y \notin A$
 proof
 assume $y \in A$
 moreover with A' have $A \neq \{y\}$ by *auto*
 ultimately have $\{y\} \sqsubset' A$ by $-(\text{rule psubset-finite-imp-set-less-aux},$
auto)
 with A' show *False* by(*rule set-less-aux-antisym*)
 qed
 have $y < \text{Min } A$ unfolding *not-le[symmetric]*
 proof
 assume $\text{Min } A \leq y$
 moreover have $\text{Min } A \neq y$ using $\langle y \notin A \rangle$ *nempty* by *clarsimp*
 ultimately have $\text{Min } A < \text{Min } \{y\}$ by *simp*
 hence $\{y\} \sqsubset' A$ by(*rule set-less-aux-Min-antimono*)(*simp-all add: nempty*)
 with A' show *False* by(*rule set-less-aux-antisym*)
 qed
 with pi *nempty* have $x \leq \text{Min } A$ by(*auto simp add: proper-interval-simps*)
 moreover
 from A obtain z where $z: z \in A \quad z \notin \text{set } (x \# xs)$
 by(*auto simp add: set-less-aux-def*)
 with $\langle x \leq \text{Min } A \rangle$ *nempty* have $x < z$ by *auto*

```

with z have  $\neg$  exhaustive-above x xs
  by(auto simp add: exhaustive-above-iff[OF xs x]) }
ultimately show ?thesis using True False by fastforce
next
case False
with  $\langle \neg x < y \rangle$  have  $x = y$  by auto
from  $\langle \neg x < y \rangle$  False
have proper-interval-set-aux xs ys =  $(\exists A. \text{set } xs \sqsubset' A \wedge A \sqsubset' \text{set } ys)$ 
  using xs ys by(rule 3.IH)
also have ... =  $(\exists A. \text{set } (x \# xs) \sqsubset' A \wedge A \sqsubset' \text{set } (y \# ys))$ 
  (is ?lhs = ?rhs)
proof
  assume ?lhs
  then obtain A where A:  $\text{set } xs \sqsubset' A$ 
    and A':  $A \sqsubset' \text{set } ys$  by blast
  hence nempty:  $A \neq \{\}$   $ys \neq []$  by auto
  let ?A = insert y A
  { assume  $\text{Min } A \leq y$ 
    also from y nempty have  $y < \text{Min } (\text{set } ys)$  by auto
    finally have  $\text{set } ys \sqsubset' A$  by(rule set-less-aux-Min-antimono)(simp-all
add: nempty)
    with A' have False by(rule set-less-aux-antisym) }
  hence  $\text{Min } A: y < \text{Min } A$  by(metis not-le)
  with nempty have  $y \notin A$  by auto
  moreover
  with  $\text{Min } A$  nempty have  $\text{Min } ?A = y$  by  $-(\text{rule } \text{Min-eqI}, \text{auto})$ 
  ultimately have A1:  $\text{set } (x \# xs) \sqsubset' ?A$  using  $\langle x = y \rangle A \text{Min}xxs \text{xnxs}$ 
    by(subst set-less-aux-rec) simp-all
  moreover
  have  $?A \sqsubset' \text{set } (y \# ys)$  using  $\langle x = y \rangle \text{Min}yA \langle y \notin A \rangle A' \text{Min}yys \text{ynys}$ 
    by(subst set-less-aux-rec) simp-all
  ultimately show ?rhs by blast
next
  assume ?rhs
  then obtain A where A:  $\text{set } (x \# xs) \sqsubset' A$ 
    and A':  $A \sqsubset' \text{set } (y \# ys)$  by blast
  let ?A =  $A - \{x\}$ 
  from A have nempty:  $A \neq \{\}$  by auto
  { assume  $x < \text{Min } A$ 
    hence  $\text{Min } (\text{set } (x \# xs)) < \text{Min } A$  unfolding  $\text{Min}xxs$  .
    hence  $A \sqsubset' \text{set } (x \# xs)$ 
      by(rule set-less-aux-Min-antimono) simp-all
    with A have False by(rule set-less-aux-antisym) }
  moreover
  { assume  $\text{Min } A < x$ 
    hence  $\text{Min } A < \text{Min } (\text{set } (y \# ys))$  unfolding  $\langle x = y \rangle \text{Min}yys$  .
    hence  $\text{set } (y \# ys) \sqsubset' A$  by(rule set-less-aux-Min-antimono)(simp-all
add: nempty)
    with A' have False by(rule set-less-aux-antisym) }

```

```

ultimately have MinA:  $\text{Min } A = x$  by(metis less-linear)
hence  $x \in A$  using nempty by(metis Min-in ‹finite A›

from A nempty Minxs xnxs have  $\text{set } xs \sqsubset' ?A$ 
  by(subst (asm) set-less-aux-rec)(auto simp add: MinA)
moreover from  $A' \langle x = y \rangle$  nempty Minyys MinA ynys have  $?A \sqsubset' \text{set } ys$ 
  by(subst (asm) set-less-aux-rec) simp-all
ultimately show ?lhs by blast
qed
finally show ?thesis using ‹ $x = y$ › by simp
qed
qed
qed
qed

lemma proper-interval-set-Compl-aux:
  assumes fin: finite (UNIV :: 'a set)
  and xs: sorted xs   distinct xs
  and ys: sorted ys   distinct ys
  shows proper-interval-set-Compl-aux None 0 xs ys  $\longleftrightarrow (\exists A. \text{set } xs \sqsubset' A \wedge A \sqsubset' - \text{set } ys)$ 
proof -
  note [simp] = finite-subset[OF subset-UNIV fin]

  define above where above = case-option UNIV (Collect o less)
  have above-simps [simp]: above None = UNIV    $\bigwedge x. \text{above } (\text{Some } x) = \{y. x < y\}$ 
  and above-upclosed:  $\bigwedge x y \text{ ao. } \llbracket x \in \text{above } \text{ao}; x < y \rrbracket \implies y \in \text{above } \text{ao}$ 
  and proper-interval-Some2:  $\bigwedge x \text{ ao. } \text{proper-interval } \text{ao } (\text{Some } x) \longleftrightarrow (\exists z \in \text{above } \text{ao}. z < x)$ 
  by(simp-all add: proper-interval-simps above-def split: option.splits)

  { fix ao n
    assume set xs  $\subseteq$  above ao   set ys  $\subseteq$  above ao
    from xs ‹set xs  $\subseteq$  above ao› ys ‹set ys  $\subseteq$  above ao›
    have proper-interval-set-Compl-aux ao (card (UNIV - above ao)) xs ys  $\longleftrightarrow$ 
       $(\exists A \subseteq \text{above } \text{ao}. \text{set } xs \sqsubset' A \wedge A \sqsubset' - \text{set } ys \cap \text{above } \text{ao})$ 
    proof(induct ao n  $\equiv$  card (UNIV - above ao)) xs ys rule: proper-interval-set-Compl-aux.induct
      case (1 ao)
      have card (UNIV - above ao) + 1 < CARD('a)  $\longleftrightarrow (\exists A \subseteq \text{above } \text{ao}. A \neq \{\} \wedge A \sqsubset' \text{above } \text{ao})$ 
        (is ?lhs  $\longleftrightarrow$  ?rhs)
      proof
        assume ?lhs
        hence card (UNIV - (UNIV - above ao)) > 1 by(simp add: card-Diff-subset)
        from card-gt-1D[OF this]
        obtain x y where above:  $x \in \text{above } \text{ao}$     $y \in \text{above } \text{ao}$ 
          and neq:  $x \neq y$  by blast
        hence  $\{x\} \sqsubset' \{x, y\} \cap \text{above } \text{ao}$ 

```

```

    by(simp-all add: psubsetI psubset-finite-imp-set-less-aux)
  also have ...  $\sqsubseteq'$  above ao
    by(auto intro: subset-finite-imp-set-less-eq-aux)
  finally show ?rhs using above by blast
next
  assume ?rhs
  then obtain A where nempty:  $A \cap$  above ao  $\neq \{\}$ 
    and subset:  $A \subseteq$  above ao
    and less:  $A \sqsubseteq'$  above ao by blast
  from nempty obtain x where x:  $x \in A$   $x \in$  above ao by blast
  show ?lhs
  proof(rule ccontr)
    assume  $\neg$  ?lhs
    moreover have  $CARD('a) \geq$  card (UNIV - above ao) by(rule card-mono)
simp-all
    moreover from card-Un-disjoint[of UNIV - above ao above ao]
    have  $CARD('a) =$  card (UNIV - above ao) + card (above ao) by auto
    ultimately have card (above ao) = 1 using x
      by(cases card (above ao))(auto simp add: not-less-eq less-Suc-eq-le)
    with x have above ao = {x} unfolding card-eq-1-iff by auto
    moreover with x subset have A:  $A = \{x\}$  by auto
    ultimately show False using less by simp
  qed
  qed
  thus ?case by simp
next
  case (2 ao y ys)
  note ys =  $\langle$ sorted (y # ys) $\rangle$   $\langle$ distinct (y # ys) $\rangle$   $\langle$ set (y # ys)  $\subseteq$  above ao $\rangle$ 
  have len-ys: length ys = card (set ys)
    using ys by(auto simp add: List.card-set intro: sym)

  define m where m =  $CARD('a) -$  card (UNIV - above ao)
  have  $CARD('a) =$  card (above ao) + card (UNIV - above ao)
    using card-Un-disjoint[of above ao UNIV - above ao] by auto
  hence m-eq:  $m =$  card (above ao) unfolding m-def by simp

  have  $m \neq$  length ys + 1  $\wedge$  ( $m =$  length ys + 2  $\longrightarrow$   $\neg$  proper-interval (Some
  (last (y # ys))) None)  $\longleftrightarrow$ 
  ( $\exists A \subseteq$  above ao.  $A \neq \{\}$   $\wedge$   $A \sqsubseteq' -$  set (y # ys)  $\cap$  above ao) (is ?lhs  $\longleftrightarrow$ 
  ?rhs)
  proof
    assume ?lhs
    hence m:  $m \neq$  length ys + 1
      and pi:  $m =$  length ys + 2  $\implies$   $\neg$  proper-interval (Some (last (y # ys)))
None
    by simp-all
  have length ys + 1 = card (set (y # ys)) using ys len-ys by simp
  also have ...  $\leq$  m unfolding m-eq by(rule card-mono)(simp, rule ys)
  finally have length ys + 2  $\leq$  m using m by simp

```

```

show ?rhs
proof(cases m = length ys + 2)
  case True
  hence card (UNIV - (UNIV - above ao) - set (y # ys)) = 1
    using ys len-ys
  by(subst card-Diff-subset)(auto simp add: m-def card-Diff-subset)
  then obtain z where z: z ∈ above ao   z ≠ y   z ∉ set ys
    unfolding card-eq-1-iff by auto
  from True have ¬ proper-interval (Some (last (y # ys))) None by(rule
pi)
  hence z ≤ last (y # ys) by(simp add: proper-interval-simps not-less del:
last.simps)
  moreover have ly: last (y # ys) ∈ set (y # ys) by(rule last-in-set) simp
  with z have z ≠ last (y # ys) by auto
  ultimately have z < last (y # ys) by simp
  hence {last (y # ys)} ⊆ {z}
    using z ly ys by(auto 4 3 simp add: set-less-aux-def)
  also have ... ⊆' - set (y # ys) ∩ above ao
    using z by(auto intro: subset-finite-imp-set-less-eq-aux)
  also have {last (y # ys)} ≠ {} using ly ys by blast
  moreover have {last (y # ys)} ⊆ above ao using ys by auto
  ultimately show ?thesis by blast
next
case False
with ⟨length ys + 2 ≤ m⟩ ys len-ys
have card (UNIV - (UNIV - above ao) - set (y # ys)) > 1
  by(subst card-Diff-subset)(auto simp add: card-Diff-subset m-def)
from card-gt-1D[OF this]
obtain x x' where x: x ∈ above ao   x ≠ y   x ∉ set ys
  and x': x' ∈ above ao   x' ≠ y   x' ∉ set ys
  and neq: x ≠ x' by auto
hence {x} ⊆' {x, x'} ∩ above ao
  by(simp-all add: psubsetI psubset-finite-imp-set-less-aux)
also have ... ⊆' - set (y # ys) ∩ above ao using x x' ys
  by(auto intro: subset-finite-imp-set-less-eq-aux)
also have {x} ∩ above ao ≠ {} using x by simp
ultimately show ?rhs by blast
qed
next
assume ?rhs
then obtain A where nempty: A ≠ {}
  and less: A ⊆' - set (y # ys) ∩ above ao
  and subset: A ⊆ above ao by blast
have card (set (y # ys)) ≤ card (above ao) using ys(3) by(simp add:
card-mono)
hence length ys + 1 ≤ m unfolding m-eq using ys by(simp add: len-ys)
have m ≠ length ys + 1
proof
  assume m = length ys + 1

```

```

hence card (above ao) ≤ card (set (y # ys))
  unfolding m-eq using ys len-ys by auto
from card-seteq[OF - - this] ys have set (y # ys) = above ao by simp
with nempty less show False by(auto simp add: set-less-aux-def)
qed
moreover
{ assume m = length ys + 2
  hence card (above ao - set (y # ys)) = 1
    using ys len-ys m-eq by(auto simp add: card-Diff-subset)
  then obtain z where z: above ao - set (y # ys) = {z}
    unfolding card-eq-1-iff ..
  hence eq-z: - set (y # ys) ∩ above ao = {z} by auto
  with less have A ⊆ {z} by simp
  have ¬ proper-interval (Some (last (y # ys))) None
  proof
    assume proper-interval (Some (last (y # ys))) None
    then obtain z' where z': last (y # ys) < z'
      by(clarsimp simp add: proper-interval-simps)
    have last (y # ys) ∈ set (y # ys) by(rule last-in-set) simp
    with ys z' have z' ∈ above ao z' ∉ set (y # ys)
      using above-upclosed local.not-less local.sorted-last ys(1) z' by blast+
    with eq-z have z = z' by fastforce
    from z' have ∧x. x ∈ set (y # ys) ⇒ x < z' using ys
      by(auto dest: sorted-last simp del: sorted-wrt.simps(2))
    with eq-z ⟨z = z'⟩
    have ∧x. x ∈ above ao ⇒ x ≤ z' by(fastforce)
    with ⟨A ⊆ {z}⟩ nempty ⟨z = z'⟩ subset
    show False by(auto simp add: set-less-aux-def)
  qed }
ultimately show ?lhs by simp
qed
thus ?case by(simp add: length-last-def m-def Let-def)
next
case (∃ ao x xs)
note xs = ⟨sorted (x # xs)⟩ ⟨distinct (x # xs)⟩ ⟨set (x # xs) ⊆ above ao⟩
have len-xs: length xs = card (set xs)
  using xs by(auto simp add: List.card-set intro: sym)

define m where m = CARD('a) - card (UNIV - above ao)
have CARD('a) = card (above ao) + card (UNIV - above ao)
  using card-Un-disjoint[of above ao UNIV - above ao] by auto
hence m-eq: m = card (above ao) unfolding m-def by simp
have m ≠ length xs + 1 ∧ (m = length xs + 2 ⇒ ¬ proper-interval (Some
(last (x # xs))) None) ⇔
  (∃ A ⊆ above ao. set (x # xs) ⊆ A ∧ A ⊆ above ao) (is ?lhs ⇔ ?rhs)
proof
  assume ?lhs
  hence m: m ≠ length xs + 1
    and pi: m = length xs + 2 ⇒ ¬ proper-interval (Some (last (x # xs)))

```

None

```

  by simp-all
  have length xs + 1 = card (set (x # xs)) using xs len-xs by simp
  also have ... ≤ m unfolding m-eq by(rule card-mono)(simp, rule xs)
  finally have length xs + 2 ≤ m using m by simp
  show ?rhs
  proof(cases m = length xs + 2)
    case True
    hence card (UNIV - (UNIV - above ao) - set (x # xs)) = 1
      using xs len-xs
      by(subst card-Diff-subset)(auto simp add: m-def card-Diff-subset)
    then obtain z where z: z ∈ above ao   z ≠ x   z ∉ set xs
      unfolding card-eq-1-iff by auto
    define A where A = insert z {y. y ∈ set (x # xs) ∧ y < z}

    from True have ¬ proper-interval (Some (last (x # xs))) None by(rule
pi)
    hence z ≤ last (x # xs) by(simp add: proper-interval-simps not-less del:
last.simps)
    moreover have lx: last (x # xs) ∈ set (x # xs) by(rule last-in-set) simp
    with z have z ≠ last (x # xs) by auto
    ultimately have z < last (x # xs) by simp
    hence set (x # xs) ⊆' A
    using z xs by(auto simp add: A-def set-less-aux-def intro: rev-bexI[where
x=z])
    moreover have last (x # xs) ∉ A using xs ⟨z < last (x # xs)⟩
      by(auto simp add: A-def simp del: last.simps)
    hence A ⊂ insert (last (x # xs)) A by blast
    hence less': A ⊆' insert (last (x # xs)) A
      by(rule psubset-finite-imp-set-less-aux) simp
    have ... ⊆ above ao using xs lx z
      by(auto simp del: last.simps simp add: A-def)
    hence insert (last (x # xs)) A ⊆' above ao
      by(auto intro: subset-finite-imp-set-less-eq-aux)
    with less' have A ⊆' above ao
      by(rule set-less-trans-set-less-eq)
    moreover have A ⊆ above ao using xs z by(auto simp add: A-def)
    ultimately show ?thesis by blast
  next
  case False
  with ⟨length xs + 2 ≤ m⟩ xs len-xs
  have card (UNIV - (UNIV - above ao) - set (x # xs)) > 1
    by(subst card-Diff-subset)(auto simp add: card-Diff-subset m-def)
  from card-gt-1D[OF this]
  obtain y y' where y: y ∈ above ao   y ≠ x   y ∉ set xs
    and y': y' ∈ above ao   y' ≠ x   y' ∉ set xs
    and neq: y ≠ y' by auto
  define A where A = insert y (set (x # xs) ∩ above ao)
  hence set (x # xs) ⊂ A using y xs by auto

```

```

hence set (x # xs)  $\sqsubset'$  ...
  by(fastforce intro: psubset-finite-imp-set-less-aux)
moreover have *: ...  $\subset$  above ao
  using y y' neq by(auto simp add: A-def)
moreover from * have A  $\sqsubset'$  above ao
  by(auto intro: psubset-finite-imp-set-less-aux)
ultimately show ?thesis by blast
qed
next
assume ?rhs
then obtain A where lessA: set (x # xs)  $\sqsubset'$  A
  and Aless: A  $\sqsubset'$  above ao and subset: A  $\subseteq$  above ao by blast
  have card (set (x # xs))  $\leq$  card (above ao) using xs(3) by(simp add:
card-mono)
hence length xs + 1  $\leq$  m unfolding m-eq using xs by(simp add: len-xs)
have m  $\neq$  length xs + 1
proof
  assume m = length xs + 1
  hence card (above ao)  $\leq$  card (set (x # xs))
    unfolding m-eq using xs len-xs by auto
  from card-seteq[OF - - this] xs have set (x # xs) = above ao by simp
  with lessA Aless show False by(auto dest: set-less-aux-antisym)
qed
moreover
{ assume m = length xs + 2
  hence card (above ao - set (x # xs)) = 1
    using xs len-xs m-eq by(auto simp add: card-Diff-subset)
  then obtain z where z: above ao - set (x # xs) = {z}
    unfolding card-eq-1-iff ..
  have  $\neg$  proper-interval (Some (last (x # xs))) None
  proof
    assume proper-interval (Some (last (x # xs))) None
    then obtain z' where z': last (x # xs) < z'
      by(clarsimp simp add: proper-interval-simps)
    have last (x # xs)  $\in$  set (x # xs) by(rule last-in-set) simp
    with xs z' have z'  $\in$  above ao z'  $\notin$  set (x # xs)
      by(auto simp del: last.simps sorted-wrt.simps(2) intro: above-upclosed
dest: sorted-last)
    with z have z = z' by fastforce
    from z' have y-less:  $\bigwedge y. y \in$  set (x # xs)  $\implies$  y < z' using xs
      by(auto simp del: sorted-wrt.simps(2) dest: sorted-last)
    with z  $\langle z = z' \rangle$  have  $\bigwedge y. y \in$  above ao  $\implies$  y  $\leq$  z' by(fastforce)

  from lessA subset obtain y where y: y  $\in$  A y  $\in$  above ao y  $\notin$  set
(x # xs)
  and min:  $\bigwedge y'. \llbracket y' \in$  set (x # xs); y'  $\in$  above ao; y'  $\notin$  A  $\rrbracket \implies$  y  $\leq$  y'
    by(auto simp add: set-less-aux-def)
  with z  $\langle z = z' \rangle$  have y = z' by auto
  have set (x # xs)  $\subseteq$  A

```

```

proof
  fix  $y'$ 
  assume  $y': y' \in \text{set } (x \# xs)$ 
  show  $y' \in A$ 
  proof(rule ccontr)
    assume  $y' \notin A$ 
    from  $y' xs$  have  $y' \in \text{above } ao$  by auto
    with  $y'$  have  $y \leq y'$  using  $\langle y' \notin A \rangle$  by(rule min)
    moreover from  $y'$  have  $y' < z'$  by(rule y-less)
    ultimately show False using  $\langle y = z' \rangle$  by simp
  qed
qed
moreover from  $z xs$  have  $\text{above } ao = \text{insert } z (\text{set } (x \# xs))$  by auto
ultimately have  $A = \text{above } ao$  using  $\langle y = z' \rangle \langle z = z' \rangle$  subset by auto
with Aless show False by simp
qed }
ultimately show ?lhs by simp
qed
thus ?case by(simp add: length-last-def m-def Let-def del: last.simps)
next
case ( $\not\perp ao \ x \ xs \ y \ ys$ )
note  $xxs = \langle \text{sorted } (x \# xs) \rangle \langle \text{distinct } (x \# xs) \rangle$ 
  and  $yys = \langle \text{sorted } (y \# ys) \rangle \langle \text{distinct } (y \# ys) \rangle$ 
  and  $xxs\text{-above} = \langle \text{set } (x \# xs) \subseteq \text{above } ao \rangle$ 
  and  $yys\text{-above} = \langle \text{set } (y \# ys) \subseteq \text{above } ao \rangle$ 
from  $xxs$  have  $xs: \text{sorted } xs \quad \text{distinct } xs$  and  $x\text{-Min}: \forall x' \in \text{set } xs. x < x'$ 
  by(auto simp add: less-le)
from  $yys$  have  $ys: \text{sorted } ys \quad \text{distinct } ys$  and  $y\text{-Min}: \forall y' \in \text{set } ys. y < y'$ 
  by(auto simp add: less-le)

have  $\text{len-}xs: \text{length } xs = \text{card } (\text{set } xs)$ 
  using  $xs$  by(auto simp add: List.card-set intro: sym)
have  $\text{len-}ys: \text{length } ys = \text{card } (\text{set } ys)$ 
  using  $ys$  by(auto simp add: List.card-set intro: sym)

show ?case
proof(cases x < y)
  case True

  have proper-interval  $ao$  (Some  $x$ )  $\vee$ 
    proper-interval-set-Compl-aux (Some  $x$ ) ( $\text{card } (\text{UNIV} - \text{above } ao) + 1$ )
 $xs \ (y \# ys) \longleftrightarrow$ 
    ( $\exists A \subseteq \text{above } ao. \text{set } (x \# xs) \sqsubset' A \wedge A \sqsubset' - \text{set } (y \# ys) \cap \text{above } ao$ )
    (is ?lhs  $\longleftrightarrow$  ?rhs)
  proof(cases proper-interval  $ao$  (Some  $x$ ))
    case True
    then obtain  $z$  where  $z: z \in \text{above } ao \quad z < x$ 
      by(clarsimp simp add: proper-interval-Some2)
    moreover with  $xxs$  have  $\forall x' \in \text{set } xs. z < x'$  by(auto)

```

```

ultimately have set (x # xs)  $\sqsubset'$  {z}
  by(auto simp add: set-less-aux-def intro!: beXI[where x=z])
moreover {
  from z yys  $\langle x < y \rangle$  have z < y  $\forall y' \in \text{set } \text{yys}. z < y'$ 
    by(auto)
  hence subset: {z}  $\subseteq$  - set (y # ys)  $\cap$  above ao
    using yys  $\langle x < y \rangle$  z by auto
  moreover have x  $\in$  ... using yys xxs  $\langle x < y \rangle$  xxs-above by(auto)
  ultimately have {z}  $\subset$  ... using  $\langle z < x \rangle$  by fastforce
  hence {z}  $\sqsubset'$  ...
    by(fastforce intro: psubset-finite-imp-set-less-aux) }
moreover have {z}  $\subseteq$  above ao using z by simp
ultimately have ?rhs by blast
thus ?thesis using True by simp
next
case False
hence above-eq: above ao = insert x (above (Some x)) using xxs-above
  by(auto simp add: proper-interval-Some2 intro: above-upclosed)
moreover have card (above (Some x)) < CARD('a)
  by(rule psubset-card-mono)(auto)
ultimately have card-eq: card (UNIV - above ao) + 1 = card (UNIV
- above (Some x))
  by(simp add: card-Diff-subset)
from xxs-above x-Min have xs-above: set xs  $\subseteq$  above (Some x) by(auto)
from  $\langle x < y \rangle$  y-Min have set (y # ys)  $\subseteq$  above (Some x) by(auto)
with  $\langle x < y \rangle$  card-eq xs xs-above yys
  have proper-interval-set-Compl-aux (Some x) (card (UNIV - above ao)
+ 1) xs (y # ys)  $\longleftrightarrow$ 
    ( $\exists A \subseteq$  above (Some x). set xs  $\sqsubset'$  A  $\wedge$  A  $\sqsubset'$  - set (y # ys)  $\cap$  above
(Some x))
  by(subst card-eq)(rule 4)
also have ...  $\longleftrightarrow$  ?rhs (is ?lhs'  $\longleftrightarrow$  -)
proof
  assume ?lhs'
  then obtain A where less-A: set xs  $\sqsubset'$  A
    and A-less: A  $\sqsubset'$  - set (y # ys)  $\cap$  above (Some x)
    and subset: A  $\subseteq$  above (Some x) by blast
  let ?A = insert x A

  have Min-A': Min ?A = x using xxs-above False subset
    by(auto intro!: Min-eqI simp add: proper-interval-Some2)
  moreover have Min (set (x # xs)) = x
    using x-Min by(auto intro!: Min-eqI)
  moreover have Amx: A - {x} = A
    using False subset
    by(auto simp add: proper-interval-Some2 intro: above-upclosed)
  moreover have set xs - {x} = set xs using x-Min by auto
  ultimately have less-A': set (x # xs)  $\sqsubset'$  ?A
    using less-A xxs-above x-Min by(subst set-less-aux-rec) simp-all

```

```

have  $x \in - \text{insert } y \text{ (set } ys) \cap \text{above } ao$ 
  using  $\langle x < y \rangle \text{ } xxs\text{-above } y\text{-Min}$  by auto
hence  $- \text{insert } y \text{ (set } ys) \cap \text{above } ao \neq \{\}$  by auto
moreover have  $\text{Min } (- \text{insert } y \text{ (set } ys) \cap \text{above } ao) = x$ 
  using  $yys \text{ } y\text{-Min } xxs\text{-above } \langle x < y \rangle \text{ } False$ 
  by(auto intro!: Min-eqI simp add: proper-interval-Some2)
moreover have  $- \text{set } (y \# ys) \cap \text{above } ao - \{x\} = - \text{set } (y \# ys) \cap$ 
above (Some x)
  using  $yys\text{-above } False \text{ } xxs\text{-above}$ 
  by(auto simp add: proper-interval-Some2 intro: above-upclosed)
ultimately have  $A'\text{-less: } ?A \sqsubset' - \text{set } (y \# ys) \cap \text{above } ao$ 
  using  $\text{Min-}A' \text{ } A'\text{-less } Amx \text{ } xxs\text{-above}$  by(subst set-less-aux-rec) simp-all
moreover have  $?A \subseteq \text{above } ao$  using subset xxs-above by(auto intro:
above-upclosed)
ultimately show  $?rhs$  using less-A' by blast
next
assume  $?rhs$ 
then obtain  $A$  where  $\text{less-}A: \text{set } (x \# xs) \sqsubset' A$ 
  and  $A'\text{-less: } A \sqsubset' - \text{set } (y \# ys) \cap \text{above } ao$ 
  and  $\text{subset: } A \subseteq \text{above } ao$  by blast
let  $?A = A - \{x\}$ 

from  $\text{less-}A \text{ } \text{subset } xxs\text{-above}$  have  $\text{set } (x \# xs) \cap \text{above } ao \sqsubset' A \cap \text{above}$ 
ao
  by(simp add: Int-absorb2)
with  $False \text{ } xxs\text{-above } \text{subset}$  have  $x \in A$ 
  by(auto simp add: set-less-aux-def proper-interval-Some2)
hence  $\dots \neq \{\}$  by auto
moreover from  $\langle x \in A \rangle \text{ } False \text{ } \text{subset}$ 
have  $\text{Min-}A: \text{Min } A = x$ 
  by(auto intro!: Min-eqI simp add: proper-interval-Some2 not-less)
moreover have  $\text{Min } (\text{set } (x \# xs)) = x$ 
  using  $x\text{-Min}$  by(auto intro!: Min-eqI)
moreover have  $\text{eq-}A: ?A \subseteq \text{above } (Some \ x)$ 
  using  $xxs\text{-above } False \text{ } \text{subset}$ 
by(fastforce simp add: proper-interval-Some2 not-less intro: above-upclosed)
moreover have  $\text{set } xs - \{x\} = \text{set } xs$ 
  using  $x\text{-Min}$  by(auto)
ultimately have  $\text{less-}A': \text{set } xs \sqsubset' ?A$ 
  using  $xxs\text{-above } \text{less-}A$  by(subst (asm) set-less-aux-rec)(simp-all cong:
conj-cong)

have  $x \in - \text{insert } y \text{ (set } ys) \cap \text{above } ao$ 
  using  $\langle x < y \rangle \text{ } xxs\text{-above } y\text{-Min}$  by auto
hence  $- \text{insert } y \text{ (set } ys) \cap \text{above } ao \neq \{\}$  by auto
moreover have  $\text{Min } (- \text{set } (y \# ys) \cap \text{above } ao) = x$ 
  using  $yys \text{ } y\text{-Min } xxs\text{-above } \langle x < y \rangle \text{ } False$ 
  by(auto intro!: Min-eqI simp add: proper-interval-Some2)

```

```

moreover have  $- \text{set } (y \# ys) \cap \text{above } (\text{Some } x) = - \text{set } (y \# ys) \cap$ 
 $\text{above } ao - \{x\}$ 
by(auto simp add: above-eq)
ultimately have  $?A \sqsubset' - \text{set } (y \# ys) \cap \text{above } (\text{Some } x)$ 
using A-less  $\langle A \neq \{\} \rangle$  eq-A Min-A
by(subst (asm) set-less-aux-rec simp-all)

with less-A' eq-A show  $?lhs'$  by blast
qed
finally show  $?thesis$  using False by simp
qed
thus  $?thesis$  using True by simp
next
case False
show  $?thesis$ 
proof(cases y < x)
case True
have proper-interval ao (Some y)  $\vee$ 
proper-interval-set-Compl-aux (Some y) (card (UNIV - above ao) +
1)  $(x \# xs) ys \longleftrightarrow$ 
 $(\exists A \subseteq \text{above } ao. \text{set } (x \# xs) \sqsubset' A \wedge A \sqsubset' - \text{set } (y \# ys) \cap \text{above } ao)$ 
(is  $?lhs \longleftrightarrow ?rhs$ )
proof(cases proper-interval ao (Some y))
case True
then obtain  $z$  where  $z: z \in \text{above } ao \quad z < y$ 
by(clarsimp simp add: proper-interval-Some2)
from  $xs \langle y < x \rangle$  have  $\forall x' \in \text{set } (x \# xs). y < x'$  by(auto)
hence less-A: set  $(x \# xs) \sqsubset' \{y\}$ 
by(auto simp add: set-less-aux-def intro!: beXI[where x=y])

have  $\{y\} \sqsubset' \{z\}$ 
using  $z$  y-Min by(auto simp add: set-less-aux-def intro: beXI[where
 $x=z]$ )

also have  $\dots \subseteq - \text{set } (y \# ys) \cap \text{above } ao$  using  $z$  y-Min by auto
hence  $\{z\} \sqsubseteq' \dots$  by(auto intro: subset-finite-imp-set-less-eq-aux)
finally have  $\{y\} \sqsubset' \dots$  .
moreover have  $\{y\} \subseteq \text{above } ao$  using yys-above by auto
ultimately have  $?rhs$  using less-A by blast
thus  $?thesis$  using True by simp
next
case False
hence above-eq: above ao = insert y (above (Some y)) using yys-above
by(auto simp add: proper-interval-Some2 intro: above-upclosed)
moreover have  $\text{card } (\text{above } (\text{Some } y)) < \text{CARD}('a)$ 
by(rule psubset-card-mono)(auto)
ultimately have card-eq: card  $(\text{UNIV} - \text{above } ao) + 1 = \text{card } (\text{UNIV}$ 
 $- \text{above } (\text{Some } y))$ 
by(simp add: card-Diff-subset)
from yys-above y-Min have  $yys\text{-above}: \text{set } ys \subseteq \text{above } (\text{Some } y)$  by(auto)

```

have *eq-ys*: $- \text{set } ys \cap \text{above } (\text{Some } y) = - \text{set } (y \# ys) \cap \text{above } ao$
by(*auto simp add: above-eq*)

from $\langle y < x \rangle$ *x-Min* **have** $\text{set } (x \# xs) \subseteq \text{above } (\text{Some } y)$ **by**(*auto*)
with $\langle \neg x < y \rangle \langle y < x \rangle$ *card-eq xxs ys ys-above*
have *proper-interval-set-Compl-aux* $(\text{Some } y) (\text{card } (\text{UNIV} - \text{above } ao)$
 $+ 1) (x \# xs) ys \longleftrightarrow$
 $(\exists A \subseteq \text{above } (\text{Some } y). \text{set } (x \# xs) \sqsubset' A \wedge A \sqsubset' - \text{set } ys \cap \text{above}$
 $(\text{Some } y))$
by(*subst card-eq*)(*rule 4*)
also have $\dots \longleftrightarrow ?rhs$ (**is** $?lhs' \longleftrightarrow -$)
proof
assume $?lhs'$
then obtain A **where** $\text{set } (x \# xs) \sqsubset' A$ **and** $\text{subset}: A \subseteq \text{above } (\text{Some}$
 $y)$
and $A \sqsubset' - \text{set } ys \cap \text{above } (\text{Some } y)$ **by** *blast*
moreover from *subset* **have** $A - \{y\} = A$ **by** *auto*
ultimately have $\text{set } (x \# xs) \sqsubset' A - \{y\}$
and $A - \{y\} \sqsubset' - \text{set } (y \# ys) \cap \text{above } ao$
using *eq-ys* **by** *simp-all*
moreover from *subset* **have** $A - \{y\} \subseteq \text{above } ao$
using *ys-above* **by**(*auto intro: above-upclosed*)
ultimately show $?rhs$ **by** *blast*
next
assume $?rhs$
then obtain A **where** $\text{set } (x \# xs) \sqsubset' A$
and *A-less*: $A \sqsubset' - \text{set } (y \# ys) \cap \text{above } ao$
and *subset*: $A \subseteq \text{above } ao$ **by** *blast*
moreover
from *A-less* *False* *ys-above* **have** $y \notin A$
by(*auto simp add: set-less-aux-def proper-interval-Some2 not-less*)
ultimately have $\text{set } (x \# xs) \sqsubset' A$
and $A \sqsubset' - \text{set } ys \cap \text{above } (\text{Some } y)$
using *eq-ys* **by** *simp-all*
moreover from $\langle y \notin A \rangle$ *subset* *above-eq* **have** $A \subseteq \text{above } (\text{Some } y)$
by *auto*
ultimately show $?lhs'$ **by** *blast*
qed
finally show $?thesis$ **using** *False* **by** *simp*
qed
with *False* *True* **show** $?thesis$ **by** *simp*
next
case *False*
with $\langle \neg x < y \rangle$ **have** $x = y$ **by** *simp*
have *proper-interval* ao $(\text{Some } x) \wedge$
 $(\text{CARD}'a) - (\text{card } (\text{UNIV} - \text{above } ao) + \text{length } ys) \neq 2 \vee$
 $\text{CARD}'a - (\text{card } (\text{UNIV} - \text{above } ao) + \text{length } xs) \neq 2) \longleftrightarrow$
 $(\exists A \subseteq \text{above } ao. \text{set } (x \# xs) \sqsubset' A \wedge A \sqsubset' - \text{set } (y \# ys) \cap \text{above } ao)$

```

(is ?below  $\wedge$  ?card  $\longleftrightarrow$  ?rhs)
proof(cases ?below)
  case False
  hence  $\neg$  set (y # ys)  $\cap$  above ao  $\sqsubset'$  set (x # xs)
  using  $\langle x = y \rangle$  yys-above xxs-above y-Min
  by(auto simp add: not-less set-less-aux-def proper-interval-Some2 intro!:
  bexI[where x=y])
  with False show ?thesis by(auto dest: set-less-aux-trans)
next
  case True
  then obtain z where z: z  $\in$  above ao    z < x
  by(clarsimp simp add: proper-interval-Some2)

  have ?card  $\longleftrightarrow$  ?rhs
  proof
    assume ?rhs
    then obtain A where less-A: set (x # xs)  $\sqsubset'$  A
    and A-less: A  $\sqsubset'$   $\neg$  set (y # ys)  $\cap$  above ao
    and subset: A  $\subseteq$  above ao by blast

    {
      assume c-ys: CARD('a)  $-$  (card (UNIV  $-$  above ao) + length ys) =
      2
      and c-xs: CARD('a)  $-$  (card (UNIV  $-$  above ao) + length xs) = 2
      from c-ys yys-above len-ys y-Min have card (UNIV  $-$  (UNIV  $-$ 
      above ao)  $-$  set (y # ys)) = 1
      by(subst card-Diff-subset)(auto simp add: card-Diff-subset)
      then obtain z' where eq-y:  $\neg$  set (y # ys)  $\cap$  above ao = {z'}
      unfolding card-eq-1-iff by auto
      moreover from z have z  $\notin$  set (y # ys) using  $\langle x = y \rangle$  y-Min by
      auto
      ultimately have z' = z using z by fastforce

      from c-xs xxs-above len-xs x-Min have card (UNIV  $-$  (UNIV  $-$ 
      above ao)  $-$  set (x # xs)) = 1
      by(subst card-Diff-subset)(auto simp add: card-Diff-subset)
      then obtain z'' where eq-x:  $\neg$  set (x # xs)  $\cap$  above ao = {z''}
      unfolding card-eq-1-iff by auto
      moreover from z have z  $\notin$  set (x # xs) using x-Min by auto
      ultimately have z'' = z using z by fastforce

      from less-A subset obtain q where q  $\in$  A    q  $\in$  above ao    q  $\notin$  set
      (x # xs)

      by(auto simp add: set-less-aux-def)
      hence q  $\in$  {z''} unfolding eq-x[symmetric] by simp
      hence q = z'' by simp
      with  $\langle q \in A \rangle$   $\langle z' = z \rangle$   $\langle z'' = z \rangle$  z
      have  $\neg$  set (y # ys)  $\cap$  above ao  $\subseteq$  A
      unfolding eq-y by simp
    }
  end
end

```

```

    hence  $- \text{set } (y \# ys) \cap \text{above } ao \sqsubseteq' A$ 
      by(auto intro: subset-finite-imp-set-less-eq-aux)
    with  $A\text{-less}$  have  $\text{False}$  by(auto dest: set-less-trans-set-less-eq) }
  thus ?card by auto
next
assume ?card (is ?card-ys  $\vee$  ?card-xs)
thus ?rhs
proof
  assume ?card-ys
  let ?YS = UNIV - (UNIV - above ao) - set (y # ys)
  from  $\langle ?card\text{-}ys \rangle$  yys-above len-ys y-Min have card ?YS  $\neq 1$ 
    by(subst card-Diff-subset)(auto simp add: card-Diff-subset)
  moreover have ?YS  $\neq \{\}$  using True y-Min yys-above  $\langle x = y \rangle$ 
    by(fastforce simp add: proper-interval-Some2)
  hence card ?YS  $\neq 0$  by simp
  ultimately have card ?YS  $> 1$  by(cases card ?YS) simp-all
  from card-gt-1D[OF this] obtain  $x' y'$ 
    where  $x'$ :  $x' \in \text{above } ao$   $x' \notin \text{set } (y \# ys)$ 
      and  $y'$ :  $y' \in \text{above } ao$   $y' \notin \text{set } (y \# ys)$ 
      and neq:  $x' \neq y'$  by auto
  let ?A = {z}
  have set (x # xs)  $\sqsubseteq' ?A$  using z x-Min
    by(auto simp add: set-less-aux-def intro!: rev-beqI)
  moreover
  { have ?A  $\subseteq - \text{set } (y \# ys) \cap \text{above } ao$ 
    using z  $\langle x = y \rangle$  y-Min by(auto)
    moreover have  $x' \notin ?A \vee y' \notin ?A$  using neq by auto
    with  $x' y'$  have ?A  $\neq - \text{set } (y \# ys) \cap \text{above } ao$  by auto
    ultimately have ?A  $\subset - \text{set } (y \# ys) \cap \text{above } ao$  by(rule psubsetI)
    hence ?A  $\sqsubset' \dots$  by(fastforce intro: psubset-finite-imp-set-less-aux)
  }
  ultimately show ?thesis using z by blast
next
assume ?card-xs
let ?XS = UNIV - (UNIV - above ao) - set (x # xs)
from  $\langle ?card\text{-}xs \rangle$  xxs-above len-xs x-Min have card ?XS  $\neq 1$ 
  by(subst card-Diff-subset)(auto simp add: card-Diff-subset)
moreover have ?XS  $\neq \{\}$  using True x-Min xxs-above
  by(fastforce simp add: proper-interval-Some2)
hence card ?XS  $\neq 0$  by simp
ultimately have card ?XS  $> 1$  by(cases card ?XS) simp-all
from card-gt-1D[OF this] obtain  $x' y'$ 
  where  $x'$ :  $x' \in \text{above } ao$   $x' \notin \text{set } (x \# xs)$ 
    and  $y'$ :  $y' \in \text{above } ao$   $y' \notin \text{set } (x \# xs)$ 
    and neq:  $x' \neq y'$  by auto

define A
  where A = (if  $x' = \text{Min } (\text{above } ao)$  then insert  $y'$  (set (x # xs))
else insert  $x'$  (set (x # xs)))

```

```

hence set (x # xs) ⊆ A by auto
moreover have set (x # xs) ≠ ...
  using neq x' y' by(auto simp add: A-def)
ultimately have set (x # xs) ⊂ A ..
hence set (x # xs) ⊆' ...
  by(fastforce intro: psubset-finite-imp-set-less-aux)
moreover {
  have nempty: above ao ≠ {} using z by auto
  have A ⊆' {Min (above ao)}
    using z x' y' neq ⟨x = y⟩ x-Min xxs-above
  by(auto 6 4 simp add: set-less-aux-def A-def nempty intro!: rev-beatI
Min-eqI)
  also have Min (above ao) ≤ z using z by(simp)
  hence Min (above ao) < x using ⟨z < x⟩ by(rule le-less-trans)
  with ⟨x = y⟩ y-Min have Min (above ao) ∉ set (y # ys) by auto
  hence {Min (above ao)} ⊆ - set (y # ys) ∩ above ao
    by(auto simp add: nempty)
  hence {Min (above ao)} ⊆' ... by(auto intro: subset-finite-imp-set-less-eq-aux)
  finally have A ⊆' ... . }
  moreover have A ⊆ above ao using xxs-above yys-above x' y'
    by(auto simp add: A-def)
  ultimately show ?rhs by blast
qed
qed
thus ?thesis using True by simp
qed
thus ?thesis using ⟨x = y⟩ by simp
qed
qed
qed
qed }
from this[of None]
show ?thesis by(simp)
qed

```

lemma proper-interval-Compl-set-aux:

assumes fin: finite (UNIV :: 'a set)

and xs: sorted xs distinct xs

and ys: sorted ys distinct ys

shows proper-interval-Compl-set-aux None xs ys \longleftrightarrow $(\exists A. - \text{set } xs \sqsubseteq' A \wedge A \sqsubseteq' \text{set } ys)$

proof –

note [simp] = finite-subset[OF subset-UNIV fin]

define above **where** above = case-option UNIV (Collect o less)

have above-simps [simp]: above None = UNIV $\wedge x. \text{above } (Some\ x) = \{y. x < y\}$

and above-upclosed: $\wedge x\ y\ ao. \llbracket x \in \text{above } ao; x < y \rrbracket \implies y \in \text{above } ao$

and proper-interval-Some2: $\wedge x\ ao. \text{proper-interval } ao\ (Some\ x) \longleftrightarrow (\exists z \in \text{above } ao. z < x)$

```

by(simp-all add: proper-interval-simps above-def split: option.splits)

{ fix ao n
  assume set xs  $\subseteq$  above ao    set ys  $\subseteq$  above ao
  from xs  $\langle$ set xs  $\subseteq$  above ao $\rangle$  ys  $\langle$ set ys  $\subseteq$  above ao $\rangle$ 
  have proper-interval-Compl-set-aux ao xs ys  $\longleftrightarrow$ 
    ( $\exists A. \neg$  set xs  $\cap$  above ao  $\sqsubset'$  A  $\cap$  above ao  $\wedge$  A  $\cap$  above ao  $\sqsubset'$  set ys  $\cap$ 
above ao)
  proof(induction ao xs ys rule: proper-interval-Compl-set-aux.induct)
    case (2-1 ao ys)
    { fix A
      assume above ao  $\sqsubset'$  A  $\cap$  above ao
      also have ...  $\subseteq$  above ao by simp
      hence A  $\cap$  above ao  $\sqsubseteq'$  above ao
      by(auto intro: subset-finite-imp-set-less-eq-aux)
      finally have False by simp }
    thus ?case by auto
  next
  case (2-2 ao xs) thus ?case by simp
  next
  case (1 ao x xs y ys)
  note xxs =  $\langle$ sorted (x # xs) $\rangle$   $\langle$ distinct (x # xs) $\rangle$ 
  hence xs: sorted xs    distinct xs and x-Min:  $\forall x' \in$  set xs. x < x'
    by(auto simp add: less-le)
  note yys =  $\langle$ sorted (y # ys) $\rangle$   $\langle$ distinct (y # ys) $\rangle$ 
  hence ys: sorted ys    distinct ys and y-Min:  $\forall y' \in$  set ys. y < y'
    by(auto simp add: less-le)
  note xxs-above =  $\langle$ set (x # xs)  $\subseteq$  above ao $\rangle$ 
  note yys-above =  $\langle$ set (y # ys)  $\subseteq$  above ao $\rangle$ 

  show ?case
  proof(cases x < y)
    case True
    have  $\neg$  proper-interval ao (Some x)  $\wedge$  proper-interval-Compl-set-aux (Some
x) xs (y # ys)  $\longleftrightarrow$ 
      ( $\exists A. \neg$  set (x # xs)  $\cap$  above ao  $\sqsubset'$  A  $\cap$  above ao  $\wedge$  A  $\cap$  above ao  $\sqsubset'$ 
set (y # ys)  $\cap$  above ao)
      (is ?lhs  $\longleftrightarrow$  ?rhs)
    proof(cases proper-interval ao (Some x))
      case True
      then obtain z where z: z < x    z  $\in$  above ao
        by(auto simp add: proper-interval-Some2)
      hence nempty: above ao  $\neq$  {} by auto
      with z have Min (above ao)  $\leq$  z by auto
      hence Min (above ao) < x using  $\langle$ z < x $\rangle$  by(rule le-less-trans)
      hence set (y # ys)  $\cap$  above ao  $\sqsubset'$   $\neg$  set (x # xs)  $\cap$  above ao
        using y-Min x-Min z  $\langle$ x < y $\rangle$ 
      by(fastforce simp add: set-less-aux-def nempty intro!: Min-eqI bexI[where
x=Min (above ao)])

```

```

thus ?thesis using True by(auto dest: set-less-aux-trans set-less-aux-antisym)
next
  case False
  hence above-eq: above ao = insert x (above (Some x))
    using xxs-above by(auto simp add: proper-interval-Some2 intro:
above-upclosed)
  from x-Min have xs-above: set xs  $\subseteq$  above (Some x) by auto
  from  $\langle x < y \rangle$  y-Min have ys-above: set (y # ys)  $\subseteq$  above (Some x) by
auto

  have eq-xs:  $\neg$  set xs  $\cap$  above (Some x) =  $\neg$  set (x # xs)  $\cap$  above ao
    using above-eq by auto
  have eq-ys: set (y # ys)  $\cap$  above (Some x) = set (y # ys)  $\cap$  above ao
    using y-Min  $\langle x < y \rangle$  xxs-above by(auto intro: above-upclosed)

  from  $\langle x < y \rangle$  xs xs-above yys ys-above
  have proper-interval-Compl-set-aux (Some x) xs (y # ys)  $\longleftrightarrow$ 
    ( $\exists A. \neg$  set xs  $\cap$  above (Some x)  $\sqsubset'$  A  $\cap$  above (Some x)  $\wedge$ 
      A  $\cap$  above (Some x)  $\sqsubset'$  set (y # ys)  $\cap$  above (Some x))
    by(rule 1.IH)
  also have ...  $\longleftrightarrow$  ?rhs (is ?lhs  $\longleftrightarrow$  -)
  proof
    assume ?lhs
    then obtain A where  $\neg$  set xs  $\cap$  above (Some x)  $\sqsubset'$  A  $\cap$  above (Some
x)

      and A  $\cap$  above (Some x)  $\sqsubset'$  set (y # ys)  $\cap$  above (Some x) by blast
      moreover have A  $\cap$  above (Some x) = (A - {x})  $\cap$  above ao
        using above-eq by auto
      ultimately have  $\neg$  set (x # xs)  $\cap$  above ao  $\sqsubset'$  (A - {x})  $\cap$  above ao
        and (A - {x})  $\cap$  above ao  $\sqsubset'$  set (y # ys)  $\cap$  above ao
        using eq-xs eq-ys by simp-all
      thus ?rhs by blast
    next
      assume ?rhs
      then obtain A where  $\neg$  set (x # xs)  $\cap$  above ao  $\sqsubset'$  A  $\cap$  above ao
        and A-less: A  $\cap$  above ao  $\sqsubset'$  set (y # ys)  $\cap$  above ao by blast
      moreover have x  $\notin$  A
      proof
        assume x  $\in$  A
        hence set (y # ys)  $\cap$  above ao  $\sqsubset'$  A  $\cap$  above ao
          using y-Min  $\langle x < y \rangle$  by(auto simp add: above-eq set-less-aux-def
intro!: bexI[where x=x])
        with A-less show False by(auto dest: set-less-aux-antisym)
      qed
      hence A  $\cap$  above ao = A  $\cap$  above (Some x) using above-eq by auto
      ultimately show ?lhs using eq-xs eq-ys by auto
    qed
  finally show ?thesis using False by simp
qed

```

```

thus ?thesis using True by simp
next
case False
show ?thesis
proof(cases  $y < x$ )
  case True
  show ?thesis (is ?lhs  $\longleftrightarrow$  ?rhs)
  proof(cases proper-interval ao (Some y))
    case True
    then obtain z where  $z < y$   $z \in \text{above } ao$ 
      by(auto simp add: proper-interval-Some2)
    hence nempty: above ao  $\neq \{\}$  by auto
    with z have Min (above ao)  $\leq z$  by auto
    hence Min (above ao)  $< y$  using  $\langle z < y \rangle$  by(rule le-less-trans)
    hence set (y # ys)  $\cap$  above ao  $\sqsubset'$  - set (x # xs)  $\cap$  above ao
      using y-Min x-Min z  $\langle y < x \rangle$ 
    by(fastforce simp add: set-less-aux-def nempty intro!: Min-eqI bexI[where
 $x = \text{Min}(\text{above } ao)$ ])
    thus ?thesis using True  $\langle y < x \rangle$  by(auto dest: set-less-aux-trans
set-less-aux-antisym)
  next
  case False
  hence above-eq: above ao = insert y (above (Some y))
    using yys-above by(auto simp add: proper-interval-Some2 intro:
above-upclosed)
  from y-Min have ys-above: set ys  $\subseteq$  above (Some y) by auto
  from  $\langle y < x \rangle$  x-Min have xs-above: set (x # xs)  $\subseteq$  above (Some y) by
auto

have  $y \in - \text{set}(x \# xs) \cap \text{above } ao$  using  $\langle y < x \rangle$  x-Min yys-above by
auto

hence nempty: - set (x # xs)  $\cap$  above ao  $\neq \{\}$  by auto
have Min-x: Min (- set (x # xs)  $\cap$  above ao) = y
  using above-eq  $\langle y < x \rangle$  x-Min by(auto intro!: Min-eqI)
have Min-y: Min (set (y # ys)  $\cap$  above ao) = y
  using y-Min above-eq by(auto intro!: Min-eqI)
have eq-xs: - set (x # xs)  $\cap$  above ao - {y} = - set (x # xs)  $\cap$  above
(Some y)
  by(auto simp add: above-eq)
have eq-ys: set ys  $\cap$  above ao - {y} = set ys  $\cap$  above (Some y)
  using y-Min above-eq by auto

from  $\langle \neg x < y \rangle$   $\langle y < x \rangle$  xxs xs-above ys ys-above
have proper-interval-Compl-set-aux (Some y) (x # xs) ys  $\longleftrightarrow$ 
  ( $\exists A. - \text{set}(x \# xs) \cap \text{above}(Some\ y) \sqsubset' A \cap \text{above}(Some\ y) \wedge$ 
 $A \cap \text{above}(Some\ y) \sqsubset' \text{set } ys \cap \text{above}(Some\ y)$ )
  by(rule 1.IH)
also have ...  $\longleftrightarrow$  ?rhs (is ?lhs'  $\longleftrightarrow$  -)
proof

```

assume $?lhs'$
then obtain A **where** $less-A: - set (x \# xs) \cap above (Some y) \sqsubset' A$
 $\cap above (Some y)$
and $A-less: A \cap above (Some y) \sqsubset' set ys \cap above (Some y)$ **by** *blast*
let $?A = insert y A$

have $Min-A: Min (?A \cap above ao) = y$
using *above-eq* **by**(*auto intro!: Min-eqI*)
moreover have $A-eq: A \cap above ao - \{y\} = A \cap above (Some y)$
using *above-eq* **by** *auto*
ultimately have $less-A': - set (x \# xs) \cap above ao \sqsubset' ?A \cap above ao$
using *nempty yys-above less-A Min-x eq-xs* **by**(*subst set-less-aux-rec*)
simp-all

have $A'-less: ?A \cap above ao \sqsubset' set (y \# ys) \cap above ao$
using *yys-above nempty Min-A A-eq A-less Min-y eq-ys*
by(*subst set-less-aux-rec*) *simp-all*

with $less-A'$ **show** $?rhs$ **by** *blast*
next
assume $?rhs$
then obtain A **where** $less-A: - set (x \# xs) \cap above ao \sqsubset' A \cap above$
 ao
and $A-less: A \cap above ao \sqsubset' set (y \# ys) \cap above ao$ **by** *blast*

from $less-A$ **have** $nempty': A \cap above ao \neq \{\}$ **by** *auto*
moreover have $A-eq: A \cap above ao - \{y\} = A \cap above (Some y)$
using *above-eq* **by** *auto*
moreover have $y-in-xs: y \in - set (x \# xs) \cap above ao$
using $\langle y < x \rangle$ *x-Min yys-above* **by** *auto*
moreover have $y \in A$
proof(*rule ccontr*)
assume $y \notin A$
hence $A \cap above ao \sqsubset' - set (x \# xs) \cap above ao$
using $\langle y < x \rangle$ *x-Min y-in-xs*
by(*auto simp add: set-less-aux-def above-eq intro: bexI[where x=y]*)
with $less-A$ **show** *False* **by**(*rule set-less-aux-antisym*)
qed
hence $Min-A: Min (A \cap above ao) = y$ **using** *above-eq y-Min* **by**(*auto*
intro!: Min-eqI)
ultimately have $less-A': - set (x \# xs) \cap above (Some y) \sqsubset' A \cap$
 $above (Some y)$
using *nempty less-A Min-x eq-xs*
by(*subst (asm) set-less-aux-rec*)(*auto dest: bspec[where x=y]*)

have $A'-less: A \cap above (Some y) \sqsubset' set ys \cap above (Some y)$
using $A-less$ *nempty' yys-above Min-A Min-y A-eq eq-ys*
by(*subst (asm) set-less-aux-rec*) *simp-all*
with $less-A'$ **show** $?lhs'$ **by** *blast*

```

    qed
    finally show ?thesis using ⟨¬ x < y⟩ ⟨y < x⟩ False by simp
  qed
next
case False
with ⟨¬ x < y⟩ have x = y by auto
thus ?thesis (is ?lhs ↔ ?rhs)
proof(cases proper-interval ao (Some x))
  case True
  then obtain z where z: z < x    z ∈ above ao
    by(auto simp add: proper-interval-Some2)
  hence empty: above ao ≠ {} by auto
  with z have Min (above ao) ≤ z by auto
  hence Min (above ao) < x using ⟨z < x⟩ by(rule le-less-trans)
  hence set (y # ys) ∩ above ao ⊆' - set (x # xs) ∩ above ao
    using y-Min x-Min z ⟨x = y⟩
  by(fastforce simp add: set-less-aux-def empty intro!: Min-eqI bexI[where
x=Min (above ao)])
  thus ?thesis using True ⟨x = y⟩ by(auto dest: set-less-aux-trans
set-less-aux-antisym)
  next
  case False
  hence above-eq: above ao = insert x (above (Some x))
    using xs-above by(auto simp add: proper-interval-Some2 intro:
above-upclosed)

  have (ys = [] → xs ≠ []) ↔ ?rhs (is ?lhs' ↔ -)
  proof(intro iffI strip notI)
    assume ?lhs'
    show ?rhs
    proof(cases ys)
      case Nil
      with ⟨?lhs'⟩ obtain x' xs' where xs-eq: xs = x' # xs'
        by(auto simp add: neq-Nil-conv)
      with xs have x'-Min: ∀ x'' ∈ set xs'. x' < x''
        by(auto simp add: less-le)
      let ?A = - set (x # xs')
      have - set (x # xs) ∩ above ao ⊆ ?A ∩ above ao
        using xs-eq by auto
      moreover have x' ∉ - set (x # xs) ∩ above ao    x' ∈ ?A ∩ above ao
        using xs-eq xs-above x'-Min x-Min by auto
      ultimately have - set (x # xs) ∩ above ao ⊂ ?A ∩ above ao
        by blast
      hence - set (x # xs) ∩ above ao ⊆' ...
        by(fastforce intro: psubset-finite-imp-set-less-aux)
      moreover have ... ⊆' set (y # ys) ∩ above ao
        using Nil ⟨x = y⟩ by(auto simp add: set-less-aux-def above-eq)
      ultimately show ?thesis by blast
    next

```

```

    case (Cons y' ys')
    let ?A = {y}
    have - set (x # xs) ∩ above ao ⊆' ?A ∩ above ao
      using ⟨x = y⟩ x-Min by(auto simp add: set-less-aux-def above-eq)
    moreover have ... ⊆ set (y # ys) ∩ above ao
      using yys-above yys Cons by auto
    hence ?A ∩ above ao ⊆' ... by(fastforce intro: psubset-finite-imp-set-less-aux)
    ultimately show ?thesis by blast
  qed
next
  assume Nil: ys = [] xs = [] and ?rhs
  then obtain A where less-A: - {x} ∩ above ao ⊆' A ∩ above ao
    and A-less: A ∩ above ao ⊆' {x} using ⟨x = y⟩ above-eq by auto
  have x ∉ A using A-less by(auto simp add: set-less-aux-def above-eq)
  hence A ∩ above ao ⊆ - {x} ∩ above ao by auto
  hence A ∩ above ao ⊆' ... by(auto intro: subset-finite-imp-set-less-eq-aux)
  with less-A have ... ⊆' ... by(rule set-less-trans-set-less-eq)
  thus False by simp
  qed
  with ⟨x = y⟩ False show ?thesis by simp
  qed
  qed
  qed
  qed }
  from this[of None] show ?thesis by simp
  qed
end

```

2.6.4 Proper intervals for HOL types

```

instantiation unit :: proper-interval begin
fun proper-interval-unit :: unit proper-interval where
  proper-interval-unit None None = True
| proper-interval-unit - - = False
instance by intro-classes auto
end

```

```

instantiation bool :: proper-interval begin
fun proper-interval-bool :: bool proper-interval where
  proper-interval-bool (Some x) (Some y) ⟷ False
| proper-interval-bool (Some x) None ⟷ ¬ x
| proper-interval-bool None (Some y) ⟷ y
| proper-interval-bool None None = True
instance by intro-classes auto
end

```

```

instantiation nat :: proper-interval begin
fun proper-interval-nat :: nat proper-interval where

```

```

  proper-interval-nat no None = True
| proper-interval-nat None (Some x)  $\longleftrightarrow$   $x > 0$ 
| proper-interval-nat (Some x) (Some y)  $\longleftrightarrow$   $y - x > 1$ 
instance by intro-classes auto
end

instantiation int :: proper-interval begin
fun proper-interval-int :: int proper-interval where
  proper-interval-int (Some x) (Some y)  $\longleftrightarrow$   $y - x > 1$ 
| proper-interval-int - - = True
instance by intro-classes (auto intro: less-add-one, metis less-add-one minus-less-iff)
end

instantiation integer :: proper-interval begin
context includes integer.lifting begin
lift-definition proper-interval-integer :: integer proper-interval is proper-interval
.
instance by(intro-classes)(transfer, simp only: proper-interval-simps)+
end
lemma proper-interval-integer-simps [code]:
  includes integer.lifting fixes x y :: integer and xo yo :: integer option shows
  proper-interval (Some x) (Some y) =  $(1 < y - x)$ 
  proper-interval None yo = True
  proper-interval xo None = True
by(transfer, simp)+

instantiation natural :: proper-interval begin
context includes natural.lifting begin
lift-definition proper-interval-natural :: natural proper-interval is proper-interval
.
instance by(intro-classes)(transfer, simp only: proper-interval-simps)+
end
lemma proper-interval-natural-simps [code]:
  includes natural.lifting fixes x y :: natural and xo :: natural option shows
  proper-interval xo None = True
  proper-interval None (Some y)  $\longleftrightarrow$   $y > 0$ 
  proper-interval (Some x) (Some y)  $\longleftrightarrow$   $y - x > 1$ 
by(transfer, simp)+

lemma char-less-iff-nat-of-char:  $x < y \longleftrightarrow$  of-char  $x < (of-char y :: nat)$ 
  by (fact less-char-def)

lemma nat-of-char-inject [simp]: of-char  $x = (of-char y :: nat) \longleftrightarrow x = y$ 
  by (fact of-char-eq-iff)

lemma char-le-iff-nat-of-char:  $x \leq y \longleftrightarrow$  of-char  $x \leq (of-char y :: nat)$ 
  by (fact less-eq-char-def)

```

```

instantiation char :: proper-interval
begin

fun proper-interval-char :: char proper-interval where
  proper-interval-char None None  $\longleftrightarrow$  True
| proper-interval-char None (Some x)  $\longleftrightarrow$  x  $\neq$  CHR 0x00
| proper-interval-char (Some x) None  $\longleftrightarrow$  x  $\neq$  CHR 0xFF
| proper-interval-char (Some x) (Some y)  $\longleftrightarrow$  of-char y - of-char x > (1 :: nat)

instance proof
  fix y :: char
  { assume y  $\neq$  CHR 0x00
    have CHR 0x00 < y
    proof (rule ccontr)
      assume  $\neg$  CHR 0x00 < y
      then have of-char y = (of-char CHR 0x00 :: nat)
        by (simp add: not-less-char-le-iff-nat-of-char)
      then have y = CHR 0x00
        using nat-of-char-inject [of y CHR 0x00] by simp
      with  $\langle$ y  $\neq$  CHR 0x00 $\rangle$  show False
      by simp
    qed }
  moreover
  { fix z :: char
    assume z < CHR 0x00
    hence False
    by (simp add: char-less-iff-nat-of-char of-char-eq-iff [symmetric]) }
  ultimately show proper-interval None (Some y) = ( $\exists$  z. z < y)
  by auto

  fix x :: char
  { assume x  $\neq$  CHR 0xFF
    then have x < CHR 0xFF
      by (auto simp add: neq-iff-char-less-iff-nat-of-char)
      (insert nat-of-char-less-256 [of x], simp)
    hence  $\exists$  z. x < z .. }
  moreover
  { fix z :: char
    assume CHR 0xFF < z
    hence False
    by (simp add: char-less-iff-nat-of-char)
    (insert nat-of-char-less-256 [of z], simp) }
  ultimately show proper-interval (Some x) None = ( $\exists$  z. x < z) by auto

  { assume gt: of-char y - of-char x > (1 :: nat)
    let ?z = char-of (of-char x + (1 :: nat))
    from gt nat-of-char-less-256 [of y]
    have 255: of-char x < (255 :: nat) by arith
  }

```

```

with gt have  $x < ?z \quad ?z < y$ 
  by (simp-all add: char-less-iff-nat-of-char)
  hence  $\exists z. x < z \wedge z < y$  by blast }
moreover
{ fix z
  assume  $x < z \quad z < y$ 
  hence  $(1 :: \text{nat}) < \text{of-char } y - \text{of-char } x$ 
  by (simp add: char-less-iff-nat-of-char) }
ultimately show proper-interval (Some x) (Some y) =  $(\exists z > x. z < y)$ 
  by auto
qed simp

```

end

```

instantiation Enum.finite-1 :: proper-interval begin
definition proper-interval-finite-1 :: Enum.finite-1 proper-interval
where proper-interval-finite-1 x y  $\longleftrightarrow x = \text{None} \wedge y = \text{None}$ 
instance by intro-classes (simp-all add: proper-interval-finite-1-def less-finite-1-def)
end

```

```

instantiation Enum.finite-2 :: proper-interval begin
fun proper-interval-finite-2 :: Enum.finite-2 proper-interval where
  proper-interval-finite-2 None None  $\longleftrightarrow \text{True}$ 
| proper-interval-finite-2 None (Some x)  $\longleftrightarrow x = \text{finite-2.a}_2$ 
| proper-interval-finite-2 (Some x) None  $\longleftrightarrow x = \text{finite-2.a}_1$ 
| proper-interval-finite-2 (Some x) (Some y)  $\longleftrightarrow \text{False}$ 
instance by intro-classes (auto simp add: less-finite-2-def)
end

```

```

instantiation Enum.finite-3 :: proper-interval begin
fun proper-interval-finite-3 :: Enum.finite-3 proper-interval where
  proper-interval-finite-3 None None  $\longleftrightarrow \text{True}$ 
| proper-interval-finite-3 None (Some x)  $\longleftrightarrow x \neq \text{finite-3.a}_1$ 
| proper-interval-finite-3 (Some x) None  $\longleftrightarrow x \neq \text{finite-3.a}_3$ 
| proper-interval-finite-3 (Some x) (Some y)  $\longleftrightarrow x = \text{finite-3.a}_1 \wedge y = \text{finite-3.a}_3$ 
instance
proof
  fix x y :: Enum.finite-3
  show proper-interval None (Some y) =  $(\exists z. z < y)$ 
    by(cases y)(auto simp add: less-finite-3-def split: finite-3.split)
  show proper-interval (Some x) None =  $(\exists z. x < z)$ 
    by(cases x)(auto simp add: less-finite-3-def)
  show proper-interval (Some x) (Some y) =  $(\exists z > x. z < y)$ 
    by(auto simp add: less-finite-3-def split: finite-3.split-asm)
qed simp
end

```

2.6.5 List fusion for the order and proper intervals on $'a$ set

definition *length-last-fusion* :: ($'a$, $'s$) generator $\Rightarrow 's \Rightarrow \text{nat} \times 'a$
where *length-last-fusion* g s = *length-last* (*list.unfoldr* g s)

lemma *length-last-fusion-code* [*code*]:

length-last-fusion g s =
 (if *list.has-next* g s then
 let (x, s') = *list.next* g s
 in *fold-fusion* g (λx ($n, -$). ($n + 1, x$)) s' ($1, x$)
 else ($0, \text{undefined}$))

unfolding *length-last-fusion-def*

by(*subst list.unfoldr.simps*)(*simp add: length-last-Nil length-last-Cons-code fold-fusion-def split-beta*)

declare *length-last-fusion-def* [*symmetric, code-unfold*]

context *proper-introl* **begin**

definition *set-less-eq-aux-Compl-fusion* :: ($'a$, $'s1$) generator \Rightarrow ($'a$, $'s2$) generator
 $\Rightarrow 'a$ option $\Rightarrow 's1 \Rightarrow 's2 \Rightarrow \text{bool}$

where

set-less-eq-aux-Compl-fusion $g1$ $g2$ ao $s1$ $s2$ =
set-less-eq-aux-Compl ao (*list.unfoldr* $g1$ $s1$) (*list.unfoldr* $g2$ $s2$)

definition *Compl-set-less-eq-aux-fusion* :: ($'a$, $'s1$) generator \Rightarrow ($'a$, $'s2$) generator
 $\Rightarrow 'a$ option $\Rightarrow 's1 \Rightarrow 's2 \Rightarrow \text{bool}$

where

Compl-set-less-eq-aux-fusion $g1$ $g2$ ao $s1$ $s2$ =
Compl-set-less-eq-aux ao (*list.unfoldr* $g1$ $s1$) (*list.unfoldr* $g2$ $s2$)

definition *set-less-aux-Compl-fusion* :: ($'a$, $'s1$) generator \Rightarrow ($'a$, $'s2$) generator
 $\Rightarrow 'a$ option $\Rightarrow 's1 \Rightarrow 's2 \Rightarrow \text{bool}$

where

set-less-aux-Compl-fusion $g1$ $g2$ ao $s1$ $s2$ =
set-less-aux-Compl ao (*list.unfoldr* $g1$ $s1$) (*list.unfoldr* $g2$ $s2$)

definition *Compl-set-less-aux-fusion* :: ($'a$, $'s1$) generator \Rightarrow ($'a$, $'s2$) generator
 $\Rightarrow 'a$ option $\Rightarrow 's1 \Rightarrow 's2 \Rightarrow \text{bool}$

where

Compl-set-less-aux-fusion $g1$ $g2$ ao $s1$ $s2$ =
Compl-set-less-aux ao (*list.unfoldr* $g1$ $s1$) (*list.unfoldr* $g2$ $s2$)

definition *exhaustive-above-fusion* :: ($'a$, $'s$) generator $\Rightarrow 'a \Rightarrow 's \Rightarrow \text{bool}$

where *exhaustive-above-fusion* g a s = *exhaustive-above* a (*list.unfoldr* g s)

definition *exhaustive-fusion* :: ($'a$, $'s$) generator $\Rightarrow 's \Rightarrow \text{bool}$

where *exhaustive-fusion* g s = *exhaustive* (*list.unfoldr* g s)

definition *proper-interval-set-aux-fusion* :: ($'a$, $'s1$) generator \Rightarrow ($'a$, $'s2$) genera-

$tor \Rightarrow 's1 \Rightarrow 's2 \Rightarrow bool$

where

$proper_interval_set_aux_fusion\ g1\ g2\ s1\ s2 =$
 $proper_interval_set_aux\ (list.unfoldr\ g1\ s1)\ (list.unfoldr\ g2\ s2)$

definition $proper_interval_set_Compl_aux_fusion ::$

$('a, 's1)\ generator \Rightarrow ('a, 's2)\ generator \Rightarrow 'a\ option \Rightarrow nat \Rightarrow 's1 \Rightarrow 's2 \Rightarrow bool$

where

$proper_interval_set_Compl_aux_fusion\ g1\ g2\ ao\ n\ s1\ s2 =$
 $proper_interval_set_Compl_aux\ ao\ n\ (list.unfoldr\ g1\ s1)\ (list.unfoldr\ g2\ s2)$

definition $proper_interval_Compl_set_aux_fusion ::$

$('a, 's1)\ generator \Rightarrow ('a, 's2)\ generator \Rightarrow 'a\ option \Rightarrow 's1 \Rightarrow 's2 \Rightarrow bool$

where

$proper_interval_Compl_set_aux_fusion\ g1\ g2\ ao\ s1\ s2 =$
 $proper_interval_Compl_set_aux\ ao\ (list.unfoldr\ g1\ s1)\ (list.unfoldr\ g2\ s2)$

lemma $set_less_eq_aux_Compl_fusion_code:$

$set_less_eq_aux_Compl_fusion\ g1\ g2\ ao\ s1\ s2 \longleftrightarrow$
 $(list.has_next\ g1\ s1 \longrightarrow list.has_next\ g2\ s2 \longrightarrow$
 $(let\ (x, s1') = list.next\ g1\ s1;$
 $(y, s2') = list.next\ g2\ s2$
 $in\ if\ x < y\ then\ proper_interval\ ao\ (Some\ x) \vee set_less_eq_aux_Compl_fusion\ g1$
 $g2\ (Some\ x)\ s1'\ s2$
 $else\ if\ y < x\ then\ proper_interval\ ao\ (Some\ y) \vee set_less_eq_aux_Compl_fusion$
 $g1\ g2\ (Some\ y)\ s1\ s2'$
 $else\ proper_interval\ ao\ (Some\ y)))$

unfolding $set_less_eq_aux_Compl_fusion_def$

by($subst\ (1\ 2\ 4\ 5)\ list.unfoldr.simps$)($simp\ add: split-beta$)

lemma $Compl_set_less_eq_aux_fusion_code:$

$Compl_set_less_eq_aux_fusion\ g1\ g2\ ao\ s1\ s2 \longleftrightarrow$
 $(if\ list.has_next\ g1\ s1\ then$
 $let\ (x, s1') = list.next\ g1\ s1$
 $in\ if\ list.has_next\ g2\ s2\ then$
 $let\ (y, s2') = list.next\ g2\ s2$
 $in\ if\ x < y\ then\ \neg\ proper_interval\ ao\ (Some\ x) \wedge Compl_set_less_eq_aux_fusion$
 $g1\ g2\ (Some\ x)\ s1'\ s2$
 $else\ if\ y < x\ then\ \neg\ proper_interval\ ao\ (Some\ y) \wedge Compl_set_less_eq_aux_fusion$
 $g1\ g2\ (Some\ y)\ s1\ s2'$
 $else\ \neg\ proper_interval\ ao\ (Some\ y)$
 $else\ \neg\ proper_interval\ ao\ (Some\ x) \wedge Compl_set_less_eq_aux_fusion\ g1\ g2$
 $(Some\ x)\ s1'\ s2$
 $else\ if\ list.has_next\ g2\ s2\ then$
 $let\ (y, s2') = list.next\ g2\ s2$
 $in\ \neg\ proper_interval\ ao\ (Some\ y) \wedge Compl_set_less_eq_aux_fusion\ g1\ g2\ (Some$
 $y)\ s1\ s2'$
 $else\ \neg\ proper_interval\ ao\ None)$

unfolding *Compl-set-less-eq-aux-fusion-def*

by(subst (1 2 4 5 8 9) list.unfoldr.simps)(simp add: split-beta)

lemma *set-less-aux-Compl-fusion-code:*

set-less-aux-Compl-fusion $g1\ g2\ ao\ s1\ s2 \longleftrightarrow$
 (if list.has-next $g1\ s1$ then
 let $(x, s1')$ = list.next $g1\ s1$
 in if list.has-next $g2\ s2$ then
 let $(y, s2')$ = list.next $g2\ s2$
 in if $x < y$ then proper-interval $ao\ (Some\ x) \vee$ *set-less-aux-Compl-fusion*
 $g1\ g2\ (Some\ x)\ s1'\ s2$
 else if $y < x$ then proper-interval $ao\ (Some\ y) \vee$ *set-less-aux-Compl-fusion*
 $g1\ g2\ (Some\ y)\ s1\ s2'$
 else proper-interval $ao\ (Some\ y)$
 else proper-interval $ao\ (Some\ x) \vee$ *set-less-aux-Compl-fusion* $g1\ g2\ (Some$
 $x)\ s1'\ s2$
 else if list.has-next $g2\ s2$ then
 let $(y, s2')$ = list.next $g2\ s2$
 in proper-interval $ao\ (Some\ y) \vee$ *set-less-aux-Compl-fusion* $g1\ g2\ (Some\ y)\ s1$
 $s2'$
 else proper-interval $ao\ None$)

unfolding *set-less-aux-Compl-fusion-def*

by(subst (1 2 4 5 8 9) list.unfoldr.simps)(simp add: split-beta)

lemma *Compl-set-less-aux-fusion-code:*

Compl-set-less-aux-fusion $g1\ g2\ ao\ s1\ s2 \longleftrightarrow$
 list.has-next $g1\ s1 \wedge$ list.has-next $g2\ s2 \wedge$
 (let $(x, s1')$ = list.next $g1\ s1$;
 $(y, s2')$ = list.next $g2\ s2$
 in if $x < y$ then \neg proper-interval $ao\ (Some\ x) \wedge$ *Compl-set-less-aux-fusion* $g1$
 $g2\ (Some\ x)\ s1'\ s2$
 else if $y < x$ then \neg proper-interval $ao\ (Some\ y) \wedge$ *Compl-set-less-aux-fusion*
 $g1\ g2\ (Some\ y)\ s1\ s2'$
 else \neg proper-interval $ao\ (Some\ y)$)

unfolding *Compl-set-less-aux-fusion-def*

by(subst (1 2 4 5) list.unfoldr.simps)(simp add: split-beta)

lemma *exhaustive-above-fusion-code:*

exhaustive-above-fusion $g\ y\ s \longleftrightarrow$
 (if list.has-next $g\ s$ then
 let (x, s') = list.next $g\ s$
 in \neg proper-interval $(Some\ y)\ (Some\ x) \wedge$ *exhaustive-above-fusion* $g\ x\ s'$
 else \neg proper-interval $(Some\ y)\ None$)

unfolding *exhaustive-above-fusion-def*

by(subst list.unfoldr.simps)(simp add: split-beta)

lemma *exhaustive-fusion-code:*

exhaustive-fusion $g\ s =$
 (list.has-next $g\ s \wedge$

(let (x, s') = list.next g s
in \neg proper-interval None (Some x) \wedge exhaustive-above-fusion g x s')

unfolding exhaustive-fusion-def exhaustive-above-fusion-def
by(subst (1) list.unfoldr.simps)(simp add: split-beta)

lemma proper-interval-set-aux-fusion-code:
proper-interval-set-aux-fusion g1 g2 s1 s2 \longleftrightarrow
list.has-next g2 s2 \wedge
(let (y, s2') = list.next g2 s2
in if list.has-next g1 s1 then
let (x, s1') = list.next g1 s1
in if x < y then False
else if y < x then proper-interval (Some y) (Some x) \vee list.has-next g2
s2' \vee \neg exhaustive-above-fusion g1 x s1'
else proper-interval-set-aux-fusion g1 g2 s1' s2'
else list.has-next g2 s2' \vee proper-interval (Some y) None)

unfolding proper-interval-set-aux-fusion-def exhaustive-above-fusion-def
by(subst (1 2) list.unfoldr.simps)(simp add: split-beta)

lemma proper-interval-set-Compl-aux-fusion-code:
proper-interval-set-Compl-aux-fusion g1 g2 ao n s1 s2 \longleftrightarrow
(if list.has-next g1 s1 then
let (x, s1') = list.next g1 s1
in if list.has-next g2 s2 then
let (y, s2') = list.next g2 s2
in if x < y then
proper-interval ao (Some x) \vee
proper-interval-set-Compl-aux-fusion g1 g2 (Some x) (n + 1) s1' s2'
else if y < x then
proper-interval ao (Some y) \vee
proper-interval-set-Compl-aux-fusion g1 g2 (Some y) (n + 1) s1 s2'
else
proper-interval ao (Some x) \wedge
(let m = CARD('a) - n
in m - length-fusion g2 s2' \neq 2 \vee m - length-fusion g1 s1' \neq 2)
else
let m = CARD('a) - n; (len-x, x') = length-last-fusion g1 s1
in m \neq len-x \wedge (m = len-x + 1 \longrightarrow \neg proper-interval (Some x') None)

else if list.has-next g2 s2 then
let (y, s2') = list.next g2 s2;
m = CARD('a) - n;
(len-y, y') = length-last-fusion g2 s2
in m \neq len-y \wedge (m = len-y + 1 \longrightarrow \neg proper-interval (Some y') None)
else CARD('a) > n + 1)

unfolding proper-interval-set-Compl-aux-fusion-def length-last-fusion-def length-fusion-def
by(subst (1 2 4 5 9 10) list.unfoldr.simps)(simp add: split-beta)

lemma proper-interval-Compl-set-aux-fusion-code:

```

proper-interval-Compl-set-aux-fusion g1 g2 ao s1 s2  $\longleftrightarrow$ 
  list.has-next g1 s1  $\wedge$  list.has-next g2 s2  $\wedge$ 
  (let (x, s1') = list.next g1 s1;
       (y, s2') = list.next g2 s2
   in if x < y then
        $\neg$  proper-interval ao (Some x)  $\wedge$  proper-interval-Compl-set-aux-fusion g1 g2
     (Some x) s1' s2
     else if y < x then
        $\neg$  proper-interval ao (Some y)  $\wedge$  proper-interval-Compl-set-aux-fusion g1 g2
     (Some y) s1 s2'
     else  $\neg$  proper-interval ao (Some x)  $\wedge$  (list.has-next g2 s2'  $\vee$  list.has-next g1
s1'))
unfolding proper-interval-Compl-set-aux-fusion-def
by(subst (1 2 4 5) list.unfoldr.simps)(auto simp add: split-beta)

end

```

lemmas [code] =

```

set-less-eq-aux-Compl-fusion-code proper-intrvl.set-less-eq-aux-Compl-fusion-code
Compl-set-less-eq-aux-fusion-code proper-intrvl.Compl-set-less-eq-aux-fusion-code
set-less-aux-Compl-fusion-code proper-intrvl.set-less-aux-Compl-fusion-code
Compl-set-less-aux-fusion-code proper-intrvl.Compl-set-less-aux-fusion-code
exhaustive-above-fusion-code proper-intrvl.exhaustive-above-fusion-code
exhaustive-fusion-code proper-intrvl.exhaustive-fusion-code
proper-interval-set-aux-fusion-code proper-intrvl.proper-interval-set-aux-fusion-code
proper-interval-set-Compl-aux-fusion-code proper-intrvl.proper-interval-set-Compl-aux-fusion-code
proper-interval-Compl-set-aux-fusion-code proper-intrvl.proper-interval-Compl-set-aux-fusion-code

```

lemmas [symmetric, code-unfold] =

```

set-less-eq-aux-Compl-fusion-def proper-intrvl.set-less-eq-aux-Compl-fusion-def
Compl-set-less-eq-aux-fusion-def proper-intrvl.Compl-set-less-eq-aux-fusion-def
set-less-aux-Compl-fusion-def proper-intrvl.set-less-aux-Compl-fusion-def
Compl-set-less-aux-fusion-def proper-intrvl.Compl-set-less-aux-fusion-def
exhaustive-above-fusion-def proper-intrvl.exhaustive-above-fusion-def
exhaustive-fusion-def proper-intrvl.exhaustive-fusion-def
proper-interval-set-aux-fusion-def proper-intrvl.proper-interval-set-aux-fusion-def
proper-interval-set-Compl-aux-fusion-def proper-intrvl.proper-interval-set-Compl-aux-fusion-def
proper-interval-Compl-set-aux-fusion-def proper-intrvl.proper-interval-Compl-set-aux-fusion-def

```

2.6.6 Drop notation

context ord begin

```

no-notation set-less-aux (infix <□''> 50)
and set-less-eq-aux (infix <□''> 50)
and set-less-eq-aux' (infix <□''''> 50)
and set-less-eq-aux'' (infix <□''''''> 50)
and set-less-eq (infix <□> 50)
and set-less (infix <□> 50)

```

end

end

theory *Containers-Generator*

imports

Deriving.Generator-Aux

Deriving.Derive-Manager

HOL-Library.Phantom-Type

Containers-Auxiliary

begin

2.6.7 Introduction

In the following, we provide generators for the major classes of the container framework: `ceq`, `corder`, `cenum`, `set-impl`, and `mapping-impl`.

In this file we provide some common infrastructure on the ML-level which will be used by the individual generators.

ML-file *<containers-generator.ML>*

end

theory *Collection-Order*

imports

Set-Linorder

Containers-Generator

Deriving.Compare-Instances

begin

Chapter 3

Light-weight containers

3.1 A linear order for code generation

3.1.1 Optional comparators

```
class ccompare =
  fixes ccompare :: 'a comparator option
  assumes ccompare:  $\bigwedge$  comp. ccompare = Some comp  $\implies$  comparator comp
begin
  abbreviation ccomp :: 'a comparator where ccomp  $\equiv$  the (ID ccompare)
  abbreviation cless :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool where cless  $\equiv$  lt-of-comp (the (ID ccompare))
  abbreviation cless-eq :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool where cless-eq  $\equiv$  le-of-comp (the (ID ccompare))
end

lemma (in ccompare) ID-ccompare':
   $\bigwedge$  c. ID ccompare = Some c  $\implies$  comparator c
  unfolding ID-def id-apply using ccompare by simp

lemma (in ccompare) ID-ccompare:
   $\bigwedge$  c. ID ccompare = Some c  $\implies$  class.linorder (le-of-comp c) (lt-of-comp c)
  by (rule comparator.linorder[OF ID-ccompare'])

syntax -CCOMPARE :: type  $\implies$  logic ( $\langle(1CCOMPARE/(1'(-'))\rangle$ )

syntax-consts -CCOMPARE == ccompare

parse-translation  $\langle$ 
let
  fun ccompare-tr [ty] =
    (Syntax.const @ {syntax-const -constrain} $ Syntax.const @ {const-syntax ccompare})
$
  (Syntax.const @ {type-syntax option} $
   (Syntax.const @ {type-syntax fun} $ ty $
```

```

      (Syntax.const @{{type-syntax fun} $ ty $ Syntax.const @{{type-syntax
order}}}))
    | ccompare-tr ts = raise TERM (ccompare-tr, ts);
in [(@{{syntax-const -CCOMPARE}}, K ccompare-tr)] end
>

```

definition *is-ccompare* :: 'a :: ccompare itself \Rightarrow bool
where *is-ccompare* - \longleftrightarrow ID CCOMPARE('a) \neq None

context *compare*

begin

lemma *cless-eq-conv-cless*:

fixes *a b* :: 'a

assumes ID CCOMPARE('a) \neq None

shows *cless-eq a b* \longleftrightarrow *cless a b* \vee *a = b*

proof –

from *assms* **interpret** *linorder cless-eq cless* :: 'a \Rightarrow 'a \Rightarrow bool

by(*clarsimp simp add: ID-ccompare*)

show *?thesis* **by**(*rule le-less*)

qed

end

3.1.2 Generator for the *ccompare*-class

This generator registers itself at the derive-manager for the class *compare*. To be more precise, one can choose whether one does not want to support a comparator by passing parameter "no", one wants to register an arbitrary type which is already in class *compare* using parameter "compare", or one wants to generate a new comparator by passing no parameter. In the last case, one demands that the type is a datatype and that all non-recursive types of that datatype already provide a comparator, which can usually be achieved via "derive comparator type" or "derive compare type".

- instantiation type :: (type,...,type) (no) corder
- instantiation datatype :: (type,...,type) corder
- instantiation datatype :: (compare,...,compare) (compare) corder

If the parameter "no" is not used, then the corresponding *is-ccompare*-theorem is automatically generated and attributed with [*simp*, *code-post*].

To create a new comparator, we just invoke the functionality provided by the generator. The only difference is the boilerplate-code, which for the generator has to perform the class instantiation for a comparator, whereas here we have to invoke the methods to satisfy the corresponding locale for comparators.

This generator can be used for arbitrary types, not just datatypes. When passing no parameters, we get same limitation as for the order generator.

lemma *corder-intro*: *class.linorder le lt* $\implies a = \text{Some } (le, lt) \implies a = \text{Some } (le', lt')$
 \implies
class.linorder le' lt' **by** *auto*

lemma *comparator-subst*: *c1 = c2* \implies *comparator c1* \implies *comparator c2* **by** *blast*

lemma (**in** *compare*) *compare-subst*: \bigwedge *comp. compare = comp* \implies *comparator comp*
using *comparator-compare* **by** *blast*

ML-file \langle *ccompare-generator.ML* \rangle

3.1.3 Instantiations for HOL types

derive (*linorder*) *compare-order*

Enum.finite-1 Enum.finite-2 Enum.finite-3 natural String.literal

derive (*compare*) *ccompare*

unit bool nat int Enum.finite-1 Enum.finite-2 Enum.finite-3 integer natural char String.literal

derive (*no*) *ccompare Enum.finite-4 Enum.finite-5*

derive *ccompare sum list option prod*

derive (*no*) *ccompare fun*

lemma *is-ccompare-fun* [*simp*]: \neg *is-ccompare TYPE('a \Rightarrow 'b)*

by (*simp add: is-ccompare-def ccompare-fun-def ID-None*)

instantiation *set* :: (*compare*) *ccompare* **begin**

definition *CCOMPARE('a set)* =

map-option (λ *c. comp-of-ords* (*ord.set-less-eq* (*le-of-comp c*)) (*ord.set-less* (*le-of-comp c*))) (*ID CCOMPARE('a)*)

instance **by** (*intro-classes*) (*auto simp add: ccompare-set-def intro: comp-of-ords linorder.set-less-eq-linorder ID-ccompare*)

end

lemma *is-ccompare-set* [*simp, code-post*]:

is-ccompare TYPE('a set) \longleftrightarrow *is-ccompare TYPE('a :: ccompare)*

by (*simp add: is-ccompare-def ccompare-set-def ID-def*)

definition *cless-eq-set* :: '*a* :: *ccompare set* \Rightarrow '*a set* \Rightarrow *bool*

where [*simp, code del*]: *cless-eq-set* = *le-of-comp* (*the* (*ID CCOMPARE('a set)*))

definition *cless-set* :: '*a* :: *ccompare set* \Rightarrow '*a set* \Rightarrow *bool*

where [*simp, code del*]: *cless-set* = *lt-of-comp* (*the* (*ID CCOMPARE('a set)*))

```

lemma ccompare-set-code [code]:
  CCOMPARE('a :: ccompare set) =
    (case ID CCOMPARE('a) of None  $\Rightarrow$  None | Some -  $\Rightarrow$  Some (comp-of-ords
cless-eq-set cless-set))
  by (clarsimp simp add: ccompare-set-def ID-Some split: option.split)

```

```

derive (no) ccompare Predicate.pred

```

3.1.4 Proper intervals

```

class cproper-interval = ccompare +
  fixes cproper-interval :: 'a option  $\Rightarrow$  'a option  $\Rightarrow$  bool
  assumes cproper-interval:
    [[ ID CCOMPARE('a)  $\neq$  None; finite (UNIV :: 'a set) ]]
     $\Rightarrow$  class.proper-interval cless cproper-interval
begin

```

```

lemma ID-ccompare-interval:
  [[ ID CCOMPARE('a) = Some c; finite (UNIV :: 'a set) ]]
   $\Rightarrow$  class.linorder-proper-interval (le-of-comp c) (lt-of-comp c) cproper-interval
using cproper-interval
by(simp add: ID-ccompare class.linorder-proper-interval-def)

```

```

end

```

```

instantiation unit :: cproper-interval begin

```

```

definition cproper-interval = (proper-interval :: unit proper-interval)

```

```

instance by intro-classes (simp add: compare-order-class.ord-defs cproper-interval-unit-def
ccompare-unit-def ID-Some proper-interval-class.axioms)

```

```

end

```

```

instantiation bool :: cproper-interval begin

```

```

definition cproper-interval = (proper-interval :: bool proper-interval)

```

```

instance by(intro-classes)

```

```

  (simp add: cproper-interval-bool-def ord-defs ccompare-bool-def ID-Some proper-interval-class.axioms)

```

```

end

```

```

instantiation nat :: cproper-interval begin

```

```

definition cproper-interval = (proper-interval :: nat proper-interval)

```

```

instance by intro-classes simp

```

```

end

```

```

instantiation int :: cproper-interval begin

```

```

definition cproper-interval = (proper-interval :: int proper-interval)

```

```

instance by intro-classes

```

```

  (simp add: cproper-interval-int-def ord-defs ccompare-int-def ID-Some proper-interval-class.axioms)

```

```

end

```

```

instantiation integer :: cproper-interval begin

```

definition *cproper-interval* = (*proper-interval* :: *integer proper-interval*)

instance by *intro-classes*

(*simp add: cproper-interval-integer-def ord-defs ccompare-integer-def ID-Some proper-interval-class.axioms*)

end

instantiation *natural* :: *cproper-interval* **begin**

definition *cproper-interval* = (*proper-interval* :: *natural proper-interval*)

instance by *intro-classes* (*simp add: cproper-interval-natural-def ord-defs ccompare-natural-def ID-Some proper-interval-class.axioms*)

end

instantiation *char* :: *cproper-interval* **begin**

definition *cproper-interval* = (*proper-interval* :: *char proper-interval*)

instance by *intro-classes* (*simp add: cproper-interval-char-def ord-defs ccompare-char-def ID-Some proper-interval-class.axioms*)

end

instantiation *Enum.finite-1* :: *cproper-interval* **begin**

definition *cproper-interval* = (*proper-interval* :: *Enum.finite-1 proper-interval*)

instance by *intro-classes* (*simp add: cproper-interval-finite-1-def ord-defs ccompare-finite-1-def ID-Some proper-interval-class.axioms*)

end

instantiation *Enum.finite-2* :: *cproper-interval* **begin**

definition *cproper-interval* = (*proper-interval* :: *Enum.finite-2 proper-interval*)

instance by *intro-classes* (*simp add: cproper-interval-finite-2-def ord-defs ccompare-finite-2-def ID-Some proper-interval-class.axioms*)

end

instantiation *Enum.finite-3* :: *cproper-interval* **begin**

definition *cproper-interval* = (*proper-interval* :: *Enum.finite-3 proper-interval*)

instance by *intro-classes* (*simp add: cproper-interval-finite-3-def ord-defs ccompare-finite-3-def ID-Some proper-interval-class.axioms*)

end

instantiation *Enum.finite-4* :: *cproper-interval* **begin**

definition (*cproper-interval* :: *Enum.finite-4 proper-interval*) - - = *undefined*

instance by *intro-classes*(*simp add: ord-defs ccompare-finite-4-def ID-None*)

end

instantiation *Enum.finite-5* :: *cproper-interval* **begin**

definition (*cproper-interval* :: *Enum.finite-5 proper-interval*) - - = *undefined*

instance by *intro-classes*(*simp add: ord-defs ccompare-finite-5-def ID-None*)

end

lemma *lt-of-comp-sum*: *lt-of-comp* (*comparator-sum ca cb*) *sx sy* = (

case sx of Inl x \Rightarrow (*case sy of Inl y* \Rightarrow *lt-of-comp ca x y* | *Inr y* \Rightarrow *True*)
 | *Inr x* \Rightarrow (*case sy of Inl y* \Rightarrow *False* | *Inr y* \Rightarrow *lt-of-comp cb x y*)

by (*simp add: lt-of-comp-def le-of-comp-def split: sum.split*)

instantiation *sum* :: (*cproper-interval*, *cproper-interval*) *cproper-interval* **begin**
fun *cproper-interval-sum* :: ('a + 'b) *proper-interval* **where**
cproper-interval-sum None None \longleftrightarrow True
| *cproper-interval-sum* None (Some (Inl x)) \longleftrightarrow *cproper-interval* None (Some x)
| *cproper-interval-sum* None (Some (Inr y)) \longleftrightarrow True
| *cproper-interval-sum* (Some (Inl x)) None \longleftrightarrow True
| *cproper-interval-sum* (Some (Inl x)) (Some (Inl y)) \longleftrightarrow *cproper-interval* (Some x) (Some y)
| *cproper-interval-sum* (Some (Inl x)) (Some (Inr y)) \longleftrightarrow *cproper-interval* (Some x) None \vee *cproper-interval* None (Some y)
| *cproper-interval-sum* (Some (Inr y)) None \longleftrightarrow *cproper-interval* (Some y) None
| *cproper-interval-sum* (Some (Inr y)) (Some (Inl x)) \longleftrightarrow False
| *cproper-interval-sum* (Some (Inr x)) (Some (Inr y)) \longleftrightarrow *cproper-interval* (Some x) (Some y)
instance
proof
assume ID CCOMPARE('a + 'b) \neq None *finite* (UNIV :: ('a + 'b) set)
then obtain *c-a c-b*
where A: ID CCOMPARE('a) = Some *c-a* *finite* (UNIV :: 'a set)
and B: ID CCOMPARE('b) = Some *c-b* *finite* (UNIV :: 'b set)
by(*fastforce simp add: ccompare-sum-def ID-Some ID-None split: option.split-asm*)
note [*simp*] = *proper-interval.proper-interval-simps[OF cproper-interval]*
lt-of-comp-sum ccompare-sum-def ID-Some
and [*split*] = *sum.split*
show *class.proper-interval class* (*cproper-interval* :: ('a + 'b) *proper-interval*)
proof
fix *y* :: 'a + 'b
show *cproper-interval* None (Some *y*) = ($\exists z$. *class* *z* *y*)
using A B **by**(*cases y*)(*auto*, *case-tac z*, *auto*)

fix *x* :: 'a + 'b
show *cproper-interval* (Some *x*) None = ($\exists z$. *class* *x* *z*)
using A B **by**(*cases x*)(*auto*, *case-tac z*, *auto*)

show *cproper-interval* (Some *x*) (Some *y*) = ($\exists z$. *class* *x* *z* \wedge *class* *z* *y*)
using A B **by**(*cases x*)(*case-tac* [!] *y*, *auto*, *case-tac* [!] *z*, *auto*)
qed *simp*
qed
end

lemma *lt-of-comp-less-prod*: *lt-of-comp* (*comparator-prod* *c-a c-b*) =
less-prod (*le-of-comp* *c-a*) (*lt-of-comp* *c-a*) (*lt-of-comp* *c-b*)

unfolding *less-prod-def*

by (*intro ext, auto simp: lt-of-comp-def le-of-comp-def split: order.split-asm prod.split*)

lemma *lt-of-comp-prod*: *lt-of-comp* (*comparator-prod* *c-a c-b*) (*x1,x2*) (*y1,y2*) =

(*lt-of-comp* *c-a* *x1* *y1* \vee *le-of-comp* *c-a* *x1* *y1* \wedge *lt-of-comp* *c-b* *x2* *y2*)
unfolding *lt-of-comp-less-prod less-prod-def* **by** *simp*

instantiation *prod* :: (*cproper-interval*, *cproper-interval*) *cproper-interval* **begin**
fun *cproper-interval-prod* :: ('a \times 'b) *proper-interval* **where**
cproper-interval-prod *None* *None* \longleftrightarrow *True*
| *cproper-interval-prod* *None* (*Some* (*y1*, *y2*)) \longleftrightarrow *cproper-interval* *None* (*Some* *y1*) \vee *cproper-interval* *None* (*Some* *y2*)
| *cproper-interval-prod* (*Some* (*x1*, *x2*)) *None* \longleftrightarrow *cproper-interval* (*Some* *x1*) *None* \vee *cproper-interval* (*Some* *x2*) *None*
| *cproper-interval-prod* (*Some* (*x1*, *x2*)) (*Some* (*y1*, *y2*)) \longleftrightarrow
cproper-interval (*Some* *x1*) (*Some* *y1*) \vee
cless *x1* *y1* \wedge (*cproper-interval* (*Some* *x2*) *None* \vee *cproper-interval* *None* (*Some* *y2*)) \vee
 \neg *cless* *y1* *x1* \wedge *cproper-interval* (*Some* *x2*) (*Some* *y2*)
instance
proof
assume *ID CCOMPARE*('a \times 'b) \neq *None* *finite* (*UNIV* :: ('a \times 'b) *set*)
then obtain *c-a* *c-b*
where *A*: *ID CCOMPARE*('a) = *Some* *c-a* *finite* (*UNIV* :: 'a *set*)
and *B*: *ID CCOMPARE*('b) = *Some* *c-b* *finite* (*UNIV* :: 'b *set*)
by(*fastforce simp add: ccompare-prod-def ID-Some ID-None finite-prod split: option.split-asm*)
interpret *a*: *linorder le-of-comp* *c-a* *lt-of-comp* *c-a* **by**(*rule ID-ccompare*)(*rule A*)
note [*simp*] = *proper-interval.proper-interval-simps[OF cproper-interval]*
ccompare-prod-def lt-of-comp-prod ID-Some
show *class.proper-interval cless* (*cproper-interval* :: ('a \times 'b) *proper-interval*)
using *A B*
by (*unfold-locales, auto 4 4*)
qed
end

instantiation *list* :: (*compare*) *cproper-interval* **begin**
definition *cproper-interval-list* :: 'a *list* *proper-interval*
where *cproper-interval-list* *xso* *yso* = *undefined*
instance **by**(*intro-classes*)(*simp add: infinite-UNIV-listI*)
end

lemma *infinite-UNIV-literal*:
infinite (*UNIV* :: *String.literal* *set*)
by (*fact infinite-literal*)

instantiation *String.literal* :: *cproper-interval* **begin**
definition *cproper-interval-literal* :: *String.literal* *proper-interval*
where *cproper-interval-literal* *xso* *yso* = *undefined*
instance **by**(*intro-classes*)(*simp add: infinite-UNIV-literal*)
end

lemma *lt-of-comp-option*: *lt-of-comp* (*comparator-option* *c*) *sx sy* = (
case sx of None \Rightarrow (*case sy of None* \Rightarrow *False* | *Some y* \Rightarrow *True*)
| *Some x* \Rightarrow (*case sy of None* \Rightarrow *False* | *Some y* \Rightarrow *lt-of-comp c x y*)
by (*simp add: lt-of-comp-def le-of-comp-def split: option.split*)

instantiation *option* :: (*cproper-interval*) *cproper-interval* **begin**
fun *cproper-interval-option* :: 'a *option proper-interval* **where**
cproper-interval-option None None \longleftrightarrow *True*
| *cproper-interval-option None (Some x)* \longleftrightarrow *x* \neq *None*
| *cproper-interval-option (Some x) None* \longleftrightarrow *cproper-interval x None*
| *cproper-interval-option (Some x) (Some None)* \longleftrightarrow *False*
| *cproper-interval-option (Some x) (Some (Some y))* \longleftrightarrow *cproper-interval x (Some y)*
instance
proof
assume *ID CCOMPARE('a option)* \neq *None* *finite (UNIV :: 'a option set)*
then obtain *c-a*
where *A: ID CCOMPARE('a) = Some c-a* *finite (UNIV :: 'a set)*
by (*auto simp add: ccompare-option-def ID-def split: option.split-asm*)
note [*simp*] = *proper-interval.proper-interval-simps[OF cproper-interval]*
ccompare-option-def lt-of-comp-option ID-Some
show *class.proper-interval class (cproper-interval :: 'a option proper-interval)*
using *A*
proof (*unfold-locales*)
fix *x y :: 'a option*
show *cproper-interval (Some x) None* = ($\exists z. class x z$) **using** *A*
by (*cases x*) (*auto split: option.split intro: exI[where x=Some undefined]*)

show *cproper-interval (Some x) (Some y)* = ($\exists z. class x z \wedge class z y$) **using**
A
by (*cases x y rule: option.exhaust[case-product option.exhaust]*) (*auto cong: option.case-cong split: option.split*)
qed (*auto split: option.splits*)
qed
end

instantiation *set* :: (*cproper-interval*) *cproper-interval* **begin**
fun *cproper-interval-set* :: 'a *set proper-interval* **where**
[*code*]: *cproper-interval-set None None* \longleftrightarrow *True*
| [*code*]: *cproper-interval-set None (Some B)* \longleftrightarrow (*B* \neq $\{\}$)
| [*code*]: *cproper-interval-set (Some A) None* \longleftrightarrow (*A* \neq *UNIV*)
| *cproper-interval-set-Some-Some*: — Refine for concrete implementations
cproper-interval-set (Some A) (Some B) \longleftrightarrow *finite (UNIV :: 'a set) \wedge (\exists C. class A C \wedge class C B)*
instance
proof

```

assume ID CCOMPARE('a set)  $\neq$  None   finite (UNIV :: 'a set set)
then obtain c-a
  where A: ID CCOMPARE('a) = Some c-a   finite (UNIV :: 'a set)
  by(auto simp add: ccompare-set-def ID-def Finite-Set.finite-set)
interpret a: linorder le-of-comp c-a   lt-of-comp c-a by(rule ID-ccompare)(rule
A)
note [simp] = proper-interval.proper-interval-simps[OF cproper-interval] ccom-
pare-set-def
  ID-Some lt-of-comp-of-ords
show class.proper-interval class (cproper-interval :: 'a set proper-interval) using
A
  by (unfold-locales, auto)
qed

```

lemma *Complement-cproper-interval-set-Complement:*

```

fixes A B :: 'a set
assumes corder: ID CCOMPARE('a)  $\neq$  None
shows cproper-interval (Some (- A)) (Some (- B)) = cproper-interval (Some
B) (Some A)
using assms
by(clarsimp simp add: ccompare-set-def ID-Some lt-of-comp-of-ords) (metis dou-
ble-complement linorder.Compl-set-less-Compl[OF ID-ccompare])

```

end

instantiation fun :: (type, type) cproper-interval **begin**

No interval checks on functions needed because we have not defined an order on them.

```

definition cproper-interval = (undefined :: ('a  $\Rightarrow$  'b) proper-interval)
instance by(intro-classes)(simp add: ccompare-fun-def ID-None)
end

```

end

theory *List-Proper-Interval* **imports**

HOL-Library.List-Lexorder

Collection-Order

begin

3.2 Instantiate *proper-interval* of for 'a list

lemma *Nil-less-conv-neq-Nil:* $[] < xs \longleftrightarrow xs \neq []$

by(cases xs) simp-all

lemma *less-append-same-iff:*

```

fixes  $xs :: 'a :: preorder\ list$ 
shows  $xs < xs @ ys \longleftrightarrow [] < ys$ 
by(induct xs) simp-all

```

```

lemma less-append-same2-iff:
fixes  $xs :: 'a :: preorder\ list$ 
shows  $xs @ ys < xs @ zs \longleftrightarrow ys < zs$ 
by(induct xs) simp-all

```

```

lemma Cons-less-iff:
fixes  $x :: 'a :: preorder$  shows
 $x \# xs < ys \longleftrightarrow (\exists y\ ys'. ys = y \# ys' \wedge (x < y \vee x = y \wedge xs < ys'))$ 
by(cases ys) auto

```

```

instantiation list :: ({proper-interval, order}) proper-interval begin

```

```

definition proper-interval-list-aux :: 'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool
where proper-interval-list-aux-correct:
 $proper-interval-list-aux\ xs\ ys \longleftrightarrow (\exists zs. xs < zs \wedge zs < ys)$ 

```

```

lemma proper-interval-list-aux-simps [code]:
 $proper-interval-list-aux\ xs\ [] \longleftrightarrow False$ 
 $proper-interval-list-aux\ []\ (y \# ys) \longleftrightarrow ys \neq [] \vee proper-interval\ None\ (Some\ y)$ 
 $proper-interval-list-aux\ (x \# xs)\ (y \# ys) \longleftrightarrow x < y \vee x = y \wedge proper-interval-list-aux\ xs\ ys$ 
apply(simp-all add: proper-interval-list-aux-correct proper-interval-simps Nil-less-conv-neq-Nil)
apply(fastforce simp add: neq-Nil-conv)
apply(rule iffI)
apply(fastforce simp add: Cons-less-iff intro: less-trans)
apply(erule disjE)
apply(rule exI[where x=x # xs @ [undefined]])
apply(simp add: less-append-same-iff)
apply(auto 4 3 simp add: Cons-less-iff)
done

```

```

fun proper-interval-list :: 'a list option  $\Rightarrow$  'a list option  $\Rightarrow$  bool where
 $proper-interval-list\ None\ None \longleftrightarrow True$ 
|  $proper-interval-list\ None\ (Some\ xs) \longleftrightarrow (xs \neq [])$ 
|  $proper-interval-list\ (Some\ xs)\ None \longleftrightarrow True$ 
|  $proper-interval-list\ (Some\ xs)\ (Some\ ys) \longleftrightarrow proper-interval-list-aux\ xs\ ys$ 
instance

```

```

proof(intro-classes)
fix  $xs\ ys :: 'a\ list$ 
show  $proper-interval\ None\ (Some\ ys) \longleftrightarrow (\exists zs. zs < ys)$ 
by(auto simp add: Nil-less-conv-neq-Nil intro: exI[where x=[]])
show  $proper-interval\ (Some\ xs)\ None \longleftrightarrow (\exists zs. xs < zs)$ 
by(simp add: exI[where x=xs @ [undefined]] less-append-same-iff)
show  $proper-interval\ (Some\ xs)\ (Some\ ys) \longleftrightarrow (\exists zs. xs < zs \wedge zs < ys)$ 
by(simp add: proper-interval-list-aux-correct)

```

```
qed simp
end
```

```
end
```

```
theory Collection-Eq imports
  Containers-Auxiliary
  Containers-Generator
  Deriving.Equality-Instances
begin
```

3.3 A type class for optional equality testing

```
class ceq =
  fixes ceq :: ('a ⇒ 'a ⇒ bool) option
  assumes ceq: ceq = Some eq ⇒ eq = (=)
begin
```

```
lemma ceq-equality: ceq = Some eq ⇒ equality eq
  by (drule ceq, rule Equality-Generator.equalityI, simp)
```

```
lemma ID-ceq: ID ceq = Some eq ⇒ eq = (=)
unfolding ID-def id-apply by(rule ceq)
```

```
abbreviation ceq' :: 'a ⇒ 'a ⇒ bool where ceq' ≡ the (ID ceq)
```

```
end
```

```
syntax -CEQ :: type => logic (⟨(1CEQ/(1'(-)))⟩)
```

```
syntax-consts -CEQ == ceq
```

```
parse-translation ‹
```

```
let
```

```
  fun ceq-tr [ty] =
    (Syntax.const @ {syntax-const -constrain} $ Syntax.const @ {const-syntax ceq}
  $
    (Syntax.const @ {type-syntax option} $
      (Syntax.const @ {type-syntax fun} $ ty $
        (Syntax.const @ {type-syntax fun} $ ty $ Syntax.const @ {type-syntax
bool}))))))
```

```
  | ceq-tr ts = raise TERM (ceq-tr, ts);
in [(@ {syntax-const -CEQ}, K ceq-tr)] end
```

```
›
```

```
typed-print-translation ‹
```

```
let
```

```
  fun ceq-tr' ctxt
    (Type (@ {type-name option}, [Type (@ {type-name fun}, [T, -]))) ts =
```

```

Term.list-comb (Syntax.const @ {syntax-const -CEQ} $ Syntax-Phases.term-of-typ
  ctxt T, ts)
| ceq-tr' - - - = raise Match;
in [(@ {const-syntax ceq}, ceq-tr')]
end
>

```

definition *is-ceq* :: 'a :: ceq itself \Rightarrow bool
where *is-ceq* - \longleftrightarrow ID CEQ('a) \neq None

3.3.1 Generator for the *ceq*-class

This generator registers itself at the derive-manager for the class *ceq*. To be more precise, one can choose whether one wants to take (=) as function for *CEQ('a)* by passing "eq" as parameter, whether equality should not be supported by passing "no" as parameter, or whether an own definition for equality should be derived by not passing any parameters. The last possibility only works for datatypes.

- instantiation type :: (type,...,type) (eq) ceq
- instantiation type :: (type,...,type) (no) ceq
- instantiation datatype :: (ceq,...,ceq) ceq

If the parameter "no" is not used, then the corresponding *is-ceq*-theorem is also automatically generated and attributed with [simp, code-post].

This generator can be used for arbitrary types, not just datatypes.

lemma *equality-subst*: $c1 = c2 \Longrightarrow$ equality $c1 \Longrightarrow$ equality $c2$ by blast

ML-file \langle ceq-generator.ML \rangle

3.3.2 Type class instances for HOL types

```

derive (eq) ceq unit
declare [[code drop:  $\langle$ CEQ(unit) $\rangle$ ]]
lemma [code]: CEQ(unit) = Some ( $\lambda$ - -. True)
  by (simp add: fun-eq-iff ceq-unit-def)
derive (eq) ceq
  bool
  nat
  int
  Enum.finite-1
  Enum.finite-2
  Enum.finite-3
  Enum.finite-4
  Enum.finite-5

```

```

integer
natural
char
String.literal
derive ceq sum prod list option
derive (no) ceq fun

lemma is-ceq-fun [simp]:  $\neg$  is-ceq TYPE('a  $\Rightarrow$  'b)
  by(simp add: is-ceq-def ceq-fun-def ID-None)

definition set-eq :: '<'a set  $\Rightarrow$  'a set  $\Rightarrow$  bool>
  where [simp, code-abbrev]: <set-eq = (=)>

lemma set-eq-code [code]:
  <set-eq A B  $\longleftrightarrow$  A  $\subseteq$  B  $\wedge$  B  $\subseteq$  A>
  by auto

instantiation set :: (ceq) ceq begin
definition CEQ('a set) = (case ID CEQ('a) of None  $\Rightarrow$  None | Some -  $\Rightarrow$  Some
set-eq)
instance by(intro-classes)(simp add: ceq-set-def set-eq-def split: option.splits)
end

lemma is-ceq-set [simp, code-post]: is-ceq TYPE('a set)  $\longleftrightarrow$  is-ceq TYPE('a ::
ceq)
by(simp add: is-ceq-def ceq-set-def ID-None ID-Some split: option.split)

lemma ID-ceq-set-not-None-iff [simp]: ID CEQ('a set)  $\neq$  None  $\longleftrightarrow$  ID CEQ('a
:: ceq)  $\neq$  None
by(simp add: ceq-set-def ID-def split: option.splits)

Instantiation for 'a Predicate.pred

context fixes eq :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool begin

definition member-pred :: 'a Predicate.pred  $\Rightarrow$  'a  $\Rightarrow$  bool
where member-pred P x  $\longleftrightarrow$  ( $\exists$  y. eq x y  $\wedge$  Predicate.eval P y)

definition member-seq :: 'a Predicate.seq  $\Rightarrow$  'a  $\Rightarrow$  bool
where member-seq xp = member-pred (Predicate.pred-of-seq xp)

lemma member-seq-code [code]:
  member-seq seq.Empty x  $\longleftrightarrow$  False
  member-seq (seq.Insert y P) x  $\longleftrightarrow$  eq x y  $\vee$  member-pred P x
  member-seq (seq.Join Q xq) x  $\longleftrightarrow$  member-pred Q x  $\vee$  member-seq xq x
by(auto simp add: member-seq-def member-pred-def)

lemma member-pred-code [code]:
  member-pred (Predicate.Seq f) = member-seq (f ())
by(simp add: member-seq-def Seq-def)

```

definition $leq\text{-pred} :: 'a \text{ Predicate.pred} \Rightarrow 'a \text{ Predicate.pred} \Rightarrow \text{bool}$
where $leq\text{-pred } P \ Q \longleftrightarrow (\forall x. \text{Predicate.eval } P \ x \longrightarrow (\exists y. \text{eq } x \ y \wedge \text{Predicate.eval } Q \ y))$

definition $leq\text{-seq} :: 'a \text{ Predicate.seq} \Rightarrow 'a \text{ Predicate.pred} \Rightarrow \text{bool}$
where $leq\text{-seq } xp \ Q \longleftrightarrow leq\text{-pred } (\text{Predicate.pred-of-seq } xp) \ Q$

lemma $leq\text{-seq-code}$ [code]:
 $leq\text{-seq } seq.\text{Empty } Q \longleftrightarrow \text{True}$
 $leq\text{-seq } (seq.\text{Insert } x \ P) \ Q \longleftrightarrow \text{member-pred } Q \ x \wedge leq\text{-pred } P \ Q$
 $leq\text{-seq } (seq.\text{Join } P \ xp) \ Q \longleftrightarrow leq\text{-pred } P \ Q \wedge leq\text{-seq } xp \ Q$
by(*auto simp add: leq-seq-def leq-pred-def member-pred-def*)

lemma $leq\text{-pred-code}$ [code]:
 $leq\text{-pred } (\text{Predicate.Seq } f) \ Q \longleftrightarrow leq\text{-seq } (f \ ()) \ Q$
by(*simp add: leq-seq-def Seq-def*)

definition $predicate\text{-eq} :: 'a \text{ Predicate.pred} \Rightarrow 'a \text{ Predicate.pred} \Rightarrow \text{bool}$
where $predicate\text{-eq } P \ Q \longleftrightarrow leq\text{-pred } P \ Q \wedge leq\text{-pred } Q \ P$

context assumes $eq: eq = (=)$ **begin**

lemma $member\text{-pred-eq}: member\text{-pred} = \text{Predicate.eval}$
unfolding $fun\text{-eq-iff } member\text{-pred-def}$ **by**(*simp add: eq*)

lemma $member\text{-seq-eq}: member\text{-seq} = \text{Predicate.member}$
by(*simp add: member-seq-def fun-eq-iff eval-member member-pred-eq*)

lemma $leq\text{-pred-eq}: leq\text{-pred} = (\leq)$
unfolding $fun\text{-eq-iff } leq\text{-pred-def}$ **by**(*auto simp add: eq less-eq-pred-def*)

lemma $predicate\text{-eq-eq}: predicate\text{-eq} = (=)$
unfolding $predicate\text{-eq-def } fun\text{-eq-iff}$ **by**(*auto simp add: leq-pred-eq*)

end
end

instantiation $\text{Predicate.pred} :: (ceq) \ ceq$ **begin**

definition $\text{CEQ}('a \ \text{Predicate.pred}) = \text{map-option } predicate\text{-eq} \ (\text{ID } \text{CEQ}('a))$

instance **by**(*intro-classes*)(*auto simp add: ceq-pred-def predicate-eq-eq dest: ID-ceq*)
end

end

theory Collection-Enum **imports**

$\text{Containers-Auxiliary}$

$\text{Containers-Generator}$

begin

3.4 A type class for optional enumerations

3.4.1 Definition

```
class cEnum =
  fixes cEnum :: ('a list × (('a ⇒ bool) ⇒ bool) × (('a ⇒ bool) ⇒ bool)) option
  assumes UNIV-cEnum: cEnum = Some (enum, enum-all, enum-ex) ⇒ UNIV
= set enum
  and cEnum-all-UNIV: cEnum = Some (enum, enum-all, enum-ex) ⇒ enum-all
P = Ball UNIV P
  and cEnum-ex-UNIV: cEnum = Some (enum, enum-all, enum-ex) ⇒ enum-ex
P = Bex UNIV P
begin
```

```
lemma ID-cEnum:
  ID cEnum = Some (enum, enum-all, enum-ex)
  ⇒ UNIV = set enum ∧ enum-all = Ball UNIV ∧ enum-ex = Bex UNIV
unfolding ID-def id-apply fun-eq-iff
by(intro conjI allI UNIV-cEnum cEnum-all-UNIV cEnum-ex-UNIV fun-eq-iff)
```

```
lemma in-cEnum: ID cEnum = Some (enum, rest) ⇒ f ∈ set enum
by(cases rest)(auto dest: ID-cEnum)
```

```
abbreviation cEnum :: 'a list
where cEnum ≡ fst (the (ID cEnum))
```

```
abbreviation cEnum-all :: ('a ⇒ bool) ⇒ bool
where cEnum-all ≡ fst (snd (the (ID cEnum)))
```

```
abbreviation cEnum-ex :: ('a ⇒ bool) ⇒ bool
where cEnum-ex ≡ snd (snd (the (ID cEnum)))
```

end

```
syntax -CENUM :: type => logic (⟨(1CENUM/(1'(-)))⟩)
```

```
syntax-consts -CENUM == cEnum
```

```
parse-translation ‹
let
  fun cEnum-tr [ty] =
    (Syntax.const @ {syntax-const -constrain} $ Syntax.const @ {const-syntax cEnum})
$
  (Syntax.const @ {type-syntax option} $
   (Syntax.const @ {type-syntax prod} $
    (Syntax.const @ {type-syntax list} $ ty) $
    (Syntax.const @ {type-syntax prod} $
```

```

      (Syntax.const @{type-syntax fun} $
        (Syntax.const @{type-syntax fun} $ ty $ (Syntax.const @{type-syntax
bool})) $
      (Syntax.const @{type-syntax bool})) $
      (Syntax.const @{type-syntax fun} $
        (Syntax.const @{type-syntax fun} $ ty $ (Syntax.const @{type-syntax
bool})) $
      (Syntax.const @{type-syntax bool}))))))
    | cenum-tr ts = raise TERM (cenum-tr, ts);
in [(@{syntax-const -CENUM}, K cenum-tr)] end
>

```

typed-print-translation <

```

let
  fun cenum-tr' ctxt
    (Type (@{type-name option}, [Type (@{type-name prod}, [Type (@{type-name
list}, [T]), -]))) ts =
      Term.list-comb (Syntax.const @{syntax-const -CENUM} $ Syntax-Phases.term-of-ty
ctxt T, ts)
    | cenum-tr' - - - = raise Match;
in [(@{const-syntax cEnum}, cenum-tr')]
end
>

```

3.4.2 Generator for the *cenum*-class

This generator registers itself at the derive-manager for the class *cenum*. To be more precise, one can currently only choose to not support enumeration by passing "no" as parameter.

- instantiation type :: (type,...,type) (no) cenum

This generator can be used for arbitrary types, not just datatypes.

ML-file <*cenum-generator.ML*>

3.4.3 Instantiations

```

context fixes cenum-all :: ('a ⇒ bool) ⇒ bool begin
fun all-n-lists :: ('a list ⇒ bool) ⇒ nat ⇒ bool
where [simp del]:
  all-n-lists P n = (if n = 0 then P [] else cenum-all (λx. all-n-lists (λxs. P (x #
xs)) (n - 1)))
end

```

```

context fixes cenum-ex :: ('a ⇒ bool) ⇒ bool begin
fun ex-n-lists :: ('a list ⇒ bool) ⇒ nat ⇒ bool
where [simp del]:

```

```

    ex-n-lists P n  $\longleftrightarrow$  (if n = 0 then P [] else cenum-ex (%x. ex-n-lists (%xs. P (x
# xs)) (n - 1)))
end

```

lemma all-n-lists-iff: fixes cenum shows

```

all-n-lists (Ball (set cenum)) P n  $\longleftrightarrow$  ( $\forall$  xs  $\in$  set (List.n-lists n cenum). P xs)
proof(induct P n rule: all-n-lists.induct)

```

```

  case (1 P n)
  show ?case
  proof(cases n)
    case 0
    thus ?thesis by(simp add: all-n-lists.simps)
  next
  case (Suc n')
  thus ?thesis using 1 by(subst all-n-lists.simps) auto
qed
qed

```

lemma ex-n-lists-iff: fixes cenum shows

```

ex-n-lists (Bex (set cenum)) P n  $\longleftrightarrow$  ( $\exists$  xs  $\in$  set (List.n-lists n cenum). P xs)
proof(induct P n rule: ex-n-lists.induct)

```

```

  case (1 P n)
  show ?case
  proof(cases n)
    case 0
    thus ?thesis by(simp add: ex-n-lists.simps)
  next
  case (Suc n')
  thus ?thesis using 1 by(subst ex-n-lists.simps) auto
qed
qed

```

instantiation fun :: (cenum, cenum) cenum **begin**

definition

```

CENUM('a  $\Rightarrow$  'b) =
(case ID CENUM('a) of None  $\Rightarrow$  None | Some (enum-a, enum-all-a, enum-ex-a)
 $\Rightarrow$ 
  case ID CENUM('b) of None  $\Rightarrow$  None | Some (enum-b, enum-all-b, enum-ex-b)
 $\Rightarrow$  Some
    (map ( $\lambda$ ys. the o map-of (zip enum-a ys)) (List.n-lists (length enum-a)
enum-b),
     $\lambda$ P. all-n-lists enum-all-b ( $\lambda$ bs. P (the o map-of (zip enum-a bs))) (length
enum-a),
     $\lambda$ P. ex-n-lists enum-ex-b ( $\lambda$ bs. P (the o map-of (zip enum-a bs))) (length
enum-a)))

```

instance proof

```

fix enum enum-all enum-ex P
assume CENUM('a  $\Rightarrow$  'b) = Some (enum, enum-all, enum-ex)

```

then obtain $enum-a$ $enum-all-a$ $enum-ex-a$ $enum-b$ $enum-all-b$ $enum-ex-b$
where a : $ID\ CENUM('a) = Some\ (enum-a, enum-all-a, enum-ex-a)$
and b : $ID\ CENUM('b) = Some\ (enum-b, enum-all-b, enum-ex-b)$
and $enum$: $enum = map\ (\lambda ys. the\ o\ map-of\ (zip\ enum-a\ ys))\ (List.n-lists\ (length\ enum-a)\ enum-b)$
and $enum-all$: $enum-all = (\lambda P. all-n-lists\ enum-all-b\ (\lambda bs. P\ (the\ o\ map-of\ (zip\ enum-a\ bs))))\ (length\ enum-a)$
and $enum-ex$: $enum-ex = (\lambda P. ex-n-lists\ enum-ex-b\ (\lambda bs. P\ (the\ o\ map-of\ (zip\ enum-a\ bs))))\ (length\ enum-a)$
by($fastforce\ simp\ add$: $cEnum-fun-def\ split$: $option.split-asm$)

show $UNIV = set\ enum$

proof ($rule\ UNIV-eq-I$)

fix $f :: 'a \Rightarrow 'b$

have $f = the\ o\ map-of\ (zip\ enum-a\ (map\ f\ enum-a))$

by ($auto\ simp\ add$: $map-of-zip-map\ fun-eq-iff\ intro$: $in-cenum[OF\ a]$)

then show $f \in set\ enum$

by ($auto\ simp\ add$: $enum\ set-n-lists\ intro$: $in-cenum[OF\ b]$)

qed

show $enum-all\ P = Ball\ UNIV\ P$

proof

assume $enum-all\ P$

show $Ball\ UNIV\ P$

proof

fix $f :: 'a \Rightarrow 'b$

have $f: f = the\ o\ map-of\ (zip\ (enum-a)\ (map\ f\ enum-a))$

by ($auto\ simp\ add$: $map-of-zip-map\ fun-eq-iff\ intro$: $in-cenum[OF\ a]$)

from $\langle enum-all\ P \rangle$ **have** $P\ (the\ o\ map-of\ (zip\ enum-a\ (map\ f\ enum-a)))$

apply($simp\ add$: $enum-all\ ID-cEnum[OF\ b]\ all-n-lists-iff\ set-n-lists$)

apply($erule\ allE, erule\ mp$)

apply($auto\ simp\ add$: $in-cenum[OF\ b]$)

done

with f **show** $P\ f$ **by** $simp$

qed

next

assume $Ball\ UNIV\ P$

from $this$ **show** $enum-all\ P$

by($simp\ add$: $enum-all\ ID-cEnum[OF\ b]\ all-n-lists-iff$)

qed

show $enum-ex\ P = Bex\ UNIV\ P$

proof

assume $enum-ex\ P$

from $this$ **show** $Bex\ UNIV\ P$

by($auto\ simp\ add$: $enum-ex\ ID-cEnum[OF\ b]\ ex-n-lists-iff$)

next

assume $Bex\ UNIV\ P$

from $this$ **obtain** f **where** $P\ f ..$

```

also have  $f: f = \text{the} \circ \text{map-of} \text{ (zip (enum-a) (map f enum-a))}$ 
by (auto simp add: map-of-zip-map fun-eq-iff intro: in-cenum[OF a])
finally show enum-ex P
apply (simp add: enum-ex ID-cEnum[OF b] ex-n-lists-iff o-def)
apply (erule bexI)
apply (auto simp add: set-n-lists intro!: in-cenum[OF b])
done
qed
qed
end

instantiation set :: (cenum) cenum begin
definition
  CENUM('a set) =
  (case ID CENUM('a) of None  $\Rightarrow$  None | Some (enum-a, enum-all-a, enum-ex-a)
 $\Rightarrow$  Some
    (map set (subseqs enum-a),
      $\lambda P$ . list-all P (map set (subseqs enum-a)),
      $\lambda P$ . list-ex P (map set (subseqs enum-a))))))
instance
by (intro-classes) (auto simp add: cEnum-set-def subseqs-powset list-ex-iff list-all-iff
split: option.split-asm dest!: ID-cEnum)
end

instantiation unit :: cenum begin
definition CENUM(unit) = Some (enum-class.enum, enum-class.enum-all, enum-class.enum-ex)
instance by (intro-classes) (auto simp add: cEnum-unit-def enum-UNIV enum-all-UNIV
enum-ex-UNIV)
end

instantiation bool :: cenum begin
definition CENUM(bool) = Some (enum-class.enum, enum-class.enum-all, enum-class.enum-ex)
instance by (intro-classes) (auto simp add: cEnum-bool-def enum-UNIV enum-all-UNIV
enum-ex-UNIV)
end

instantiation prod :: (cenum, cenum) cenum begin
definition
  CENUM('a  $\times$  'b) =
  (case ID CENUM('a) of None  $\Rightarrow$  None | Some (enum-a, enum-all-a, enum-ex-a)
 $\Rightarrow$ 
    case ID CENUM('b) of None  $\Rightarrow$  None | Some (enum-b, enum-all-b, enum-ex-b)
 $\Rightarrow$  Some
      (List.product enum-a enum-b,
        $\lambda P$ . enum-all-a (%x. enum-all-b (%y. P (x, y))),
        $\lambda P$ . enum-ex-a (%x. enum-ex-b (%y. P (x, y))))))
instance
by (intro-classes) (auto 4 4 simp add: cEnum-prod-def split: option.split-asm dest!:
ID-cEnum)

```

end

instantiation *sum* :: (*cenum*, *cenum*) *cenum* **begin**

definition

$CENUM('a + 'b) =$
 $(\text{case ID } CENUM('a) \text{ of } None \Rightarrow None \mid \text{Some } (enum-a, enum-all-a, enum-ex-a)$
 \Rightarrow
 $\text{case ID } CENUM('b) \text{ of } None \Rightarrow None \mid \text{Some } (enum-b, enum-all-b, enum-ex-b)$
 $\Rightarrow \text{Some}$
 $(\text{map Inl } enum-a @ \text{map Inr } enum-b,$
 $\lambda P. enum-all-a (\lambda x. P (\text{Inl } x)) \wedge enum-all-b (\lambda x. P (\text{Inr } x)),$
 $\lambda P. enum-ex-a (\lambda x. P (\text{Inl } x)) \vee enum-ex-b (\lambda x. P (\text{Inr } x)))$

instance

by(*intro-classes*)(*auto* 4 4 *simp add: cEnum-sum-def UNIV-sum split: option.split-asm*
dest!: ID-cEnum)

end

instantiation *option* :: (*cenum*) *cenum* **begin**

definition

$CENUM('a \text{ option}) =$
 $(\text{case ID } CENUM('a) \text{ of } None \Rightarrow None \mid \text{Some } (enum-a, enum-all-a, enum-ex-a)$
 $\Rightarrow \text{Some}$
 $(None \# \text{map Some } enum-a,$
 $\lambda P. P \text{ None} \wedge enum-all-a (\lambda x. P (\text{Some } x)),$
 $\lambda P. P \text{ None} \vee enum-ex-a (\lambda x. P (\text{Some } x)))$

instance

by(*intro-classes*)(*auto simp add: cEnum-option-def UNIV-option-conv split: option.split-asm* *dest: ID-cEnum*)

end

instantiation *Enum.finite-1* :: *cenum* **begin**

definition $CENUM(Enum.finite-1) = \text{Some } (enum-class.enum, enum-class.enum-all,$
 $enum-class.enum-ex)$

instance **by**(*intro-classes*)(*auto simp add: cEnum-finite-1-def enum-UNIV enum-all-UNIV*
 $enum-ex-UNIV$)

end

instantiation *Enum.finite-2* :: *cenum* **begin**

definition $CENUM(Enum.finite-2) = \text{Some } (enum-class.enum, enum-class.enum-all,$
 $enum-class.enum-ex)$

instance **by**(*intro-classes*)(*auto simp add: cEnum-finite-2-def enum-UNIV enum-all-UNIV*
 $enum-ex-UNIV$)

end

instantiation *Enum.finite-3* :: *cenum* **begin**

definition $CENUM(Enum.finite-3) = \text{Some } (enum-class.enum, enum-class.enum-all,$
 $enum-class.enum-ex)$

instance **by**(*intro-classes*)(*auto simp add: cEnum-finite-3-def enum-UNIV enum-all-UNIV*
 $enum-ex-UNIV$)

end

instantiation *Enum.finite-4* :: *cenum* **begin**

definition *CENUM*(*Enum.finite-4*) = *Some* (*enum-class.enum*, *enum-class.enum-all*,
enum-class.enum-ex)

instance by(*intro-classes*)(*auto simp add: cEnum-finite-4-def enum-UNIV enum-all-UNIV*
enum-ex-UNIV)

end

instantiation *Enum.finite-5* :: *cenum* **begin**

definition *CENUM*(*Enum.finite-5*) = *Some* (*enum-class.enum*, *enum-class.enum-all*,
enum-class.enum-ex)

instance by(*intro-classes*)(*auto simp add: cEnum-finite-5-def enum-UNIV enum-all-UNIV*
enum-ex-UNIV)

end

instantiation *char* :: *cenum* **begin**

definition *CENUM*(*char*) = *Some* (*enum-class.enum*, *enum-class.enum-all*, *enum-class.enum-ex*)

instance by(*intro-classes*)(*auto simp add: cEnum-char-def enum-UNIV enum-all-UNIV*
enum-ex-UNIV)

end

derive (*no*) *cenum list nat int integer natural String.literal*

end

theory *Equal* **imports** *Main* **begin**

3.5 Locales to abstract over HOL equality

locale *equal-base* = **fixes** *equal* :: 'a ⇒ 'a ⇒ bool

locale *equal* = *equal-base* +
assumes *equal-eq*: *equal* = (=)

begin

lemma *equal-conv-eq*: *equal* *x y* ↔ *x = y*

by(*simp add: equal-eq*)

end

end

theory *RBT-ext*

imports

HOL-Library.RBT-Impl

Containers-Auxiliary

List-Fusion

begin

3.6 More on red-black trees

3.6.1 More lemmas

context *linorder* **begin**

lemma *is-rbt-fold-rbt-insert-impl*:

is-rbt t \implies *is-rbt (RBT-Impl.fold rbt-insert t' t)*

by(*simp add: rbt-insert-def is-rbt-fold-rbt-insertwk*)

lemma *rbt-sorted-fold-insert*: *rbt-sorted t* \implies *rbt-sorted (RBT-Impl.fold rbt-insert t' t)*

by(*induct t' arbitrary: t*)(*simp-all add: rbt-insert-rbt-sorted*)

lemma *rbt-lookup-rbt-insert'*: *rbt-sorted t* \implies *rbt-lookup (rbt-insert k v t) = (rbt-lookup t)(k \mapsto v)*

by(*simp add: rbt-insert-def rbt-lookup-rbt-insertwk fun-eq-iff split: option.split*)

lemma *rbt-lookup-fold-rbt-insert-impl*:

rbt-sorted t2 \implies

rbt-lookup (RBT-Impl.fold rbt-insert t1 t2) = rbt-lookup t2 ++ map-of (rev (RBT-Impl.entries t1))

proof(*induction t1 arbitrary: t2*)

case *Empty* **thus** *?case* **by** *simp*

next

case (*Branch c l x k r*)

show *?case* **using** *Branch.prem*s

by(*simp add: map-add-def Branch.IH rbt-insert-rbt-sorted rbt-sorted-fold-insert rbt-lookup-rbt-insert' fun-eq-iff split: option.split*)

qed

end

3.6.2 Build the cross product of two RBTs

context *fixes f :: 'a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd \Rightarrow 'e* **begin**

definition *alist-product* :: (*'a \times 'b*) *list* \Rightarrow (*'c \times 'd*) *list* \Rightarrow ((*'a \times 'c*) \times *'e*) *list*

where *alist-product xs ys = concat (map ($\lambda(a, b).$ map ($\lambda(c, d).$ ((*a, c*), *f a b c d*)) *ys*) *xs*)*

lemma *alist-product-simps* [*simp*]:

alist-product [] ys = []

alist-product xs [] = []

*alist-product ((*a, b*) # *xs*) *ys* = map ($\lambda(c, d).$ ((*a, c*), *f a b c d*)) *ys* @ *alist-product xs ys**

by(*simp-all add: alist-product-def*)

lemma *append-alist-product-conv-fold*:

$zs @ alist-product\ xs\ ys = rev\ (fold\ (\lambda(a, b). fold\ (\lambda(c, d)\ rest.\ ((a, c), f\ a\ b\ c\ d)\ #\ rest)\ ys)\ xs\ (rev\ zs))$

proof(*induction xs arbitrary: zs*)

case *Nil* **thus** *?case by simp*

next

case (*Cons x xs*)

obtain *a b* **where** $x = (a, b)$ **by**(*cases x*)

have $\bigwedge zs.\ fold\ (\lambda(c, d).\ (#)\ ((a, c), f\ a\ b\ c\ d))\ ys\ zs = rev\ (map\ (\lambda(c, d).\ ((a, c), f\ a\ b\ c\ d))\ ys)\ @\ zs$

by(*induct ys auto*)

with *Cons.IH*[*of zs @ map (\lambda(c, d).\ ((a, c), f a b c d)) ys*] *x*

show *?case by simp*

qed

lemma *alist-product-code* [*code*]:

$alist-product\ xs\ ys =$

$rev\ (fold\ (\lambda(a, b).\ fold\ (\lambda(c, d)\ rest.\ ((a, c), f\ a\ b\ c\ d)\ #\ rest)\ ys)\ xs\ [])$

using *append-alist-product-conv-fold*[*of [] xs ys*]

by *simp*

lemma *set-alist-product*:

$set\ (alist-product\ xs\ ys) =$

$(\lambda((a, b), (c, d)). ((a, c), f\ a\ b\ c\ d))\ ` (set\ xs \times set\ ys)$

by(*auto 4 3 simp add: alist-product-def intro: rev-image-eqI rev-bexI*)

lemma *distinct-alist-product*:

$[distinct\ (map\ fst\ xs); distinct\ (map\ fst\ ys)]$

$\implies distinct\ (map\ fst\ (alist-product\ xs\ ys))$

proof(*induct xs*)

case *Nil* **thus** *?case by simp*

next

case (*Cons x xs*)

obtain *a b* **where** $x = (a, b)$ **by**(*cases x*)

have $distinct\ (map\ (fst \circ (\lambda(c, d).\ ((a, c), f\ a\ b\ c\ d)))\ ys)$

using $\langle distinct\ (map\ fst\ ys) \rangle$ **by**(*induct ys*)(*auto intro: rev-image-eqI*)

thus *?case using Cons x by*(*auto simp add: set-alist-product intro: rev-image-eqI*)

qed

lemma *map-of-alist-product*:

$map-of\ (alist-product\ xs\ ys)\ (a, c) =$

$(case\ map-of\ xs\ a\ of\ None \implies None$

$| Some\ b \implies map-option\ (f\ a\ b\ c)\ (map-of\ ys\ c))$

proof(*induction xs*)

case *Nil* **thus** *?case by simp*

next

case (*Cons x xs*)

```

obtain  $a\ b$  where  $x: x = (a, b)$  by (cases  $x$ )
let  $?map = map\ (\lambda(c, d). ((a, c), f\ a\ b\ c\ d))\ ys$ 
have  $map\text{-of}\ ?map\ (a, c) = map\text{-option}\ (f\ a\ b\ c)\ (map\text{-of}\ ys\ c)$ 
  by(induct  $ys$ ) auto
moreover {
  fix  $a'$  assume  $a' \neq a$ 
  hence  $map\text{-of}\ ?map\ (a', c) = None$ 
  by(induct  $ys$ ) auto }
ultimately show  $?case$  using  $x\ Cons.IH$ 
by(auto simp add: map-add-def split: option.split)
qed

```

definition $rbt\text{-product} :: ('a, 'b)\ rbt \Rightarrow ('c, 'd)\ rbt \Rightarrow ('a \times 'c, 'e)\ rbt$

where

```

 $rbt\text{-product}\ rbt1\ rbt2 = rbtreeify\ (alist\text{-product}\ (RBT\text{-Impl.entries}\ rbt1)\ (RBT\text{-Impl.entries}\ rbt2))$ 

```

lemma $rbt\text{-product-code}$ [*code*]:

```

 $rbt\text{-product}\ rbt1\ rbt2 =$ 
 $rbtreeify\ (rev\ (RBT\text{-Impl.fold}\ (\lambda a\ b.\ RBT\text{-Impl.fold}\ (\lambda c\ d\ rest.\ ((a, c), f\ a\ b\ c\ d)\ \# rest)\ rbt2)\ rbt1\ []))$ 

```

unfolding $rbt\text{-product-def}\ alist\text{-product-code}\ RBT\text{-Impl.fold-def}$..

end

context

fixes $leq\text{-}a :: 'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\langle \sqsubseteq_a \rangle$ 50)

and $less\text{-}a :: 'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\langle \sqsubset_a \rangle$ 50)

and $leq\text{-}b :: 'b \Rightarrow 'b \Rightarrow bool$ (**infix** $\langle \sqsubseteq_b \rangle$ 50)

and $less\text{-}b :: 'b \Rightarrow 'b \Rightarrow bool$ (**infix** $\langle \sqsubset_b \rangle$ 50)

assumes $lin\text{-}a: class.linorder\ leq\text{-}a\ less\text{-}a$

and $lin\text{-}b: class.linorder\ leq\text{-}b\ less\text{-}b$

begin

abbreviation (*input*) $less\text{-}eq\text{-}prod' :: ('a \times 'b) \Rightarrow ('a \times 'b) \Rightarrow bool$ (**infix** $\langle \sqsubseteq \rangle$ 50)

where $less\text{-}eq\text{-}prod' \equiv less\text{-}eq\text{-}prod\ leq\text{-}a\ less\text{-}a\ leq\text{-}b$

abbreviation (*input*) $less\text{-}prod' :: ('a \times 'b) \Rightarrow ('a \times 'b) \Rightarrow bool$ (**infix** $\langle \sqsubset \rangle$ 50)

where $less\text{-}prod' \equiv less\text{-}prod\ leq\text{-}a\ less\text{-}a\ less\text{-}b$

lemmas $linorder\text{-}prod = linorder\text{-}prod[OF\ lin\text{-}a\ lin\text{-}b]$

lemma $sorted\text{-}alist\text{-}product$:

assumes $xs: linorder.sorted\ leq\text{-}a\ (map\ fst\ xs)$ $distinct\ (map\ fst\ xs)$

and $ys: linorder.sorted\ (\sqsubseteq_b)\ (map\ fst\ ys)$

shows $linorder.sorted\ (\sqsubseteq)\ (map\ fst\ (alist\text{-}product\ f\ xs\ ys))$

proof –

interpret $a: linorder\ (\sqsubseteq_a)\ (\sqsubset_a)$ **by**(*fact* $lin\text{-}a$)

```

note [simp] =
  linorder.sorted0[OF linorder-prod] linorder.sorted1[OF linorder-prod]
  linorder.sorted-append[OF linorder-prod]
  linorder.sorted1[OF lin-b]

show ?thesis using xs
proof(induction xs)
  case Nil show ?case by simp
next
  case (Cons x xs)
  obtain a b where x: x = (a, b) by(cases x)
  have linorder.sorted (□) (map fst (map (λ(c, d). ((a, c), f a b c d)) ys))
    using ys by(induct ys) auto
  thus ?case using x Cons
    by(fastforce simp add: set-alist-product a.not-less dest: bspec a.order-antisym
intro: rev-image-eqI)
  qed
qed

lemma is-rbt-rbt-product:
  [| ord.is-rbt (□a) rbt1; ord.is-rbt (□b) rbt2 |]
  ⇒ ord.is-rbt (□) (rbt-product f rbt1 rbt2)
unfolding rbt-product-def
by(blast intro: linorder.is-rbt-rbtreeify[OF linorder-prod] sorted-alist-product linorder.rbt-sorted-entries[OF
lin-a] ord.is-rbt-rbt-sorted linorder.distinct-entries[OF lin-a] linorder.rbt-sorted-entries[OF
lin-b] distinct-alist-product linorder.distinct-entries[OF lin-b])

lemma rbt-lookup-rbt-product:
  [| ord.is-rbt (□a) rbt1; ord.is-rbt (□b) rbt2 |]
  ⇒ ord.rbt-lookup (□) (rbt-product f rbt1 rbt2) (a, c) =
    (case ord.rbt-lookup (□a) rbt1 a of None ⇒ None
     | Some b ⇒ map-option (f a b c) (ord.rbt-lookup (□b) rbt2 c))
by(simp add: rbt-product-def linorder.rbt-lookup-rbtreeify[OF linorder-prod] linorder.is-rbt-rbtreeify[OF
linorder-prod] sorted-alist-product linorder.rbt-sorted-entries[OF lin-a] ord.is-rbt-rbt-sorted
linorder.distinct-entries[OF lin-a] linorder.rbt-sorted-entries[OF lin-b] distinct-alist-product
linorder.distinct-entries[OF lin-b] map-of-alist-product linorder.map-of-entries[OF
lin-a] linorder.map-of-entries[OF lin-b] cong: option.case-cong)

end

hide-const less-eq-prod' less-prod'

```

3.6.3 Build an RBT where keys are paired with themselves

```

primrec RBT-Impl-diag :: ('a, 'b) rbt ⇒ ('a × 'a, 'b) rbt
where
  RBT-Impl-diag rbt.Empty = rbt.Empty
  | RBT-Impl-diag (rbt.Branch c l k v r) = rbt.Branch c (RBT-Impl-diag l) (k, k) v
  (RBT-Impl-diag r)

```

lemma *entries-RBT-Impl-diag*:

$RBT-Impl.entries (RBT-Impl-diag t) = map (\lambda(k, v). ((k, k), v)) (RBT-Impl.entries t)$
by(*induct t simp-all*)

lemma *keys-RBT-Impl-diag*:

$RBT-Impl.keys (RBT-Impl-diag t) = map (\lambda k. (k, k)) (RBT-Impl.keys t)$
by(*simp add: RBT-Impl.keys-def entries-RBT-Impl-diag split-beta*)

lemma *rbt-sorted-RBT-Impl-diag*:

$ord.rbt-sorted lt t \implies ord.rbt-sorted (less-prod leq lt lt) (RBT-Impl-diag t)$
by(*induct t*)(*auto simp add: ord.rbt-sorted.simps ord.rbt-less-prop ord.rbt-greater-prop keys-RBT-Impl-diag*)

lemma *bheight-RBT-Impl-diag*:

$bheight (RBT-Impl-diag t) = bheight t$
by(*induct t simp-all*)

lemma *inv-RBT-Impl-diag*:

assumes *inv1 t inv2 t*
shows $inv1 (RBT-Impl-diag t) \quad inv2 (RBT-Impl-diag t)$
and $color-of t = color.B \implies color-of (RBT-Impl-diag t) = color.B$
using *assms* **by**(*induct t*)(*auto simp add: bheight-RBT-Impl-diag*)

lemma *is-rbt-RBT-Impl-diag*:

$ord.is-rbt lt t \implies ord.is-rbt (less-prod leq lt lt) (RBT-Impl-diag t)$
by(*simp add: ord.is-rbt-def rbt-sorted-RBT-Impl-diag inv-RBT-Impl-diag*)

lemma (**in** *linorder*) *rbt-lookup-RBT-Impl-diag*:

$ord.rbt-lookup (less-prod (\leq) (<) (<)) (RBT-Impl-diag t) =$
 $(\lambda(k, k'). \text{if } k = k' \text{ then } ord.rbt-lookup (<) t k \text{ else None})$
by(*induct t*)(*auto simp add: ord.rbt-lookup.simps fun-eq-iff*)

3.6.4 Folding and quantifiers over RBTs

definition *RBT-Impl-fold1* :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow ('a, unit) RBT-Impl.rbt \Rightarrow 'a$
where $RBT-Impl-fold1 f rbt = fold f (tl (RBT-Impl.keys rbt)) (hd (RBT-Impl.keys rbt))$

lemma *RBT-Impl-fold1-simps* [*simp, code*]:

$RBT-Impl-fold1 f rbt.Empty = undefined$
 $RBT-Impl-fold1 f (Branch c rbt.Empty k v r) = RBT-Impl.fold (\lambda k v. f k) r k$
 $RBT-Impl-fold1 f (Branch c (Branch c' l' k' v' r') k v r) =$
 $RBT-Impl.fold (\lambda k v. f k) r (f k (RBT-Impl-fold1 f (Branch c' l' k' v' r')))$
by(*simp-all add: RBT-Impl-fold1-def RBT-Impl.keys-def fold-map RBT-Impl.fold-def split-def o-def tl-append hd-def split: list.split*)

definition *RBT-Impl-rbt-all* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a, 'b) rbt \Rightarrow bool$

where `[code del]: RBT-Impl-rbt-all P rbt = (∀ (k, v) ∈ set (RBT-Impl.entries rbt). P k v)`

lemma `RBT-Impl-rbt-all-simps [simp, code]:`

`RBT-Impl-rbt-all P rbt.Empty ⟷ True`

`RBT-Impl-rbt-all P (Branch c l k v r) ⟷ P k v ∧ RBT-Impl-rbt-all P l ∧ RBT-Impl-rbt-all P r`

by(`auto simp add: RBT-Impl-rbt-all-def`)

definition `RBT-Impl-rbt-ex :: ('a ⇒ 'b ⇒ bool) ⇒ ('a, 'b) rbt ⇒ bool`

where `[code del]: RBT-Impl-rbt-ex P rbt = (∃ (k, v) ∈ set (RBT-Impl.entries rbt). P k v)`

lemma `RBT-Impl-rbt-ex-simps [simp, code]:`

`RBT-Impl-rbt-ex P rbt.Empty ⟷ False`

`RBT-Impl-rbt-ex P (Branch c l k v r) ⟷ P k v ∨ RBT-Impl-rbt-ex P l ∨ RBT-Impl-rbt-ex P r`

by(`auto simp add: RBT-Impl-rbt-ex-def`)

3.6.5 List fusion for RBTs

type-synonym `('a, 'b, 'c) rbt-generator-state = ('c × ('a, 'b) RBT-Impl.rbt) list × ('a, 'b) RBT-Impl.rbt`

fun `rbt-has-next :: ('a, 'b, 'c) rbt-generator-state ⇒ bool`

where

`rbt-has-next ([], rbt.Empty) = False`

| `rbt-has-next - = True`

fun `rbt-keys-next :: ('a, 'b, 'a) rbt-generator-state ⇒ 'a × ('a, 'b, 'a) rbt-generator-state`

where

`rbt-keys-next ((k, t) # kts, rbt.Empty) = (k, kts, t)`

| `rbt-keys-next (kts, rbt.Branch c l k v r) = rbt-keys-next ((k, r) # kts, l)`

lemma `rbt-generator-induct [case-names empty split shuffle]:`

assumes `P ([], rbt.Empty)`

and `∧ k t kts. P (kts, t) ⇒ P ((k, t) # kts, rbt.Empty)`

and `∧ kts c l k v r. P ((f k v, r) # kts, l) ⇒ P (kts, Branch c l k v r)`

shows `P ktst`

using `assms`

apply(`induction-schema`)

apply `pat-completeness`

apply(`relation measure (λ(kts, t). size-list (λ(k, t). size-rbt (λ-. 1) (λ-. 1) t) kts + size-rbt (λ-. 1) (λ-. 1) t)`)

apply `simp-all`

done

lemma `terminates-rbt-keys-generator:`

`terminates (rbt-has-next, rbt-keys-next)`

proof

fix $ktst :: ('a \times ('a, 'b) \text{rbt}) \text{list} \times ('a, 'b) \text{rbt}$

show $ktst \in \text{terminates-on} (\text{rbt-has-next}, \text{rbt-keys-next})$

by(*induct ktst taking: $\lambda k \cdot k$ rule: rbt-generator-induct*)(*auto 4 3 intro: terminates-on.intros elim: terminates-on.cases*)

qed

lift-definition $\text{rbt-keys-generator} :: ('a, ('a, 'b, 'a) \text{rbt-generator-state}) \text{generator}$
is $(\text{rbt-has-next}, \text{rbt-keys-next})$

by(*rule terminates-rbt-keys-generator*)

definition $\text{rbt-init} :: ('a, 'b) \text{rbt} \Rightarrow ('a, 'b, 'c) \text{rbt-generator-state}$

where $\text{rbt-init} = \text{Pair} []$

lemma $\text{has-next-rbt-keys-generator}$ [*simp*]:

$\text{list.has-next rbt-keys-generator} = \text{rbt-has-next}$

by *transfer simp*

lemma $\text{next-rbt-keys-generator}$ [*simp*]:

$\text{list.next rbt-keys-generator} = \text{rbt-keys-next}$

by *transfer simp*

lemma $\text{unfoldr-rbt-keys-generator-aux}$:

$\text{list.unfoldr rbt-keys-generator} (kts, t) =$

$\text{RBT-Impl.keys } t @ \text{concat} (\text{map} (\lambda(k, t). k \# \text{RBT-Impl.keys } t) kts)$

proof(*induct (kts, t) arbitrary: kts t taking: $\lambda k \cdot k$ rule: rbt-generator-induct*)

case empty thus ?case

by(*simp add: list.unfoldr.simps*)

next

case split thus ?case

by(*subst list.unfoldr.simps*) *simp*

next

case shuffle thus ?case

by(*subst list.unfoldr.simps*)(*subst (asm) list.unfoldr.simps, simp*)

qed

corollary $\text{unfoldr-rbt-keys-generator}$:

$\text{list.unfoldr rbt-keys-generator} (\text{rbt-init } t) = \text{RBT-Impl.keys } t$

by(*simp add: unfoldr-rbt-keys-generator-aux rbt-init-def*)

fun $\text{rbt-entries-next} ::$

$('a, 'b, 'a \times 'b) \text{rbt-generator-state} \Rightarrow ('a \times 'b) \times ('a, 'b, 'a \times 'b) \text{rbt-generator-state}$

where

$\text{rbt-entries-next} ((kv, t) \# kts, \text{rbt.Empty}) = (kv, kts, t)$

$|\text{rbt-entries-next} (kts, \text{rbt.Branch } c \ l \ k \ v \ r) = \text{rbt-entries-next} (((k, v), r) \# kts, l)$

lemma $\text{terminates-rbt-entries-generator}$:

$\text{terminates} (\text{rbt-has-next}, \text{rbt-entries-next})$

```

proof(rule terminatesI)
  fix ktst :: ('a, 'b, 'a × 'b) rbt-generator-state
  show ktst ∈ terminates-on (rbt-has-next, rbt-entries-next)
    by(induct ktst taking: Pair rule: rbt-generator-induct)(auto 4 3 intro: termi-
      nates-on.intros elim: terminates-on.cases)
qed

```

```

lift-definition rbt-entries-generator :: ('a × 'b, ('a, 'b, 'a × 'b) rbt-generator-state)
  generator
  is (rbt-has-next, rbt-entries-next)
by(rule terminates-rbt-entries-generator)

```

```

lemma has-next-rbt-entries-generator [simp]:
  list.has-next rbt-entries-generator = rbt-has-next
by transfer simp

```

```

lemma next-rbt-entries-generator [simp]:
  list.next rbt-entries-generator = rbt-entries-next
by transfer simp

```

```

lemma unfoldr-rbt-entries-generator-aux:
  list.unfoldr rbt-entries-generator (kts, t) =
    RBT-Impl.entries t @ concat (map (λ(k, t). k # RBT-Impl.entries t) kts)
proof(induct (kts, t) arbitrary: kts t taking: Pair rule: rbt-generator-induct)
  case empty thus ?case
    by(simp add: list.unfoldr.simps)
next
  case split thus ?case
    by(subst list.unfoldr.simps) simp
next
  case shuffle thus ?case
    by(subst list.unfoldr.simps)(subst (asm) list.unfoldr.simps, simp)
qed

```

```

corollary unfoldr-rbt-entries-generator:
  list.unfoldr rbt-entries-generator (rbt-init t) = RBT-Impl.entries t
by(simp add: unfoldr-rbt-entries-generator-aux rbt-init-def)

```

```

end

```

```

theory RBT-Mapping2
imports
  Collection-Order
  RBT-ext
  Deriving.RBT-Comparator-Impl
begin

```

3.7 Mappings implemented by red-black trees

lemma *distinct-map-filterI*: $\text{distinct} (\text{map } f \text{ } xs) \implies \text{distinct} (\text{map } f (\text{filter } P \text{ } xs))$
by(*induct xs*) *auto*

lemma *map-of-filter-apply*:
 $\text{distinct} (\text{map } \text{fst } xs)$
 $\implies \text{map-of} (\text{filter } P \text{ } xs) \text{ } k =$
(case map-of xs k of None \Rightarrow None | Some v \Rightarrow if P (k, v) then Some v else None)
by(*induct xs*)(*auto simp add: map-of-eq-None-iff split: option.split*)

3.7.1 Type definition

typedef (**overloaded**) (*'a*, *'b*) *mapping-rbt*
 $= \{t :: ('a :: \text{ccompare}, 'b) \text{RBT-Impl.rbt. ord.is-rbt cless } t \vee \text{ID CCOMPARE}('a)$
 $= \text{None}\}$
morphisms *impl-of Mapping-RBT'*
proof
show *RBT-Impl.Empty* \in *?mapping-rbt* **by**(*simp add: ord.Empty-is-rbt*)
qed

definition *Mapping-RBT* :: (*'a* :: *ccompare*, *'b*) *rbt* \Rightarrow (*'a*, *'b*) *mapping-rbt*
where
 $\text{Mapping-RBT } t = \text{Mapping-RBT}'$
(if ord.is-rbt cless t \vee ID CCOMPARE('a) = None then t
else RBT-Impl.fold (ord.rbt-insert cless) t rbt.Empty)

lemma *Mapping-RBT-inverse*:
fixes *y* :: (*'a* :: *ccompare*, *'b*) *rbt*
assumes $y \in \{t. \text{ord.is-rbt cless } t \vee \text{ID CCOMPARE}('a) = \text{None}\}$
shows *impl-of (Mapping-RBT y) = y*
using *assms* **by**(*auto simp add: Mapping-RBT-def Mapping-RBT'-inverse*)

lemma *impl-of-inverse: Mapping-RBT (impl-of t) = t*
by(*cases t*)(*auto simp add: Mapping-RBT'-inverse Mapping-RBT-def*)

lemma *type-definition-mapping-rbt'*:
type-definition impl-of Mapping-RBT
 $\{t :: ('a, 'b) \text{rbt. ord.is-rbt cless } t \vee \text{ID CCOMPARE}('a :: \text{ccompare}) = \text{None}\}$
by *unfold-locales(rule mapping-rbt.impl-of impl-of-inverse Mapping-RBT-inverse)+*

lemmas *Mapping-RBT-cases*[*cases type: mapping-rbt*] =
type-definition.Abs-cases[OF type-definition-mapping-rbt']
and *Mapping-RBT-induct*[*induct type: mapping-rbt*] =
type-definition.Abs-induct[OF type-definition-mapping-rbt'] **and**
Mapping-RBT-inject = type-definition.Abs-inject[OF type-definition-mapping-rbt']

lemma *rbt-eq-iff*:
 $t1 = t2 \iff \text{impl-of } t1 = \text{impl-of } t2$

by (*simp add: impl-of-inject*)

lemma *rbt-eqI*:

impl-of t1 = impl-of t2 \implies t1 = t2

by (*simp add: rbt-eq-iff*)

lemma *Mapping-RBT-impl-of* [*simp*]:

Mapping-RBT (impl-of t) = t

by (*simp add: impl-of-inverse*)

3.7.2 Operations

setup-lifting *type-definition-mapping-rbt'*

context *fixes dummy :: 'a :: ccompare begin*

lift-definition *lookup :: ('a, 'b) mapping-rbt \Rightarrow 'a \rightarrow 'b is rbt-comp-lookup ccomp*

.

lift-definition *empty :: ('a, 'b) mapping-rbt is RBT-Impl.Empty*

by(*simp add: ord.Empty-is-rbt*)

lift-definition *insert :: 'a \Rightarrow 'b \Rightarrow ('a, 'b) mapping-rbt \Rightarrow ('a, 'b) mapping-rbt is rbt-comp-insert ccomp*

by(*auto 4 3 intro: linorder.rbt-insert-is-rbt ID-ccompare simp: rbt-comp-insert[OF ID-ccompare[?]]*)

lift-definition *delete :: 'a \Rightarrow ('a, 'b) mapping-rbt \Rightarrow ('a, 'b) mapping-rbt is rbt-comp-delete ccomp*

by(*auto 4 3 intro: linorder.rbt-delete-is-rbt ID-ccompare simp: rbt-comp-delete[OF ID-ccompare[?]]*)

lift-definition *bulkload :: ('a \times 'b) list \Rightarrow ('a, 'b) mapping-rbt is rbt-comp-bulkload ccomp*

by(*auto 4 3 intro: linorder.rbt-bulkload-is-rbt ID-ccompare simp: rbt-comp-bulkload[OF ID-ccompare[?]]*)

lift-definition *map-entry :: 'a \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a, 'b) mapping-rbt \Rightarrow ('a, 'b) mapping-rbt is*

rbt-comp-map-entry ccomp

by(*auto simp: ord.rbt-map-entry-is-rbt rbt-comp-map-entry[OF ID-ccompare[?]]*)

lift-definition *map :: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a, 'b) mapping-rbt \Rightarrow ('a, 'c) mapping-rbt is RBT-Impl.map*

by(*simp add: ord.map-is-rbt*)

lift-definition *entries :: ('a, 'b) mapping-rbt \Rightarrow ('a \times 'b) list is RBT-Impl.entries*

.

lift-definition *keys* :: ('a, 'b) mapping-rbt ⇒ 'a set **is** set ◦ RBT-Impl.keys .

lift-definition *fold* :: ('a ⇒ 'b ⇒ 'c ⇒ 'c) ⇒ ('a, 'b) mapping-rbt ⇒ 'c ⇒ 'c **is** RBT-Impl.fold .

lift-definition *is-empty* :: ('a, 'b) mapping-rbt ⇒ bool **is** case-rbt True (λ- - - - . False) .

lift-definition *filter* :: ('a × 'b ⇒ bool) ⇒ ('a, 'b) mapping-rbt ⇒ ('a, 'b) mapping-rbt **is**

λP t. rbtreeify (List.filter P (RBT-Impl.entries t))

by(auto intro!: linorder.is-rbt-rbtreeify ID-ccompare linorder.sorted-filter linorder.rbt-sorted-entries ord.is-rbt-rbt-sorted linorder.distinct-entries distinct-map-filterI simp add: filter-map[symmetric])

lift-definition *join* ::

('a ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ ('a, 'b) mapping-rbt ⇒ ('a, 'b) mapping-rbt ⇒ ('a, 'b) mapping-rbt

is rbt-comp-union-with-key ccomp

by(auto 4 3 intro: linorder.is-rbt-rbt-unionwk ID-ccompare simp: rbt-comp-union-with-key[OF ID-ccompare])

lift-definition *meet* ::

('a ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ ('a, 'b) mapping-rbt ⇒ ('a, 'b) mapping-rbt ⇒ ('a, 'b) mapping-rbt

is rbt-comp-inter-with-key ccomp

by(auto 4 3 intro: linorder.rbt-interwk-is-rbt ID-ccompare ord.is-rbt-rbt-sorted simp: rbt-comp-inter-with-key[OF ID-ccompare])

lift-definition *all* :: ('a ⇒ 'b ⇒ bool) ⇒ ('a, 'b) mapping-rbt ⇒ bool

is RBT-Impl-rbt-all .

lift-definition *ex* :: ('a ⇒ 'b ⇒ bool) ⇒ ('a, 'b) mapping-rbt ⇒ bool

is RBT-Impl-rbt-ex .

lift-definition *product* ::

('a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ 'e) ⇒ ('a, 'b) mapping-rbt

⇒ ('c :: ccompare, 'd) mapping-rbt ⇒ ('a × 'c, 'e) mapping-rbt

is rbt-product

by (fastforce intro: is-rbt-rbt-product ID-ccompare simp add: lt-of-comp-less-prod ccompare-prod-def ID-Some ID-None split: option.split-asm)

lift-definition *diag* ::

('a, 'b) mapping-rbt ⇒ ('a × 'a, 'b) mapping-rbt

is RBT-Impl-diag

by(auto simp add: lt-of-comp-less-prod ccompare-prod-def ID-Some ID-None is-rbt-RBT-Impl-diag split: option.split-asm)

lift-definition *init* :: ('a, 'b) mapping-rbt ⇒ ('a, 'b, 'c) rbt-generator-state

is rbt-init .

end

3.7.3 Properties

lemma *unfoldr-rbt-entries-generator*:

list.unfoldr rbt-entries-generator (init t) = entries t
by *transfer(simp add: unfoldr-rbt-entries-generator)*

lemma *lookup-RBT*:

ord.is-rbt cless t \implies
lookup (Mapping-RBT t) = rbt-comp-lookup ccomp t
by(*simp add: lookup-def Mapping-RBT-inverse*)

lemma *lookup-impl-of*:

rbt-comp-lookup ccomp (impl-of t) = lookup t
by(*transfer*) *simp*

lemma *entries-impl-of*:

RBT-Impl.entries (impl-of t) = entries t
by *transfer simp*

lemma *keys-impl-of*:

set (RBT-Impl.keys (impl-of t)) = keys t
by (*simp add: keys-def*)

lemma *lookup-empty [simp]*:

lookup empty = Map.empty
by *transfer (simp add: fun-eq-iff ord.rbt-lookup.simps)*

lemma *fold-conv-fold*:

fold f t = List.fold (case-prod f) (entries t)
by *transfer(simp add: RBT-Impl.fold-def)*

lemma *is-empty-empty [simp]*:

is-empty t \iff t = empty
by *transfer (simp split: rbt.split)*

context assumes *ID-ccompare-neq-None: ID CCOMPARE('a :: ccompare) \neq None*
begin

lemma *mapping-linorder: class.linorder (cless-eq :: 'a \Rightarrow 'a \Rightarrow bool) cless*
using *ID-ccompare-neq-None* **by**(*clarsimp*)(*rule ID-ccompare*)

lemma *mapping-comparator: comparator (ccomp :: 'a comparator)*
using *ID-ccompare-neq-None* **by**(*clarsimp*)(*rule ID-ccompare'*)

lemmas *rbt-comp[simp] = rbt-comp-simps[OF mapping-comparator]*

lemma *is-rbt-impl-of* [*simp*, *intro*]:
fixes $t :: ('a, 'b)$ *mapping-rbt*
shows *ord.is-rbt cless (impl-of t)*
using *ID-ccompare-neq-None impl-of [of t] by auto*

lemma *lookup-insert* [*simp*]:
 $lookup (insert (k :: 'a) v t) = (lookup t)(k \mapsto v)$
by transfer (*simp add: ID-ccompare-neq-None*
linorder.rbt-lookup-rbt-insert[OF mapping-linorder])

lemma *lookup-delete* [*simp*]:
 $lookup (delete (k :: 'a) t) = (lookup t)(k := None)$
by transfer(*simp add: ID-ccompare-neq-None linorder.rbt-lookup-rbt-delete[OF mapping-linorder] restrict-complement-singleton-eq*)

lemma *map-of-entries* [*simp*]:
 $map-of (entries (t :: ('a, 'b)$ *mapping-rbt*)) = *lookup t*
by transfer(*simp add: ID-ccompare-neq-None linorder.map-of-entries[OF mapping-linorder] ord.is-rbt-rbt-sorted*)

lemma *entries-lookup*:
 $entries (t1 :: ('a, 'b)$ *mapping-rbt*) = *entries t2* \longleftrightarrow $lookup t1 = lookup t2$
by transfer(*simp add: ID-ccompare-neq-None linorder.entries-rbt-lookup[OF mapping-linorder] ord.is-rbt-rbt-sorted*)

lemma *lookup-bulkload* [*simp*]:
 $lookup (bulkload xs) = map-of (xs :: ('a \times 'b)$ *list*)
by transfer(*simp add: linorder.rbt-lookup-rbt-bulkload[OF mapping-linorder]*)

lemma *lookup-map-entry* [*simp*]:
 $lookup (map-entry (k :: 'a) f t) = (lookup t)(k := map-option f (lookup t k))$
by transfer(*simp add: ID-ccompare-neq-None linorder.rbt-lookup-rbt-map-entry[OF mapping-linorder]*)

lemma *lookup-map* [*simp*]:
 $lookup (map f t) (k :: 'a) = map-option (f k) (lookup t k)$
by transfer(*simp add: linorder.rbt-lookup-map[OF mapping-linorder]*)

lemma *RBT-lookup-empty* [*simp*]:
 $ord.rbt-lookup cless (t :: ('a, 'b)$ *RBT-Impl.rbt*) = *Map.empty* \longleftrightarrow $t = RBT-Impl.Empty$
proof –
interpret *linorder cless-eq :: 'a* \Rightarrow *'a* \Rightarrow *bool cless* **by**(*rule mapping-linorder*)
show *?thesis* **by**(*cases t*)(*auto simp add: fun-eq-iff*)
qed

lemma *lookup-empty-empty* [*simp*]:
 $lookup t = Map.empty \longleftrightarrow (t :: ('a, 'b)$ *mapping-rbt*) = *empty*
by transfer *simp*

lemma *finite-dom-lookup* [*simp*]: *finite* (*dom* (*lookup* (*t* :: ('a, 'b) *mapping-rbt*)))
by *transfer*(*auto simp add: linorder.finite-dom-rbt-lookup*[*OF mapping-linorder*])

lemma *card-com-lookup* [*unfolded length-map, simp*]:

card (*dom* (*lookup* (*t* :: ('a, 'b) *mapping-rbt*))) = *length* (*List.map fst* (*entries t*))
by *transfer*(*auto simp add: linorder.rbt-lookup-keys*[*OF mapping-linorder*] *linorder.distinct-entries*[*OF mapping-linorder*] *RBT-Impl.keys-def* *ord.is-rbt-rbt-sorted* *ID-ccompare-neq-None* *List.card-set simp del: set-map length-map*)

lemma *lookup-join*:

lookup (*join f* (*t1* :: ('a, 'b) *mapping-rbt*) *t2*) =
(λk . *case lookup t1 k of* *None* \Rightarrow *lookup t2 k* | *Some v1* \Rightarrow *Some* (*case lookup t2 k of* *None* \Rightarrow *v1* | *Some v2* \Rightarrow *f k v1 v2*))
by *transfer*(*auto simp add: fun-eq-iff linorder.rbt-lookup-rbt-unionwk*[*OF mapping-linorder*] *ord.is-rbt-rbt-sorted* *ID-ccompare-neq-None split: option.splits*)

lemma *lookup-meet*:

lookup (*meet f* (*t1* :: ('a, 'b) *mapping-rbt*) *t2*) =
(λk . *case lookup t1 k of* *None* \Rightarrow *None* | *Some v1* \Rightarrow *case lookup t2 k of* *None* \Rightarrow *None* | *Some v2* \Rightarrow *Some* (*f k v1 v2*))
by *transfer*(*auto simp add: fun-eq-iff linorder.rbt-lookup-rbt-interwk*[*OF mapping-linorder*] *ord.is-rbt-rbt-sorted* *ID-ccompare-neq-None split: option.splits*)

lemma *lookup-filter* [*simp*]:

lookup (*filter P* (*t* :: ('a, 'b) *mapping-rbt*)) *k* =
(*case lookup t k of* *None* \Rightarrow *None* | *Some v* \Rightarrow *if P* (*k*, *v*) *then Some v else None*)
by *transfer*(*simp split: option.split add: ID-ccompare-neq-None linorder.rbt-lookup-rbtreeify*[*OF mapping-linorder*] *linorder.sorted-filter*[*OF mapping-linorder*] *ord.is-rbt-rbt-sorted* *linorder.rbt-sorted-entries*[*OF mapping-linorder*] *distinct-map-filterI* *linorder.distinct-entries*[*OF mapping-linorder*] *map-of-filter-apply* *linorder.map-of-entries*[*OF mapping-linorder*])

lemma *all-conv-all-lookup*:

all P t \iff (\forall (*k* :: 'a) *v*. *lookup t k* = *Some v* \longrightarrow *P k v*)
by *transfer*(*auto simp add: ID-ccompare-neq-None linorder.rbt-lookup-keys*[*OF mapping-linorder*] *ord.is-rbt-rbt-sorted* *RBT-Impl.keys-def* *RBT-Impl-rbt-all-def* *linorder.map-of-entries*[*OF mapping-linorder, symmetric*] *linorder.distinct-entries*[*OF mapping-linorder*] *dest: map-of-SomeD* *intro: map-of-is-SomeI*)

lemma *ex-conv-ex-lookup*:

ex P t \iff (\exists (*k* :: 'a) *v*. *lookup t k* = *Some v* \wedge *P k v*)
by *transfer*(*auto simp add: ID-ccompare-neq-None linorder.rbt-lookup-keys*[*OF mapping-linorder*] *ord.is-rbt-rbt-sorted* *RBT-Impl.keys-def* *RBT-Impl-rbt-ex-def* *linorder.map-of-entries*[*OF mapping-linorder, symmetric*] *linorder.distinct-entries*[*OF mapping-linorder*] *intro: map-of-is-SomeI*)

lemma *diag-lookup*:

lookup (*diag t*) = (λ (*k* :: 'a, *k'*). *if k = k'* *then lookup t k else None*)
using *linorder.rbt-lookup-RBT-Impl-diag*[**where** *?b='b*, *OF mapping-linorder*]
apply *transfer*

```

apply (clarsimp simp add: ID-ccompare-neq-None ccompare-prod-def lt-of-comp-less-prod[symmetric]

  rbt-comp-lookup[OF comparator-prod[OF mapping-comparator mapping-comparator],
  symmetric]
  ID-Some split: option.split)
apply (unfold rbt-comp-lookup[OF mapping-comparator], simp)
done

context assumes ID-ccompare-neq-None': ID CCOMPARE('b :: ccompare) ≠
None
begin

lemma mapping-linorder': class.linorder (cless-eq :: 'b ⇒ 'b ⇒ bool) cless
using ID-ccompare-neq-None' by(clarsimp)(rule ID-ccompare)

lemma mapping-comparator': comparator (ccomp :: 'b comparator)
using ID-ccompare-neq-None' by(clarsimp)(rule ID-ccompare)

lemmas rbt-comp'[simp] = rbt-comp-simps[OF mapping-comparator']

lemma ccomp-comparator-prod:
  ccomp = (comparator-prod ccomp ccomp :: ('a × 'b) comparator)
  by(simp add: ccompare-prod-def lt-of-comp-less-prod ID-ccompare-neq-None ID-ccompare-neq-None'
  ID-Some split: option.splits)

lemma lookup-product:
  lookup (product f rbt1 rbt2) (a :: 'a, b :: 'b) =
  (case lookup rbt1 a of None ⇒ None
  | Some c ⇒ map-option (f a c b) (lookup rbt2 b))
using mapping-linorder mapping-linorder'
apply transfer
apply (unfold ccomp-comparator-prod rbt-comp-lookup[OF comparator-prod[OF map-
  ping-comparator mapping-comparator']]
  rbt-comp rbt-comp' lt-of-comp-less-prod)
apply (simp add: ID-ccompare-neq-None ID-ccompare-neq-None' rbt-lookup-rbt-product)
done
end

end

hide-const (open) impl-of lookup empty insert delete
  entries keys bulkload map-entry map fold join meet filter all ex product diag init

end

theory AssocList imports
  HOL-Library.DAList
begin

```

3.8 Additional operations for associative lists

3.8.1 Operations on the raw type

primrec *update-with-aux* :: 'val \Rightarrow 'key \Rightarrow ('val \Rightarrow 'val) \Rightarrow ('key \times 'val) list \Rightarrow ('key \times 'val) list

where

update-with-aux v k f [] = [(k, f v)]
| *update-with-aux* v k f (p # ps) = (if (fst p = k) then (k, f (snd p)) # ps else p # *update-with-aux* v k f ps)

Do not use *AList.delete* because this traverses all the list even if it has found the key. We do not have to keep going because we use the invariant that keys are distinct.

fun *delete-aux* :: 'key \Rightarrow ('key \times 'val) list \Rightarrow ('key \times 'val) list

where

delete-aux k [] = []
| *delete-aux* k ((k', v) # xs) = (if k = k' then xs else (k', v) # *delete-aux* k xs)

definition *zip-with-index-from* :: nat \Rightarrow 'a list \Rightarrow (nat \times 'a) list **where**

zip-with-index-from n xs = zip [n.. $n + \text{length } xs$] xs

abbreviation *zip-with-index* :: 'a list \Rightarrow (nat \times 'a) list **where**

zip-with-index \equiv *zip-with-index-from* 0

lemma *update-conv-update-with-aux*:

AList.update k v xs = *update-with-aux* v k (λ -. v) xs

by(*induct* xs) *simp-all*

lemma *map-of-update-with-aux'*:

map-of (*update-with-aux* v k f ps) k' = ((*map-of* ps)(k \mapsto (case *map-of* ps k of None \Rightarrow f v | Some v \Rightarrow f v))) k'

by(*induct* ps) *auto*

lemma *map-of-update-with-aux*:

map-of (*update-with-aux* v k f ps) = (*map-of* ps)(k \mapsto (case *map-of* ps k of None \Rightarrow f v | Some v \Rightarrow f v))

by(*simp add: fun-eq-iff map-of-update-with-aux'*)

lemma *dom-update-with-aux*: *fst* ' set (*update-with-aux* v k f ps) = {k} \cup *fst* ' set ps

by (*induct* ps) *auto*

lemma *distinct-update-with-aux [simp]*:

distinct (*map* *fst* (*update-with-aux* v k f ps)) = *distinct* (*map* *fst* ps)

by(*induct* ps)(*auto simp add: dom-update-with-aux*)

lemma *set-update-with-aux*:

distinct (*map* *fst* xs)

$\implies \text{set } (\text{update-with-aux } v \ k \ f \ xs) = (\text{set } xs - \{k\} \times UNIV \cup \{(k, f \ (\text{case } \text{map-of } xs \ k \ \text{of } \text{None} \Rightarrow v \mid \text{Some } v \Rightarrow v))\})$
by(*induct xs*)(*auto intro: rev-image-eqI*)

lemma *set-delete-aux*: $\text{distinct } (\text{map } \text{fst } xs) \implies \text{set } (\text{delete-aux } k \ xs) = \text{set } xs - \{k\} \times UNIV$
apply(*induct xs*)
apply *simp-all*
apply *clarsimp*
apply(*fastforce intro: rev-image-eqI*)
done

lemma *dom-delete-aux*: $\text{distinct } (\text{map } \text{fst } ps) \implies \text{fst } ' \text{set } (\text{delete-aux } k \ ps) = \text{fst } ' \text{set } ps - \{k\}$
by(*auto simp add: set-delete-aux*)

lemma *distinct-delete-aux* [*simp*]:
 $\text{distinct } (\text{map } \text{fst } ps) \implies \text{distinct } (\text{map } \text{fst } (\text{delete-aux } k \ ps))$
proof(*induct ps*)
case *Nil* **thus** *?case by simp*
next
case (*Cons a ps*)
obtain *k' v* **where** *a = (k', v)* **by**(*cases a*)
show *?case*
proof(*cases k' = k*)
case *True* **with** *Cons a* **show** *?thesis by simp*
next
case *False*
with *Cons a* **have** $k' \notin \text{fst } ' \text{set } ps$ **distinct** (*map fst ps*) **by** *simp-all*
with *False a* **have** $k' \notin \text{fst } ' \text{set } (\text{delete-aux } k \ ps)$
by(*auto dest!: dom-delete-aux[where k=k]*)
with *Cons a* **show** *?thesis by simp*
qed
qed

lemma *map-of-delete-aux'*:
 $\text{distinct } (\text{map } \text{fst } xs) \implies \text{map-of } (\text{delete-aux } k \ xs) = (\text{map-of } xs)(k := \text{None})$
by(*induct xs*)(*fastforce simp add: map-of-eq-None-iff fun-upd-twist*)

lemma *map-of-delete-aux*:
 $\text{distinct } (\text{map } \text{fst } xs) \implies \text{map-of } (\text{delete-aux } k \ xs) \ k' = ((\text{map-of } xs)(k := \text{None})) \ k'$
by(*simp add: map-of-delete-aux'*)

lemma *delete-aux-eq-Nil-conv*: $\text{delete-aux } k \ ts = [] \longleftrightarrow ts = [] \vee (\exists v. ts = [(k, v)])$
by(*cases ts*)(*auto split: if-split-asm*)

lemma *zip-with-index-from-simps* [*simp, code*]:

$zip-with-index-from\ n\ [] = []$
 $zip-with-index-from\ n\ (x \# xs) = (n, x) \# zip-with-index-from\ (Suc\ n)\ xs$
by(*simp-all add: zip-with-index-from-def upt-rec del: upt.upt-Suc*)

lemma *zip-with-index-from-append* [*simp*]:
 $zip-with-index-from\ n\ (xs\ @\ ys) = zip-with-index-from\ n\ xs\ @\ zip-with-index-from\ (n + length\ xs)\ ys$
by(*simp add: zip-with-index-from-def zip-append[symmetric] upt-add-eq-append[symmetric] del: zip-append (simp add: add.assoc)*)

lemma *zip-with-index-from-conv-nth*:
 $zip-with-index-from\ n\ xs = map\ (\lambda i. (n + i, xs\ !\ i))\ [0..<length\ xs]$
by(*induction xs rule: rev-induct(auto simp add: nth-append)*)

lemma *map-of-zip-with-index-from* [*simp*]:
 $map-of\ (zip-with-index-from\ n\ xs)\ i = (if\ i \geq n \wedge i < n + length\ xs\ then\ Some\ (xs\ !\ (i - n))\ else\ None)$
by(*auto simp add: zip-with-index-from-def set-zip intro: exI[where $x=i - n$]*)

lemma *map-of-map'*: $map-of\ (map\ (\lambda(k, v). (k, f\ k\ v))\ xs)\ x = map-option\ (f\ x)\ (map-of\ xs\ x)$
by (*induct xs(auto)*)

3.8.2 Operations on the abstract type ('a, 'b) alist

lift-definition *update-with* :: '*v* \Rightarrow '*k* \Rightarrow ('*v* \Rightarrow '*v*) \Rightarrow ('*k*, '*v*) *alist* \Rightarrow ('*k*, '*v*) *alist*
is *update-with-aux* **by** *simp*

lift-definition *delete* :: '*k* \Rightarrow ('*k*, '*v*) *alist* \Rightarrow ('*k*, '*v*) *alist* **is** *delete-aux*
by *simp*

lift-definition *keys* :: ('*k*, '*v*) *alist* \Rightarrow '*k* *set* **is** *set* \circ *map fst* .

lift-definition *set* :: ('*key*, '*val*) *alist* \Rightarrow ('*key* \times '*val*) *set*
is *List.set* .

lift-definition *map-values* :: ('*key* \Rightarrow '*val* \Rightarrow '*val*') \Rightarrow ('*key*, '*val*) *alist* \Rightarrow ('*key*, '*val*') *alist* **is**
 $\lambda f. map\ (\lambda(x,y). (x, f\ x\ y))$
by(*simp add: o-def split-def*)

lemma *lookup-update-with* [*simp*]:
 $DAList.lookup\ (update-with\ v\ k\ f\ al) = (DAList.lookup\ al)(k \mapsto case\ DAList.lookup\ al\ k\ of\ None \Rightarrow f\ v \mid Some\ v \Rightarrow f\ v)$
by *transfer(simp add: map-of-update-with-aux)*

lemma *lookup-delete* [*simp*]: $DAList.lookup\ (delete\ k\ al) = (DAList.lookup\ al)(k := None)$

by *transfer*(*simp add: map-of-delete-aux'*)

lemma *finite-dom-lookup* [*simp, intro!*]: *finite* (*dom* (*DAList.lookup* *m*))
by *transfer*(*simp add: finite-dom-map-of*)

lemma *update-conv-update-with*: *DAList.update* *k v = update-with* *v k* ($\lambda\cdot. v$)
by(*rule ext*)(*transfer, simp add: update-conv-update-with-aux*)

lemma *lookup-update* [*simp*]: *DAList.lookup* (*DAList.update* *k v al*) = (*DAList.lookup* *al*)(*k* \mapsto *v*)
by(*simp add: update-conv-update-with split: option.split*)

lemma *dom-lookup-keys*: *dom* (*DAList.lookup* *al*) = *keys* *al*
by *transfer*(*simp add: dom-map-of-conv-image-fst*)

lemma *keys-empty* [*simp*]: *keys* *DAList.empty* = {}
by *transfer simp*

lemma *keys-update-with* [*simp*]: *keys* (*update-with* *v k f al*) = *insert* *k* (*keys* *al*)
by(*simp add: dom-lookup-keys[symmetric]*)

lemma *keys-update* [*simp*]: *keys* (*DAList.update* *k v al*) = *insert* *k* (*keys* *al*)
by(*simp add: update-conv-update-with*)

lemma *keys-delete* [*simp*]: *keys* (*delete* *k al*) = *keys* *al* - {*k*}
by(*simp add: dom-lookup-keys[symmetric]*)

lemma *set-empty* [*simp*]: *set* *DAList.empty* = {}
by *transfer simp*

lemma *set-update-with*:
 set (*update-with* *v k f al*) =
 $(set$ *al* - {*k*} \times *UNIV* \cup {(*k*, *f* (*case* *DAList.lookup* *al* *k* of *None* \Rightarrow *v* | *Some* *v* \Rightarrow *v*)))
by *transfer*(*simp add: set-update-with-aux*)

lemma *set-update*: *set* (*DAList.update* *k v al*) = (*set* *al* - {*k*} \times *UNIV* \cup {(*k*, *v*)})
by(*simp add: update-conv-update-with set-update-with*)

lemma *set-delete*: *set* (*delete* *k al*) = *set* *al* - {*k*} \times *UNIV*
by *transfer*(*simp add: set-delete-aux*)

lemma *size-dalist-transfer* [*transfer-rule*]:
includes *lifting-syntax*
shows (*pcr-alist* (=) (=) \implies (=)) *length size*
unfolding *size-alist-def*[*abs-def*]
by *transfer-prover*

lemma *size-eq-card-dom-lookup*: $\text{size } al = \text{card } (\text{dom } (DAList.lookup\ al))$
by *transfer* (*metis comp-apply distinct-card dom-map-of-conv-image-fst image-set length-map*)

hide-const (**open**) *update-with keys set delete*

end

theory *DList-Set imports*

Collection-Eq

Equal

begin

3.9 Sets implemented by distinct lists

3.9.1 Operations on the raw type with parametrised equality

context *equal-base begin*

primrec *list-member* :: $'a\ list \Rightarrow 'a \Rightarrow bool$

where

$list_member\ []\ y \longleftrightarrow False$

| $list_member\ (x\ \#\ xs)\ y \longleftrightarrow equal\ x\ y \vee list_member\ xs\ y$

primrec *list-distinct* :: $'a\ list \Rightarrow bool$

where

$list_distinct\ [] \longleftrightarrow True$

| $list_distinct\ (x\ \#\ xs) \longleftrightarrow \neg list_member\ xs\ x \wedge list_distinct\ xs$

definition *list-insert* :: $'a \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**

$list_insert\ x\ xs = (\text{if } list_member\ xs\ x \text{ then } xs \text{ else } x\ \#\ xs)$

primrec *list-remove1* :: $'a \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**

$list_remove1\ x\ [] = []$

| $list_remove1\ x\ (y\ \#\ xs) = (\text{if } equal\ x\ y \text{ then } xs \text{ else } y\ \#\ list_remove1\ x\ xs)$

primrec *list-remdups* :: $'a\ list \Rightarrow 'a\ list$ **where**

$list_remdups\ [] = []$

| $list_remdups\ (x\ \#\ xs) = (\text{if } list_member\ xs\ x \text{ then } list_remdups\ xs \text{ else } x\ \#\ list_remdups\ xs)$

lemma *list-member-filterD*: $list_member\ (filter\ P\ xs)\ x \Longrightarrow list_member\ xs\ x$

by (*induct xs*) (*auto split: if-split-asm*)

lemma *list-distinct-filter* [*simp*]: $list_distinct\ xs \Longrightarrow list_distinct\ (filter\ P\ xs)$

by (*induct xs*) (*auto dest: list-member-filterD*)

lemma *list-distinct-tl* [*simp*]: $list_distinct\ xs \Longrightarrow list_distinct\ (tl\ xs)$

by(*cases xs*) *simp-all*

end

lemmas [*code*] =
equal-base.list-member.simps
equal-base.list-distinct.simps
equal-base.list-insert-def
equal-base.list-remove1.simps
equal-base.list-remdups.simps

lemmas [*simp*] =
equal-base.list-member.simps
equal-base.list-distinct.simps
equal-base.list-remove1.simps
equal-base.list-remdups.simps

lemma *list-member-conv-member* [*simp*]:
equal-base.list-member (=) = *List.member*
proof(*intro ext*)
fix *xs and x :: 'a*
show *equal-base.list-member* (=) *xs x = List.member xs x*
by (*induct xs*) *auto*
qed

lemma *list-distinct-conv-distinct* [*simp*]:
equal-base.list-distinct (=) = *List.distinct*
proof
fix *xs :: 'a list*
show *equal-base.list-distinct* (=) *xs = distinct xs*
by (*induct xs*) *auto*
qed

lemma *list-insert-conv-insert* [*simp*]:
equal-base.list-insert (=) = *List.insert*
unfolding *equal-base.list-insert-def[abs-def]* *List.insert-def[abs-def]*
by *simp*

lemma *list-remove1-conv-remove1* [*simp*]:
equal-base.list-remove1 (=) = *List.remove1*
unfolding *equal-base.list-remove1-def* *List.remove1-def* ..

lemma *list-remdups-conv-remdups* [*simp*]:
equal-base.list-remdups (=) = *List.remdups*
by (*simp add: equal-base.list-remdups-def remdups-def*)

context *equal* **begin**

lemma *member-insert* [*simp*]: *list-member* (*list-insert x xs*) *y* \longleftrightarrow *equal x y* \vee

list-member xs y
by(*auto simp add: equal-eq*)

lemma *member-remove1 [simp]*:
 $\neg \text{equal } x \ y \implies \text{list-member } (\text{list-remove1 } x \ xs) \ y = \text{list-member } xs \ y$
by(*simp add: equal-eq*)

lemma *distinct-remove1*:
 $\text{list-distinct } xs \implies \text{list-distinct } (\text{list-remove1 } x \ xs)$
by(*simp add: equal-eq*)

lemma *distinct-member-remove1 [simp]*:
 $\text{list-distinct } xs \implies \text{list-member } (\text{list-remove1 } x \ xs) = (\text{list-member } xs)(x := \text{False})$
by(*auto simp add: equal-eq fun-eq-iff*)

end

lemma *ID-ceq*:
 $ID \ CEQ('a :: \text{ceq}) = \text{Some } eq \implies \text{equal } eq$
by(*unfold-locales*)(*clarsimp simp add: ID-ceq*)

3.9.2 The type of distinct lists

typedef (**overloaded**) *'a :: ceq set-dlist =*
 $\{xs :: 'a \ \text{list. equal-base.list-distinct } ceq' \ xs \vee ID \ CEQ('a) = \text{None}\}$
morphisms *list-of-dlist Abs-dlist'*

proof
show $\square \in ?\text{set-dlist}$ **by**(*simp*)
qed

definition *Abs-dlist :: 'a :: ceq list \Rightarrow 'a set-dlist*
where

$Abs-dlist \ xs = Abs-dlist'$
(if equal-base.list-distinct ceq' xs \vee ID CEQ('a) = None then xs
else equal-base.list-remdups ceq' xs)

lemma *Abs-dlist-inverse*:
fixes $y :: 'a :: \text{ceq list}$
assumes $y \in \{xs. \text{equal-base.list-distinct } ceq' \ xs \vee ID \ CEQ('a) = \text{None}\}$
shows $\text{list-of-dlist } (Abs-dlist \ y) = y$
using *assms* **by**(*auto simp add: Abs-dlist-def Abs-dlist'-inverse*)

lemma *list-of-dlist-inverse*: $Abs-dlist \ (\text{list-of-dlist } dxs) = dxs$
by(*cases dxs*)(*simp add: Abs-dlist'-inverse Abs-dlist-def*)

lemma *type-definition-set-dlist'*:
 $\text{type-definition list-of-dlist Abs-dlist}$
 $\{xs :: 'a :: \text{ceq list. equal-base.list-distinct } ceq' \ xs \vee ID \ CEQ('a) = \text{None}\}$

by(*unfold-locales*)(*rule set-dlist.list-of-dlist Abs-dlist-inverse list-of-dlist-inverse*)+

lemmas *Abs-dlist-cases*[*cases type: set-dlist*] =
type-definition.Abs-cases[*OF type-definition-set-dlist*']
and *Abs-dlist-induct*[*induct type: set-dlist*] =
type-definition.Abs-induct[*OF type-definition-set-dlist*'] **and**
Abs-dlist-inject = *type-definition.Abs-inject*[*OF type-definition-set-dlist*']

setup-lifting *type-definition-set-dlist'*

3.9.3 Operations

lift-definition *empty* :: 'a :: ceq set-dlist **is** []
by *simp*

lift-definition *insert* :: 'a :: ceq ⇒ 'a set-dlist ⇒ 'a set-dlist **is**
equal-base.list-insert ceq'
by(*simp add: equal-base.list-insert-def*)

lift-definition *remove* :: 'a :: ceq ⇒ 'a set-dlist ⇒ 'a set-dlist **is**
equal-base.list-remove1 ceq'
by(*auto simp: equal.distinct-remove1 ID-ceq*)

lift-definition *filter* :: ('a :: ceq ⇒ bool) ⇒ 'a set-dlist ⇒ 'a set-dlist **is** *List.filter*
by(*auto simp add: equal-base.list-distinct-filter*)

Derived operations:

lift-definition *null* :: 'a :: ceq set-dlist ⇒ bool **is** *List.null* .

lift-definition *member* :: 'a :: ceq set-dlist ⇒ 'a ⇒ bool **is** *equal-base.list-member ceq'* .

lift-definition *length* :: 'a :: ceq set-dlist ⇒ nat **is** *List.length* .

lift-definition *fold* :: ('a :: ceq ⇒ 'b ⇒ 'b) ⇒ 'a set-dlist ⇒ 'b ⇒ 'b **is** *List.fold* .

lift-definition *foldr* :: ('a :: ceq ⇒ 'b ⇒ 'b) ⇒ 'a set-dlist ⇒ 'b ⇒ 'b **is** *List.foldr*
.

lift-definition *hd* :: 'a :: ceq set-dlist ⇒ 'a **is** *List.hd* .

lift-definition *tl* :: 'a :: ceq set-dlist ⇒ 'a set-dlist **is** *List.tl*
by(*auto simp add: equal-base.list-distinct-tl*)

lift-definition *dlist-all* :: ('a ⇒ bool) ⇒ 'a :: ceq set-dlist ⇒ bool **is** *list-all* .

lift-definition *dlist-ex* :: ('a ⇒ bool) ⇒ 'a :: ceq set-dlist ⇒ bool **is** *list-ex* .

definition *union* :: 'a :: ceq set-dlist ⇒ 'a set-dlist ⇒ 'a set-dlist **where**

union = *fold insert*

lift-definition *product* :: 'a :: ceq set-dlist \Rightarrow 'b :: ceq set-dlist \Rightarrow ('a \times 'b) set-dlist
is $\lambda xs\ ys.$ *rev (concat (map ($\lambda x.$ map (Pair x) ys) xs))*

proof –

fix *xs* :: 'a list **and** *ys* :: 'b list
assume *: *equal-base.list-distinct ceq' xs* \vee *ID CEQ('a) = None*
equal-base.list-distinct ceq' ys \vee *ID CEQ('b) = None*
let *?product* = *concat (map ($\lambda x.$ map (Pair x) ys) xs)*
{ **assume** *neg: ID CEQ('a) \neq None* *ID CEQ('b) \neq None*
hence *ceq': ceq' = ((=) :: 'a \Rightarrow 'a \Rightarrow bool)* *ceq' = ((=) :: 'b \Rightarrow 'b \Rightarrow bool)*
by(*auto intro: equal.equal-eq[OF ID-ceq]*)
with * *neg* **have** *dist: distinct xs* *distinct ys* **by** *simp-all*
hence *distinct ?product*
by(*cases ys = []*)(*auto simp add: distinct-map map-replicate-const intro!*:
inj-onI distinct-concat)
hence *distinct (rev ?product)* **by** *simp*
moreover **have** *ceq' = ((=) :: ('a \times 'b) \Rightarrow ('a \times 'b) \Rightarrow bool)*
using *neg ceq'* **by** (*auto simp add: ceq-prod-def ID-Some fun-eq-iff list-all-eq-def*)
ultimately **have** *equal-base.list-distinct ceq' (rev ?product)* **by** *simp* **}**
with *
show *equal-base.list-distinct ceq' (rev ?product) \vee ID CEQ('a \times 'b) = None*
by(*fastforce simp add: ceq-prod-def ID-def split: option.split-asm*)
qed

lift-definition *Id-on* :: 'a :: ceq set-dlist \Rightarrow ('a \times 'a) set-dlist

is *map ($\lambda x.$ (x, x))*

proof –

fix *xs* :: 'a list
assume *ceq: equal-base.list-distinct ceq' xs* \vee *ID CEQ('a) = None*
{
assume *ceq: ID CEQ('a \times 'a) \neq None*
and *xs: equal-base.list-distinct ceq' xs*
from *ceq* **have** *ID CEQ('a) \neq None*
and *ceq' = ((=) :: 'a \Rightarrow 'a \Rightarrow bool)*
and *ceq' = ((=) :: ('a \times 'a) \Rightarrow ('a \times 'a) \Rightarrow bool)*
by(*auto simp add: equal.equal-eq[OF ID-ceq] ceq-prod-def ID-None ID-Some*
split: option.split-asm)
hence *?thesis xs* **using** *xs* **by**(*auto simp add: distinct-map intro: inj-onI*) **}**
thus *?thesis xs* **using** *ceq* **by**(*auto dest: equal.equal-eq[OF ID-ceq] simp add:*
ceq-prod-def ID-None)
qed

3.9.4 Properties

lemma *member-empty* [*simp*]: *member empty = ($\lambda.$ False)*

by *transfer (simp add: fun-eq-iff)*

lemma *null-iff* [*simp*]: *null xs \longleftrightarrow xs = empty*

by *transfer simp*

lemma *list-of-dlist-empty* [*simp*]: *list-of-dlist DList-Set.empty* = []
by(*rule empty.rep-eq*)

lemma *list-of-dlist-insert* [*simp*]: $\neg \text{member } dxs\ x \implies \text{list-of-dlist } (\text{insert } x\ dxs) = x \# \text{list-of-dlist } dxs$
by(*cases dxs*)(*auto simp add: DList-Set.insert-def DList-Set.member-def Abs-dlist-inverse Abs-dlist-inject equal-base.list-insert-def intro: Abs-dlist-inverse*)

lemma *list-of-dlist-eq-Nil-iff* [*simp*]: *list-of-dlist dxs* = [] $\longleftrightarrow dxs = \text{empty}$
by(*cases dxs*)(*auto simp add: Abs-dlist-inverse Abs-dlist-inject DList-Set.empty-def*)

lemma *fold-empty* [*simp*]: *DList-Set.fold f empty b* = *b*
by(*transfer simp*)

lemma *fold-insert* [*simp*]: $\neg \text{member } dxs\ x \implies \text{DList-Set.fold } f\ (\text{insert } x\ dxs)\ b = \text{DList-Set.fold } f\ dxs\ (f\ x\ b)$
by(*transfer*)(*simp add: equal-base.list-insert-def*)

lemma *no-memb-fold-insert*:
 $\neg \text{member } dxs\ x \implies \text{fold } f\ (\text{insert } x\ dxs)\ b = \text{fold } f\ dxs\ (f\ x\ b)$
by(*transfer*)(*simp add: equal-base.list-insert-def*)

lemma *set-fold-insert*: *set (List.fold List.insert xs1 xs2)* = *set xs1* \cup *set xs2*
by(*induct xs1 arbitrary: xs2*) *simp-all*

lemma *list-of-dlist-eq-singleton-conv*:
list-of-dlist dxs = [*x*] $\longleftrightarrow dxs = \text{DList-Set.insert } x\ \text{DList-Set.empty}$
by *transfer(case-tac dxs, auto simp add: equal-base.list-insert-def)*

lemma *product-code* [*code abstract*]:
list-of-dlist (product dxs1 dxs2) = *fold* ($\lambda a. \text{fold } (\lambda c\ \text{rest}. (a, c) \# \text{rest})\ dxs2$)
dxs1 []

proof –

```
{ fix xs ys and zs :: ('a × 'b) list
  have rev (concat (map (λx. map (Pair x) ys) xs)) @ zs =
    List.fold (λa. List.fold (λc rest. (a, c) # rest) ys) xs zs
  proof (induction xs arbitrary: zs)
    case Nil thus ?case by simp
  next
    case (Cons x xs)
    have List.fold (λc rest. (x, c) # rest) ys zs = rev (map (Pair x) ys) @ zs
      by (induct ys arbitrary: zs) simp-all
    with Cons.IH[of rev (map (Pair x) ys) @ zs]
    show ?case by simp
  qed }
```

from *this*[of *list-of-dlist dxs2 list-of-dlist dxs1*] []
show ?thesis by(*simp add: product.rep-eq fold.rep-eq*)

qed

lemma *set-list-of-dlist-Abs-dlist*:

set (list-of-dlist (Abs-dlist xs)) = set xs

by(*clarsimp simp add: Abs-dlist-def Abs-dlist'-inverse*)(*subst Abs-dlist'-inverse, auto dest: equal.equal-eq[OF ID-ceq]*)

context assumes *ID-ceq-neq-None: ID CEQ('a :: ceq) ≠ None*

begin

lemma *equal-ceq: equal (ceq' :: 'a ⇒ 'a ⇒ bool)*

using *ID-ceq-neq-None* **by**(*clarsimp*)(*rule ID-ceq*)

declare *Domainp-forall-transfer*[**where** *A = pcr-set-dlist (=), simplified set-dlist.domain-eq, transfer-rule*]

lemma *set-dlist-induct* [*case-names Nil insert, induct type: set-dlist*]:

fixes *dxs :: 'a :: ceq set-dlist*

assumes *Nil: P empty and Cons: $\bigwedge a dxs. [\neg \text{member } dxs a; P dxs] \implies P$*

(*insert a dxs*)

shows *P dxs*

using *assms*

proof *transfer*

fix *P :: 'a list ⇒ bool and xs :: 'a list*

assume *NIL: P []*

and *Insert: $\bigwedge xs. \text{equal-base.list-distinct } ceq' xs \vee ID CEQ('a) = None$*

$\implies (\bigwedge x. [\neg \text{equal-base.list-member } ceq' xs x; P xs] \implies P$

(*equal-base.list-insert ceq' x xs*))

and *Eq: equal-base.list-distinct ceq' xs $\vee ID CEQ('a) = None$*

from *Eq show P xs*

proof(*induction xs*)

case Nil show ?case **by**(*rule NIL*)

next

case (*Cons x xs*) **thus** *?case using Insert[of xs x] equal.equal-eq[OF equal-ceq]*

ID-ceq-neq-None

by(*auto simp add: simp del: not-None-eq*)

qed

qed

context includes *lifting-syntax*

begin

lemma *fold-transfer2* [*transfer-rule*]:

assumes *is-equality A*

shows (*(A \implies pcr-set-dlist (=) \implies pcr-set-dlist (=) \implies*

(pcr-set-dlist (=) :: 'a list ⇒ 'a set-dlist ⇒ bool) \implies pcr-set-dlist (=) \implies

pcr-set-dlist (=))

List.fold DList-Set.fold

```

unfolding Transfer.Rel-def set-dlist.pcr-cr-eq
proof(rule rel-funI)+
  fix f :: 'a ⇒ 'b list ⇒ 'b list and g and xs :: 'a list and ys and b :: 'b list and c
  assume fg: (A ===> cr-set-dlist ===> cr-set-dlist) f g
  assume cr-set-dlist xs ys cr-set-dlist b c
  thus cr-set-dlist (List.fold f xs b) (DList-Set.fold g ys c)
  proof(induct ys arbitrary: xs b c rule: set-dlist-induct)
    case Nil thus ?case by(simp add: cr-set-dlist-def)
  next
    case (insert y dxs)
    have A y y and cr-set-dlist (list-of-dlist c) c
      using assms by(simp-all add: cr-set-dlist-def is-equality-def)
    with fg have cr-set-dlist (f y (list-of-dlist c)) (g y c)
      by -(drule (1) rel-funD)+
    thus ?case using insert by(simp add: cr-set-dlist-def)
  qed
qed

end

lemma distinct-list-of-dlist:
  distinct (list-of-dlist (dxs :: 'a set-dlist))
using list-of-dlist[of dxs] equal.equal-eq[OF equal-ceq]
by(simp add: ID-ceq-neq-None)

lemma member-empty-empty: (∀ x :: 'a. ¬ member dxs x) ⟷ dxs = empty
by(transfer)(simp add: equal.equal-eq[OF equal-ceq])

lemma Collect-member: Collect (member (dxs :: 'a set-dlist)) = set (list-of-dlist dxs)
  by (auto simp add: member-def equal.equal-eq [OF equal-ceq])

lemma member-insert: member (insert (x :: 'a) xs) = (member xs)(x := True)
by(transfer)(simp add: fun-eq-iff ID-ceq-neq-None equal.equal-eq[OF equal-ceq])

lemma member-remove:
  member (remove (x :: 'a) xs) = (member xs)(x := False)
by transfer (auto simp add: fun-eq-iff ID-ceq-neq-None equal.equal-eq[OF equal-ceq])

lemma member-union: member (union (xs1 :: 'a set-dlist) xs2) x ⟷ member xs1 x ∨ member xs2 x
unfolding union-def
by(transfer)(simp add: equal.equal-eq[OF equal-ceq] set-fold-insert)

lemma member-fold-insert: member (List.fold insert xs dxs) (x :: 'a) ⟷ member dxs x ∨ x ∈ set xs
by transfer(auto simp add: ID-ceq-neq-None equal.equal-eq[OF equal-ceq] set-fold-insert)

lemma card-eq-length [simp]:

```

$card (Collect (member (dxs :: 'a set-dlist))) = length dxs$
by transfer (auto simp add: ID-ceq-neq-None equal.equal-eq [OF equal-ceq] distinct-card)

lemma finite-member [simp]:
 $finite (Collect (member (dxs :: 'a set-dlist)))$
by transfer (auto simp add: ID-ceq-neq-None equal.equal-eq [OF equal-ceq])

lemma member-filter [simp]: $member (filter P xs) = (\lambda x :: 'a. member xs x \wedge P x)$
by transfer (auto simp add: ID-ceq-neq-None equal.equal-eq [OF equal-ceq])

lemma dlist-all-conv-member: $dlist-all P dxs \longleftrightarrow (\forall x :: 'a. member dxs x \longrightarrow P x)$
by transfer (auto simp add: ID-ceq-neq-None equal.equal-eq [OF equal-ceq] list-all-iff)

lemma dlist-ex-conv-member: $dlist-ex P dxs \longleftrightarrow (\exists x :: 'a. member dxs x \wedge P x)$
by transfer (auto simp add: ID-ceq-neq-None equal.equal-eq [OF equal-ceq] list-ex-iff)

lemma member-Id-on: $member (Id-on dxs) = (\lambda(x :: 'a, y). x = y \wedge member dxs x)$

proof –

have $ID\ CEQ('a \times 'a) = Some (=)$
using equal.equal-eq[**where** $?'a='a$, OF equal-ceq]
by (auto simp add: ceq-prod-def list-all-eq-def ID-ceq-neq-None ID-Some fun-eq-iff split: option.split)
thus $?thesis$
using equal.equal-eq[**where** $?'a='a$, OF equal-ceq]
by transfer auto

qed

end

lemma product-member:

assumes $ID\ CEQ('a :: ceq) \neq None$ $ID\ CEQ('b :: ceq) \neq None$
shows $member (product dxs1 dxs2) = (\lambda(a :: 'a, b :: 'b). member dxs1 a \wedge member dxs2 b)$

proof –

from *assms* **have** $ceq' = ((=) :: 'a \Rightarrow 'a \Rightarrow bool)$ $ceq' = ((=) :: 'b \Rightarrow 'b \Rightarrow bool)$

by (auto intro: equal.equal-eq[OF ID-ceq])

moreover with *assms* **have** $ceq' = ((=) :: ('a \times 'b) \Rightarrow ('a \times 'b) \Rightarrow bool)$

by (auto simp add: ceq-prod-def list-all-eq-def ID-Some fun-eq-iff)

ultimately show $?thesis$ **by transfer** (auto simp add: List.member-iff [abs-def])

qed

hide-const (open) empty insert remove null member length fold foldr union filter hd tl dlist-all product Id-on

end

theory *RBT-Set2*
imports
 RBT-Mapping2
begin

3.10 Sets implemented by red-black trees

lemma *map-of-map-Pair-const*:

map-of (map (λx. (x, v)) xs) = (λx. if x ∈ set xs then Some v else None)
by(*induct xs*) *auto*

lemma *map-of-rev-unit [simp]*:

fixes *xs* :: ('a * unit) list
shows *map-of (rev xs) = map-of xs*
by(*induct xs rule: rev-induct*)(*auto simp add: map-add-def split: option.split*)

lemma *fold-split-conv-map-fst*: *fold (λ(x, y). f x) xs = fold f (map fst xs)*

by(*simp add: fold-map o-def split-def*)

lemma *foldr-split-conv-map-fst*: *foldr (λ(x, y). f x) xs = foldr f (map fst xs)*

by(*simp add: foldr-map o-def split-def fun-eq-iff*)

lemma *set-foldr-Cons*:

set (foldr (λx xs. if P x xs then x # xs else xs) as []) ⊆ set as
by(*induct as*) *auto*

lemma *distinct-fst-foldr-Cons*:

distinct (map f as) ⇒ distinct (map f (foldr (λx xs. if P x xs then x # xs else xs) as []))

proof(*induct as*)

case (*Cons a as*)

with *set-foldr-Cons[of P as]* **show** *?case* **by** *auto*

qed *simp*

lemma *filter-conv-foldr*:

filter P xs = foldr (λx xs. if P x then x # xs else xs) xs []
by(*induct xs*) *simp-all*

lemma *map-of-map-Pair-key*: *map-of (map (λk. (k, f k)) xs) x = (if x ∈ set xs then Some (f x) else None)*

by(*induct xs*) *simp-all*

lemma *neq-Empty-conv*: *t ≠ rbt.Empty ⇔ (∃ c l k v r. t = Branch c l k v r)*

by(*cases t*) *simp-all*

context *linorder* **begin**

lemma *is-rbt-RBT-fold-rbt-insert* [*simp*]:

$is\text{-rbt } t \implies is\text{-rbt } (fold (\lambda(k, v). rbt\text{-insert } k v) xs t)$

by(*induct xs arbitrary: t*)(*simp-all add: split-beta*)

lemma *rbt-lookup-RBT-fold-rbt-insert* [*simp*]:

$is\text{-rbt } t \implies rbt\text{-lookup } (fold (\lambda(k, v). rbt\text{-insert } k v) xs t) = rbt\text{-lookup } t ++ map\text{-of } (rev xs)$

apply(*induct xs arbitrary: t rule: rev-induct*)

apply(*simp-all add: split-beta fun-eq-iff rbt-lookup-rbt-insert*)

done

lemma *is-rbt-fold-rbt-delete* [*simp*]:

$is\text{-rbt } t \implies is\text{-rbt } (fold rbt\text{-delete } xs t)$

by(*induct xs arbitrary: t*)(*simp-all*)

lemma *rbt-lookup-fold-rbt-delete* [*simp*]:

$is\text{-rbt } t \implies rbt\text{-lookup } (fold rbt\text{-delete } xs t) = rbt\text{-lookup } t |' (- set xs)$

apply(*induct xs rule: rev-induct*)

apply(*simp-all add: rbt-lookup-rbt-delete ext*)

apply(*metis Un-insert-right compl-sup sup-bot-right*)

done

lemma *is-rbt-fold-rbt-insert: is-rbt t* $\implies is\text{-rbt } (fold (\lambda k. rbt\text{-insert } k (f k)) xs t)$

by(*induct xs rule: rev-induct*) *simp-all*

lemma *rbt-lookup-fold-rbt-insert*:

$is\text{-rbt } t \implies$

$rbt\text{-lookup } (fold (\lambda k. rbt\text{-insert } k (f k)) xs t) =$

$rbt\text{-lookup } t ++ map\text{-of } (map (\lambda k. (k, f k)) xs)$

by(*induct xs arbitrary: t*)(*auto simp add: rbt-lookup-rbt-insert map-add-def fun-eq-iff*)

map-of-map-Pair-key split: option.splits)

end

definition *fold-rev* :: $('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'c) \Rightarrow ('a, 'b) rbt \Rightarrow 'c \Rightarrow 'c$

where *fold-rev f t* = *List.foldr* $(\lambda(k, v). f k v)$ (*RBT-Impl.entries t*)

lemma *fold-rev-simps* [*simp, code*]:

$fold\text{-rev } f \text{ RBT-Impl.Empty} = id$

$fold\text{-rev } f (Branch c l k v r) = fold\text{-rev } f l o f k v o fold\text{-rev } f r$

by(*simp-all add: fold-rev-def fun-eq-iff*)

context *linorder* **begin**

lemma *sorted-fst-foldr-Cons*:

$sorted (map f as) \implies sorted (map f (foldr (\lambda x xs. if P x xs then x \# xs else xs) as []))$

proof(*induct as*)

```

    case (Cons a as)
    with set-foldr-Cons[of P as] show ?case by(auto)
qed simp

end

```

3.10.1 Type and operations

type-synonym $'a$ set-rbt = ($'a$, unit) mapping-rbt

translations

(type) $'a$ set-rbt <= (type) ($'a$, unit) mapping-rbt

abbreviation (input) Set-RBT :: ($'a$:: ccompare, unit) RBT-Impl.rbt \Rightarrow $'a$ set-rbt
where Set-RBT \equiv Mapping-RBT

3.10.2 Primitive operations

lift-definition member :: $'a$:: ccompare set-rbt \Rightarrow $'a \Rightarrow$ bool **is**
 $\lambda t x. x \in \text{dom} (\text{rbt-comp-lookup ccomp } t) .$

abbreviation empty :: $'a$:: ccompare set-rbt
where empty \equiv RBT-Mapping2.empty

abbreviation insert :: $'a$:: ccompare \Rightarrow $'a$ set-rbt \Rightarrow $'a$ set-rbt
where insert k \equiv RBT-Mapping2.insert k ()

abbreviation remove :: $'a$:: ccompare \Rightarrow $'a$ set-rbt \Rightarrow $'a$ set-rbt
where remove \equiv RBT-Mapping2.delete

lift-definition bulkload :: $'a$:: ccompare list \Rightarrow $'a$ set-rbt **is**
 $\text{rbt-comp-bulkload ccomp} \circ \text{map } (\lambda x. (x, ()))$
by(auto 4 3 intro: linorder.rbt-bulkload-is-rbt ID-ccompare simp: rbt-comp-bulkload[OF ID-ccompare[†]])

abbreviation is-empty :: $'a$:: ccompare set-rbt \Rightarrow bool
where is-empty \equiv RBT-Mapping2.is-empty

abbreviation union :: $'a$:: ccompare set-rbt \Rightarrow $'a$ set-rbt \Rightarrow $'a$ set-rbt
where union \equiv RBT-Mapping2.join ($\lambda - . \text{id}$)

abbreviation inter :: $'a$:: ccompare set-rbt \Rightarrow $'a$ set-rbt \Rightarrow $'a$ set-rbt
where inter \equiv RBT-Mapping2.meet ($\lambda - . \text{id}$)

lift-definition inter-list :: $'a$:: ccompare set-rbt \Rightarrow $'a$ list \Rightarrow $'a$ set-rbt **is**
 $\lambda t xs. \text{fold } (\lambda k. \text{rbt-comp-insert ccomp } k ()) [x \leftarrow xs. \text{rbt-comp-lookup ccomp } t x \neq \text{None}] \text{ RBT-Impl.Empty}$
by(auto 4 3 intro: ID-ccompare linorder.is-rbt-fold-rbt-insert ord.Empty-is-rbt simp: rbt-comp-simps[OF ID-ccompare[†]])

lift-definition *minus* :: 'a :: ccompare set-rbt ⇒ 'a set-rbt ⇒ 'a set-rbt **is**
rbt-comp-minus ccomp
by(*auto 4 3 intro: linorder.rbt-minus-is-rbt ID-ccompare simp: rbt-comp-minus[OF ID-ccompare']*)

abbreviation *filter* :: ('a :: ccompare ⇒ bool) ⇒ 'a set-rbt ⇒ 'a set-rbt
where *filter* P ≡ *RBT-Mapping2.filter* (P ◦ *fst*)

lift-definition *fold* :: ('a :: ccompare ⇒ 'b ⇒ 'b) ⇒ 'a set-rbt ⇒ 'b ⇒ 'b **is** λf.
RBT-Impl.fold (λa -. f a) .

lift-definition *fold1* :: ('a :: ccompare ⇒ 'a ⇒ 'a) ⇒ 'a set-rbt ⇒ 'a **is** *RBT-Impl.fold1*
 .

lift-definition *keys* :: 'a :: ccompare set-rbt ⇒ 'a list **is** *RBT-Impl.keys* .

abbreviation *all* :: ('a :: ccompare ⇒ bool) ⇒ 'a set-rbt ⇒ bool
where *all* P ≡ *RBT-Mapping2.all* (λk -. P k)

abbreviation *ex* :: ('a :: ccompare ⇒ bool) ⇒ 'a set-rbt ⇒ bool
where *ex* P ≡ *RBT-Mapping2.ex* (λk -. P k)

definition *product* :: 'a :: ccompare set-rbt ⇒ 'b :: ccompare set-rbt ⇒ ('a × 'b)
set-rbt
where *product* rbt1 rbt2 = *RBT-Mapping2.product* (λ- - - . ()) rbt1 rbt2

abbreviation *Id-on* :: 'a :: ccompare set-rbt ⇒ ('a × 'a) set-rbt
where *Id-on* ≡ *RBT-Mapping2.diag*

abbreviation *init* :: 'a :: ccompare set-rbt ⇒ ('a, unit, 'a) rbt-generator-state
where *init* ≡ *RBT-Mapping2.init*

3.10.3 Properties

lemma *member-empty* [*simp*]:
member empty = (λ-. *False*)
by(*simp add: member-def empty-def Mapping-RBT-inverse ord.Empty-is-rbt ord.rbt-lookup.simps fun-eq-iff*)

lemma *fold-conv-fold-keys*: *RBT-Set2.fold* f rbt b = *List.fold* f (*RBT-Set2.keys*
rbt) b
by(*simp add: RBT-Set2.fold-def RBT-Set2.keys-def RBT-Impl.fold-def RBT-Impl.keys-def fold-map o-def split-def*)

lemma *fold-conv-fold-keys'*:
fold f t = *List.fold* f (*RBT-Impl.keys* (*RBT-Mapping2.impl-of* t))
by(*simp add: fold.rep-eq RBT-Impl.fold-def RBT-Impl.keys-def fold-map o-def split-def*)

lemma *member-lookup* [*code*]: *member* t x ↔ *RBT-Mapping2.lookup* t x = *Some*

()
by *transfer auto*

lemma *unfoldr-rbt-keys-generator*:
list.unfoldr rbt-keys-generator (init t) = keys t
by *transfer(simp add: unfoldr-rbt-keys-generator)*

lemma *keys-eq-Nil-iff* [*simp*]: *keys rbt = [] \longleftrightarrow rbt = empty*
by *transfer(case-tac rbt, simp-all)*

lemma *fold1-conv-fold*: *fold1 f rbt = List.fold f (tl (keys rbt)) (hd (keys rbt))*
by *transfer(simp add: RBT-Impl-fold1-def)*

context assumes *ID-ccompare-neq-None*: *ID CCOMPARE('a :: ccompare) \neq None*
begin

lemma *set-linorder*: *class.linorder (class-eq :: 'a \Rightarrow 'a \Rightarrow bool) class*
using *ID-ccompare-neq-None* **by** (*clarsimp*)(*rule ID-ccompare*)

lemma *ccomp-comparator*: *comparator (ccomp :: 'a comparator)*
using *ID-ccompare-neq-None* **by** (*clarsimp*)(*rule ID-ccompare'*)

lemmas *rbt-comps = rbt-comp-simps[OF ccomp-comparator] rbt-comp-minus[OF ccomp-comparator]*

lemma *is-rbt-impl-of* [*simp, intro*]:
fixes *t :: 'a set-rbt*
shows *ord.is-rbt class (RBT-Mapping2.impl-of t)*
using *ID-ccompare-neq-None impl-of [of t]* **by** *auto*

lemma *member-RBT*:
ord.is-rbt class t \Longrightarrow member (Set-RBT t) (x :: 'a) \longleftrightarrow ord.rbt-lookup class t x = Some ()
by (*auto simp add: member-def Mapping-RBT-inverse rbt-comps*)

lemma *member-impl-of*:
ord.rbt-lookup class (RBT-Mapping2.impl-of t) (x :: 'a) = Some () \longleftrightarrow member t x
by *transfer (auto simp: rbt-comps)*

lemma *member-insert* [*simp*]:
member (insert x (t :: 'a set-rbt)) = (member t)(x := True)
by *transfer (simp add: fun-eq-iff linorder.rbt-lookup-rbt-insert[OF set-linorder] ID-ccompare-neq-None)*

lemma *member-fold-insert* [*simp*]:
member (List.fold insert xs (t :: 'a set-rbt)) = ($\lambda x. member t x \vee x \in set xs$)
by (*induct xs arbitrary: t*) *auto*

lemma *member-remove* [*simp*]:

$member (remove (x :: 'a) t) = (member t)(x := False)$
by transfer (*simp add: linorder.rbt-lookup-rbt-delete*[*OF set-linorder*] *ID-ccompare-neq-None*
fun-eq-iff)

lemma member-bulkload [*simp*]:
 $member (bulkload xs) (x :: 'a) \longleftrightarrow x \in set xs$
by transfer (*auto simp add: linorder.rbt-lookup-rbt-bulkload*[*OF set-linorder*] *rbt-comps*
map-of-map-Pair-const split: if-split-asm)

lemma member-conv-keys: $member t = (\lambda x :: 'a. x \in set (keys t))$
by(*transfer*)(*simp add: ID-ccompare-neq-None linorder.rbt-lookup-keys*[*OF set-linorder*]
ord.is-rbt-rbt-sorted)

lemma is-empty-empty [*simp*]:
 $is-empty t \longleftrightarrow t = empty$
by transfer (*simp split: rbt.split*)

lemma RBT-lookup-empty [*simp*]:
 $ord.rbt-lookup cless (t :: ('a, unit) rbt) = Map.empty \longleftrightarrow t = RBT-Impl.Empty$
proof –
interpret *linorder cless-eq :: 'a \Rightarrow 'a \Rightarrow bool cless* **by**(*rule set-linorder*)
show *?thesis* **by**(*cases t*)(*auto simp add: fun-eq-iff*)
qed

lemma member-empty-empty [*simp*]:
 $member t = (\lambda-. False) \longleftrightarrow (t :: 'a set-rbt) = empty$
by transfer(*simp add: ID-ccompare-neq-None fun-eq-iff RBT-lookup-empty*[*symmetric*])

lemma member-union [*simp*]:
 $member (union (t1 :: 'a set-rbt) t2) = (\lambda x. member t1 x \vee member t2 x)$
by(*auto simp add: member-lookup fun-eq-iff lookup-join*[*OF ID-ccompare-neq-None*]
split: option.split)

lemma member-minus [*simp*]:
 $member (minus (t1 :: 'a set-rbt) t2) = (\lambda x. member t1 x \wedge \neg member t2 x)$
by(*transfer*)(*auto simp add: ID-ccompare-neq-None fun-eq-iff rbt-comps linorder.rbt-lookup-rbt-minus*[*OF*
set-linorder] *ord.is-rbt-rbt-sorted*)

lemma member-inter [*simp*]:
 $member (inter (t1 :: 'a set-rbt) t2) = (\lambda x. member t1 x \wedge member t2 x)$
by(*auto simp add: member-lookup fun-eq-iff lookup-meet*[*OF ID-ccompare-neq-None*]
split: option.split)

lemma member-inter-list [*simp*]:
 $member (inter-list (t :: 'a set-rbt) xs) = (\lambda x. member t x \wedge x \in set xs)$
by transfer(*auto simp add: ID-ccompare-neq-None fun-eq-iff linorder.rbt-lookup-fold-rbt-insert*[*OF*
set-linorder] *ord.Empty-is-rbt map-of-map-Pair-key ord.rbt-lookup.simps rel-option-iff*
split: if-split-asm option.split-asm)

lemma *member-filter* [*simp*]:

$member (filter P (t :: 'a set-rbt)) = (\lambda x. member t x \wedge P x)$

by (*simp add: member-lookup fun-eq-iff lookup-filter* [*OF ID-ccompare-neq-None*] *split: option.split*)

lemma *distinct-keys* [*simp*]:

$distinct (keys (rbt :: 'a set-rbt))$

by *transfer* (*simp add: ID-ccompare-neq-None RBT-Impl.keys-def ord.is-rbt-rbt-sorted linorder.distinct-entries* [*OF set-linorder*])

lemma *all-conv-all-member*:

$all P t \iff (\forall x :: 'a. member t x \longrightarrow P x)$

by (*simp add: member-lookup all-conv-all-lookup* [*OF ID-ccompare-neq-None*])

lemma *ex-conv-ex-member*:

$ex P t \iff (\exists x :: 'a. member t x \wedge P x)$

by (*simp add: member-lookup ex-conv-ex-lookup* [*OF ID-ccompare-neq-None*])

lemma *finite-member*: $finite (Collect (RBT-Set2.member (t :: 'a set-rbt)))$

by *transfer* (*simp add: rbt-comps linorder.finite-dom-rbt-lookup* [*OF set-linorder*])

lemma *member-Id-on*: $member (Id-on t) = (\lambda(k :: 'a, k'). k = k' \wedge member t k)$

by (*simp add: member-lookup* [*abs-def*] *diag-lookup* [*OF ID-ccompare-neq-None*] *fun-eq-iff*)

context **assumes** *ID-ccompare-neq-None'*: $ID\ CCOMPARE('b :: ccompare) \neq None$

begin

lemma *set-linorder'*: $class.linorder (cless-eq :: 'b \Rightarrow 'b \Rightarrow bool) cless$

using *ID-ccompare-neq-None'* **by** (*clarsimp*)(*rule ID-ccompare*)

lemma *member-product*:

$member (product rbt1 rbt2) = (\lambda ab :: 'a \times 'b. ab \in Collect (member rbt1) \times Collect (member rbt2))$

by (*auto simp add: fun-eq-iff member-lookup product-def RBT-Mapping2.lookup-product ID-ccompare-neq-None ID-ccompare-neq-None'* *split: option.splits*)

end

end

lemma *sorted-RBT-Set-keys*:

$ID\ CCOMPARE('a :: ccompare) = Some\ c$

$\implies linorder.sorted (le-of-comp\ c) (RBT-Set2.keys\ rbt)$

by *transfer* (*auto simp add: RBT-Set2.keys.rep-eq RBT-Impl.keys-def linorder.rbt-sorted-entries* [*OF ID-ccompare*] *ord.is-rbt-rbt-sorted*)

context **assumes** *ID-ccompare-neq-None*: $ID\ CCOMPARE('a :: \{ccompare, lattice\}) \neq None$

begin

lemma *set-linorder2*: *class.linorder (cless-eq :: 'a ⇒ 'a ⇒ bool) cless*
using *ID-ccompare-neq-None* **by**(*clarsimp*)(*rule ID-ccompare*)

end

lemma *set-keys-Mapping-RBT*: *set (keys (Mapping-RBT t)) = set (RBT-Impl.keys t)*

proof(*cases t*)

case *Empty* **thus** *?thesis*

by(*clarsimp simp add: Mapping-RBT-def keys.rep-eq is-ccompare-def Mapping-RBT'-inverse ord.is-rbt-def ord.rbt-sorted.simps*)

next

case (*Branch c l k v r*)

show *?thesis*

proof(*cases is-ccompare TYPE('a) ∧ ¬ ord.is-rbt cless (Branch c l k v r)*)

case *False* **thus** *?thesis using Branch*

by(*auto simp add: Mapping-RBT-def keys.rep-eq is-ccompare-def Mapping-RBT'-inverse simp del: not-None-eq*)

next

case *True*

thus *?thesis using Branch*

by(*clarsimp simp add: Mapping-RBT-def keys.rep-eq is-ccompare-def Mapping-RBT'-inverse RBT-ext.linorder.is-rbt-fold-rbt-insert-impl[OF ID-ccompare] linorder.rbt-insert-is-rbt[OF ID-ccompare] ord.Empty-is-rbt(subst linorder.rbt-lookup-keys[OF ID-ccompare, symmetric], assumption, auto simp add: linorder.rbt-sorted-fold-insert[OF ID-ccompare] RBT-ext.linorder.rbt-lookup-fold-rbt-insert-impl[OF ID-ccompare] RBT-ext.linorder.rbt-lookup-rbt-insert'[OF ID-ccompare] linorder.rbt-insert-rbt-sorted[OF ID-ccompare] ord.is-rbt-rbt-sorted ord.Empty-is-rbt dom-map-of-conv-image-fst RBT-Impl.keys-def ord.rbt-lookup.simps*)

qed

qed

hide-const (**open**) *member empty insert remove bulkload union minus*
keys fold fold-rev filter all ex product Id-on init

end

theory *Closure-Set* **imports** *Equal* **begin**

3.11 Sets implemented as Closures

context *equal-base* **begin**

definition *fun-upd* :: (*'a ⇒ 'b*) ⇒ *'a ⇒ 'b ⇒ 'a ⇒ 'b*

where *fun-upd-apply*: *fun-upd f a b a' = (if equal a a' then b else f a')*

end

lemmas [*code*] = *equal-base.fun-upd-apply*
lemmas [*simp*] = *equal-base.fun-upd-apply*

lemma *fun-upd-conv-fun-upd*: *equal-base.fun-upd (=) = fun-upd*
by(*simp add: fun-eq-iff*)

end

theory *Set-Impl imports*

Collection-Enum

DList-Set

RBT-Set2

Closure-Set

Containers-Generator

Complex-Main

begin

3.12 Different implementations of sets

3.12.1 Auxiliary functions

A simple quicksort implementation

context *ord begin*

function (*sequential*) *quicksort-acc* :: *'a list* \Rightarrow *'a list* \Rightarrow *'a list*
and *quicksort-part* :: *'a list* \Rightarrow *'a* \Rightarrow *'a list* \Rightarrow *'a list* \Rightarrow *'a list* \Rightarrow *'a list* \Rightarrow *'a list*

where

quicksort-acc *ac* [] = *ac*

| *quicksort-acc* *ac* [*x*] = *x* # *ac*

| *quicksort-acc* *ac* (*x* # *xs*) = *quicksort-part* *ac* *x* [] [] *xs*

| *quicksort-part* *ac* *x* *lts* *eqs* *gts* [] = *quicksort-acc* (*eqs* @ *x* # *quicksort-acc* *ac* *gts*)
lts

| *quicksort-part* *ac* *x* *lts* *eqs* *gts* (*z* # *zs*) =

(*if* *z* > *x* *then* *quicksort-part* *ac* *x* *lts* *eqs* (*z* # *gts*) *zs*

else if *z* < *x* *then* *quicksort-part* *ac* *x* (*z* # *lts*) *eqs* *gts* *zs*

else *quicksort-part* *ac* *x* *lts* (*z* # *eqs*) *gts* *zs*)

by *pat-completeness simp-all*

lemma *length-quicksort-accp*:

quicksort-acc-quicksort-part-dom (*Inl* (*ac*, *xs*)) \Longrightarrow *length* (*quicksort-acc* *ac* *xs*)
= *length* *ac* + *length* *xs*

and *length-quicksort-partp*:

quicksort-acc-quicksort-part-dom (*Inr* (*ac*, *x*, *lts*, *eqs*, *gts*, *zs*))

\Longrightarrow *length* (*quicksort-part* *ac* *x* *lts* *eqs* *gts* *zs*) = *length* *ac* + 1 + *length* *lts* +
length *eqs* + *length* *gts* + *length* *zs*

apply(*induct rule: quicksort-acc-quicksort-part.pinduct*)

apply(*simp-all add: quicksort-acc.psims quicksort-part.psims*)
done

termination

apply(*relation measure (case-sum ($\lambda(-, xs). 2 * \text{length } xs \wedge 2$) ($\lambda(-, -, lts, eqs, gts, zs). 2 * (\text{length } lts + \text{length } eqs + \text{length } gts + \text{length } zs) \wedge 2 + \text{length } zs + 1$)))*)
apply(*simp-all add: power2-eq-square add-mult-distrib add-mult-distrib2 length-quicksort-accp*)
done

definition *quicksort* :: 'a list \Rightarrow 'a list
where *quicksort* = *quicksort-acc* []

lemma *set-quicksort-acc* [*simp*]: *set (quicksort-acc ac xs) = set ac \cup set xs*
and *set-quicksort-part* [*simp*]:
set (quicksort-part ac x lts eqs gts zs) =
set ac \cup {x} \cup set lts \cup set eqs \cup set gts \cup set zs
by(*induct ac xs and ac x lts eqs gts zs rule: quicksort-acc-quicksort-part.induct*)(*auto split: if-split-asm*)

lemma *set-quicksort* [*simp*]: *set (quicksort xs) = set xs*
by(*simp add: quicksort-def*)

lemma *distinct-quicksort-acc*:
distinct (quicksort-acc ac xs) = distinct (ac @ xs)
and *distinct-quicksort-part*:
distinct (quicksort-part ac x lts eqs gts zs) = distinct (ac @ [x] @ lts @ eqs @ gts @ zs)
by(*induct ac xs and ac x lts eqs gts zs rule: quicksort-acc-quicksort-part.induct*)
auto

lemma *distinct-quicksort* [*simp*]: *distinct (quicksort xs) = distinct xs*
by(*simp add: quicksort-def distinct-quicksort-acc*)

end

lemmas [*code*] =
ord.quicksort-acc.sims quicksort-acc.sims
ord.quicksort-part.sims quicksort-part.sims
ord.quicksort-def quicksort-def

context *linorder* **begin**

lemma *sorted-quicksort-acc*:
[[*sorted ac; $\forall x \in \text{set } xs. \forall a \in \text{set } ac. x < a$*]]
 \Rightarrow *sorted (quicksort-acc ac xs)*
and *sorted-quicksort-part*:
[[*sorted ac; $\forall y \in \text{set } lts \cup \{x\} \cup \text{set } eqs \cup \text{set } gts \cup \text{set } zs. \forall a \in \text{set } ac. y < a;$*
 $\forall y \in \text{set } lts. y < x; \forall y \in \text{set } eqs. y = x; \forall y \in \text{set } gts. y > x$]]
 \Rightarrow *sorted (quicksort-part ac x lts eqs gts zs)*

```

proof(induction ac xs and ac x lts eqs gts zs rule: quicksort-acc-quicksort-part.induct)
  case 1 thus ?case by simp
next
  case 2 thus ?case by(auto)
next
  case 3 thus ?case by simp
next
  case (4 ac x lts eqs gts)
  note  $ac\text{-greater} = \langle \forall y \in set\ lts \cup \{x\} \cup set\ eqs \cup set\ gts \cup set\ []. \forall a \in set\ ac. y < a \rangle$ 
  have sorted eqs  $set\ eqs \subseteq \{x\}$  using  $\langle \forall y \in set\ eqs. y = x \rangle$ 
  by(induct eqs)(simp-all)
  moreover have  $\forall y \in set\ ac \cup set\ gts. x \leq y$ 
  using  $\langle \forall a \in set\ gts. x < a \rangle$  ac-greater by auto
  moreover have sorted (quicksort-acc ac gts)
  using  $\langle sorted\ ac \rangle$  ac-greater by(auto intro: 4.IH)
  ultimately have sorted ( $eqs @ x \# quicksort\text{-acc}\ ac\ gts$ )
  by(auto simp add: sorted-append)
  moreover have  $\forall y \in set\ lts. \forall a \in set\ (eqs @ x \# quicksort\text{-acc}\ ac\ gts). y < a$ 
  using  $\langle \forall y \in set\ lts. y < x \rangle$  ac-greater  $\langle \forall a \in set\ gts. x < a \rangle$   $\langle \forall y \in set\ eqs. y = x \rangle$ 
  by fastforce
  ultimately show ?case by(simp add: 4.IH)
next
  case 5 thus ?case by(simp add: not-less order-eq-iff)
qed

```

lemma *sorted-quicksort* [*simp*]: *sorted (quicksort xs)*
by(*simp add: quicksort-def sorted-quicksort-acc*)

lemma *insort-key-append1*:
 $\forall y \in set\ ys. f\ x < f\ y \implies insort\text{-key}\ f\ x\ (xs @ ys) = insort\text{-key}\ f\ x\ xs @ ys$
proof(*induct xs*)
case Nil
thus ?case by(*cases ys*) *auto*
qed *simp*

lemma *insort-key-append2*:
 $\forall y \in set\ xs. f\ x > f\ y \implies insort\text{-key}\ f\ x\ (xs @ ys) = xs @ insort\text{-key}\ f\ x\ ys$
by(*induct xs*) *auto*

lemma *sort-key-append*:
 $\forall x \in set\ xs. \forall y \in set\ ys. f\ x < f\ y \implies sort\text{-key}\ f\ (xs @ ys) = sort\text{-key}\ f\ xs @ sort\text{-key}\ f\ ys$
by(*induct xs*)(*simp-all add: insort-key-append1*)

definition *single-list* :: 'a \Rightarrow 'a list **where** *single-list a* = [a]

lemma *to-single-list*: $x \# xs = single\text{-list}\ x @ xs$
by(*simp add: single-list-def*)

lemma *sort-snoc*: $\text{sort } (xs @ [x]) = \text{insort } x (\text{sort } xs)$
by(*induct xs*)(*simp-all add: insort-left-comm*)

lemma *sort-append-swap*: $\text{sort } (xs @ ys) = \text{sort } (ys @ xs)$
by(*induct xs arbitrary: ys rule: rev-induct*)(*simp-all add: sort-snoc[symmetric]*)

lemma *sort-append-swap2*: $\text{sort } (xs @ ys @ zs) = \text{sort } (ys @ xs @ zs)$
by(*induct xs*)(*simp-all, subst (1 2) sort-append-swap, simp*)

lemma *sort-Cons-append-swap*: $\text{sort } (x \# xs) = \text{sort } (xs @ [x])$
by(*subst sort-append-swap*) *simp*

lemma *sort-append-Cons-swap*: $\text{sort } (ys @ x \# xs) = \text{sort } (ys @ xs @ [x])$
apply(*induct ys*)
apply(*simp only: append.simps sort-Cons-append-swap*)
apply *simp*
done

lemma *quicksort-acc-conv-sort*:
 $\text{quicksort-acc } ac \ xs = \text{sort } xs @ ac$
and *quicksort-part-conv-sort*:
 $\llbracket \forall y \in \text{set } lts. y < x; \forall y \in \text{set } eqs. y = x; \forall y \in \text{set } gts. y > x \rrbracket$
 $\implies \text{quicksort-part } ac \ x \ lts \ eqs \ gts \ zs = \text{sort } (lts @ eqs @ gts @ x \# zs) @ ac$
proof(*induct ac xs and ac x lts eqs gts zs rule: quicksort-acc-quicksort-part.induct*)
case 1 thus ?*case by simp*
next
case 2 thus ?*case by simp*
next
case 3 thus ?*case by simp*
next
case (4 *ac x lts eqs gts*)
note $eqs = \langle \forall y \in \text{set } eqs. y = x \rangle$
{ fix *eqs*
assume $\forall y \in \text{set } eqs. y = x$
hence $\text{insort } x \ eqs = x \# eqs$ **by**(*induct eqs*) *simp-all* }
note [*simp*] = *this*
from *eqs* **have** [*simp*]: $\text{sort } eqs = eqs$ **by**(*induct eqs*) *simp-all*
from *eqs* **have** [*simp*]: $eqs @ [x] = x \# eqs$ **by**(*induct eqs*) *simp-all*

show ?*case using 4*
apply(*subst sort-key-append*)
apply(*auto 4 3 dest: bspec*)[1]
apply(*simp add: append-assoc[symmetric] sort-snoc del: append-assoc*)
apply(*subst sort-key-append*)
apply(*auto 4 3 simp add: insort-key-append1 dest: bspec*)
done
next

```

case (5 ac x lts eqs gts z zs)
have  $\llbracket \neg z < x; \neg x < z \rrbracket \implies z = x$  by simp
thus ?case using 5
  apply(simp del: sort-key-simps)
  apply(safe, simp-all del: sort-key-simps add: to-single-list)
  apply(subst sort-append-swap)
  apply(fold append-assoc)
  apply(subst (2) sort-append-swap)
  apply(subst sort-append-swap2)
  apply(unfold append-assoc)
  apply(rule refl)
  apply(subst (1 5) append-assoc[symmetric])
  apply(subst (1 2) sort-append-swap)
  apply(unfold append-assoc)
  apply(subst sort-append-swap2)
  apply(subst (1 2) sort-append-swap)
  apply(unfold append-assoc)
  apply(subst sort-append-swap2)
  apply(rule refl)
  apply(subst (2 6) append-assoc[symmetric])
  apply(subst (2 5) append-assoc[symmetric])
  apply(subst (1 2) sort-append-swap2)
  apply(subst (4) append-assoc)
  apply(subst (2) sort-append-swap2)
  apply simp
done
qed

```

lemma quicksort-conv-sort: quicksort xs = sort xs
by(simp add: quicksort-def quicksort-acc-conv-sort)

lemma sort-remdups: sort (remdups xs) = remdups (sort xs)
by(rule sorted-distinct-set-unique) simp-all

end

Removing duplicates from a sorted list

context ord **begin**

```

fun remdups-sorted :: 'a list  $\Rightarrow$  'a list
where
  remdups-sorted [] = []
| remdups-sorted [x] = [x]
| remdups-sorted (x#y#xs) = (if x < y then x # remdups-sorted (y#xs) else
  remdups-sorted (y#xs))

```

end

lemmas [code] = ord.remdups-sorted.simps

context *linorder* **begin**

lemma [*simp*]:
assumes *sorted xs*
shows *sorted-remdups-sorted: sorted (remdups-sorted xs)*
and *set-remdups-sorted: set (remdups-sorted xs) = set xs*
using *assms* **by**(*induct xs rule: remdups-sorted.induct*)(*auto*)

lemma *distinct-remdups-sorted* [*simp*]: *sorted xs \implies distinct (remdups-sorted xs)*
by(*induct xs rule: remdups-sorted.induct*)(*auto*)

lemma *remdups-sorted-conv-remdups: sorted xs \implies remdups-sorted xs = remdups xs*
by(*induct xs rule: remdups-sorted.induct*)(*auto*)

end

An specialised operation to convert a finite set into a sorted list

definition *csorted-list-of-set* :: '*a* :: *ccompare set* \Rightarrow '*a list*

where

csorted-list-of-set A =
(if ID CCOMPARE('a) = None \vee \neg finite A then undefined else linorder.sorted-list-of-set cless-eq A)

lemma *csorted-list-of-set-set* [*simp*]:
 \llbracket *ID CCOMPARE('a :: ccompare) = Some c; linorder.sorted (le-of-comp c) xs;*
distinct xs \rrbracket
 \implies *linorder.sorted-list-of-set (le-of-comp c) (set xs) = xs*
by(*simp add: distinct-remdups-id linorder.sorted-list-of-set-sort-remdups[OF ID-ccompare]*
linorder.sorted-sort-id[OF ID-ccompare])

lemma *csorted-list-of-set-split*:

fixes *A :: 'a :: ccompare set* **shows**
 P (*csorted-list-of-set A*) \longleftrightarrow
 $(\forall xs. ID CCOMPARE('a) \neq None \longrightarrow finite A \longrightarrow A = set xs \longrightarrow distinct xs$
 $\longrightarrow linorder.sorted cless-eq xs \longrightarrow P xs) \wedge$
 $(ID CCOMPARE('a) = None \vee \neg finite A \longrightarrow P undefined)$
by(*auto simp add: csorted-list-of-set-def linorder.sorted-list-of-set[OF ID-ccompare]*)

code-identifier code-module *Set* \rightarrow (SML) *Set-Impl*

| **code-module** *Set-Impl* \rightarrow (SML) *Set-Impl*

3.12.2 Delete code equation with set as constructor

lemma *is-empty-unfold* [*code-unfold*]:
set-eq A {} = Set.is-empty A
set-eq {} A = Set.is-empty A
by (*auto simp add: set-eq-def*)

definition *is-UNIV* :: 'a set \Rightarrow bool
where *is-UNIV* A \longleftrightarrow A = UNIV

lemma *is-UNIV-unfold* [*code-unfold*]:
 A = UNIV \longleftrightarrow *is-UNIV* A
 UNIV = A \longleftrightarrow *is-UNIV* A
 set-eq A UNIV \longleftrightarrow *is-UNIV* A
 set-eq UNIV A \longleftrightarrow *is-UNIV* A
by(*auto simp add: is-UNIV-def set-eq-def*)

3.12.3 Set implementations

definition *Collect-set* :: ('a \Rightarrow bool) \Rightarrow 'a set
where [*simp*]: *Collect-set* = Collect

definition *DList-set* :: 'a :: ceq set-dlist \Rightarrow 'a set
where *DList-set* = Collect o *DList-Set.member*

definition *RBT-set* :: 'a :: ccompare set-rbt \Rightarrow 'a set
where *RBT-set* = Collect o *RBT-Set2.member*

definition *Complement* :: 'a set \Rightarrow 'a set
where [*simp*]: *Complement* A = - A

definition *Set-Monad* :: 'a list \Rightarrow 'a set
where [*simp*]: *Set-Monad* = set

code-datatype *Collect-set DList-set RBT-set Set-Monad Complement*

lemma *DList-set-empty* [*simp*]: *DList-set DList-Set.empty* = {}
by(*simp add: DList-set-def*)

lemma *RBT-set-empty* [*simp*]: *RBT-set RBT-Set2.empty* = {}
by(*simp add: RBT-set-def*)

lemma *RBT-set-conv-keys*:
 ID CCOMPARE('a :: ccompare) \neq None
 \implies *RBT-set* (t :: 'a set-rbt) = set (*RBT-Set2.keys* t)
by(*clarsimp simp add: RBT-set-def member-conv-keys*)

3.12.4 Set operations

named-theorems *set-base-code* <Code equations not involving set complement>

Various fold operations over sets

typedef ('a, 'b) *comp-fun-commute* = {f :: 'a \Rightarrow 'b \Rightarrow 'b. *comp-fun-commute* f}
morphisms *comp-fun-commute-apply Abs-comp-fun-commute*
by(*rule exI[where x= λ -. id]*)(*simp, unfold-locales, auto*)

setup-lifting *type-definition-comp-fun-commute*

lemma *comp-fun-commute-apply'* [*simp*]:

comp-fun-commute-on UNIV (comp-fun-commute-apply f)

using *comp-fun-commute-apply[of f]* **by** (*simp add: comp-fun-commute-def'*)

lift-definition *set-fold-cfc* :: ('a, 'b) *comp-fun-commute* \Rightarrow 'b \Rightarrow 'a *set* \Rightarrow 'b **is** *Finite-Set.fold* .

lemma *set-fold-cfc-code* [*code*]:

fixes *xs* :: 'a :: *ceq list*

and *dxs* :: 'a :: *ceq set-dlist*

and *rbt* :: 'b :: *compare set-rbt*

shows [*set-base-code*]:

set-fold-cfc f'' b (RBT-set rbt) =

(*case ID CCOMPARE('b) of None* \Rightarrow *Code.abort (STR "set-fold-cfc RBT-set: ccompare = None")* (λ -. *set-fold-cfc f'' b (RBT-set rbt)*)

| *Some* - \Rightarrow *RBT-Set2.fold (comp-fun-commute-apply f'') rbt*

b)

(**is** *?RBT-set*)

set-fold-cfc f' b (DList-set dxs) =

(*case ID CEQ('a) of None* \Rightarrow *Code.abort (STR "set-fold-cfc DList-set: ceq = None")* (λ -. *set-fold-cfc f' b (DList-set dxs)*)

| *Some* - \Rightarrow *DList-Set.fold (comp-fun-commute-apply f') dxs b*)

(**is** *?DList-set*)

set-fold-cfc f b (Set-Monad xs) =

(*case ID CEQ('a) of None* \Rightarrow *Code.abort (STR "set-fold-cfc Set-Monad: ceq = None")* (λ -. *set-fold-cfc f b (Set-Monad xs)*)

| *Some eq* \Rightarrow *List.fold (comp-fun-commute-apply f) (equal-base.list-remdups*

eq xs) b)

(**is** *?Set-Monad*)

set-fold-cfc f''' b (Collect-set P) = Code.abort (STR "set-fold-cfc not supported on Collect-set") (λ -. *set-fold-cfc f''' b (Collect-set P)*)

and *set-fold-cfc-Complement*:

set-fold-cfc f''' b (Complement A) = Code.abort (STR "set-fold-cfc not supported on Complement") (λ -. *set-fold-cfc f''' b (Complement A)*)

proof -

note *fold-set-fold-remdups = comp-fun-commute-def' comp-fun-commute-on.fold-set-fold-remdups[OF - subset-UNIV]*

show *?Set-Monad*

by(*auto split: option.split dest!: Collection-Eq.ID-ceq simp add: set-fold-cfc-def fold-set-fold-remdups*)

show *?DList-set*

apply(*auto split: option.splits simp add: DList-set-def*)

apply *transfer*

apply(*auto dest: Collection-Eq.ID-ceq simp add: List.member-iff [abs-def] fold-set-fold-remdups distinct-remdups-id*)

done

```

show ?RBT-set
  apply(auto split: option.split simp add: RBT-set-conv-keys fold-conv-fold-keys)
  apply transfer
  apply(simp add: fold-set-fold-remdups distinct-remdups-id linorder.distinct-keys[OF
ID-ccompare] ord.is-rbt-rbt-sorted)
  done
qed simp-all

```

```

typedef ('a, 'b) comp-fun-idem = {f :: 'a ⇒ 'b ⇒ 'b. comp-fun-idem f}
  morphisms comp-fun-idem-apply Abs-comp-fun-idem
by(rule exI[where x=λ-. id])(simp, unfold-locales, auto)

```

```

setup-lifting type-definition-comp-fun-idem

```

```

lemma comp-fun-idem-apply' [simp]:
  comp-fun-idem-on UNIV (comp-fun-idem-apply f)
using comp-fun-idem-apply[of f] by (simp add: comp-fun-idem-def')

```

```

lift-definition set-fold-cfi :: ('a, 'b) comp-fun-idem ⇒ 'b ⇒ 'a set ⇒ 'b is Finite-Set.fold .

```

```

lemma set-fold-cfi-code [code]:
  fixes xs :: 'a list
  and dxs :: 'b :: ceq set-dlist
  and rbt :: 'c :: ccompare set-rbt
  shows
    set-fold-cfi f'' b (RBT-set rbt) =
      (case ID CCOMPARE('c) of None ⇒ Code.abort (STR "set-fold-cfi RBT-set:
ccompare = None") (λ-. set-fold-cfi f'' b (RBT-set rbt))
      | Some - ⇒ RBT-Set2.fold (comp-fun-idem-apply f'') rbt b)
    (is ?RBT-set)
    set-fold-cfi f' b (DList-set dxs) =
      (case ID CEQ('b) of None ⇒ Code.abort (STR "set-fold-cfi DList-set: ceq =
None") (λ-. set-fold-cfi f' b (DList-set dxs))
      | Some - ⇒ DList-Set.fold (comp-fun-idem-apply f') dxs b)
    (is ?DList-set)
    set-fold-cfi f b (Set-Monad xs) = List.fold (comp-fun-idem-apply f) xs b
    (is ?Set-Monad)
    set-fold-cfi f b (Collect-set P) = Code.abort (STR "set-fold-cfi not supported on
Collect-set") (λ-. set-fold-cfi f b (Collect-set P))
    set-fold-cfi f b (Complement A) = Code.abort (STR "set-fold-cfi not supported
on Complement") (λ-. set-fold-cfi f b (Complement A))
  proof –
    show ?Set-Monad
    by(auto split: option.split dest!: Collection-Eq.ID-ceq simp add: set-fold-cfi-def
comp-fun-idem-def' comp-fun-idem-on.fold-set-fold[OF - subset-UNIV])
    show ?DList-set
    apply(auto split: option.split simp add: DList-set-def)
    apply transfer

```

```

apply(auto dest: Collection-Eq.ID-ceq simp add: List.member-iff [abs-def]
comp-fun-idem-def' comp-fun-idem-on.fold-set-fold[OF - subset-UNIV])
done
show ?RBT-set
apply(auto split: option.split simp add: RBT-set-conv-keys fold-conv-fold-keys)
apply transfer
apply(simp add: comp-fun-idem-def' comp-fun-idem-on.fold-set-fold[OF - subset-UNIV])
done
qed simp-all

```

```

typedef 'a semilattice-set = {f :: 'a ⇒ 'a ⇒ 'a. semilattice-set f}
morphisms semilattice-set-apply Abs-semilattice-set
proof
show (λx y. if x = y then x else undefined) ∈ ?semilattice-set
unfolding mem-Collect-eq by(unfold-locales) simp-all
qed

```

setup-lifting *type-definition-semilattice-set*

```

lemma semilattice-set-apply' [simp]:
semilattice-set (semilattice-set-apply f)
using semilattice-set-apply[of f] by simp

```

```

lemma comp-fun-idem-semilattice-set-apply [simp]:
comp-fun-idem-on UNIV (semilattice-set-apply f)
proof –
interpret semilattice-set semilattice-set-apply f by simp
show ?thesis by(unfold-locales)(simp-all add: fun-eq-iff left-commute)
qed

```

lift-definition *set-fold1* :: 'a *semilattice-set* ⇒ 'a *set* ⇒ 'a **is** *semilattice-set.F* .

```

lemma (in semilattice-set) F-set-conv-fold:
xs ≠ [] ⇒ F (set xs) = Finite-Set.fold f (hd xs) (set (tl xs))
by(clarsimp simp add: neq-Nil-conv eq-fold)

```

```

lemma set-fold1-code [code]:
fixes rbt :: 'a :: {ccompare, lattice} set-rbt
and dxs :: 'b :: {ceq, lattice} set-dlist
shows [set-base-code]:
set-fold1 f'' (RBT-set rbt) =
  (case ID CCOMPARE('a) of None ⇒ Code.abort (STR "set-fold1 RBT-set:
ccompare = None") (λ-. set-fold1 f'' (RBT-set rbt))
  | Some - ⇒ if RBT-Set2.is-empty rbt then Code.abort (STR
"set-fold1 RBT-set: empty set") (λ-. set-fold1 f'' (RBT-set rbt))
  else RBT-Set2.fold1 (semilattice-set-apply f'') rbt)
  (is ?RBT-set)
set-fold1 f' (DList-set dxs) =

```

```

      (case ID CEQ('b) of None => Code.abort (STR "set-fold1 DList-set: ceq =
None") (λ-. set-fold1 f' (DList-set dxs))
      | Some - => if DList-Set.null dxs then Code.abort (STR "set-fold1
DList-set: empty set") (λ-. set-fold1 f' (DList-set dxs))
      else DList-Set.fold (semilattice-set-apply f') (DList-Set.tl
dxs) (DList-Set.hd dxs))
    (is ?DList-set)
    set-fold1 f (Set-Monad (x # xs)) = fold (semilattice-set-apply f) xs x
    (is ?Set-Monad)
    set-fold1 f (Collect-set P) = Code.abort (STR "set-fold1: Collect-set") (λ-.
set-fold1 f (Collect-set P))
  and set-fold1-Complement:
    set-fold1 f (Complement A) = Code.abort (STR "set-fold1: Complement") (λ-.
set-fold1 f (Complement A))
proof -
  note fold-set-fold = comp-fun-idem-def' comp-fun-idem-on.fold-set-fold[OF - sub-
set-UNIV]
  show ?Set-Monad
  by(simp add: set-fold1-def semilattice-set.eq-fold fold-set-fold)
  show ?DList-set
  by(simp add: set-fold1-def semilattice-set.F-set-conv-fold fold-set-fold DList-set-def
DList-Set.Collect-member split: option.split)(transfer, simp)
  show ?RBT-set
  by(simp add: set-fold1-def semilattice-set.F-set-conv-fold fold-set-fold RBT-set-def
RBT-Set2.member-conv-keys RBT-Set2.fold1-conv-fold split: option.split)
qed simp-all

```

Implementation of set operations

lemma *Collect-code* [code, set-base-code]:

fixes $P :: 'a :: cenum \Rightarrow bool$ **shows**

$Collect\ P =$

(case ID CENUM('a) of None \Rightarrow Collect-set P

| Some (enum, -) \Rightarrow Set-Monad (filter P enum))

by(auto split: option.split dest: in-cenum)

lemma *finite-code* [code]:

fixes $dxs :: 'a :: ceq\ set\ dlist$

and $rbt :: 'b :: ccompare\ set\ rbt$

and $A :: 'c :: finite\ UNIV\ set$ **and** $P :: 'c \Rightarrow bool$

shows [set-base-code]:

$finite\ (Collect\ set\ P) \longleftrightarrow$

$of\ phantom\ (finite\ UNIV :: 'c\ finite\ UNIV) \vee Code.abort\ (STR\ "finite\ Col\ lect\ set")\ (\lambda\cdot.\ finite\ (Collect\ set\ P))$

$finite\ (Set\ Monad\ xs) = True$

and *finite-Complement*:

$finite\ (Complement\ A) \longleftrightarrow$

(if of-phantom (finite-UNIV :: 'c finite-UNIV) then True

else if finite A then False

else Code.abort (STR "finite Complement: infinite set") (λ-. finite (Complement

```

A)))
and [set-base-code]:
  finite (RBT-set rbt) =
    (case ID CCOMPARE('b) of None  $\Rightarrow$  Code.abort (STR "finite RBT-set:
ccompare = None") ( $\lambda$ -. finite (RBT-set rbt))
    | Some -  $\Rightarrow$  True)
  finite (DList-set dxs) =
    (case ID CEQ('a) of None  $\Rightarrow$  Code.abort (STR "finite DList-set: ceq = None")
( $\lambda$ -. finite (DList-set dxs))
    | Some -  $\Rightarrow$  True)
by (auto simp add: DList-set-def RBT-set-def member-conv-keys card-gt-0-iff fi-
nite-UNIV split: option.split elim: finite-subset [rotated 1])

```

```

lemma CARD-code [code-unfold]:
  CARD('a :: card-UNIV) = of-phantom (card-UNIV :: 'a card-UNIV)
by (simp add: card-UNIV)

```

```

lemma card-code [code]:
fixes dxs :: 'a :: ceq set-dlist and xs :: 'a list
and rbt :: 'b :: ccompare set-rbt
and A :: 'c :: card-UNIV set
shows card-Complement:
  card (Complement A) =
    (let a = card A; s = CARD('c)
    in if s > 0 then s - a
    else if finite A then 0
    else Code.abort (STR "card Complement: infinite") ( $\lambda$ -. card (Complement
A)))

```

```

and [set-base-code]:
  card (Set-Monad xs) =
    (case ID CEQ('a) of None  $\Rightarrow$  Code.abort (STR "card Set-Monad: ceq =
None") ( $\lambda$ -. card (Set-Monad xs))
    | Some eq  $\Rightarrow$  length (equal-base.list-remdups eq xs))
  card (RBT-set rbt) =
    (case ID CCOMPARE('b) of None  $\Rightarrow$  Code.abort (STR "card RBT-set:
ccompare = None") ( $\lambda$ -. card (RBT-set rbt))
    | Some -  $\Rightarrow$  length (RBT-Set2.keys rbt))
  card (DList-set dxs) =
    (case ID CEQ('a) of None  $\Rightarrow$  Code.abort (STR "card DList-set: ceq = None")
( $\lambda$ -. card (DList-set dxs))
    | Some -  $\Rightarrow$  DList-Set.length dxs)
by(auto simp add: RBT-set-def member-conv-keys distinct-card DList-set-def Let-def
card-UNIV Compl-eq-Diff-UNIV card-Diff-subset-Int card-gt-0-iff finite-subset[of A
UNIV] List.card-set dest: Collection-Eq.ID-ceq split: option.split)

```

```

lemma is-UNIV-code [code, set-base-code]:
fixes rbt :: 'a :: {cproper-interval, finite-UNIV} set-rbt
and A :: 'b :: card-UNIV set

```

```

shows
  is-UNIV (RBT-set rbt) =
    (case ID CCOMPARE('a) of None  $\Rightarrow$  Code.abort (STR "is-UNIV RBT-set:
  ccompare = None") ( $\lambda$ -. is-UNIV (RBT-set rbt))
      | Some -  $\Rightarrow$  of-phantom (finite-UNIV :: 'a finite-UNIV)  $\wedge$ 
  proper-intrvl.exhaustive-fusion cproper-interval rbt-keys-generator (RBT-Set2.init
  rbt))
    (is ?rbt)
  is-UNIV A  $\longleftrightarrow$ 
    (let a = CARD('b);
      b = card A
      in if a > 0 then a = b
        else if b > 0 then False
          else Code.abort (STR "is-UNIV called on infinite type and set") ( $\lambda$ -.
  is-UNIV A))
    (is ?generic)
proof -
  {
    fix c
    assume linorder: ID CCOMPARE('a) = Some c
    have is-UNIV (RBT-set rbt) =
      (finite (UNIV :: 'a set)  $\wedge$  proper-intrvl.exhaustive cproper-interval (RBT-Set2.keys
  rbt))
      (is ?lhs  $\longleftrightarrow$  ?rhs)
    proof
      assume ?lhs
      have finite (UNIV :: 'a set)
        unfolding <?lhs>[unfolded is-UNIV-def, symmetric]
        using linorder
        by(simp add: finite-code)
      moreover
      hence proper-intrvl.exhaustive cproper-interval (RBT-Set2.keys rbt)
        using linorder <?lhs>
      by(simp add: linorder-proper-interval.exhaustive-correct[OF ID-ccompare-interval[OF
  linorder]] sorted-RBT-Set-keys is-UNIV-def RBT-set-conv-keys)
      ultimately show ?rhs ..
    next
      assume ?rhs
      thus ?lhs using linorder
      by(auto simp add: linorder-proper-interval.exhaustive-correct[OF ID-ccompare-interval[OF
  linorder]] sorted-RBT-Set-keys is-UNIV-def RBT-set-conv-keys)
      qed }
    thus ?rbt
      by(auto simp add: finite-UNIV proper-intrvl.exhaustive-fusion-def unfoldr-rbt-keys-generator
  is-UNIV-def split: option.split)

    show ?generic
      by(auto simp add: Let-def is-UNIV-def dest: card-seteq[of UNIV A] dest!:
  card-ge-0-finite)
  
```

qed

lemma *is-empty-code* [code]:

fixes *dxs* :: 'a :: ceq set-dlist

and *rbt* :: 'b :: ccompare set-rbt

and *A* :: 'c set

shows *is-empty-Complement*:

Set.is-empty (Complement A) \longleftrightarrow is-UNIV A

(**is** ?Complement)

and [set-base-code]:

Set.is-empty (RBT-set rbt) \longleftrightarrow

(*case ID CCOMPARE('b) of None \Rightarrow Code.abort (STR "is-empty RBT-set: ccompare = None")* (λ -. *Set.is-empty (RBT-set rbt)*)

| *Some - \Rightarrow RBT-Set2.is-empty rbt*)

(**is** ?RBT-set)

Set.is-empty (DList-set dxs) \longleftrightarrow

(*case ID CEQ('a) of None \Rightarrow Code.abort (STR "is-empty DList-set: ceq = None")* (λ -. *Set.is-empty (DList-set dxs)*)

| *Some - \Rightarrow DList-Set.null dxs*)

(**is** ?DList-set)

Set.is-empty (Set-Monad xs) \longleftrightarrow xs = []

proof –

show ?DList-set

by(*clarsimp simp add: DList-set-def DList-Set.member-empty-empty split: option.split*)

show ?RBT-set

by(*clarsimp simp add: RBT-set-def RBT-Set2.member-empty-empty[symmetric] fun-eq-iff simp del: RBT-Set2.member-empty-empty split: option.split*)

show ?Complement

by(*auto simp add: is-UNIV-def*)

qed *simp-all*

lemma *Set-insert-code* [code]:

fixes *dxs* :: 'a :: ceq set-dlist

and *rbt* :: 'b :: ccompare set-rbt

shows *insert-Complement*:

$\bigwedge x. \text{Set.insert } x \text{ (Complement } X) = \text{Complement (Set.remove } x \text{ } X)$

and [set-base-code]:

$\bigwedge x. \text{Set.insert } x \text{ (RBT-set rbt) =}$

(*case ID CCOMPARE('b) of None \Rightarrow Code.abort (STR "insert RBT-set: ccompare = None")* (λ -. *Set.insert x (RBT-set rbt)*)

| *Some - \Rightarrow RBT-set (RBT-Set2.insert x rbt)*)

$\bigwedge x. \text{Set.insert } x \text{ (DList-set dxs) =}$

(*case ID CEQ('a) of None \Rightarrow Code.abort (STR "insert DList-set: ceq = None")* (λ -. *Set.insert x (DList-set dxs)*)

| *Some - \Rightarrow DList-set (DList-Set.insert x dxs)*)

$\bigwedge x. \text{Set.insert } x \text{ (Set-Monad xs) = Set-Monad (x \# xs)}$

```

 $\bigwedge x. \text{Set.insert } x \text{ (Collect-set } A) =$ 
  (case ID CEQ('a) of None  $\Rightarrow$  Code.abort (STR "insert Collect-set: ceq =
None") ( $\lambda\cdot. \text{Set.insert } x \text{ (Collect-set } A)$ )
  | Some eq  $\Rightarrow$  Collect-set (equal-base.fun-upd eq A x True))
by(auto split: option.split dest: equal.equal-eq[OF ID-ceq] simp add: DList-set-def
DList-Set.member-insert RBT-set-def)

```

lemma *Set-member-code* [code]:

```

fixes xs :: 'a :: ceq list
shows [set-base-code]:
 $\bigwedge x. x \in \text{Set-Monad } xs \longleftrightarrow$ 
  (case ID CEQ('a) of None  $\Rightarrow$  Code.abort (STR "member Set-Monad: ceq =
None") ( $\lambda\cdot. x \in \text{Set-Monad } xs$ )
  | Some eq  $\Rightarrow$  equal-base.list-member eq xs x)
and mem-Complement:
 $\bigwedge x. x \in \text{Complement } X \longleftrightarrow x \notin X$ 
and [set-base-code]:
 $\bigwedge x. x \in \text{RBT-set } rbt \longleftrightarrow \text{RBT-Set2.member } rbt x$ 
 $\bigwedge x. x \in \text{DList-set } dxs \longleftrightarrow \text{DList-Set.member } dxs x$ 
 $\bigwedge x. x \in \text{Collect-set } A \longleftrightarrow A x$ 
by(auto simp add: DList-set-def RBT-set-def split: option.split dest!: Collection-Eq.ID-ceq)

```

lemma *Set-remove-code* [code]:

```

fixes rbt :: 'a :: ccompare set-rbt
and dxs :: 'b :: ceq set-dlist
shows remove-Complement:
 $\bigwedge x A. \text{Set.remove } x \text{ (Complement } A) = \text{Complement } (\text{Set.insert } x A)$ 
and [set-base-code]:
 $\bigwedge x. \text{Set.remove } x \text{ (RBT-set } rbt) =$ 
  (case ID CCOMPARE('a) of None  $\Rightarrow$  Code.abort (STR "remove RBT-set:
ccompare = None") ( $\lambda\cdot. \text{Set.remove } x \text{ (RBT-set } rbt)$ )
  | Some -  $\Rightarrow$  RBT-set (RBT-Set2.remove x rbt))
 $\bigwedge x. \text{Set.remove } x \text{ (DList-set } dxs) =$ 
  (case ID CEQ('b) of None  $\Rightarrow$  Code.abort (STR "remove DList-set: ceq =
None") ( $\lambda\cdot. \text{Set.remove } x \text{ (DList-set } dxs)$ )
  | Some -  $\Rightarrow$  DList-set (DList-Set.remove x dxs))
 $\bigwedge x. \text{Set.remove } x \text{ (Collect-set } A) =$ 
  (case ID CEQ('b) of None  $\Rightarrow$  Code.abort (STR "remove Collect: ceq = None")
( $\lambda\cdot. \text{Set.remove } x \text{ (Collect-set } A)$ )
  | Some eq  $\Rightarrow$  Collect-set (equal-base.fun-upd eq A x False))
by(auto split: option.split if-split-asm dest: equal.equal-eq[OF ID-ceq] simp add:
DList-set-def DList-Set.member-remove RBT-set-def)

```

lemma *Set-uminus-code* [code]:

```

- (Complement B) = B
- (Collect-set P) = Collect-set ( $\lambda x. \neg P x$ )
- A = Complement A
by auto

```

These equations represent complements as true complements. If you want

that the complement operations returns an explicit enumeration of the elements, use the following set of equations which use *cenum*.

lemma *Set-uminus-cenum*:

fixes $A :: 'a :: cenum\ set$

shows $- A =$

(*case ID CENUM('a) of None \Rightarrow Complement A*
 | *Some (enum, -) \Rightarrow Set-Monad (filter ($\lambda x. x \notin A$) enum)*)

and $- (\text{Complement } B) = B$

by(*auto split: option.split dest: ID-cEnum*)

lemma *Set-minus-code* [*code, set-base-code*]:

fixes $rbt1\ rbt2 :: 'a :: ccompare\ set-rbt$

shows

$RBT-set\ rbt1 - RBT-set\ rbt2 =$
 (*case ID CCOMPARE('a) of None \Rightarrow Code.abort (STR "minus RBT-set*
RBT-set: ccompare = None") ($\lambda-. RBT-set\ rbt1 - RBT-set\ rbt2$)
 | *Some - \Rightarrow RBT-set (RBT-Set2.minus rbt1 rbt2)*)

$A - B = A \cap (- B)$

by (*auto simp: Set-member-code(3) split: option.splits*)

lemma *Set-union-code* [*code*]:

fixes $rbt1\ rbt2 :: 'a :: ccompare\ set-rbt$

and $rbt :: 'b :: \{ccompare, ceq\}\ set-rbt$

and $dxs :: 'b\ set-dlist$

and $dxs1\ dxs2 :: 'c :: ceq\ set-dlist$

shows *Set-union-Complement*:

$B' \cup \text{Complement } B = \text{Complement } (- B' \cap B)$

$\text{Complement } B \cup B' = \text{Complement } (B \cap - B')$

and [*set-base-code*]:

$RBT-set\ rbt1 \cup RBT-set\ rbt2 =$
 (*case ID CCOMPARE('a) of None \Rightarrow Code.abort (STR "union RBT-set*
RBT-set: ccompare = None") ($\lambda-. RBT-set\ rbt1 \cup RBT-set\ rbt2$)
 | *Some - \Rightarrow RBT-set (RBT-Set2.union rbt1 rbt2)*) (**is**
?RBT-set-RBT-set)

$RBT-set\ rbt \cup DList-set\ dxs =$
 (*case ID CCOMPARE('b) of None \Rightarrow Code.abort (STR "union RBT-set*
DList-set: ccompare = None") ($\lambda-. RBT-set\ rbt \cup DList-set\ dxs$)
 | *Some - \Rightarrow*

case ID CEQ('b) of None \Rightarrow Code.abort (STR "union RBT-set DList-set:
ceq = None") ($\lambda-. RBT-set\ rbt \cup DList-set\ dxs$)
 | *Some - \Rightarrow RBT-set (DList-Set.fold RBT-Set2.insert dxs rbt)*)

(**is** *?RBT-set-DList-set*)

$DList-set\ dxs \cup RBT-set\ rbt =$
 (*case ID CCOMPARE('b) of None \Rightarrow Code.abort (STR "union DList-set*
RBT-set: ccompare = None") ($\lambda-. RBT-set\ rbt \cup DList-set\ dxs$)
 | *Some - \Rightarrow*

case ID CEQ('b) of None \Rightarrow Code.abort (STR "union DList-set RBT-set:
ceq = None") ($\lambda-. RBT-set\ rbt \cup DList-set\ dxs$)
 | *Some - \Rightarrow RBT-set (DList-Set.fold RBT-Set2.insert dxs rbt)*)

```

(is ?DList-set-RBT-set)
  DList-set dxs1  $\cup$  DList-set dxs2 =
    (case ID CEQ('c) of None  $\Rightarrow$  Code.abort (STR "union DList-set DList-set:
ceq = None") ( $\lambda$ -. DList-set dxs1  $\cup$  DList-set dxs2)
      | Some -  $\Rightarrow$  DList-set (DList-Set.union dxs1 dxs2)) (is
?DList-set-DList-set)
  Set-Monad zs  $\cup$  RBT-set rbt2 =
    (case ID CCOMPARE('a) of None  $\Rightarrow$  Code.abort (STR "union Set-Monad
RBT-set: ccompare = None") ( $\lambda$ -. Set-Monad zs  $\cup$  RBT-set rbt2)
      | Some -  $\Rightarrow$  RBT-set (fold RBT-Set2.insert zs rbt2)) (is
?Set-Monad-RBT-set)
  RBT-set rbt1  $\cup$  Set-Monad zs =
    (case ID CCOMPARE('a) of None  $\Rightarrow$  Code.abort (STR "union RBT-set
Set-Monad: ccompare = None") ( $\lambda$ -. RBT-set rbt1  $\cup$  Set-Monad zs)
      | Some -  $\Rightarrow$  RBT-set (fold RBT-Set2.insert zs rbt1)) (is
?RBT-set-Set-Monad)
  Set-Monad ws  $\cup$  DList-set dxs2 =
    (case ID CEQ('c) of None  $\Rightarrow$  Code.abort (STR "union Set-Monad DList-set:
ceq = None") ( $\lambda$ -. Set-Monad ws  $\cup$  DList-set dxs2)
      | Some -  $\Rightarrow$  DList-set (fold DList-Set.insert ws dxs2)) (is
?Set-Monad-DList-set)
  DList-set dxs1  $\cup$  Set-Monad ws =
    (case ID CEQ('c) of None  $\Rightarrow$  Code.abort (STR "union DList-set Set-Monad:
ceq = None") ( $\lambda$ -. DList-set dxs1  $\cup$  Set-Monad ws)
      | Some -  $\Rightarrow$  DList-set (fold DList-Set.insert ws dxs1)) (is
?DList-set-Set-Monad)
  Set-Monad xs  $\cup$  Set-Monad ys = Set-Monad (xs @ ys)
  B  $\cup$  Collect-set A = Collect-set ( $\lambda$ x. A x  $\vee$  x  $\in$  B)
  Collect-set A  $\cup$  B = Collect-set ( $\lambda$ x. A x  $\vee$  x  $\in$  B)
proof -
  show ?RBT-set-RBT-set ?Set-Monad-RBT-set ?RBT-set-Set-Monad
  by(auto split: option.split simp add: RBT-set-def)

  show ?RBT-set-DList-set ?DList-set-RBT-set
  by(auto split: option.split simp add: RBT-set-def DList-set-def DList-Set.fold-def
DList-Set.member-def dest: equal.equal-eq[OF ID-ceq])

  show ?DList-set-Set-Monad ?Set-Monad-DList-set
  by(auto split: option.split simp add: DList-set-def DList-Set.member-fold-insert)

  show ?DList-set-DList-set
  by(auto split: option.split simp add: DList-set-def DList-Set.member-union)
qed auto

lemma Set-inter-code [code]:
fixes rbt1 rbt2 :: 'a :: ccompare set-rbt
and rbt :: 'b :: {ccompare, ceq} set-rbt
and dxs :: 'b set-dlist
and dxs1 dxs2 :: 'c :: ceq set-dlist

```

```

and  $xs1\ xs2 :: 'c\ list$ 
shows [set-base-code]:
   $RBT\text{-set}\ rbt1 \cap Set\text{-Monad}\ xs =$ 
    (case ID CCOMPARE('a) of None  $\Rightarrow$  Code.abort (STR "inter RBT-set
Set-Monad: ccompare = None") ( $\lambda$ -.  $RBT\text{-set}\ rbt1 \cap Set\text{-Monad}\ xs$ )
      | Some -  $\Rightarrow$   $RBT\text{-set}\ (RBT\text{-Set2.inter-list}\ rbt1\ xs)$ ) (is
?rbt-monad)

   $Set\text{-Monad}\ xs \cap RBT\text{-set}\ rbt1 =$ 
    (case ID CCOMPARE('a) of None  $\Rightarrow$  Code.abort (STR "inter Set-Monad
RBT-set: ccompare = None") ( $\lambda$ -.  $RBT\text{-set}\ rbt1 \cap Set\text{-Monad}\ xs$ )
      | Some -  $\Rightarrow$   $RBT\text{-set}\ (RBT\text{-Set2.inter-list}\ rbt1\ xs)$ ) (is
?monad-rbt)

   $Set\text{-Monad}\ xs' \cap DList\text{-set}\ dxs2 =$ 
    (case ID CEQ('c) of None  $\Rightarrow$  Code.abort (STR "inter Set-Monad DList-set:
ceq = None") ( $\lambda$ -.  $Set\text{-Monad}\ xs' \cap DList\text{-set}\ dxs2$ )
      | Some eq  $\Rightarrow$   $DList\text{-set}\ (DList\text{-Set.filter}\ (equal\text{-base.list-member}\ eq\ xs')\ dxs2)$ ) (is ?monad-dlist)

   $Set\text{-Monad}\ xs1 \cap Set\text{-Monad}\ xs2 =$ 
    (case ID CEQ('c) of None  $\Rightarrow$  Code.abort (STR "inter Set-Monad Set-Monad:
ceq = None") ( $\lambda$ -.  $Set\text{-Monad}\ xs1 \cap Set\text{-Monad}\ xs2$ )
      | Some eq  $\Rightarrow$   $Set\text{-Monad}\ (filter\ (equal\text{-base.list-member}\ eq\ xs2)\ xs1)$ ) (is ?monad)

   $RBT\text{-set}\ rbt \cap DList\text{-set}\ dxs =$ 
    (case ID CCOMPARE('b) of None  $\Rightarrow$  Code.abort (STR "inter RBT-set
DList-set: ccompare = None") ( $\lambda$ -.  $RBT\text{-set}\ rbt \cap DList\text{-set}\ dxs$ )
      | Some -  $\Rightarrow$ 
        case ID CEQ('b) of None  $\Rightarrow$  Code.abort (STR "inter RBT-set DList-set:
ceq = None") ( $\lambda$ -.  $RBT\text{-set}\ rbt \cap DList\text{-set}\ dxs$ )
          | Some -  $\Rightarrow$   $RBT\text{-set}\ (RBT\text{-Set2.inter-list}\ rbt\ (list\text{-of-dlist}\ dxs))$ )
(is ?rbt-dlist)

   $RBT\text{-set}\ rbt1 \cap RBT\text{-set}\ rbt2 =$ 
    (case ID CCOMPARE('a) of None  $\Rightarrow$  Code.abort (STR "inter RBT-set
RBT-set: ccompare = None") ( $\lambda$ -.  $RBT\text{-set}\ rbt1 \cap RBT\text{-set}\ rbt2$ )
      | Some -  $\Rightarrow$   $RBT\text{-set}\ (RBT\text{-Set2.inter}\ rbt1\ rbt2)$ ) (is ?rbt-rbt)

   $DList\text{-set}\ dxs1 \cap Set\text{-Monad}\ xs' =$ 
    (case ID CEQ('c) of None  $\Rightarrow$  Code.abort (STR "inter DList-set Set-Monad:
ceq = None") ( $\lambda$ -.  $DList\text{-set}\ dxs1 \cap Set\text{-Monad}\ xs'$ )
      | Some eq  $\Rightarrow$   $DList\text{-set}\ (DList\text{-Set.filter}\ (equal\text{-base.list-member}\ eq\ xs')\ dxs1)$ ) (is ?dlist-monad)

   $DList\text{-set}\ dxs1 \cap DList\text{-set}\ dxs2 =$ 
    (case ID CEQ('c) of None  $\Rightarrow$  Code.abort (STR "inter DList-set DList-set: ceq
= None") ( $\lambda$ -.  $DList\text{-set}\ dxs1 \cap DList\text{-set}\ dxs2$ )
      | Some -  $\Rightarrow$   $DList\text{-set}\ (DList\text{-Set.filter}\ (DList\text{-Set.member}\ dxs2)\ dxs1)$ ) (is ?dlist)

   $DList\text{-set}\ dxs \cap RBT\text{-set}\ rbt =$ 
    (case ID CCOMPARE('b) of None  $\Rightarrow$  Code.abort (STR "inter DList-set
RBT-set: ccompare = None") ( $\lambda$ -.  $DList\text{-set}\ dxs \cap RBT\text{-set}\ rbt$ )
      | Some -  $\Rightarrow$ 

```

```

      case ID CEQ('b) of None => Code.abort (STR "inter DList-set RBT-set: ceq
= None") (λ-. DList-set dxs ∩ RBT-set rbt)
      | Some - => RBT-set (RBT-Set2.inter-list rbt (list-of-dlist dxs))
(is ?dlist-rbt)
and Set-inter-Complement:
  Complement B'' ∩ Complement B''' = Complement (B'' ∪ B''') (is ?complement)
and [set-base-code]:
  G ∩ RBT-set rbt2 =
    (case ID CCOMPARE('a) of None => Code.abort (STR "inter RBT-set2:
ccompare = None") (λ-. G ∩ RBT-set rbt2)
    | Some - => RBT-set (RBT-Set2.filter (λx. x ∈ G) rbt2)) (is
?rbt2)
  RBT-set rbt1 ∩ G =
    (case ID CCOMPARE('a) of None => Code.abort (STR "inter RBT-set1:
ccompare = None") (λ-. RBT-set rbt1 ∩ G)
    | Some - => RBT-set (RBT-Set2.filter (λx. x ∈ G) rbt1)) (is
?rbt1)
  H ∩ DList-set dxs2 =
    (case ID CEQ('c) of None => Code.abort (STR "inter DList-set2: ceq =
None") (λ-. H ∩ DList-set dxs2)
    | Some eq => DList-set (DList-Set.filter (λx. x ∈ H) dxs2)) (is
?dlist2)
  DList-set dxs1 ∩ H =
    (case ID CEQ('c) of None => Code.abort (STR "inter DList-set1: ceq =
None") (λ-. DList-set dxs1 ∩ H)
    | Some eq => DList-set (DList-Set.filter (λx. x ∈ H) dxs1)) (is
?dlist1)
  I ∩ Set-Monad xs'' = Set-Monad (filter (λx. x ∈ I) xs'') (is ?monad2)
  Set-Monad xs'' ∩ I = Set-Monad (filter (λx. x ∈ I) xs'') (is ?monad1)
  J ∩ Collect-set A'' = Collect-set (λx. A'' x ∧ x ∈ J) (is ?collect2)
  Collect-set A'' ∩ J = Collect-set (λx. A'' x ∧ x ∈ J) (is ?collect1)
proof -
  show ?rbt-rbt ?rbt1 ?rbt2 ?rbt-dlist ?rbt-monad ?dlist-rbt ?monad-rbt
  by(auto simp add: RBT-set-def DList-set-def DList-Set.member-def dest: equal.equal-eq[OF
ID-ceq] split: option.split)
  show ?dlist ?dlist1 ?dlist2 ?dlist-monad ?monad-dlist ?monad ?monad1 ?monad2
?collect1 ?collect2 ?complement
  by(auto simp add: DList-set-def dest!: Collection-Eq.ID-ceq split: option.splits)
qed

```

lemma *Set-bind-code* [code, set-base-code]:

```

fixes dxs :: 'a :: ceq set-dlist
and rbt :: 'b :: ccompare set-rbt
shows
  Set.bind (RBT-set rbt) f'' =
    (case ID CCOMPARE('b) of None => Code.abort (STR "bind RBT-set:
ccompare = None") (λ-. Set.bind (RBT-set rbt) f'')
    | Some - => RBT-Set2.fold (union ∘ f'') rbt { }) (is ?RBT)
  Set.bind (DList-set dxs) f' =

```

```

    (case ID CEQ('a) of None => Code.abort (STR "bind DList-set: ceq = None")
  (λ-. Set.bind (DList-set dxs) f')
    | Some - => DList-Set.fold (union ∘ f') dxs {}) (is ?DList)
  Set.bind (Set-Monad xs) f = fold ((∪) ∘ f) xs (Set-Monad []) (is ?Set-Monad)
proof -
  show ?Set-Monad by (simp add: set-bind-conv-fold)
  show ?DList by (auto simp add: DList-set-def DList-Set.member-def List.member-iff
[abs-def] set-bind-conv-fold DList-Set.fold-def split: option.split dest: equal.equal-eq[OF
ID-ceq] ID-ceq)
  show ?RBT by (clarsimp split: option.split simp add: RBT-set-def RBT-Set2.fold-conv-fold-keys
RBT-Set2.member-conv-keys set-bind-conv-fold)
qed

```

lemma UNIV-code [code, set-base-code]:

UNIV = - {}

by simp

lift-definition inf-sls :: 'a :: lattice semilattice-set

is inf

..

lemma Inf-fin-code [code, set-base-code]:

Inf-fin A = set-fold1 inf-sls A

by transfer (simp add: Inf-fin-def)

lift-definition sup-sls :: 'a :: lattice semilattice-set

is sup

..

lemma Sup-fin-code [code, set-base-code]:

Sup-fin A = set-fold1 sup-sls A

by transfer (simp add: Sup-fin-def)

lift-definition inf-cfi :: ('a :: lattice, 'a) comp-fun-idem

is inf

by (rule comp-fun-idem-inf)

lemma Inf-code:

fixes A :: 'a :: complete-lattice set

shows Inf A = (if finite A then set-fold-cfi inf-cfi top A else Code.abort (STR "Inf: infinite")) (λ-. Inf A)

by transfer (simp add: Inf-fold-inf)

lift-definition sup-cfi :: ('a :: lattice, 'a) comp-fun-idem

is sup

by (rule comp-fun-idem-sup)

lemma Sup-code:

```

fixes A :: 'a :: complete-lattice set
shows Sup A = (if finite A then set-fold-cfi sup-cfi bot A else Code.abort (STR
"Sup: infinite") (λ-. Sup A))
by transfer (simp add: Sup-fold-sup)

```

```

lemmas Inter-code [code, set-base-code] = Inf-code[where ?'a = - :: type set]
lemmas Union-code [code, set-base-code] = Sup-code[where ?'a = - :: type set]
lemmas Predicate-Inf-code [code, set-base-code] = Inf-code[where ?'a = - :: type
Predicate.pred]
lemmas Predicate-Sup-code [code, set-base-code] = Sup-code[where ?'a = - :: type
Predicate.pred]
lemmas Inf-fun-code [code, set-base-code] = Inf-code[where ?'a = - :: type ⇒ - ::
complete-lattice]
lemmas Sup-fun-code [code, set-base-code] = Sup-code[where ?'a = - :: type ⇒ -
:: complete-lattice]

```

```

lift-definition min-sls :: 'a :: linorder semilattice-set
is min
..

```

```

lemma Min-code [code, set-base-code]:
  Min A = set-fold1 min-sls A
by transfer (simp add: Min-def)

```

```

lift-definition max-sls :: 'a :: linorder semilattice-set
is max
..

```

```

lemma Max-code [code, set-base-code]:
  Max A = set-fold1 max-sls A
by transfer (simp add: Max-def)

```

We do not implement *Ball*, *Bex*, and *sorted-list-of-set* for *Collect-set* using *CENUM*('a), because it should already have been converted to an explicit list of elements if that is possible.

```

lemma Ball-code [code, set-base-code]:
fixes rbt :: 'a :: ccompare set-rbt
and dxs :: 'b :: ceq set-dlist
shows
  Ball (RBT-set rbt) P'' =
    (case ID CCOMPARE('a) of None ⇒ Code.abort (STR "Ball RBT-set:
ccompare = None") (λ-. Ball (RBT-set rbt) P'')
    | Some - ⇒ RBT-Set2.all P'' rbt)
  Ball (DList-set dxs) P' =
    (case ID CEQ('b) of None ⇒ Code.abort (STR "Ball DList-set: ceq = None")
(λ-. Ball (DList-set dxs) P')
    | Some - ⇒ DList-Set.dlist-all P' dxs)
  Ball (Set-Monad xs) P = list-all P xs
by (simp-all add: DList-set-def RBT-set-def list-all-iff dlist-all-conv-member RBT-Set2.all-conv-all-men

```

split: option.splits)

lemma *Bex-code* [*code, set-base-code*]:

fixes *rbt* :: 'a :: *ccompare set-rbt*
and *dxs* :: 'b :: *ceq set-dlist*
shows
Bex (RBT-set rbt) P'' =
(case ID CCOMPARE('a) of None ⇒ Code.abort (STR "Bex RBT-set: ccompare = None") (λ-. Bex (RBT-set rbt) P'')
| Some - ⇒ RBT-Set2.ex P'' rbt)
Bex (DList-set dxs) P' =
(case ID CEQ('b) of None ⇒ Code.abort (STR "Bex DList-set: ceq = None")
(λ-. Bex (DList-set dxs) P')
| Some - ⇒ DList-Set.dlist-ex P' dxs)
Bex (Set-Monad xs) P = list-ex P xs
by (*simp-all add: DList-set-def RBT-set-def list-ex-iff dlist-ex-conv-member RBT-Set2.ex-conv-ex-member split: option.splits*)

lemma *csorted-list-of-set-code* [*code, set-base-code*]:

fixes *rbt* :: 'a :: *ccompare set-rbt*
and *dxs* :: 'b :: {*ccompare, ceq*} *set-dlist*
and *xs* :: 'a :: *ccompare list*
shows
csorted-list-of-set (Set-Monad xs) =
(case ID CCOMPARE('a) of None ⇒ Code.abort (STR "csorted-list-of-set
Set-Monad: ccompare = None") (λ-. csorted-list-of-set (Set-Monad xs))
| Some c ⇒ ord.remdups-sorted (lt-of-comp c) (ord.quick-sort
(lt-of-comp c) xs))
csorted-list-of-set (DList-set dxs) =
(case ID CEQ('b) of None ⇒ Code.abort (STR "csorted-list-of-set DList-set:
ceq = None") (λ-. csorted-list-of-set (DList-set dxs))
| Some - ⇒
case ID CCOMPARE('b) of None ⇒ Code.abort (STR "csorted-list-of-set
DList-set: ccompare = None") (λ-. csorted-list-of-set (DList-set dxs))
| Some c ⇒ ord.quick-sort (lt-of-comp c) (list-of-dlist dxs))
csorted-list-of-set (RBT-set rbt) =
(case ID CCOMPARE('a) of None ⇒ Code.abort (STR "csorted-list-of-set
RBT-set: ccompare = None") (λ-. csorted-list-of-set (RBT-set rbt))
| Some - ⇒ RBT-Set2.keys rbt)
by (*auto split: option.split simp add: RBT-set-def DList-Set.Collect-member member-conv-keys sorted-RBT-Set-keys linorder.sorted-list-of-set-sort-remdups[OF ID-ccompare] linorder.quick-sort-conv-sort[OF ID-ccompare] distinct-remdups-id distinct-list-of-dlist linorder.remdups-sorted-conv-remdups[OF ID-ccompare] linorder.sorted-sort[OF ID-ccompare] linorder.sort-remdups[OF ID-ccompare] csorted-list-of-set-def*)

lemma *cless-set-code* [*code*]:

fixes *rbt rbt'* :: 'a :: *ccompare set-rbt*
and *rbt1 rbt2* :: 'b :: *cproper-interval set-rbt*
and *A B* :: 'a *set*

```

and A' B' :: 'b set
shows cless-set-rbt-Complement1:
  cless-set (Complement (RBT-set rbt1)) (RBT-set rbt2)  $\longleftrightarrow$ 
  (case ID CCOMPARE('b) of None  $\Rightarrow$  Code.abort (STR "cless-set (Complement
  RBT-set) RBT-set: ccompare = None") ( $\lambda$ -. cless-set (Complement (RBT-set rbt1))
  (RBT-set rbt2))
    | Some c  $\Rightarrow$ 
      finite (UNIV :: 'b set)  $\wedge$ 
      proper-intrvl.Compl-set-less-aux-fusion (lt-of-comp c) cproper-interval
      rbt-keys-generator rbt-keys-generator None (RBT-Set2.init rbt1) (RBT-Set2.init
      rbt2))
    (is ?Compl-rbt)
and cless-set-rbt-Complement2:
  cless-set (RBT-set rbt1) (Complement (RBT-set rbt2))  $\longleftrightarrow$ 
  (case ID CCOMPARE('b) of None  $\Rightarrow$  Code.abort (STR "cless-set RBT-set
  (Complement RBT-set): ccompare = None") ( $\lambda$ -. cless-set (RBT-set rbt1) (Complement
  (RBT-set rbt2)))
    | Some c  $\Rightarrow$ 
      finite (UNIV :: 'b set)  $\longrightarrow$ 
      proper-intrvl.set-less-aux-Compl-fusion (lt-of-comp c) cproper-interval
      rbt-keys-generator rbt-keys-generator None (RBT-Set2.init rbt1) (RBT-Set2.init
      rbt2))
    (is ?rbt-Compl)
and [set-base-code]:
  cless-set (RBT-set rbt) (RBT-set rbt')  $\longleftrightarrow$ 
  (case ID CCOMPARE('a) of None  $\Rightarrow$  Code.abort (STR "cless-set RBT-set
  RBT-set: ccompare = None") ( $\lambda$ -. cless-set (RBT-set rbt) (RBT-set rbt'))
    | Some c  $\Rightarrow$  ord.lexord-fusion ( $\lambda$ x y. lt-of-comp c y x) rbt-keys-generator
      rbt-keys-generator (RBT-Set2.init rbt) (RBT-Set2.init rbt'))
    (is ?rbt-rbt)
and cless-set-Complement12:
  cless-set (Complement A) (Complement B)  $\longleftrightarrow$ 
  (case ID CCOMPARE('a) of None  $\Rightarrow$  Code.abort (STR "cless-set Complement
  Complement: ccompare = None") ( $\lambda$ -. cless-set (Complement A) (Complement B))
    | Some -  $\Rightarrow$  cless B A)
    (is ?Compl-Compl)
and cless-set-Complement1:
  cless-set (Complement A) B'  $\longleftrightarrow$ 
  (case ID CCOMPARE('b) of None  $\Rightarrow$  Code.abort (STR "cless-set Comple-
  ment1: ccompare = None") ( $\lambda$ -. cless-set (Complement A) B'))
    | Some c  $\Rightarrow$ 
      if finite A'  $\wedge$  finite B' then
        finite (UNIV :: 'b set)  $\wedge$ 
        proper-intrvl.Compl-set-less-aux (lt-of-comp c) cproper-interval None
        (csorted-list-of-set A') (csorted-list-of-set B')
      else Code.abort (STR "cless-set Complement1: infinite set") ( $\lambda$ -. cless-set
      (Complement A) B'))
    (is ?Compl-fin-fin)
and cless-set-Complement2:

```

```

cless-set A' (Complement B')  $\longleftrightarrow$ 
  (case ID CCOMPARE('b) of None  $\Rightarrow$  Code.abort (STR "cless-set Complement2: ccompare = None") ( $\lambda$ -. cless-set A' (Complement B'))
    | Some c  $\Rightarrow$ 
      if finite A'  $\wedge$  finite B' then
        finite (UNIV :: 'b set)  $\longrightarrow$ 
          proper-intrvl.set-less-aux-Compl (lt-of-comp c) cproper-interval None
          (csorted-list-of-set A') (csorted-list-of-set B')
        else Code.abort (STR "cless-set Complement2: infinite set") ( $\lambda$ -. cless-set
A' (Complement B'))
      (is ?fin-Compl-fin)
  and [set-base-code]:
    cless-set A B  $\longleftrightarrow$ 
      (case ID CCOMPARE('a) of None  $\Rightarrow$  Code.abort (STR "cless-set: ccompare = None") ( $\lambda$ -. cless-set A B)
        | Some c  $\Rightarrow$ 
          if finite A  $\wedge$  finite B then ord.lexordp ( $\lambda$ x y. lt-of-comp c y x) (csorted-list-of-set
A) (csorted-list-of-set B)
          else Code.abort (STR "cless-set: infinite set") ( $\lambda$ -. cless-set A B)
        (is ?fin-fin)

```

proof –

```

note [split] = option.split csorted-list-of-set-split
and [simp] =
  le-of-comp-of-ords-linorder[OF ID-ccompare]
  lt-of-comp-of-ords
  finite-subset[OF subset-UNIV] ccompare-set-def ID-Some
  ord.lexord-fusion-def
  proper-intrvl.Compl-set-less-aux-fusion-def
  proper-intrvl.set-less-aux-Compl-fusion-def
  unfoldr-rbt-keys-generator
  RBT-set-def sorted-RBT-Set-keys member-conv-keys
  linorder.set-less-finite-iff[OF ID-ccompare]
  linorder.set-less-aux-code[OF ID-ccompare, symmetric]
  linorder.Compl-set-less-Compl[OF ID-ccompare]
  linorder.infinite-set-less-Complement[OF ID-ccompare]
  linorder.infinite-Complement-set-less[OF ID-ccompare]
  linorder-proper-interval.set-less-aux-Compl2-conv-set-less-aux-Compl[OF ID-ccompare-interval, symmetric]
  linorder-proper-interval.Compl1-set-less-aux-conv-Compl-set-less-aux[OF ID-ccompare-interval, symmetric]

```

show *?Compl-Compl* **by** *simp*

show *?rbt-rbt*

by *auto*

show *?rbt-Compl* **by**(*cases* *finite* (*UNIV* :: 'b *set*)) *auto*

show *?Compl-rbt* **by**(*cases* *finite* (*UNIV* :: 'b *set*)) *auto*

show *?fin-fin* **by** *auto*

show *?fin-Compl-fin* **by**(*cases* *finite* (*UNIV* :: 'b *set*), *auto*)

show *?Compl-fin-fin* **by**(*cases* *finite* (*UNIV* :: 'b *set*)) *auto*

qed

lemma *le-of-comp-set-less-eq*:

le-of-comp (comp-of-ords (ord.set-less-eq le) (ord.set-less le)) = ord.set-less-eq le
by (rule *le-of-comp-of-ords-gen*, simp add: *ord.set-less-def*)

lemma *cless-eq-set-code* [code]:

fixes *rbt rbt' :: 'a :: ccompare set-rbt*

and *rbt1 rbt2 :: 'b :: cproper-interval set-rbt*

and *A B :: 'a set*

and *A' B' :: 'b set*

shows *cless-eq-set-rbt-Complement1*:

cless-eq-set (Complement (RBT-set rbt1)) (RBT-set rbt2) \longleftrightarrow

(case ID CCOMPARE('b) of None \Rightarrow Code.abort (STR "cless-eq-set (Complement RBT-set) RBT-set: ccompare = None") (λ -. cless-eq-set (Complement (RBT-set rbt1)) (RBT-set rbt2))

| Some c \Rightarrow

finite (UNIV :: 'b set) \wedge

proper-intrvl.Compl-set-less-eq-aux-fusion (lt-of-comp c) cproper-interval rbt-keys-generator rbt-keys-generator None (RBT-Set2.init rbt1) (RBT-Set2.init rbt2))

(is ?Compl-rbt)

and *cless-eq-set-rbt-Complement2*:

cless-eq-set (RBT-set rbt1) (Complement (RBT-set rbt2)) \longleftrightarrow

(case ID CCOMPARE('b) of None \Rightarrow Code.abort (STR "cless-eq-set RBT-set (Complement RBT-set): ccompare = None") (λ -. cless-eq-set (RBT-set rbt1) (Complement (RBT-set rbt2)))

| Some c \Rightarrow

finite (UNIV :: 'b set) \longrightarrow

proper-intrvl.set-less-eq-aux-Compl-fusion (lt-of-comp c) cproper-interval rbt-keys-generator rbt-keys-generator None (RBT-Set2.init rbt1) (RBT-Set2.init rbt2))

(is ?rbt-Compl)

and [set-base-code]:

cless-eq-set (RBT-set rbt) (RBT-set rbt') \longleftrightarrow

(case ID CCOMPARE('a) of None \Rightarrow Code.abort (STR "cless-eq-set RBT-set RBT-set: ccompare = None") (λ -. cless-eq-set (RBT-set rbt) (RBT-set rbt'))

| Some c \Rightarrow ord.lexord-eq-fusion ($\lambda x y. lt-of-comp c y x$)

rbt-keys-generator rbt-keys-generator (RBT-Set2.init rbt) (RBT-Set2.init rbt'))

(is ?rbt-rbt)

and *cless-eq-set-Complement12*:

cless-eq-set (Complement A) (Complement B) \longleftrightarrow

(case ID CCOMPARE('a) of None \Rightarrow Code.abort (STR "cless-eq-set Complement Complement: ccompare = None") (λ -. cless-eq (Complement A) (Complement B))

| Some c \Rightarrow cless-eq-set B A)

(is ?Compl-Compl)

and *cless-eq-set-Complement1*:

cless-eq-set (Complement A) B' \longleftrightarrow

```

(case ID CCOMPARE('b) of None ⇒ Code.abort (STR "cless-eq-set Complement1: ccompare = None") (λ-. cless-eq-set (Complement A') B')
  | Some c ⇒
    if finite A' ∧ finite B'
    then finite (UNIV :: 'b set) ∧
      proper-intrvl.Compl-set-less-eq-aux (lt-of-comp c) cproper-interval None
      (csorted-list-of-set A') (csorted-list-of-set B')
    else Code.abort (STR "cless-eq-set Complement1: infinite set") (λ-. cless-eq-set
      (Complement A') B')
    (is ?Compl-fin-fin)
and cless-eq-set-Complement2:
  cless-eq-set A' (Complement B') ↔
    (case ID CCOMPARE('b) of None ⇒ Code.abort (STR "cless-eq-set Complement2: ccompare = None") (λ-. cless-eq-set A' (Complement B'))
      | Some c ⇒
        if finite A' ∧ finite B'
        then finite (UNIV :: 'b set) →
          proper-intrvl.set-less-eq-aux-Compl (lt-of-comp c) cproper-interval None
          (csorted-list-of-set A') (csorted-list-of-set B')
        else Code.abort (STR "cless-eq-set Complement2: infinite set") (λ-. cless-eq-set
          A' (Complement B'))
        (is ?fin-Compl-fin)
    and [set-base-code]:
      cless-eq-set A B ↔
        (case ID CCOMPARE('a) of None ⇒ Code.abort (STR "cless-eq-set: ccompare = None") (λ-. cless-eq-set A B)
          | Some c ⇒
            if finite A ∧ finite B
            then ord.lexordp-eq (λx y. lt-of-comp c y x) (csorted-list-of-set A) (csorted-list-of-set
              B)
            else Code.abort (STR "cless-eq-set: infinite set") (λ-. cless-eq-set A B)
          (is ?fin-fin)

```

proof –

```

note [split] = option.split csorted-list-of-set-split
and [simp] =
  le-of-comp-set-less-eq
  finite-subset[OF subset-UNIV] ccompare-set-def ID-Some
  ord.lexord-eq-fusion-def proper-intrvl.Compl-set-less-eq-aux-fusion-def
  proper-intrvl.set-less-eq-aux-Compl-fusion-def
  unfoldr-rbt-keys-generator
  RBT-set-def sorted-RBT-Set-keys member-conv-keys
  linorder.set-less-eq-finite-iff[OF ID-ccompare]
  linorder.set-less-eq-aux-code[OF ID-ccompare, symmetric]
  linorder.Compl-set-less-eq-Compl[OF ID-ccompare]
  linorder.infinite-set-less-eq-Complement[OF ID-ccompare]
  linorder.infinite-Complement-set-less-eq[OF ID-ccompare]
  linorder-proper-interval.set-less-eq-aux-Compl2-conv-set-less-eq-aux-Compl[OF
  ID-ccompare-interval, symmetric]
  linorder-proper-interval.Compl1-set-less-eq-aux-conv-Compl-set-less-eq-aux[OF

```

ID-ccompare-interval, symmetric]

show *?Compl-Compl* **by** *simp*

show *?rbt-rbt*

by *auto*

show *?rbt-Compl* **by** *(cases finite (UNIV :: 'b set)) auto*

show *?Compl-rbt* **by** *(cases finite (UNIV :: 'b set)) auto*

show *?fin-fin* **by** *auto*

show *?fin-Compl-fin* **by** *(cases finite (UNIV :: 'b set), auto)*

show *?Compl-fin-fin* **by** *(cases finite (UNIV :: 'b set)) auto*

qed

lemma *cproper-interval-set-Some-Some-code* [*code*]:

fixes *rbt1 rbt2 :: 'a :: cproper-interval set-rbt*

and *A B :: 'a set*

shows *cproper-interval-set-Some-Complement-Some-rbt:*

cproper-interval (Some (Complement (RBT-set rbt1))) (Some (RBT-set rbt2))

\longleftrightarrow

(case ID CCOMPARE('a) of None \Rightarrow Code.abort (STR "cproper-interval (Complement RBT-set) RBT-set: ccompare = None") (λ -. cproper-interval (Some (Complement (RBT-set rbt1))) (Some (RBT-set rbt2))))

| Some c \Rightarrow

finite (UNIV :: 'a set) \wedge proper-interval.proper-interval-Compl-set-aux-fusion (lt-of-comp c) cproper-interval rbt-keys-generator rbt-keys-generator None (RBT-Set2.init rbt1) (RBT-Set2.init rbt2))

(is ?Compl-rbt-rbt)

and *cproper-interval-set-Some-rbt-Some-Complement:*

cproper-interval (Some (RBT-set rbt1)) (Some (Complement (RBT-set rbt2)))

\longleftrightarrow

(case ID CCOMPARE('a) of None \Rightarrow Code.abort (STR "cproper-interval RBT-set (Complement RBT-set): ccompare = None") (λ -. cproper-interval (Some (RBT-set rbt1)) (Some (Complement (RBT-set rbt2))))

| Some c \Rightarrow

finite (UNIV :: 'a set) \wedge proper-interval.proper-interval-set-Compl-aux-fusion (lt-of-comp c) cproper-interval rbt-keys-generator rbt-keys-generator None 0 (RBT-Set2.init rbt1) (RBT-Set2.init rbt2))

(is ?rbt-Compl-rbt)

and [*set-base-code*]:

cproper-interval (Some (RBT-set rbt1)) (Some (RBT-set rbt2)) \longleftrightarrow

(case ID CCOMPARE('a) of None \Rightarrow Code.abort (STR "cproper-interval RBT-set RBT-set: ccompare = None") (λ -. cproper-interval (Some (RBT-set rbt1)) (Some (RBT-set rbt2))))

| Some c \Rightarrow

finite (UNIV :: 'a set) \wedge proper-interval.proper-interval-set-aux-fusion (lt-of-comp c) cproper-interval rbt-keys-generator rbt-keys-generator (RBT-Set2.init rbt1) (RBT-Set2.init rbt2))

(is ?rbt-rbt)

and *cproper-interval-set-Some-Complement-Some-Complement:*

```

    cproper-interval (Some (Complement A)) (Some (Complement B))  $\longleftrightarrow$ 
      (case ID CCOMPARE('a) of None  $\Rightarrow$  Code.abort (STR "cproper-interval Com-
plement Complement: ccompare = None") ( $\lambda$ -. cproper-interval (Some (Complement
A)) (Some (Complement B))))
      | Some -  $\Rightarrow$  cproper-interval (Some B) (Some A))
  (is ?Compl-Compl)
and cproper-interval-set-Some-Complement-Some:
    cproper-interval (Some (Complement A)) (Some B)  $\longleftrightarrow$ 
      (case ID CCOMPARE('a) of None  $\Rightarrow$  Code.abort (STR "cproper-interval
Complement1: ccompare = None") ( $\lambda$ -. cproper-interval (Some (Complement A))
(Some B))
      | Some c  $\Rightarrow$ 
        finite (UNIV :: 'a set)  $\wedge$  proper-intrvl.proper-interval-Compl-set-aux (lt-of-comp
c) cproper-interval None (csorted-list-of-set A) (csorted-list-of-set B))
        (is ?Compl-fin-fin)
and cproper-interval-set-Some-Some-Complement:
    cproper-interval (Some A) (Some (Complement B))  $\longleftrightarrow$ 
      (case ID CCOMPARE('a) of None  $\Rightarrow$  Code.abort (STR "cproper-interval Com-
plement2: ccompare = None") ( $\lambda$ -. cproper-interval (Some A) (Some (Complement
B))))
      | Some c  $\Rightarrow$ 
        finite (UNIV :: 'a set)  $\wedge$  proper-intrvl.proper-interval-set-Compl-aux (lt-of-comp
c) cproper-interval None 0 (csorted-list-of-set A) (csorted-list-of-set B))
        (is ?fin-Compl-fin)
and [set-base-code]:
    cproper-interval (Some A) (Some B)  $\longleftrightarrow$ 
      (case ID CCOMPARE('a) of None  $\Rightarrow$  Code.abort (STR "cproper-interval:
ccompare = None") ( $\lambda$ -. cproper-interval (Some A) (Some B))
      | Some c  $\Rightarrow$ 
        finite (UNIV :: 'a set)  $\wedge$  proper-intrvl.proper-interval-set-aux (lt-of-comp c)
cproper-interval (csorted-list-of-set A) (csorted-list-of-set B))
        (is ?fin-fin)
proof -
note [split] = option.split csorted-list-of-set-split
and [simp] =
  lt-of-comp-of-ords
  finite-subset[OF subset-UNIV] ccompare-set-def ID-Some
  linorder.set-less-finite-iff[OF ID-ccompare]
  RBT-set-def sorted-RBT-Set-keys member-conv-keys
  linorder.distinct-entries[OF ID-ccompare]
  unfoldr-rbt-keys-generator
  proper-intrvl.proper-interval-set-aux-fusion-def
  proper-intrvl.proper-interval-set-Compl-aux-fusion-def
  proper-intrvl.proper-interval-Compl-set-aux-fusion-def
  linorder-proper-interval.proper-interval-set-aux[OF ID-ccompare-interval]
  linorder-proper-interval.proper-interval-set-Compl-aux[OF ID-ccompare-interval]
  linorder-proper-interval.proper-interval-Compl-set-aux[OF ID-ccompare-interval]
and [cong] = conj-cong

```

```

show ?Compl-Compl
  by(clarsimp simp add: Complement-cproper-interval-set-Complement simp del:
cproper-interval-set-Some-Some)

```

```

show ?rbt-rbt ?rbt-Compl-rbt ?Compl-rbt-rbt by auto
show ?fin-fin ?fin-Compl-fin ?Compl-fin-fin by auto
qed

```

```

context ord begin

```

```

fun sorted-list-subset :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ 'a list ⇒ bool
where
  sorted-list-subset eq [] ys = True
| sorted-list-subset eq (x # xs) [] = False
| sorted-list-subset eq (x # xs) (y # ys) ←→
  (if eq x y then sorted-list-subset eq xs ys
   else x > y ∧ sorted-list-subset eq (x # xs) ys)

```

```

end

```

```

context linorder begin

```

```

lemma sorted-list-subset-correct:
  [| sorted xs; distinct xs; sorted ys; distinct ys |]
  ⇒ sorted-list-subset (=) xs ys ←→ set xs ⊆ set ys
apply(induct (=) :: 'a ⇒ 'a ⇒ bool xs ys rule: sorted-list-subset.induct)
  apply(auto 6 2)
  using order-antisym apply auto
  done

```

```

end

```

```

context ord begin

```

```

definition sorted-list-subset-fusion :: ('a ⇒ 'a ⇒ bool) ⇒ ('a, 's1) generator ⇒
('a, 's2) generator ⇒ 's1 ⇒ 's2 ⇒ bool
where sorted-list-subset-fusion eq g1 g2 s1 s2 = sorted-list-subset eq (list.unfoldr
g1 s1) (list.unfoldr g2 s2)

```

```

lemma sorted-list-subset-fusion-code:
  sorted-list-subset-fusion eq g1 g2 s1 s2 =
  (if list.has-next g1 s1 then
    let (x, s1') = list.next g1 s1
    in list.has-next g2 s2 ∧ (
      let (y, s2') = list.next g2 s2
      in if eq x y then sorted-list-subset-fusion eq g1 g2 s1' s2'
        else y < x ∧ sorted-list-subset-fusion eq g1 g2 s1 s2')
    else True)
unfolding sorted-list-subset-fusion-def

```

by(*subst* (1 2 5) *list.unfoldr.simps*)(*simp add: split-beta Let-def*)

end

lemmas [*code, set-base-code*] = *ord.sorted-list-subset-fusion-code*

lemma *subset-eq-code* [*code*]:

fixes *A1 A2* :: 'a *set*

and *rbt* :: 'b :: *compare set-rbt*

and *rbt1 rbt2* :: 'd :: {*compare, ceq*} *set-rbt*

and *dxs* :: 'c :: *ceq set-dlist*

and *xs* :: 'c *list*

shows [*set-base-code*]:

$RBT\text{-set } rbt1 \subseteq RBT\text{-set } rbt2 \iff$

(*case ID CCOMPARE*('d) of *None* \Rightarrow *Code.abort* (*STR* "subset *RBT-set*
RBT-set: compare = None") (λ -. $RBT\text{-set } rbt1 \subseteq RBT\text{-set } rbt2$)

| *Some c* \Rightarrow

(*case ID CEQ*('d) of *None* \Rightarrow *ord.sorted-list-subset-fusion* (*lt-of-comp c*)

(λ *x y. c x y = Eq*) *rbt-keys-generator rbt-keys-generator* (*RBT-Set2.init rbt1*)
(*RBT-Set2.init rbt2*)

| *Some eq* \Rightarrow *ord.sorted-list-subset-fusion* (*lt-of-comp c*) *eq*
rbt-keys-generator rbt-keys-generator (*RBT-Set2.init rbt1*) (*RBT-Set2.init rbt2*)))

(**is** ?*rbt-rbt*)

and *Complement-subset-eq-Complement*:

$Complement\ A1 \subseteq Complement\ A2 \iff A2 \subseteq A1$ (**is** ?*Compl*)

and *Collect-subset-eq-Complement*:

$Collect\text{-set } P \subseteq Complement\ A \iff A \subseteq \{x. \neg P\ x\}$ (**is** ?*Collect-set-Compl*)

and [*set-base-code*]:

$RBT\text{-set } rbt \subseteq B \iff$

(*case ID CCOMPARE*('b) of *None* \Rightarrow *Code.abort* (*STR* "subset *RBT-set1*:
compare = None") (λ -. $RBT\text{-set } rbt \subseteq B$)

| *Some -* \Rightarrow *list-all-fusion rbt-keys-generator* ($\lambda x. x \in B$)

(*RBT-Set2.init rbt*)) (**is** ?*rbt*)

$DList\text{-set } dxs \subseteq C \iff$

(*case ID CEQ*('c) of *None* \Rightarrow *Code.abort* (*STR* "subset *DList-set1*: *ceq =*
None") (λ -. $DList\text{-set } dxs \subseteq C$)

| *Some -* \Rightarrow *DList-Set.dlist-all* ($\lambda x. x \in C$) *dxs*) (**is** ?*dlist*)

Set-Monad xs $\subseteq C \iff$ *list-all* ($\lambda x. x \in C$) *xs*) (**is** ?*Set-Monad*)

proof –

show ?*rbt-rbt*

by (*auto simp add: comparator.eq[OF ID-ccompare]* *RBT-set-def member-conv-keys*
unfoldr-rbt-keys-generator ord.sorted-list-subset-fusion-def linorder.sorted-list-subset-correct[OF
ID-ccompare] *sorted-RBT-Set-keys split: option.split dest!: ID-ceq[THEN equal.equal-eq]*
del: iffI)

show ?*rbt*

by (*auto simp add: RBT-set-def member-conv-keys list-all-fusion-def unfoldr-rbt-keys-generator*
keys.rep-eq list-all-iff split: option.split)

show ?*dlist* **by** (*auto simp add: DList-set-def dlist-all-conv-member split: op-*

```

tion.split)
  show ?Set-Monad by(auto simp add: list-all-iff split: option.split)
  show ?Collect-set-Compl ?Compl by auto
qed

lemma set-eq-code [code]:
  fixes rbt1 rbt2 :: 'b :: {ccompare, ceq} set-rbt
  shows [set-base-code]:
    set-eq (RBT-set rbt1) (RBT-set rbt2) =
      (case ID CCOMPARE('b) of None  $\Rightarrow$  Code.abort (STR "set-eq RBT-set
RBT-set: ccompare = None") ( $\lambda$ -. set-eq (RBT-set rbt1) (RBT-set rbt2))
      | Some c  $\Rightarrow$ 
        (case ID CEQ('b) of None  $\Rightarrow$  list-all2-fusion ( $\lambda$  x y. c x y = Eq) rbt-keys-generator
rbt-keys-generator (RBT-Set2.init rbt1) (RBT-Set2.init rbt2)
        | Some eq  $\Rightarrow$  list-all2-fusion eq rbt-keys-generator rbt-keys-generator
(RBT-Set2.init rbt1) (RBT-Set2.init rbt2)))
    (is ?rbt-rbt)
  and set-eq-Complement-Complement:
    set-eq (Complement A) (Complement B) = set-eq A B
  and [set-base-code]: set-eq A B  $\longleftrightarrow$  A  $\subseteq$  B  $\wedge$  B  $\subseteq$  A
proof -
  show ?rbt-rbt
    by (auto 4 3 split: option.split simp add: comparator.eq[OF ID-ccompare]
sorted-RBT-Set-keys list-all2-fusion-def unfoldr-rbt-keys-generator RBT-set-conv-keys
set-eq-def list.rel-eq dest!: ID-ceq[THEN equal.equal-eq] intro: linorder.sorted-distinct-set-unique[OF
ID-ccompare])
  qed (auto simp add: set-eq-def)

lemma Set-project-code [code, set-base-code]:
  Set.filter P A = A  $\cap$  Collect-set P
  by (simp add: set-eq-iff)

lemma Set-image-code [code]:
  fixes dxs :: 'a :: ceq set-dlist
  and rbt :: 'b :: ccompare set-rbt
  shows [set-base-code]:
    image h (RBT-set rbt) =
      (case ID CCOMPARE('b) of None  $\Rightarrow$  Code.abort (STR "image RBT-set:
ccompare = None") ( $\lambda$ -. image h (RBT-set rbt))
      | Some -  $\Rightarrow$  RBT-Set2.fold (insert  $\circ$  h) rbt {})
    (is ?rbt)
    image g (DList-set dxs) =
      (case ID CEQ('a) of None  $\Rightarrow$  Code.abort (STR "image DList-set: ceq =
None") ( $\lambda$ -. image g (DList-set dxs))
      | Some -  $\Rightarrow$  DList-Set.fold (insert  $\circ$  g) dxs {})
    (is ?dlist)
  and image-Complement-Complement:
    image f (Complement (Complement B)) = image f B
  and [set-base-code]:

```

$image\ f\ (Collect\text{-}set\ A) = Code.abort\ (STR\ "image\ Collect\text{-}set")\ (\lambda\cdot.\ image\ f\ (Collect\text{-}set\ A))$

$image\ f\ (Set\text{-}Monad\ xs) = Set\text{-}Monad\ (map\ f\ xs)$

proof –

{ **fix** xs **have** $fold\ (insert\ \circ\ g)\ xs\ \{\} = g\ 'set\ xs$

by($induct\ xs\ rule:\ rev\text{-}induct\ simp\text{-}all$ **}**

thus $?dlist$

by($simp\ add:\ DList\text{-}set\text{-}def\ DList\text{-}Set.\text{fold}\text{-}def\ DList\text{-}Set.\text{Collect}\text{-}member\ split:\ option.\text{split}$)

{ **fix** xs **have** $fold\ (insert\ \circ\ h)\ xs\ \{\} = h\ 'set\ xs$

by($induct\ xs\ rule:\ rev\text{-}induct\ simp\text{-}all$ **}**

thus $?rbt$ **by**($auto\ simp\ add:\ RBT\text{-}set\text{-}def\ fold\text{-}conv\text{-}fold\text{-}keys\ member\text{-}conv\text{-}keys\ split:\ option.\text{split}$)

qed $simp\text{-}all$

lemma $the\text{-}elem\text{-}code$ [$code$]:

fixes $dxs :: 'a :: ceq\ set\text{-}dlist$

and $rbt :: 'b :: ccompare\ set\text{-}rbt$

shows

$the\text{-}elem\ (RBT\text{-}set\ rbt) =$

($case\ ID\ CCOMPARE('b)$ of $None \Rightarrow Code.abort\ (STR\ "the\text{-}elem\ RBT\text{-}set:\ ccompare = None")\ (\lambda\cdot.\ the\text{-}elem\ (RBT\text{-}set\ rbt))$

| $Some\ - \Rightarrow$

$case\ RBT\text{-}Mapping2.\text{impl}\text{-}of\ rbt$ of $RBT\text{-}Impl.Branch - RBT\text{-}Impl.Empty\ x - RBT\text{-}Impl.Empty \Rightarrow x$

| $- \Rightarrow Code.abort\ (STR\ "the\text{-}elem\ RBT\text{-}set:\ not\ unique")\ (\lambda\cdot.\ the\text{-}elem\ (RBT\text{-}set\ rbt))$)

$the\text{-}elem\ (DList\text{-}set\ dxs) =$

($case\ ID\ CEQ('a)$ of $None \Rightarrow Code.abort\ (STR\ "the\text{-}elem\ DList\text{-}set:\ ceq = None")\ (\lambda\cdot.\ the\text{-}elem\ (DList\text{-}set\ dxs))$

| $Some\ - \Rightarrow$

$case\ list\text{-}of\text{-}dlist\ dxs$ of $[x] \Rightarrow x$

| $- \Rightarrow Code.abort\ (STR\ "the\text{-}elem\ DList\text{-}set:\ not\ unique")\ (\lambda\cdot.\ the\text{-}elem\ (DList\text{-}set\ dxs))$)

$the\text{-}elem\ (Set\text{-}Monad\ [x]) = x$

by ($auto\ simp\ add:\ RBT\text{-}set\text{-}def\ DList\text{-}set\text{-}def\ DList\text{-}Set.\text{Collect}\text{-}member\ the\text{-}elem\text{-}def\ member\text{-}conv\text{-}keys\ split:\ option.\text{split}\ list.\text{split}\ rbt.\text{split}$)($simp\ add:\ RBT\text{-}Set2.\text{keys}\text{-}def$)

lemma $Pow\text{-}set\text{-}conv\text{-}fold$:

$Pow\ (set\ xs\ \cup\ A) = fold\ (\lambda x\ A.\ A\ \cup\ insert\ x\ 'A)\ xs\ (Pow\ A)$

by($induct\ xs\ rule:\ rev\text{-}induct$)($auto\ simp\ add:\ Pow\text{-}insert$)

lemma $Pow\text{-}code$ [$code$, $set\text{-}base\text{-}code$]:

fixes $dxs :: 'a :: ceq\ set\text{-}dlist$

and $rbt :: 'b :: ccompare\ set\text{-}rbt$

shows

$Pow\ (RBT\text{-}set\ rbt) =$

($case\ ID\ CCOMPARE('b)$ of $None \Rightarrow Code.abort\ (STR\ "Pow\ RBT\text{-}set:\ ccompare = None")\ (\lambda\cdot.\ Pow\ (RBT\text{-}set\ rbt))$)

```

      | Some - => RBT-Set2.fold (λx A. A ∪ insert x ' A) rbt {{{}})
Pow (DList-set dxs) =
  (case ID CEQ('a) of None => Code.abort (STR "Pow DList-set: ceq = None")
(λ-. Pow (DList-set dxs))
      | Some - => DList-Set.fold (λx A. A ∪ insert x ' A) dxs {{{}})
Pow (Set-Monad xs) = fold (λx A. A ∪ insert x ' A) xs {{{}}
Pow A = Collect-set (λB. B ⊆ A)
by (auto simp add: DList-set-def DList-Set.Collect-member DList-Set.fold-def
RBT-set-def fold-conv-fold-keys member-conv-keys Pow-set-conv-fold[where A={},
simplified] split: option.split)

```

lemma fold-singleton: *Finite-Set.fold f x {y} = f y x*
by(fastforce simp add: Finite-Set.fold-def intro: fold-graph.intros elim: fold-graph.cases)

lift-definition *sum-cfc* :: ('a => 'b :: comm-monoid-add) => ('a, 'b) comp-fun-commute
is λf :: 'a => 'b. plus ∘ f
by(unfold-locales)(simp add: fun-eq-iff add.left-commute)

lemma sum-code [code]:
sum f A = (if finite A then set-fold-cfc (sum-cfc f) 0 A else 0)
by transfer(simp add: sum.eq-fold)

lift-definition *prod-cfc* :: ('a => 'b :: comm-monoid-mult) => ('a, 'b) comp-fun-commute
is λf :: 'a => 'b. times ∘ f
by standard (simp add: fun-eq-iff mult.left-commute)

lemma prod-code [code]:
prod f A = (if finite A then set-fold-cfc (prod-cfc f) 1 A else 1)
by transfer (simp add: prod.eq-fold)

lemma product-code [code]:
fixes *dxs* :: 'a :: ceq set-dlist
and *dys* :: 'b :: ceq set-dlist
and *rbt1* :: 'c :: ccompare set-rbt
and *rbt2* :: 'd :: ccompare set-rbt
shows
Product-Type.product (RBT-set rbt1) (RBT-set rbt2) =
 (case ID CCOMPARE('c) of None => Code.abort (STR "product RBT-set
RBT-set: ccompare1 = None") (λ-. Product-Type.product (RBT-set rbt1) (RBT-set
rbt2))
 | Some - =>
 case ID CCOMPARE('d) of None => Code.abort (STR "product RBT-set
RBT-set: ccompare2 = None") (λ-. Product-Type.product (RBT-set rbt1) (RBT-set
rbt2))
 | Some - => RBT-set (RBT-Set2.product rbt1 rbt2))

Product-Type.product A2 (RBT-set rbt2) =
 (case ID CCOMPARE('d) of None => Code.abort (STR "product RBT-set:
ccompare2 = None") (λ-. Product-Type.product A2 (RBT-set rbt2))

```

      | Some - => RBT-Set2.fold (λy rest. (λx. (x, y)) ‘ A2 ∪ rest)
rbt2 {}
  (is ?rbt2)

```

```

Product-Type.product (RBT-set rbt1) B2 =
  (case ID CCOMPARE('c) of None => Code.abort (STR "product RBT-set:
ccompare1 = None") (λ-. Product-Type.product (RBT-set rbt1) B2)
  | Some - => RBT-Set2.fold (λx rest. Pair x ‘ B2 ∪ rest) rbt1
  {})
  (is ?rbt1)

```

```

Product-Type.product (DList-set dxs) (DList-set dys) =
  (case ID CEQ('a) of None => Code.abort (STR "product DList-set DList-set:
ceq1 = None") (λ-. Product-Type.product (DList-set dxs) (DList-set dys))
  | Some - =>
    case ID CEQ('b) of None => Code.abort (STR "product DList-set DList-set:
ceq2 = None") (λ-. Product-Type.product (DList-set dxs) (DList-set dys))
    | Some - => DList-Set (DList-Set.product dxs dys)

```

```

Product-Type.product A1 (DList-set dys) =
  (case ID CEQ('b) of None => Code.abort (STR "product DList-set2: ceq =
None") (λ-. Product-Type.product A1 (DList-set dys))
  | Some - => DList-Set.fold (λy rest. (λx. (x, y)) ‘ A1 ∪ rest) dys
  {})
  (is ?dlist2)

```

```

Product-Type.product (DList-set dxs) B1 =
  (case ID CEQ('a) of None => Code.abort (STR "product DList-set1: ceq =
None") (λ-. Product-Type.product (DList-set dxs) B1)
  | Some - => DList-Set.fold (λx rest. Pair x ‘ B1 ∪ rest) dxs {})
  (is ?dlist1)

```

```

Product-Type.product (Set-Monad xs) (Set-Monad ys) =
  Set-Monad (fold (λx. fold (λy rest. (x, y) # rest) ys) xs [])
  (is ?Set-Monad)

```

```

Product-Type.product A B = Collect-set (λ(x, y). x ∈ A ∧ y ∈ B)

```

proof –

```

have [simp]: ∧a zs. fold (λy. (#) (a, y)) ys zs = rev (map (Pair a) ys) @ zs
by(induct ys) simp-all
have [simp]: ∧zs. fold (λx. fold (λy rest. (x, y) # rest) ys) xs zs = rev (concat
(map (λx. map (Pair x) ys) xs)) @ zs
by(induct xs) simp-all
show ?Set-Monad by(auto simp add: Product-Type.product-def)

```

```

{ fix xs :: 'a list
  have fold (λx. (∪) (Pair x ‘ B1)) xs {} = set xs × B1
  by(induct xs rule: rev-induct) auto }

```

```

thus ?dlist1
  by(simp add: Product-Type.product-def DList-set-def DList-Set.fold.rep-eq DList-Set.Collect-member
split: option.split)

{ fix ys :: 'b list
  have fold ( $\lambda y. (\cup) ((\lambda x. (x, y)) \text{' } A1))$  ys {} =  $A1 \times \text{set } ys$ 
  by(induct ys rule: rev-induct) auto }
thus ?dlist2
  by(simp add: Product-Type.product-def DList-set-def DList-Set.fold.rep-eq DList-Set.Collect-member
split: option.split)

{ fix xs :: 'c list
  have fold ( $\lambda x. (\cup) (\text{Pair } x \text{' } B2))$  xs {} =  $\text{set } xs \times B2$ 
  by(induct xs rule: rev-induct) auto }
thus ?rbt1
  by(simp add: Product-Type.product-def RBT-set-def RBT-Set2.member-product
RBT-Set2.member-conv-keys fold-conv-fold-keys split: option.split)

{ fix ys :: 'd list
  have fold ( $\lambda y. (\cup) ((\lambda x. (x, y)) \text{' } A2))$  ys {} =  $A2 \times \text{set } ys$ 
  by(induct ys rule: rev-induct) auto }
thus ?rbt2
  by(simp add: Product-Type.product-def RBT-set-def RBT-Set2.member-product
RBT-Set2.member-conv-keys fold-conv-fold-keys split: option.split)
qed(auto simp add: RBT-set-def DList-set-def Product-Type.product-def DList-Set.product-member
RBT-Set2.member-product split: option.split)

lemma Id-on-code [code]:
  fixes A :: 'a :: ceq set
  and dxs :: 'a set-dlist
  and P :: 'a  $\Rightarrow$  bool
  and rbt :: 'b :: ccompare set-rbt
  shows [set-base-code]:
    Id-on (RBT-set rbt) =
      (case ID CCOMPARE('b) of None  $\Rightarrow$  Code.abort (STR "Id-on RBT-set:
ccompare = None") ( $\lambda\cdot$ . Id-on (RBT-set rbt))
      | Some -  $\Rightarrow$  RBT-set (RBT-Set2.Id-on rbt))
    Id-on (DList-set dxs) =
      (case ID CEQ('a) of None  $\Rightarrow$  Code.abort (STR "Id-on DList-set: ceq = None")
( $\lambda\cdot$ . Id-on (DList-set dxs))
      | Some -  $\Rightarrow$  DList-set (DList-Set.Id-on dxs))
    Id-on (Collect-set P) =
      (case ID CEQ('a) of None  $\Rightarrow$  Code.abort (STR "Id-on Collect-set: ceq =
None") ( $\lambda\cdot$ . Id-on (Collect-set P))
      | Some eq  $\Rightarrow$  Collect-set ( $\lambda(x, y). \text{eq } x \text{ } y \wedge P \text{ } x$ ))
  and Id-on-Complement:
    Id-on (Complement A) =
      (case ID CEQ('a) of None  $\Rightarrow$  Code.abort (STR "Id-on Complement: ceq =
None") ( $\lambda\cdot$ . Id-on (Complement A))

```

```

      | Some eq  $\Rightarrow$  Collect-set ( $\lambda(x, y). eq\ x\ y \wedge x \notin A$ )
and [set-base-code]:
  Id-on B = ( $\lambda x. (x, x)$ ) ‘ B
by (auto simp add: DList-set-def RBT-set-def DList-Set.member-Id-on RBT-Set2.member-Id-on
dest: equal.equal-eq[OF ID-ceq] split: option.split)

lemma Image-code [code, set-base-code]:
fixes dxs :: ('a :: ceq  $\times$  'b :: ceq) set-dlist
and rbt :: ('c :: ccompare  $\times$  'd :: ccompare) set-rbt
shows
  RBT-set rbt “ C =
    (case ID CCOMPARE('c) of None  $\Rightarrow$  Code.abort (STR "Image RBT-set:
ccompare1 = None") ( $\lambda\cdot$ . RBT-set rbt “ C)
      | Some -  $\Rightarrow$ 
        case ID CCOMPARE('d) of None  $\Rightarrow$  Code.abort (STR "Image RBT-set:
ccompare2 = None") ( $\lambda\cdot$ . RBT-set rbt “ C)
          | Some -  $\Rightarrow$ 
            RBT-Set2.fold ( $\lambda(x, y) acc. if\ x \in C\ then\ insert\ y\ acc\ else\ acc$ ) rbt {}
    (is ?RBT-set)
  DList-set dxs “ B =
    (case ID CEQ('a) of None  $\Rightarrow$  Code.abort (STR "Image DList-set: ceq1 =
None") ( $\lambda\cdot$ . DList-set dxs “ B)
      | Some -  $\Rightarrow$ 
        case ID CEQ('b) of None  $\Rightarrow$  Code.abort (STR "Image DList-set: ceq2 =
None") ( $\lambda\cdot$ . DList-set dxs “ B)
          | Some -  $\Rightarrow$ 
            DList-Set.fold ( $\lambda(x, y) acc. if\ x \in B\ then\ insert\ y\ acc\ else\ acc$ ) dxs {}
    (is ?DList-set)
  Set-Monad rxs “ A = Set-Monad (fold ( $\lambda(x, y) rest. if\ x \in A\ then\ y \# rest$ 
else rest) rxs [])
    (is ?Set-Monad)
  X “ Y = snd ‘ Set.filter ( $\lambda(x, y). x \in Y$ ) X
    (is ?generic)
proof –
  show ?generic by(auto intro: rev-image-eqI)

  have set (fold ( $\lambda(x, y) rest. if\ x \in A\ then\ y \# rest\ else\ rest$ ) rxs []) = set rxs “
A
  by(induct rxs rule: rev-induct)(auto split: if-split-asm)
  thus ?Set-Monad by(auto)

  { fix dxs :: ('a  $\times$  'b) list
    have fold ( $\lambda(x, y) acc. if\ x \in B\ then\ insert\ y\ acc\ else\ acc$ ) dxs {} = set dxs “
B
    by(induct dxs rule: rev-induct)(auto split: if-split-asm) }
  thus ?DList-set
  by(clarsimp simp add: DList-set-def Collect-member ceq-prod-def ID-Some
DList-Set.fold.rep-eq split: option.split)

```

```

{ fix rbt :: (('c × 'd) × unit) list
  have fold (λ(a, -). case a of (x, y) ⇒ λacc. if x ∈ C then insert y acc else acc)
rbt {} = (fst ' set rbt) “ C
  by(induct rbt rule: rev-induct)(auto simp add: split-beta split: if-split-asm) }
thus ?RBT-set
  by(clarsimp simp add: RBT-set-def ccompare-prod-def ID-Some RBT-Set2.fold.rep-eq
member-conv-keys RBT-Set2.keys.rep-eq RBT-Impl.fold-def RBT-Impl.keys-def split:
option.split)
qed

```

lemma *insert-relcomp*: $insert\ (a, b)\ A\ O\ B = A\ O\ B \cup \{a\} \times \{c.\ (b, c) \in B\}$
by *auto*

lemma *trancl-code* [*code*, *set-base-code*]:
trancl A =
(if finite A then ntrancl (card A - 1) A else Code.abort (STR "trancl: infinite set") (λ-. trancl A))
by (*simp add: finite-trancl-ntrancl*)

lemma *set-relcomp-set*:
set xs O set ys = fold (λ(x, y). fold (λ(y', z) A. if y = y' then insert (x, z) A else A) ys) xs {}
proof(*induct xs rule: rev-induct*)
case *Nil show ?case by simp*
next
case (*snoc x xs*)
note [*show-types*]
{ **fix** a :: 'a **and** b :: 'c **and** X :: ('a × 'b) set
have fold (λ(y', z) A. if b = y' then insert (a, z) A else A) ys X = X ∪ {a}
× {c. (b, c) ∈ set ys}
by(*induct ys arbitrary: X rule: rev-induct*)(*auto split: if-split-asm*) }
thus ?case **using** *snoc by*(*cases x*)(*simp add: insert-relcomp*)
qed

lemma *If-not*: $(if\ \neg\ a\ then\ b\ else\ c) = (if\ a\ then\ c\ else\ b)$
by *auto*

lemma *relcomp-code* [*code*, *set-base-code*]:
fixes *rbt1* :: ('a :: ccompare × 'b :: ccompare) set-rbt
and *rbt2* :: ('b × 'c :: ccompare) set-rbt
and *rbt3* :: ('a × 'd :: {ccompare, ceq}) set-rbt
and *rbt4* :: ('d × 'a) set-rbt
and *rbt5* :: ('b × 'a) set-rbt
and *drs1* :: ('d × 'e :: ceq) set-dlist
and *drs2* :: ('e × 'd) set-dlist
and *drs3* :: ('e × 'f :: ceq) set-dlist
and *drs4* :: ('f × 'g :: ceq) set-dlist
and *xs1* :: ('h × 'i :: ceq) list

```

and  $xs2 :: ('i \times 'j)$  list
and  $xs3 :: ('b \times 'h)$  list
and  $xs4 :: ('h \times 'b)$  list
and  $xs5 :: ('f \times 'h)$  list
and  $xs6 :: ('h \times 'f)$  list
shows
  RBT-set  $rbt1$  O RBT-set  $rbt2$  =
    (case ID CCOMPARE('a) of None  $\Rightarrow$  Code.abort (STR "relcomp RBT-set
RBT-set: ccompare1 = None") ( $\lambda$ -. RBT-set  $rbt1$  O RBT-set  $rbt2$ )
      | Some -  $\Rightarrow$ 
      case ID CCOMPARE('b) of None  $\Rightarrow$  Code.abort (STR "relcomp RBT-set
RBT-set: ccompare2 = None") ( $\lambda$ -. RBT-set  $rbt1$  O RBT-set  $rbt2$ )
        | Some  $c-b$   $\Rightarrow$ 
        case ID CCOMPARE('c) of None  $\Rightarrow$  Code.abort (STR "relcomp RBT-set
RBT-set: ccompare3 = None") ( $\lambda$ -. RBT-set  $rbt1$  O RBT-set  $rbt2$ )
          | Some -  $\Rightarrow$  RBT-Set2.fold ( $\lambda(x, y).$  RBT-Set2.fold ( $\lambda(y',$ 
z) A. if  $c-b$   $y$   $y' \neq Eq$  then A else insert (x, z) A)  $rbt2$ )  $rbt1$  { })
            (is ?rbt-rbt)

  RBT-set  $rbt3$  O DList-set  $dxs1$  =
    (case ID CCOMPARE('a) of None  $\Rightarrow$  Code.abort (STR "relcomp RBT-set
DList-set: ccompare1 = None") ( $\lambda$ -. RBT-set  $rbt3$  O DList-set  $dxs1$ )
      | Some -  $\Rightarrow$ 
      case ID CCOMPARE('d) of None  $\Rightarrow$  Code.abort (STR "relcomp RBT-set
DList-set: ccompare2 = None") ( $\lambda$ -. RBT-set  $rbt3$  O DList-set  $dxs1$ )
        | Some -  $\Rightarrow$ 
        case ID CEQ('d) of None  $\Rightarrow$  Code.abort (STR "relcomp RBT-set
DList-set: ceq2 = None") ( $\lambda$ -. RBT-set  $rbt3$  O DList-set  $dxs1$ )
          | Some  $eq$   $\Rightarrow$ 
          case ID CEQ('e) of None  $\Rightarrow$  Code.abort (STR "relcomp RBT-set
DList-set: ceq3 = None") ( $\lambda$ -. RBT-set  $rbt3$  O DList-set  $dxs1$ )
            | Some -  $\Rightarrow$  RBT-Set2.fold ( $\lambda(x, y).$  DList-Set.fold ( $\lambda(y',$ 
z) A. if  $eq$   $y$   $y'$  then insert (x, z) A else A)  $dxs1$ )  $rbt3$  { })
              (is ?rbt-dlist)

  DList-set  $dxs2$  O RBT-set  $rbt4$  =
    (case ID CEQ('e) of None  $\Rightarrow$  Code.abort (STR "relcomp DList-set RBT-set:
ceq1 = None") ( $\lambda$ -. DList-set  $dxs2$  O RBT-set  $rbt4$ )
      | Some -  $\Rightarrow$ 
      case ID CCOMPARE('d) of None  $\Rightarrow$  Code.abort (STR "relcomp DList-set
RBT-set: ceq2 = None") ( $\lambda$ -. DList-set  $dxs2$  O RBT-set  $rbt4$ )
        | Some -  $\Rightarrow$ 
        case ID CEQ('d) of None  $\Rightarrow$  Code.abort (STR "relcomp DList-set RBT-set:
ccompare2 = None") ( $\lambda$ -. DList-set  $dxs2$  O RBT-set  $rbt4$ )
          | Some  $eq$   $\Rightarrow$ 
          case ID CCOMPARE('a) of None  $\Rightarrow$  Code.abort (STR "relcomp DList-set
RBT-set: ccompare3 = None") ( $\lambda$ -. DList-set  $dxs2$  O RBT-set  $rbt4$ )
            | Some -  $\Rightarrow$  DList-Set.fold ( $\lambda(x, y).$  RBT-Set2.fold ( $\lambda(y',$ 
z) A. if  $eq$   $y$   $y'$  then insert (x, z) A else A)  $rbt4$ )  $dxs2$  { })

```

(is ?dlist-rbt)

DList-set *dxs3* *O* *DList-set* *dxs4* =
 (case ID CEQ('e) of None \Rightarrow Code.abort (STR "relcomp *DList-set* *DList-set*:
 ceq1 = None") (λ -. *DList-set* *dxs3* *O* *DList-set* *dxs4*)
 | Some - \Rightarrow
 case ID CEQ('f) of None \Rightarrow Code.abort (STR "relcomp *DList-set* *DList-set*:
 ceq2 = None") (λ -. *DList-set* *dxs3* *O* *DList-set* *dxs4*)
 | Some eq \Rightarrow
 case ID CEQ('g) of None \Rightarrow Code.abort (STR "relcomp *DList-set* *DList-set*:
 ceq3 = None") (λ -. *DList-set* *dxs3* *O* *DList-set* *dxs4*)
 | Some - \Rightarrow *DList-Set.fold* ($\lambda(x, y)$. *DList-Set.fold* ($\lambda(y', z)$
 A. if eq y y' then insert (x, z) A else A) *dxs4* *dxs3* {})
 (is ?dlist-dlist)

Set-Monad *xs1* *O* *Set-Monad* *xs2* =
 (case ID CEQ('i) of None \Rightarrow Code.abort (STR "relcomp *Set-Monad* *Set-Monad*:
 ceq = None") (λ -. *Set-Monad* *xs1* *O* *Set-Monad* *xs2*)
 | Some eq \Rightarrow fold ($\lambda(x, y)$. fold ($\lambda(y', z)$ A. if eq y y' then insert
 (x, z) A else A) *xs2*) *xs1* {})
 (is ?monad-monad)

RBT-set *rbt1* *O* *Set-Monad* *xs3* =
 (case ID CCOMPARE('a) of None \Rightarrow Code.abort (STR "relcomp *RBT-set* *Set-Monad*:
 ccompare1 = None") (λ -. *RBT-set* *rbt1* *O* *Set-Monad* *xs3*)
 | Some - \Rightarrow
 case ID CCOMPARE('b) of None \Rightarrow Code.abort (STR "relcomp *RBT-set* *Set-Monad*:
 ccompare2 = None") (λ -. *RBT-set* *rbt1* *O* *Set-Monad* *xs3*)
 | Some c-b \Rightarrow *RBT-Set2.fold* ($\lambda(x, y)$. fold ($\lambda(y', z)$ A. if c-b y y' \neq
 Eq then A else insert (x, z) A) *xs3*) *rbt1* {})
 (is ?rbt-monad)

Set-Monad *xs4* *O* *RBT-set* *rbt5* =
 (case ID CCOMPARE('a) of None \Rightarrow Code.abort (STR "relcomp *Set-Monad* *RBT-set*:
 ccompare1 = None") (λ -. *Set-Monad* *xs4* *O* *RBT-set* *rbt5*)
 | Some - \Rightarrow
 case ID CCOMPARE('b) of None \Rightarrow Code.abort (STR "relcomp *Set-Monad* *RBT-set*:
 ccompare2 = None") (λ -. *Set-Monad* *xs4* *O* *RBT-set* *rbt5*)
 | Some c-b \Rightarrow fold ($\lambda(x, y)$. *RBT-Set2.fold* ($\lambda(y', z)$ A. if c-b y y' \neq
 Eq then A else insert (x, z) A) *rbt5*) *xs4* {})
 (is ?monad-rbt)

DList-set *dxs3* *O* *Set-Monad* *xs5* =
 (case ID CEQ('e) of None \Rightarrow Code.abort (STR "relcomp *DList-set* *Set-Monad*:
 ceq1 = None") (λ -. *DList-set* *dxs3* *O* *Set-Monad* *xs5*)
 | Some - \Rightarrow
 case ID CEQ('f) of None \Rightarrow Code.abort (STR "relcomp *DList-set* *Set-Monad*:
 ceq2 = None") (λ -. *DList-set* *dxs3* *O* *Set-Monad* *xs5*)
 | Some eq \Rightarrow *DList-Set.fold* ($\lambda(x, y)$. fold ($\lambda(y', z)$ A. if eq y y'

then insert (x, z) A else A) xs5) dxs3 {}}
 (is ?dlist-monad)

Set-Monad xs6 O DList-set dxs4 =
 (case ID CEQ('f) of None \Rightarrow Code.abort (STR "relcomp Set-Monad DList-set:
 ceq1 = None") (λ -. Set-Monad xs6 O DList-set dxs4)
 | Some eq \Rightarrow
 case ID CEQ('g) of None \Rightarrow Code.abort (STR "relcomp Set-Monad DList-set:
 ceq2 = None") (λ -. Set-Monad xs6 O DList-set dxs4)
 | Some - \Rightarrow fold ($\lambda(x, y)$. DList-Set.fold ($\lambda(y', z)$ A. if eq y y'
 then insert (x, z) A else A) dxs4) xs6 {}}
 (is ?monad-dlist)

proof –

show ?rbt-rbt ?rbt-monad ?monad-rbt

by(auto simp add: comparator.eq[OF ID-ccompare] RBT-set-def ccompare-prod-def
 member-conv-keys ID-Some RBT-Set2.fold-conv-fold-keys' RBT-Set2.keys.rep-eq If-not
 set-relcomp-set split: option.split del: equalityI)

show ?rbt-dlist ?dlist-rbt ?dlist-dlist ?monad-monad ?dlist-monad ?monad-dlist

by(auto simp add: RBT-set-def DList-set-def member-conv-keys ID-Some ccom-
 pare-prod-def ceq-prod-def Collect-member RBT-Set2.fold-conv-fold-keys' RBT-Set2.keys.rep-eq
 DList-Set.fold.rep-eq set-relcomp-set dest: equal.equal-eq[OF ID-ceq] split: option.split
 del: equalityI)

qed

lemma irrefl-on-code [code, set-base-code]:

fixes r :: ('a :: {ceq, ccompare} \times 'a) set **shows**

irrefl-on A r \longleftrightarrow

(case ID CEQ('a) of Some eq \Rightarrow ($\forall (x, y) \in r. x \in A \longrightarrow y \in A \longrightarrow \neg eq x y$) |
 None \Rightarrow

case ID CCOMPARE('a) of None \Rightarrow Code.abort (STR "irrefl-on: ceq = None
 & ccompare = None") (λ -. irrefl-on A r)

| Some c \Rightarrow ($\forall (x, y) \in r. x \in A \longrightarrow y \in A \longrightarrow c x y \neq Eq$))

apply(auto simp add: irrefl-on-distinct comparator.eq[OF ID-ccompare] split:
 option.split dest!: ID-ceq[THEN equal.equal-eq])

done

lemma wf-code [code, set-base-code]:

fixes rbt :: ('a :: ccompare \times 'a) set-rbt

and dxs :: ('b :: ceq \times 'b) set-dlist

shows

wf-code (DList-set dxs) =

(case ID CEQ('b) of None \Rightarrow Code.abort (STR "wf-code DList-set: ceq =
 None") (λ -. wf-code (DList-set dxs))

| Some - \Rightarrow acyclic (DList-set dxs))

wf-code (RBT-set rbt) =

(case ID CCOMPARE('a) of None \Rightarrow Code.abort (STR "wf-code RBT-set:
 ccompare = None") (λ -. wf-code (RBT-set rbt))

$| \text{Some } - \Rightarrow \text{acyclic } (\text{RBT-set } \text{rbt}))$
 $\text{wf-code } (\text{Set-Monad } \text{xs}) = \text{acyclic } (\text{Set-Monad } \text{xs})$
by (*auto simp add: wf-iff-acyclic-if-finite wf-code-def split: option.split del: iffI*)
(simp-all add: wf-iff-acyclic-if-finite finite-code ccompare-prod-def ceq-prod-def ID-Some)

lemma *bacc-code* [*code, set-base-code*]:

$\text{bacc } R \ 0 = - \text{snd } 'R$
 $\text{bacc } R \ (\text{Suc } n) = (\text{let } \text{rec} = \text{bacc } R \ n \ \text{in } \text{rec} \cup - \text{snd } '(\text{Set.filter } (\lambda(y, x). y \notin \text{rec}) R))$
by(*auto intro: rev-image-eqI simp add: Let-def*)

lemma *acc-code* [*code, set-base-code*]:

fixes $A :: ('a :: \{\text{finite, card-UNIV}\} \times 'a) \text{ set}$ **shows**
 $\text{Wellfounded.acc } A = \text{bacc } A \ (\text{of-phantom } (\text{card-UNIV} :: 'a \ \text{card-UNIV}))$
by(*simp add: card-UNIV acc-bacc-eq*)

lemma *sorted-list-of-set-code* [*code, set-base-code*]:

fixes $\text{dxs} :: 'a :: \{\text{linorder, ceq}\} \text{ set-dlist}$
and $\text{rbt} :: 'b :: \{\text{linorder, ccompare}\} \text{ set-rbt}$
shows
 $\text{sorted-list-of-set } (\text{RBT-set } \text{rbt}) =$
 $(\text{case ID CCOMPARE}'b) \ \text{of } \text{None} \Rightarrow \text{Code.abort } (\text{STR } \text{"sorted-list-of-set RBT-set: ccompare = None"}) \ (\lambda-. \text{sorted-list-of-set } (\text{RBT-set } \text{rbt}))$
 $| \text{Some } - \Rightarrow \text{sort } (\text{RBT-Set2.keys } \text{rbt})$
— We must sort the keys because *ccompare*'s ordering need not coincide with *linorder*'s.
 $\text{sorted-list-of-set } (\text{DList-set } \text{dxs}) =$
 $(\text{case ID CEQ}'a) \ \text{of } \text{None} \Rightarrow \text{Code.abort } (\text{STR } \text{"sorted-list-of-set DList-set: ceq = None"}) \ (\lambda-. \text{sorted-list-of-set } (\text{DList-set } \text{dxs}))$
 $| \text{Some } - \Rightarrow \text{sort } (\text{list-of-dlist } \text{dxs})$
 $\text{sorted-list-of-set } (\text{Set-Monad } \text{xs}) = \text{sort } (\text{remdups } \text{xs})$
by (*auto simp add: DList-set-def RBT-set-def sorted-list-of-set-sort-remdups Collect-member distinct-remdups-id distinct-list-of-dlist member-conv-keys split: option.split*)

lemma *image-filter-set-conv-fold*:

$\langle \text{Option.image-filter } f \ (\text{set } \text{xs}) =$
 $\text{fold } (\lambda x \ A. \ \text{case } f \ x \ \text{of } \text{None} \Rightarrow A \ | \ \text{Some } y \Rightarrow \text{insert } y \ A) \ \text{xs } \{\}\rangle$
by (*induction xs rule: rev-induct*) (*simp-all add: Option.image-filter-eq split: option.split*)

lemma *image-filter-code* [*code, set-base-code*]:

fixes $\text{dxs} :: 'a :: \text{ceq set-dlist}$
and $\text{rbt} :: 'b :: \text{ccompare set-rbt}$
shows
 $\text{Option.image-filter } h \ (\text{RBT-set } \text{rbt}) =$

```

    (case ID CCOMPARE('b) of None ⇒ Code.abort (STR "Option.image-filter
RBT-set: ccompare = None") (λ-. Option.image-filter h (RBT-set rbt))
    | Some - ⇒ RBT-Set2.fold (λx A. case h x of None ⇒ A |
Some y ⇒ insert y A) rbt {})
    (is ?rbt)
    Option.image-filter g (DList-set dxs) =
    (case ID CEQ('a) of None ⇒ Code.abort (STR "Option.image-filter DList-set:
ceq = None") (λ-. Option.image-filter g (DList-set dxs))
    | Some - ⇒ DList-Set.fold (λx A. case g x of None ⇒ A | Some y
⇒ insert y A) dxs {})
    (is ?dlist)
    Option.image-filter f (Set-Monad xs) = Set-Monad (List.map-filter f xs)
    (is ?set-monad)

```

proof –

```

    show ?dlist ?rbt
    by (auto split: option.split simp add: RBT-set-def DList-set-def DList-Set.fold.rep-eq
Collect-member
image-filter-set-conv-fold RBT-Set2.fold-conv-fold-keys member-conv-keys del:
equalityI)
    show ?set-monad
    by (auto simp add: Option.image-filter-eq Option.these-eq Option.is-none-def
List.map-filter-def)
qed

```

lemma *these-code* [code, set-base-code]:

```

⟨Option.these A = Option.image-filter (λx. x) A⟩
by (simp add: Option.image-filter-eq)

```

lemma *can-select-code*:

```

fixes xs :: 'a :: ceq list
and dxs :: 'a :: ceq set-dlist
and rbt :: 'b :: ccompare set-rbt
shows
Set.can-select R (RBT-set rbt) =
  (case ID CCOMPARE('b) of None ⇒ Code.abort (STR "Set.can-select
RBT-set: ccompare = None") (λ-. Set.can-select R (RBT-set rbt))
  | Some - ⇒ singleton-list-fusion (filter-generator R
rbt-keys-generator) (RBT-Set2.init rbt))
  (is ?rbt)
Set.can-select Q (DList-set dxs) =
  (case ID CEQ('a) of None ⇒ Code.abort (STR "Set.can-select DList-set: ceq
= None") (λ-. Set.can-select Q (DList-set dxs))
  | Some - ⇒ DList-Set.length (DList-Set.filter Q dxs) = 1)
  (is ?dlist)
Set.can-select P (Set-Monad xs) =
  (case ID CEQ('a) of None ⇒ Code.abort (STR "Set.can-select Set-Monad:
ceq = None") (λ-. Set.can-select P (Set-Monad xs))
  | Some eq ⇒ case filter P xs of Nil ⇒ False | x # xs ⇒ list-all
(eq x) xs)

```

```

      (is ?Set-Monad)
proof –
  show ?Set-Monad
    apply(auto split: option.split list.split dest!: ID-ceq[THEN equal.equal-eq] dest:
filter-eq-ConsD simp add: filter-empty-conv list-all-iff)
    apply(drule filter-eq-ConsD, fastforce)
    apply(drule filter-eq-ConsD, clarsimp, blast)
  done

show ?dlist
  by(clarsimp simp add: card-eq-length[symmetric] Set-member-code card-eq-Suc-0-ex1
simp del: card-eq-length split: option.split)

note [simp del] = distinct-keys
show ?rbt
  using distinct-keys[of rbt]
  apply(auto simp add: singleton-list-fusion-def unfoldr-filter-generator unfoldr-rbt-keys-generator
Set-member-code member-conv-keys filter-empty-conv empty-filter-conv split: op-
tion.split list.split dest: filter-eq-ConsD)
  apply(drule filter-eq-ConsD, fastforce)
  apply(drule filter-eq-ConsD, fastforce simp add: empty-filter-conv)
  apply(drule filter-eq-ConsD)
  apply clarsimp
  apply(drule Cons-eq-filterD)
  apply clarify
  apply(simp (no-asm-use))
  apply blast
done
qed

lemma [code]:
  ‹Set.can-select P A  $\longleftrightarrow$  is-singleton (Set.filter P A)›
  by (fact Set.can-select-iff-is-singleton)

lemma pred-of-set-code [code, set-base-code]:
  fixes dxs :: 'a :: ceq set-dlist
  and rbt :: 'b :: ccompare set-rbt
  shows
    pred-of-set (RBT-set rbt) =
      (case ID CCOMPARE('b) of None  $\Rightarrow$  Code.abort (STR "pred-of-set RBT-set:
ccompare = None") ( $\lambda$ -. pred-of-set (RBT-set rbt))
      | Some -  $\Rightarrow$  RBT-Set2.fold (sup  $\circ$  Predicate.single) rbt bot)
    pred-of-set (DList-set dxs) =
      (case ID CEQ('a) of None  $\Rightarrow$  Code.abort (STR "pred-of-set DList-set: ceq =
None") ( $\lambda$ -. pred-of-set (DList-set dxs))
      | Some -  $\Rightarrow$  DList-Set.fold (sup  $\circ$  Predicate.single) dxs bot)
    pred-of-set (Set-Monad xs) = fold (sup  $\circ$  Predicate.single) xs bot
  by (auto simp add: pred-of-set-set-fold-sup fold-map DList-set-def RBT-set-def Col-
lect-member member-conv-keys DList-Set.fold.rep-eq fold-conv-fold-keys split: op-

```

tion.split)

'a *Predicate.pred* is implemented as a monad, so we keep the monad when converting to 'a *set*. For this case, *insert-monad* and *union-monad* avoid the unnecessary dictionary construction.

definition *insert-monad* :: 'a \Rightarrow 'a *set* \Rightarrow 'a *set*
where [*simp*]: *insert-monad* = *insert*

definition *union-monad* :: 'a *set* \Rightarrow 'a *set* \Rightarrow 'a *set*
where [*simp*]: *union-monad* = (\cup)

lemma *insert-monad-code* [*code*, *set-base-code*]:
insert-monad *x* (*Set-Monad* *xs*) = *Set-Monad* (*x* # *xs*)
by *simp*

lemma *union-monad-code* [*code*, *set-base-code*]:
union-monad (*Set-Monad* *xs*) (*Set-Monad* *ys*) = *Set-Monad* (*xs* @ *ys*)
by(*simp*)

lemma *set-of-pred-code* [*code*, *set-base-code*]:
set-of-pred (*Predicate.Seq* *f*) =
(case *f* () of *seq.Empty* \Rightarrow *Set-Monad* []
| *seq.Insert* *x* *P* \Rightarrow *insert-monad* *x* (*set-of-pred* *P*)
| *seq.Join* *P* *xq* \Rightarrow *union-monad* (*set-of-pred* *P*) (*set-of-seq* *xq*))
by(*simp* *add*: *of-pred-code* *cong*: *seq.case-cong*)

lemma *set-of-seq-code* [*code*, *set-base-code*]:
set-of-seq *seq.Empty* = *Set-Monad* []
set-of-seq (*seq.Insert* *x* *P*) = *insert-monad* *x* (*set-of-pred* *P*)
set-of-seq (*seq.Join* *P* *xq*) = *union-monad* (*set-of-pred* *P*) (*set-of-seq* *xq*)
by(*simp-all* *add*: *of-seq-code*)

hide-const (**open**) *insert-monad* *union-monad*

3.12.5 Type class instantiations

datatype *set-impl* = *Set-IMPL*

declare

set-impl.eq.simps [*code del*]
set-impl.size [*code del*]
set-impl.rec [*code del*]
set-impl.case [*code del*]

lemma [*code*, *set-base-code*]:
fixes *x* :: *set-impl*
shows *size* *x* = 0
and *size-set-impl* *x* = 0
by(*case-tac* [!] *x*) *simp-all*

definition *set-Choose* :: *set-impl* **where** [*simp*]: *set-Choose* = *Set-IMPL*

definition *set-Collect* :: *set-impl* **where** [*simp*]: *set-Collect* = *Set-IMPL*

definition *set-DList* :: *set-impl* **where** [*simp*]: *set-DList* = *Set-IMPL*

definition *set-RBT* :: *set-impl* **where** [*simp*]: *set-RBT* = *Set-IMPL*

definition *set-Monad* :: *set-impl* **where** [*simp*]: *set-Monad* = *Set-IMPL*

code-datatype *set-Choose set-Collect set-DList set-RBT set-Monad*

definition *set-empty-choose* :: 'a *set* **where** [*simp*]: *set-empty-choose* = {}

lemma *set-empty-choose-code* [*code*, *set-base-code*]:

(*set-empty-choose* :: 'a :: {*ceq*, *compare*} *set*) =
 (case *CCOMPARE*('a) of *Some* - \Rightarrow *RBT-set RBT-Set2.empty*
 | *None* \Rightarrow case *CEQ*('a) of *None* \Rightarrow *Set-Monad []* | *Some* - \Rightarrow *DList-set*
 (*DList-Set.empty*))
by(*simp split: option.split*)

definition *set-impl-choose2* :: *set-impl* \Rightarrow *set-impl* \Rightarrow *set-impl*

where [*simp*]: *set-impl-choose2* = (λ - . *Set-IMPL*)

lemma *set-impl-choose2-code* [*code*, *set-base-code*]:

set-impl-choose2 set-Monad set-Monad = *set-Monad*
set-impl-choose2 set-RBT set-RBT = *set-RBT*
set-impl-choose2 set-DList set-DList = *set-DList*
set-impl-choose2 set-Collect set-Collect = *set-Collect*
set-impl-choose2 x y = *set-Choose*
by *simp-all*

definition *set-empty* :: *set-impl* \Rightarrow 'a *set*

where [*simp*]: *set-empty* = (λ - . {})

lemma *set-empty-code* [*code*, *set-base-code*]:

set-empty set-Collect = *Collect-set* (λ - . *False*)
set-empty set-DList = *DList-set DList-Set.empty*
set-empty set-RBT = *RBT-set RBT-Set2.empty*
set-empty set-Monad = *Set-Monad []*
set-empty set-Choose = *set-empty-choose*
by(*simp-all*)

class *set-impl* =

fixes *set-impl* :: ('a, *set-impl*) *phantom*

syntax (*input*)

-SET-IMPL :: *type* \Rightarrow *logic* (\langle (*1SET'-IMPL*/(*1'(-)*)) \rangle)

syntax-consts

-SET-IMPL == *set-impl*

parse-translation \langle

```

let
  fun set-impl-tr [ty] =
    (Syntax.const @ {syntax-const -constrain} $ Syntax.const @ {const-syntax set-impl}
 $
    (Syntax.const @ {type-syntax phantom} $ ty $ Syntax.const @ {type-syntax
set-impl}))
  | set-impl-tr ts = raise TERM (set-impl-tr, ts);
in [(@ {syntax-const -SET-IMPL}, K set-impl-tr)] end
>

```

```

lemma empty-code [code, set-base-code, code-unfold]:
  ({} :: 'a :: set-impl set) = set-empty (of-phantom SET-IMPL('a))
by simp

```

3.12.6 Generator for the *set-impl*-class

This generator registers itself at the derive-manager for the classes *set-impl*. Here, one can choose the desired implementation via the parameter.

- instantiation type :: (type, ..., type) (rbt, dlist, collect, monad, choose, or arbitrary constant name) set-impl

This generator can be used for arbitrary types, not just datatypes.

ML-file <set-impl-generator.ML>

```

derive (dlist) set-impl unit bool
derive (rbt) set-impl nat
derive (set-RBT) set-impl int
derive (dlist) set-impl Enum.finite-1 Enum.finite-2 Enum.finite-3
derive (rbt) set-impl integer natural
derive (rbt) set-impl char

instantiation sum :: (set-impl, set-impl) set-impl begin
definition SET-IMPL('a + 'b) = Phantom('a + 'b)
  (set-impl-choose2 (of-phantom SET-IMPL('a)) (of-phantom SET-IMPL('b)))
instance ..
end

instantiation prod :: (set-impl, set-impl) set-impl begin
definition SET-IMPL('a * 'b) = Phantom('a * 'b)
  (set-impl-choose2 (of-phantom SET-IMPL('a)) (of-phantom SET-IMPL('b)))
instance ..
end

derive (choose) set-impl list
derive (rbt) set-impl String.literal

instantiation option :: (set-impl) set-impl begin

```

```

definition SET-IMPL('a option) = Phantom('a option) (of-phantom SET-IMPL('a))
instance ..
end

```

```

derive (monad) set-impl fun
derive (choose) set-impl set

```

```

instantiation phantom :: (type, set-impl) set-impl begin
definition SET-IMPL(('a, 'b) phantom) = Phantom (('a, 'b) phantom) (of-phantom
SET-IMPL('b))
instance ..
end

```

We enable automatic implementation selection for sets constructed by *set*, although they could be directly converted using *Set-Monad* in constant time. However, then it is more likely that the parameters of binary operators have different implementations, which can lead to less efficient execution.

However, we test whether automatic selection picks *Set-Monad* anyway and take a short-cut.

```

definition set-aux :: set-impl  $\Rightarrow$  'a list  $\Rightarrow$  'a set
where [simp]: set-aux - = set

```

```

lemma set-aux-code [code, set-base-code]:
  defines conv  $\equiv$  foldl ( $\lambda s (x :: 'a). \text{insert } x s$ )
  shows
    set-aux set-Monad = Set-Monad
    set-aux set-Choose =
      (case CCOMPARE('a :: {compare, ceq}) of Some -  $\Rightarrow$  conv (RBT-set
RBT-Set2.empty)
      | None  $\Rightarrow$  case CEQ('a) of None  $\Rightarrow$  Set-Monad
      | Some -  $\Rightarrow$  conv (DList-set DList-Set.empty)) (is ?thesis2)
    set-aux impl = conv (set-empty impl) (is ?thesis1)

```

```

proof -
  have conv {} = set
  by(rule ext)(induct-tac x rule: rev-induct, simp-all add: conv-def)
  thus ?thesis1 ?thesis2
  by(simp-all split: option.split)
qed simp

```

```

lemma set-code [code, set-base-code]:
  fixes xs :: 'a :: set-impl list
  shows set xs = set-aux (of-phantom (ID SET-IMPL('a))) xs
by(simp)

```

3.12.7 Pretty printing for sets

code-post marks contexts (as hypothesis) in which we use *code_post* as a decision procedure rather than a pretty-printing engine. The intended use

is to enable more rules when proving assumptions of rewrite rules.

definition `code-post` :: `bool` **where** `code-post` = `True`

lemma `conj-code-post` [`code-post`]:
assumes `code-post`
shows `True & x <=> x` `False & x <=> False`
by `simp-all`

A flag to switch post-processing of sets on and off. Use `declare pretty_sets[code_post del]` to disable pretty printing of sets in value.

definition `code-post-set` :: `bool`
where `pretty_sets` [`code-post`, `simp`]: `code-post-set` = `True`

definition `collapse-RBT-set` :: `'a set-rbt` \Rightarrow `'a` :: `ccompare set` \Rightarrow `'a set`
where `collapse-RBT-set` `r` `M` = `set (RBT-Set2.keys r) \cup M`

lemma `RBT-set-collapse-RBT-set` [`code-post`]:
fixes `r` :: `'a` :: `ccompare set-rbt`
assumes `code-post` \Longrightarrow `is-ccompare TYPE('a)` **and** `code-post-set`
shows `RBT-set r` = `collapse-RBT-set r {}`
using `assms`
by(`clarsimp simp add: code-post-def is-ccompare-def RBT-set-def member-conv-keys collapse-RBT-set-def`)

lemma `collapse-RBT-set-Branch` [`code-post`]:
`collapse-RBT-set (Mapping-RBT (Branch c l x v r)) M` =
`collapse-RBT-set (Mapping-RBT l) (insert x (collapse-RBT-set (Mapping-RBT r) M))`
unfolding `collapse-RBT-set-def`
by(`auto simp add: is-ccompare-def set-keys-Mapping-RBT`)

lemma `collapse-RBT-set-Empty` [`code-post`]:
`collapse-RBT-set (Mapping-RBT rbt.Empty) M` = `M`
by(`auto simp add: collapse-RBT-set-def set-keys-Mapping-RBT`)

definition `collapse-DList-set` :: `'a` :: `ceq set-dlist` \Rightarrow `'a set`
where `collapse-DList-set` `dxs` = `set (DList-Set.list-of-dlist dxs)`

lemma `DList-set-collapse-DList-set` [`code-post`]:
fixes `dxs` :: `'a` :: `ceq set-dlist`
assumes `code-post` \Longrightarrow `is-ceq TYPE('a)` **and** `code-post-set`
shows `DList-set dxs` = `collapse-DList-set dxs`
using `assms`
by(`clarsimp simp add: code-post-def DList-set-def is-ceq-def collapse-DList-set-def Collect-member`)

lemma `collapse-DList-set-empty` [`code-post`]: `collapse-DList-set (Abs-dlist [])` = `{}`
by(`simp add: collapse-DList-set-def Abs-dlist-inverse`)

```

lemma collapse-DList-set-Cons [code-post]:
  collapse-DList-set (Abs-dlist (x # xs)) = insert x (collapse-DList-set (Abs-dlist
xs))
by (simp add: collapse-DList-set-def set-list-of-dlist-Abs-dlist)

```

```

lemma Set-Monad-code-post [code-post]:
  assumes code-post-set
  shows Set-Monad [] = {}
  and Set-Monad (x#xs) = insert x (Set-Monad xs)
by simp-all

```

```

declare [[code drop:
  <Gcd :: - => nat>
  <Lcm :: - => nat>
  <Gcd :: - => int>
  <Lcm :: - => int>
  Gcd-fin
  Lcm-fin
  <Inf :: - => 'a Predicate.pred>
  <Sup :: - => 'a Predicate.pred>
  ]]

```

end

```

theory Mapping-Impl imports
  RBT-Mapping2
  AssocList
  HOL-Library.Mapping
  Set-Impl
  Containers-Generator
begin

```

3.13 Different implementations of maps

3.13.1 Map implementations

```

definition Assoc-List-Mapping :: ('a, 'b) alist => ('a, 'b) mapping
where [simp]: Assoc-List-Mapping al = Mapping.Mapping (DAList.lookup al)

```

```

definition RBT-Mapping :: ('a :: compare, 'b) mapping-rbt => ('a, 'b) mapping
where [simp]: RBT-Mapping t = Mapping.Mapping (RBT-Mapping2.lookup t)

```

```

code-datatype Assoc-List-Mapping RBT-Mapping Mapping

```

3.13.2 Map operations

```

lemma lookup-Mapping-code [code, no-atp]:
  Mapping.lookup (RBT-Mapping t) = RBT-Mapping2.lookup t

```

$Mapping.lookup (Assoc-List-Mapping\ al) = DAList.lookup\ al$
by(simp-all)(transfer, rule)+

lemma *is-empty-transfer* [transfer-rule]:

includes *lifting-syntax*
shows (pcr-mapping (=) (=) ==> (=)) ($\lambda m. m = Map.empty$) *Mapping.is-empty*
unfolding *mapping.pcr-cr-eq*
apply(rule rel-funI)
apply(case-tac y)
apply(simp add: *Mapping.is-empty-def cr-mapping-def Mapping-inverse Mapping.keys.rep-eq*)
done

lemma *is-empty-Mapping* [code]:

fixes $t :: ('a :: ccompare, 'b) mapping-rbt$ **shows**
 $Mapping.is-empty (RBT-Mapping\ t) \longleftrightarrow$
(case ID CCOMPARE('a) of None \Rightarrow Code.abort (STR "is-empty RBT-Mapping:
ccompare = None") ($\lambda-. Mapping.is-empty (RBT-Mapping\ t)$)
| Some - $\Rightarrow RBT-Mapping2.is-empty\ t$)
 $Mapping.is-empty (Assoc-List-Mapping\ al) \longleftrightarrow al = DAList.empty$
apply(simp-all split: option.split)
apply(transfer, case-tac al, simp-all)
apply(transfer, simp)
done

lemma *update-Mapping* [code]:

fixes $t :: ('a :: ccompare, 'b) mapping-rbt$ **shows**
 $Mapping.update\ k\ v (RBT-Mapping\ t) =$
(case ID CCOMPARE('a) of None \Rightarrow Code.abort (STR "update RBT-Mapping:
ccompare = None") ($\lambda-. Mapping.update\ k\ v (RBT-Mapping\ t)$)
| Some - $\Rightarrow RBT-Mapping (RBT-Mapping2.insert\ k\ v\ t)$) (is
?RBT)
 $Mapping.update\ k\ v (Assoc-List-Mapping\ al) = Assoc-List-Mapping (DAList.update\ k\ v\ al)$
 $Mapping.update\ k\ v (Mapping\ m) = Mapping (m(k \mapsto v))$
by(simp-all split: option.split)(transfer, simp)+

lemma *delete-Mapping* [code]:

fixes $t :: ('a :: ccompare, 'b) mapping-rbt$ **shows**
 $Mapping.delete\ k (RBT-Mapping\ t) =$
(case ID CCOMPARE('a) of None \Rightarrow Code.abort (STR "delete RBT-Mapping:
ccompare = None") ($\lambda-. Mapping.delete\ k (RBT-Mapping\ t)$)
| Some - $\Rightarrow RBT-Mapping (RBT-Mapping2.delete\ k\ t)$)
 $Mapping.delete\ k (Assoc-List-Mapping\ al) = Assoc-List-Mapping (AssocList.delete\ k\ al)$
 $Mapping.delete\ k (Mapping\ m) = Mapping (m(k := None))$
by(simp-all split: option.split)(transfer, simp)+

theorem *rbt-comp-lookup-map-const*: $rbt-comp-lookup\ c (RBT-Impl.map (\lambda-. f)\ t)$
 $= map-option\ f \circ rbt-comp-lookup\ c\ t$

by(*induct t*)(*auto simp: fun-eq-iff split: order.split*)

lemma *keys-Mapping* [*code*]:

fixes *t* :: ('a :: *ccompare*, 'b) *mapping-rbt* **shows**

Mapping.keys (RBT-Mapping t) = RBT-set (RBT-Mapping2.map (λ-. .) t)
(is ?RBT)

Mapping.keys (Assoc-List-Mapping al) = AssocList.keys al **(is ?Assoc-List)**

Mapping.keys (Mapping m) = Collect (λk. m k ≠ None) **(is ?Mapping)**

proof –

show *?Mapping* **by** *transfer auto*

show *?Assoc-List* **by** *simp(transfer, auto intro: rev-image-eqI)*

show *?RBT*

by(*simp add: RBT-set-def, transfer, auto simp add: rbt-comp-lookup-map-const o-def*)

qed

lemma *Mapping-size-transfer* [*transfer-rule*]:

includes *lifting-syntax*

shows (*pcr-mapping* (=) (=) \implies (=)) (*card* ∘ *dom*) *Mapping.size*

apply(*rule rel-funI*)

apply(*case-tac y*)

apply(*simp add: Mapping.size-def Mapping.keys.rep-eq Mapping-inverse mapping.pcr-cr-eq cr-mapping-def*)

done

lemma *size-Mapping* [*code*]:

fixes *t* :: ('a :: *ccompare*, 'b) *mapping-rbt* **shows**

Mapping.size (RBT-Mapping t) =

(*case ID CCOMPARE('a) of None* \implies *Code.abort (STR "size RBT-Mapping: ccompare = None")* (λ-. *Mapping.size (RBT-Mapping t)*)
| *Some -* \implies *length (RBT-Mapping2.entries t)*)

Mapping.size (Assoc-List-Mapping al) = size al

subgoal

apply (*simp split: option.split*)

apply *transfer*

apply (*clarsimp simp add: size-eq-card-dom-lookup*)

apply (*simp add: linorder.rbt-lookup-keys [OF ID-ccompare] ord.is-rbt-rbt-sorted RBT-Impl.keys-def distinct-card linorder.distinct-entries [OF ID-ccompare] flip: set-map*)

done

subgoal

apply *simp*

apply *transfer*

apply (*simp add: dom-map-of-conv-image-fst distinct-card flip: set-map*)

done

done

declare *tabulate-fold* [*code*]

declare *ordered-keys-def* [*code*]

declare *Mapping.lookup-default-def* [code]

lemma *filter-code* [code]:

fixes $t :: ('a :: ccompare, 'b) \text{ mapping-rbt}$ **shows**
 $\text{Mapping.filter } P \text{ (RBT-Mapping } t) =$
 $(\text{case ID CCOMPARE('a) of None} \Rightarrow \text{Code.abort (STR "filter RBT-Mapping:}$
 $\text{ccompare = None") } (\lambda-. \text{Mapping.filter } P \text{ (RBT-Mapping } t))$
 $\quad | \text{Some } - \Rightarrow \text{RBT-Mapping (RBT-Mapping2.filter } (\lambda(k, v). P$
 $k \ v) \ t))$
 $\text{Mapping.filter } P \text{ (Assoc-List-Mapping } al) = \text{Assoc-List-Mapping (DAList.filter}$
 $(\lambda(k, v). P \ k \ v) \ al)$
 $\text{Mapping.filter } P \text{ (Mapping } m) = \text{Mapping } (\lambda k. \text{case } m \ k \ \text{of None} \Rightarrow \text{None} \ | \ \text{Some}$
 $v \Rightarrow \text{if } P \ k \ v \ \text{then Some } v \ \text{else None})$
subgoal by (*clarsimp simp add: Mapping-inject Mapping.filter.abs-eq fun-eq-iff*
split: option.split)
subgoal by (*simp, transfer*)(*simp add: map-of-filter-apply fun-eq-iff cong: if-cong*
option.case-cong)
subgoal by *transfer simp*
done

lemma *map-values-code* [code]:

fixes $t :: ('a :: ccompare, 'b) \text{ mapping-rbt}$ **shows**
 $\text{Mapping.map-values } f \text{ (RBT-Mapping } t) =$
 $(\text{case ID CCOMPARE('a) of None} \Rightarrow \text{Code.abort (STR "map-values RBT-Mapping:}$
 $\text{ccompare = None") } (\lambda-. \text{Mapping.map-values } f \text{ (RBT-Mapping } t))$
 $\quad | \text{Some } - \Rightarrow \text{RBT-Mapping (RBT-Mapping2.map } f \ t))$
 $\text{Mapping.map-values } f \text{ (Assoc-List-Mapping } al) = \text{Assoc-List-Mapping (AssocList.map-values}$
 $f \ al)$
 $\text{Mapping.map-values } f \text{ (Mapping } m) = \text{Mapping } (\lambda k. \text{map-option } (f \ k) \ (m \ k))$
subgoal by (*clarsimp simp add: Mapping-inject Mapping.map-values.abs-eq fun-eq-iff*
split: option.split)
subgoal by (*simp, transfer*) (*simp add: fun-eq-iff map-of-map'*)
subgoal by *transfer simp*
done

lemma *bulkload-code* [code]:

$\text{Mapping.bulkload } vs = \text{RBT-Mapping (RBT-Mapping2.bulkload (zip-with-index}$
 $vs))$
by (*simp add: Mapping.bulkload.abs-eq Mapping-inject ccompare-nat-def ID-def*
fun-eq-iff)

datatype *mapping-impl* = *Mapping-IMPL*

lemma [code]:

fixes $x :: \text{mapping-impl}$
shows $\text{size } x = 0$
and $\text{size-mapping-impl } x = 0$
subgoal by (*cases x*) *simp*

subgoal by (*cases x*) *simp*
done

definition *mapping-Choose* :: *mapping-impl*
where [*simp*]: *mapping-Choose* = *Mapping-IMPL*

definition *mapping-Assoc-List* :: *mapping-impl*
where [*simp*]: *mapping-Assoc-List* = *Mapping-IMPL*

definition *mapping-RBT* :: *mapping-impl*
where [*simp*]: *mapping-RBT* = *Mapping-IMPL*

definition *mapping-Mapping* :: *mapping-impl*
where [*simp*]: *mapping-Mapping* = *Mapping-IMPL*

code-datatype *mapping-Choose mapping-Assoc-List mapping-RBT mapping-Mapping*

definition *mapping-empty-choose* :: ('a, 'b) *mapping*
where [*simp*]: *mapping-empty-choose* = *Mapping.empty*

lemma *mapping-empty-choose-code* [*code*]:
(*mapping-empty-choose* :: ('a :: *ccompare*, 'b) *mapping*) =
(*case ID CCOMPARE*('a) of *Some* - \Rightarrow *RBT-Mapping RBT-Mapping2.empty*
| *None* \Rightarrow *Assoc-List-Mapping DAList.empty*)
by (*auto split: option.split simp add: DAList.lookup-empty [abs-def] Mapping.empty-def*)

definition *mapping-impl-choose2* :: *mapping-impl* \Rightarrow *mapping-impl* \Rightarrow *mapping-impl*
where [*simp*]: *mapping-impl-choose2* = (λ - . *Mapping-IMPL*)

lemma *mapping-impl-choose2-code* [*code*]:
mapping-impl-choose2 mapping-RBT mapping-RBT = *mapping-RBT*
mapping-impl-choose2 mapping-Assoc-List mapping-Assoc-List = *mapping-Assoc-List*
mapping-impl-choose2 mapping-Mapping mapping-Mapping = *mapping-Mapping*
mapping-impl-choose2 x y = *mapping-Choose*
by *simp-all*

definition *mapping-empty* :: ('a, 'b) *mapping*
where [*simp*]: *mapping-empty* = (λ - . *Mapping.empty*)

lemma *mapping-empty-code* [*code*]:
mapping-empty mapping-RBT = *RBT-Mapping RBT-Mapping2.empty*
mapping-empty mapping-Assoc-List = *Assoc-List-Mapping DAList.empty*
mapping-empty mapping-Mapping = *Mapping* (λ - . *None*)
mapping-empty mapping-Choose = *mapping-empty-choose*
by (*simp-all add: Mapping.empty-def DAList.lookup-empty [abs-def]*)

3.13.3 Type classes

class *mapping-impl* =

```

fixes mapping-impl :: ('a, mapping-impl) phantom

syntax (input)
  -MAPPING-IMPL :: type => logic (⟨(1MAPPING'-IMPL/(1'(-'))⟩)⟩)

syntax-consts
  -MAPPING-IMPL == mapping-impl

parse-translation ⟨
  let
    fun mapping-impl-tr [ty] =
      (Syntax.const @{syntax-const -constrain} $ Syntax.const @{const-syntax mapping-impl}
  $
    (Syntax.const @{type-syntax phantom} $ ty $ Syntax.const @{type-syntax
mapping-impl}))
    | mapping-impl-tr ts = raise TERM (mapping-impl-tr, ts);
  in [(@{syntax-const -MAPPING-IMPL}, K mapping-impl-tr)] end
  ⟩

lemma Mapping-empty-code [code, code-unfold]:
  (Mapping.empty :: ('a :: mapping-impl, 'b) mapping) =
    mapping-empty (of-phantom MAPPING-IMPL('a))
by simp

```

3.13.4 Generator for the *mapping-impl-class*

This generator registers itself at the derive-manager for the classes *mapping-impl*. Here, one can choose the desired implementation via the parameter.

- instantiation type :: (type,...,type) (rbt,assoclist,mapping,choose, or arbitrary constant name) *mapping-impl*

This generator can be used for arbitrary types, not just datatypes.

ML-file ⟨*mapping-impl-generator.ML*⟩

```

derive (assoclist) mapping-impl unit bool
derive (rbt) mapping-impl nat
derive (mapping-RBT) mapping-impl int
derive (assoclist) mapping-impl Enum.finite-1 Enum.finite-2 Enum.finite-3
derive (rbt) mapping-impl integer natural
derive (rbt) mapping-impl char

instantiation sum :: (mapping-impl, mapping-impl) mapping-impl
begin

definition MAPPING-IMPL('a + 'b) = Phantom('a + 'b)

```

(mapping-impl-choose2 (of-phantom MAPPING-IMPL('a)) (of-phantom MAPPING-IMPL('b)))

instance ..

end

instantiation *prod* :: (*mapping-impl*, *mapping-impl*) *mapping-impl* **begin**

definition *MAPPING-IMPL('a * 'b)* = *Phantom('a * 'b)*
(mapping-impl-choose2 (of-phantom MAPPING-IMPL('a)) (of-phantom MAPPING-IMPL('b)))

instance ..

end

derive (*choose*) *mapping-impl list*
derive (*rbt*) *mapping-impl String.literal*

instantiation *option* :: (*mapping-impl*) *mapping-impl*
begin

definition *MAPPING-IMPL('a option)* = *Phantom('a option)* (*of-phantom MAPPING-IMPL('a)*)

instance ..

end

derive (*choose*) *mapping-impl set*

instantiation *phantom* :: (*type*, *mapping-impl*) *mapping-impl*
begin

definition *MAPPING-IMPL(('a, 'b) phantom)* = *Phantom (('a, 'b) phantom)*
(of-phantom MAPPING-IMPL('b))

instance ..

end

declare [[*code drop*:
rec-mapping-impl
case-mapping-impl
<HOL.equal :: mapping-impl => ->
Mapping.combine-with-key
Mapping.combine
]]

```

code-identifier
  code-module Mapping  $\rightarrow$  (SML) Mapping-Impl
| code-module Mapping-Impl  $\rightarrow$  (SML) Mapping-Impl

end

```

```

theory Map-To-Mapping imports
  Mapping-Impl
begin

```

3.14 Infrastructure for operation identification

To convert theorems from *'a* \Rightarrow *'b* *option* to *('a, 'b)* *mapping* using lifting / transfer, we first introduce constants for the empty map and map lookup, then apply lifting / transfer, and finally eliminate the non-converted constants again.

Dynamic theorem list of rewrite rules that are applied before `Transfer.transferred`

```

ML <
  structure Containers-Pre = Named-Thms
  (
    val name = @{binding containers-pre}
    val description = Preprocessing rewrite rules in operation identification for Containers
  )
  >
setup <Containers-Pre.setup>

```

Dynamic theorem list of rewrite rules that are applied after `Transfer.transferred`

```

ML <
  structure Containers-Post = Named-Thms
  (
    val name = @{binding containers-post}
    val description = Postprocessing rewrite rules in operation identification for Containers
  )
  >
setup <Containers-Post.setup>

```

```

context includes lifting-syntax
begin

```

```

definition map-empty :: 'a  $\Rightarrow$  'b option

```

```

where [code-unfold]: map-empty = Map.empty

declare map-empty-def[containers-post, symmetric, containers-pre]

declare Mapping.empty.transfer[transfer-rule del]

lemma map-empty-transfer [transfer-rule]:
  (pcr-mapping A B) map-empty Mapping.empty
unfolding map-empty-def by(rule Mapping.empty.transfer)

```

```

definition map-apply :: ('a ⇒ 'b option) ⇒ 'a ⇒ 'b option
where [code-unfold]: map-apply = (λm. m)

```

```

lemma eq-map-apply: m x ≡ map-apply m x
by(simp add: map-apply-def)

```

```

declare eq-map-apply[symmetric, abs-def, containers-post]

```

We cannot use `eq-map-apply` as a fold rule for operator identification, because it would loop. We use a `simproc` instead.

```

simproc-setup passive map-apply (f x :: 'a option) = ⟨
  fn - => fn ctxt => fn ct =>
  (case Thm.term-of ct of
    Const (@{const-name map-apply}, -) $ - $ - => NONE
  | f $ x =>
    let
      val cTr =
        Thm.typ-of-cterm ct
        |> dest-Type
        |> snd |> hd
        |> Thm.ctyp-of ctxt;
      val cTx = Thm.ctyp-of ctxt (fastype-of x);
      val cts = map (SOME o Thm.cterm-of ctxt) [f, x];
    in
      SOME (Thm.instantiate' [SOME cTr, SOME cTx] cts @ {thm eq-map-apply})
    end
  | - => NONE)
⟩

```

```

lemma map-apply-parametric [transfer-rule]:
  ((A ==> B) ==> A ==> B) map-apply map-apply
unfolding map-apply-def by(transfer-prover)

```

```

lemma map-apply-transfer [transfer-rule]:
  (pcr-mapping A B ==> A ==> rel-option B) map-apply Mapping.lookup
by(auto simp add: pcr-mapping-def cr-mapping-def Mapping.lookup-def map-apply-def
  dest: rel-funD)

```

definition $map\text{-}update :: 'a \Rightarrow 'b\ option \Rightarrow ('a \Rightarrow 'b\ option) \Rightarrow ('a \Rightarrow 'b\ option)$
where $map\text{-}update\ x\ y\ f = f(x := y)$

lemma $map\text{-}update\text{-}parametric$ [*transfer-rule*]:
assumes [*transfer-rule*]: $bi\text{-}unique\ A$
shows $(A ==> rel\text{-}option\ B ==> (A ==> rel\text{-}option\ B) ==> (A ==> rel\text{-}option\ B))\ map\text{-}update\ map\text{-}update$
unfolding $map\text{-}update\text{-}def$ [*abs-def*] **by** $transfer\text{-}prover$

context begin

local-setup $\langle Local\text{-}Theory.map\text{-}background\text{-}naming\ (Name\text{-}Space.mandatory\text{-}path\ Mapping) \rangle$

lift-definition $update' :: 'a \Rightarrow 'b\ option \Rightarrow ('a, 'b)\ mapping \Rightarrow ('a, 'b)\ mapping$
is $map\text{-}update$ **parametric** $map\text{-}update\text{-}parametric$.

lemma $update'\text{-}code$ [*simp, code, code-unfold*]:
 $update'\ x\ None = Mapping.delete\ x$
 $update'\ x\ (Some\ y) = Mapping.update\ x\ y$
by($transfer,$ $simp\ add: map\text{-}update\text{-}def\ fun\text{-}eq\text{-}iff$) $+$

end

declare $map\text{-}update\text{-}def$ [*abs-def, containers-post*] $map\text{-}update\text{-}def$ [*symmetric, containers-pre*]

definition $map\text{-}is\text{-}empty :: ('a \Rightarrow 'b\ option) \Rightarrow bool$
where $map\text{-}is\text{-}empty\ m \longleftrightarrow m = Map.empty$

lemma $map\text{-}is\text{-}empty\text{-}folds$:
 $m = map\text{-}empty \longleftrightarrow map\text{-}is\text{-}empty\ m$
 $map\text{-}empty = m \longleftrightarrow map\text{-}is\text{-}empty\ m$
by($auto\ simp\ add: map\text{-}is\text{-}empty\text{-}def\ map\text{-}empty\text{-}def$)

declare $map\text{-}is\text{-}empty\text{-}folds$ [*containers-pre*]
 $map\text{-}is\text{-}empty\text{-}def$ [*abs-def, containers-post*]

lemma $map\text{-}is\text{-}empty\text{-}transfer$ [*transfer-rule*]:
assumes $bi\text{-}total\ A$
shows $(pcr\text{-}mapping\ A\ B ==> (=))\ map\text{-}is\text{-}empty\ Mapping.is\text{-}empty$
unfolding $map\text{-}is\text{-}empty\text{-}def$ [*abs-def*] $Mapping.is\text{-}empty\text{-}def$ [*abs-def*] $dom\text{-}eq\text{-}empty\text{-}conv$ [*symmetric*]
by($rule\ rel\text{-}funI$) $+$ ($auto\ simp\ del: dom\text{-}eq\text{-}empty\text{-}conv\ dest: rel\text{-}setD2\ rel\text{-}setD1\ Mapping.keys.transfer$ [*THEN rel-funD, OF assms*])

end

ML \langle

```

signature CONTAINERS = sig
  val identify : Context.generic -> thm -> thm;
  val identify-attribute : attribute;
end

structure Containers: CONTAINERS =
struct

fun identify context thm =
  let
    val ctxt' = Context.proof-of context
    val ss = put-simpset HOL-basic-ss ctxt'
    val ctxt1 = ss |> Simplifier.add-simps (Containers-Pre.get ctxt')
      |> Simplifier.add-proc simproc <map-apply>
    val ctxt2 = ss |> Simplifier.add-simps (Containers-Post.get ctxt')

    (* Hack to recover Transfer.transferred function from attribute *)
    fun transfer-transferred thm = Transfer.transferred-attribute [] (context, thm)
  |> snd |> the
  in
    thm
    |> full-simplify ctxt1
    |> transfer-transferred
    |> full-simplify ctxt2
  end

val identify-attribute = Thm.rule-attribute [] identify

end
>

attribute-setup containers-identify =
  <Scan.succeed Containers.identify-attribute>
  Transfer theorems for operator identification in Containers

hide-const (open) map-apply map-empty map-is-empty map-update
hide-fact (open) map-apply-def map-empty-def eq-map-apply

end

theory Containers imports
  Set-Linorder
  Collection-Order
  Collection-Eq
  Collection-Enum
  Equal
  Mapping-Impl
  Map-To-Mapping

```

begin

end

3.15 Compatibility with Regular-Sets

theory *Compatibility-Containers-Regular-Sets* **imports**

Containers

Regular-Sets.Regexp-Method

begin

Adaptation theory to make *regexp* work when *Containers.Containers* are loaded.

Warning: Each invocation of *regexp* takes longer than without *Containers.Containers* because the code generator takes longer to generate the evaluation code for *regexp*.

datatype-compat *regexp*

derive *ceq regexp*

derive *compare regexp*

derive (*choose*) *set-impl regexp*

notepad begin

fix $r\ s :: ('a \times 'a)\ set$

have $(r \cup s^+)^* = (r \cup s)^*$ **by** *regexp*

end

end

Chapter 4

User guide

This user guide shows how to use and extend the lightweight containers framework (LC). For a more theoretical discussion, see [5]. This user guide assumes that you are familiar with refinement in the code generator [1, 2]. The theory *Containers-Userguide* generates it; so if you want to experiment with the examples, you can find their source code there. Further examples can be found in the `Examples` folder.

4.1 Characteristics

- **Separate type classes for code generation**

LC follows the ideal that type classes for code generation should be separate from the standard type classes in Isabelle. LC's type classes are designed such that every type can become an instance, so well-sortedness errors during code generation can always be remedied.

- **Multiple implementations**

LC supports multiple simultaneous implementations of the same container type. For example, the following implements at the same time (i) the set of *bool* as a distinct list of the elements, (ii) *int set* as a RBT of the elements or as the RBT of the complement, and (iii) sets of functions as monad-style lists:

```
value ({True}, {1 :: int}, - {2 :: int, 3}, {λx :: int. x * x, λy. y + 1})
```

The LC type classes are the key to simultaneously supporting different implementations.

- **Extensibility**

The LC framework is designed for being extensible. You can add new containers, implementations and element types any time.

4.2 Getting started

Add the entry theory *Containers.Containers* for LC to the end of your imports. This will reconfigure the code generator such that it implements the types *'a set* for sets and *('a, 'b) mapping* for maps with one of the data structures supported. As with all the theories that adapt the code generator setup, it is important that *Containers.Containers* comes at the end of the imports.

Note: LC should not be used together with the theory *HOL-Library.Code-Cardinality*. Run the following command, e.g., to check that LC works correctly and implements sets of *ints* as red-black trees (RBT):

```
value [code] {1 :: int}
```

This should produce $\{1\}$. Without LC, sets are represented as (complements of) a list of elements, i.e., *set [1]* in the example.

If your exported code does not use your own types as elements of sets or maps and you have not declared any code equation for these containers, then your **export-code** command will use LC to implement *'a set* and *('a, 'b) mapping*.

Our running example will be arithmetic expressions. The function *vars e* computes the variables that occur in the expression *e*

```
type-synonym vname = string
datatype expr = Var vname | Lit int | Add expr expr
fun vars :: expr  $\Rightarrow$  vname set where
  vars (Var v) = {v}
| vars (Lit i) = {}
| vars (Add e1 e2) = vars e1  $\cup$  vars e2
```

```
value vars (Var "x")
```

To illustrate how to deal with type variables, we will use the following variant where variable names are polymorphic:

```
datatype 'a expr' = Var' 'a | Lit' int | Add' 'a expr' 'a expr'
fun vars' :: 'a expr'  $\Rightarrow$  'a set where
  vars' (Var' v) = {v}
| vars' (Lit' i) = {}
| vars' (Add' e1 e2) = vars' e1  $\cup$  vars' e2
```

```
value vars' (Var' (1 :: int))
```

4.3 New types as elements

This section explains LC's type classes and shows how to instantiate them. If you want to use your own types as the elements of sets or the keys of maps, you must instantiate up to eight type classes: *ceq* (§4.3.1), *ccompare* (§4.3.2), *set-impl* (§4.3.3), *mapping-impl* (§4.3.3), *cenum* (§4.3.4), *finite-UNIV* (§4.3.5), *card-UNIV* (§4.3.5), and *cproper-interval* (§4.3.5). Otherwise, well-sortedness errors like the following will occur:

```
*** Wellsortedness error:
*** Type expr not of sort {ceq,ccompare}
*** No type arity expr :: ceq
*** At command "value"
```

In detail, the sort requirements on the element type *'a* are:

- *ceq* (§4.3.1), *ccompare* (§4.3.2), and *set-impl* (§4.3.3) for *'a set* in general
- *cenum* (§4.3.4) for set comprehensions $\{x. P x\}$,
- *card-UNIV*, *cproper-interval* for *'a set set* and any deeper nesting of sets (§4.3.5),¹ and
- *equal*,² *ccompare* (§4.3.2) and *mapping-impl* (§4.3.3) for (*'a*, *'b*) *mapping*.

4.3.1 Equality testing

The type class *ceq* defines the operation $CEQ('a) :: ('a \Rightarrow 'a \Rightarrow bool) option$ for testing whether two elements are equal.³ The test is embedded in an *option* value to allow for types that do not support executable equality test such as $'a \Rightarrow 'b$. Whenever possible, $CEQ('a)$ should provide an executable equality operator. Otherwise, membership tests on such sets will raise an exception at run-time.

¹These type classes are only required for set complements (see §4.7.2).

²We deviate here from the strict separation of type classes, because it does not make sense to store types in a map on which we do not have equality, because the most basic operation *Mapping.lookup* inherently requires equality.

³Technically, the type class *ceq* defines the operation *ceq*. As usage often does not fully determine *ceq*'s type, we use the notation $CEQ('a)$ that explicitly mentions the type. In detail, $CEQ('a)$ is translated to $CEQ('a) :: ('a \Rightarrow 'a \Rightarrow bool) option$ including the type constraint. We do the same for the other type class operators: *ccompare* constrains the operation *ccompare* (§4.3.2), *SET-IMPL('a)* constrains the operation *set-impl*, (§4.3.3), *MAPPING-IMPL('a)* (constrains the operation *mapping-impl*, (§4.3.3), and *CENUM('a)* constrains the operation *cenum*, §4.3.4.

For data types, the *derive* command can automatically instantiate of *ceq*, we only have to tell it whether an equality operation should be provided or not (parameter *no*).

```
derive (eq) ceq expr
```

```
datatype example = Example
derive (no) ceq example
```

In the remainder of this subsection, we look at how to manually instantiate a type for *ceq*. First, the simple case of a type constructor *simple-tycon* without parameters that already is an instance of *equal*:

```
typedecl simple-tycon
axiomatization where simple-tycon-equal: OFCLASS(simple-tycon, equal-class)
instance simple-tycon :: equal by (rule simple-tycon-equal)
```

```
instantiation simple-tycon :: ceq begin
definition CEQ(simple-tycon) = Some (=)
instance by(intro-classes)(simp add: ceq-simple-tycon-def)
end
```

For polymorphic types, this is a bit more involved, as the next example with *'a expr'* illustrates (note that we could have delegated all this to *derive*). First, we need an operation that implements equality tests with respect to a given equality operation on the polymorphic type. For data types, we can use the relator which the transfer package (method *transfer*) requires and the BNF package generates automatically. As we have used the old datatype package for *'a expr'*, we must define it manually:

```
context fixes R :: 'a  $\Rightarrow$  'b  $\Rightarrow$  bool begin
fun expr'-rel :: 'a expr'  $\Rightarrow$  'b expr'  $\Rightarrow$  bool
where
  expr'-rel (Var' v)      (Var' v')       $\longleftrightarrow$  R v v'
| expr'-rel (Lit' i)      (Lit' i')        $\longleftrightarrow$  i = i'
| expr'-rel (Add' e1 e2) (Add' e1' e2')  $\longleftrightarrow$  expr'-rel e1 e1'  $\wedge$  expr'-rel e2 e2'
| expr'-rel -             -               $\longleftrightarrow$  False
end
```

If we give HOL equality as parameter, the relator is equality:

```
lemma expr'-rel-eq: expr'-rel (=) e1 e2  $\longleftrightarrow$  e1 = e2
by(induct e1 e2 rule: expr'-rel.induct) simp-all
```

Then, the instantiation is again canonical:

```
instantiation expr' :: (ceq) ceq begin
```

definition

```
CEQ('a expr') =
  (case ID CEQ('a) of None => None | Some eq => Some (expr'-rel eq))
```

instance

```
by(intro-classes)
  (auto simp add: ceq-expr'-def expr'-rel-eq[abs-def]
    dest: Collection-Eq.ID-ceq
    split: option.split-asm)
```

end

Note the following two points: First, the instantiation should avoid to use $(=)$ on terms of the polymorphic type. This keeps the LC framework separate from the type class *equal*, i.e., every choice of *'a* in *'a expr'* can be of sort *ceq*. The easiest way to achieve this is to obtain the equality test from *CEQ('a)*. Second, we use *ID CEQ('a)* instead of *CEQ('a)*. In proofs, we want that the simplifier uses assumptions like *CEQ('a) = Some ...* for rewriting. However, *CEQ('a)* is a nullary constant, so the simplifier reverses such an equation, i.e., it only rewrites *Some ...* to *CEQ('a)*. Applying the identity function *ID* to *CEQ('a)* avoids this, and the code generator eliminates all occurrences of *ID*. Although *ID = id* by definition, do not use the conventional *id* instead of *ID*, because *id CEQ('a)* immediately simplifies to *CEQ('a)*.

4.3.2 Ordering

LC takes the order for storing elements in search trees from the type class *ccompare* rather than *compare*, because we cannot instantiate *compare* for some types (e.g., *'a set* as (\subseteq) is not linear). Similar to *CEQ('a)* in class *CEQ('b)*, the class *ccompare* specifies an optional comparator *CCOMPARE('a) :: (('a => 'a => order)) option*. If you cannot or do not want to implement a comparator on your type, you can default to *None*. In that case, you will not be able to use your type as elements of sets or as keys in maps implemented by search trees.

If the type is a data type or instantiates *compare* and we wish to use that comparator also for the search tree, instantiation is again canonical: For our data type *expr*, *derive* does everything!

```
derive ccompare expr
```

In general, the pattern for type constructors without parameters looks as follows:

```
axiomatization where simple-tycon-compare: OFCLASS(simple-tycon, com-
  pare-class)
```

```
instance simple-tycon :: compare by (rule simple-tycon-compare)
```

derive (compare) ccompare simple-tycon

For polymorphic types like *'a expr'*, we should not do everything manually: First, we must define a comparator that takes the comparator on the type variable *'a* as a parameter. This is necessary to maintain the separation between Isabelle/HOL's type classes (like *compare*) and LC's. Such a comparator is again easily defined by *derive*.

derive ccompare *expr'*

thm ccompare-*expr'*-def comparator-*expr'*-simps

4.3.3 Heuristics for picking an implementation

Now, we have defined the necessary operations on *expr* and *'a expr'* to store them in a set or use them as the keys in a map. But before we can actually do so, we also have to say which data structure to use. The type classes *set-impl* and *mapping-impl* are used for this.

They define the overloaded operations *SET-IMPL('a) :: ('a, set-impl) phantom* and *MAPPING-IMPL('a) :: ('a, mapping-impl) phantom*, respectively. The phantom type *('a, 'b) phantom* from theory *HOL-Library.Pantom-Type* is isomorphic to *'b*, but formally depends on *'a*. This way, the type class operations meet the requirement that their type contains exactly one type variable. The Haskell and ML compiler will get rid of the extra type constructor again.

For sets, you can choose between *set-Collect* (characteristic function *P* like in $\{x. P x\}$), *set-DList* (distinct list), *set-RBT* (red-black tree), and *set-Monad* (list with duplicates). Additionally, you can define *set-impl* as *set-Choose* which picks the implementation based on the available operations (RBT if *ccompare* provides a linear order, else distinct lists if *CEQ('a)* provides equality testing, and lists with duplicates otherwise). *set-Choose* is the safest choice because it picks only a data structure when the required operations are actually available. If *set-impl* picks a specific implementation, Isabelle does not ensure that all required operations are indeed available.

For maps, the choices are *mapping-Assoc-List* (associative list without duplicates), *mapping-RBT* (red-black tree), and *mapping-Mapping* (closures with function update). Again, there is also the *mapping-Choose* heuristics.

For simple cases, *derive* can be used again (even if the type is not a data type). Consider, e.g., the following instantiations: *expr set* uses RBTs, (*expr*, -) *mapping* and *'a expr' set* use the heuristics, and (*'a expr'*, -) *mapping* uses the same implementation as (*'a*, -) *mapping*.

derive (rbt) set-impl *expr*

derive (choose) mapping-impl *expr*

derive (choose) set-impl *expr'*

More complex cases such as taking the implementation preference of a type parameter must be done manually.

instantiation *expr'* :: (mapping-impl) mapping-impl **begin**

definition

MAPPING-IMPL('a *expr'*) =

Phantom('a *expr'*) (of-phantom MAPPING-IMPL('a))

instance ..

end

To see the effect of the different configurations, consider the following examples where *empty* refers to *Mapping.empty*. For that, we must disable pretty printing for sets as follows:

declare pretty-sets[code-post del]

value [code]	result
$\{\}$:: <i>expr set</i>	<i>RBT-set (Mapping-RBT Empty)</i>
<i>empty</i> :: (<i>expr</i> , <i>unit</i>) <i>mapping</i>	<i>RBT-Mapping (Mapping-RBT Empty)</i>
$\{\}$:: <i>string expr' set</i>	<i>RBT-set (Mapping-RBT Empty)</i>
$\{\}$:: (<i>nat</i> \Rightarrow <i>nat</i>) <i>expr' set</i>	<i>Set-Monad []</i>
$\{\}$:: <i>bool expr' set</i>	<i>RBT-set (Mapping-RBT Empty)</i>
<i>empty</i> :: (<i>bool expr'</i> , <i>unit</i>) <i>mapping</i>	<i>Assoc-List-Mapping (Alist [])</i>

For *expr*, *mapping-Choose* picks RBTs, because *compare* provides a comparison operation for *expr*. For 'a *expr'*, the effect of *set-Choose* is more pronounced: *compare* is not *None*, so neither is *compare*, and *set-Choose* picks RBTs. As *nat* \Rightarrow *nat* neither provides equality tests (*ceq*) nor comparisons (*compare*), neither does (*nat* \Rightarrow *nat*) *expr'*, so we use lists with duplicates. The last two examples show the difference between inheriting a choice and choosing freshly: By default, *bool* prefers distinct (associative) lists over RBTs, because there are just two elements. As *bool expr'* inherits the choice for maps from *bool*, an associative list implements *empty* :: (*bool expr'*, *unit*) *mapping*. For sets, in contrast, *set-impl* discards 'a's preferences and picks RBTs, because there is a comparison operation.

Finally, let's enable pretty-printing for sets again:

declare pretty-sets [code-post]

4.3.4 Set comprehensions

If you use the default code generator setup that comes with Isabelle, set comprehensions $\{x. P x\}$:: 'a *set* are only executable if the type 'a has sort

enum. Internally, Isabelle's code generator transforms set comprehensions into an explicit list of elements which it obtains from the list *enum* of all of *'a*'s elements. Thus, the type must be an instance of *enum*, i.e., finite in particular. For example, $\{c. CHR "A" \leq c \wedge c \leq CHR "D"\}$ evaluates to *set "ABCD"*, the set of the characters A, B, C, and D.

For compatibility, LC also implements such an enumeration strategy, but avoids the finiteness restriction. The type class *cenum* mimicks *enum*, but its single parameter $cEnum :: ('a\ list \times (('a \Rightarrow bool) \Rightarrow bool) \times (('a \Rightarrow bool) \Rightarrow bool))\ option$ combines all of *enum*'s parameters, namely a list of all elements, a universal and an existential quantifier. *option* ensures that every type can be an instance as *CENUM('a)* can always default to *None*.

For types that define *CENUM('a)*, set comprehensions evaluate to a list of their elements. Otherwise, set comprehensions are represented as a closure. This means that if the generated code contains at least one set comprehension, all element types of a set must instantiate *cenum*. Infinite types default to *None*, and enumerations for finite types are canonical, see *Containers.Collection-Enum* for examples.

instantiation *expr* :: *cenum begin*

definition *CENUM(expr) = None*

instance *by(intro-classes)(simp-all add: cEnum-expr-def)*

end

derive (no) *cenum expr'*

derive *compare-order expr*

For example, **value** ($\{b. b = True\}, \{x. compare\ x\ (Lit\ 0) = Lt\}$) yields $(\{True\},\ Collect-set\ -)$

LC keeps complements of such enumerated set comprehensions, i.e., $-\{b. b = True\}$ evaluates to *Complement {True}*. If you want that the complement operation actually computes the elements of the complements, you have to replace the code equations for *uminus* as follows:

declare *Set-uminus-code[code del] Set-uminus-cenum[code]*

Then, $-\{b. b = True\}$ becomes $\{False\}$, but this applies to all complement invocations. For example, *UNIV :: bool set* becomes $\{False, True\}$.

4.3.5 Nested sets

To deal with nested sets such as *expr set set*, the element type must provide three operations from three type classes:

- *finite-UNIV* from theory *HOL-Library.Cardinality* defines the constant *finite-UNIV :: ('a, bool) phantom* which designates whether the

type is finite.

- *card-UNIV* from theory *HOL-Library.Cardinality* defines the constant *card-UNIV* :: ('a, nat) phantom which returns *CARD('a)*, i.e., the number of values in 'a. If 'a is infinite, *CARD('a) = 0*.
- *cproper-interval* from theory *Containers.Collection-Order* defines the function *cproper-interval* :: 'a option \Rightarrow 'a option \Rightarrow bool. If the type 'a is finite and *ccompare* yields a linear order on 'a, then *cproper-interval* *x y* returns whether the open interval between *x* and *y* is non-empty. The bound *None* denotes unboundedness.

Note that the type class *finite-UNIV* must not be confused with the type class *finite*. *finite-UNIV* allows the generated code to examine whether a type is finite whereas *finite* requires that the type in fact is finite.

For datatypes, the theory *Containers.Card-Datatype* defines some machinery to assist in proving that the type is (in)finite and has a given number of elements – see *Examples/Card_Datatype_Ex.thy* for examples. With this, it is easy to instantiate *card-UNIV* for our running examples:

```
lemma inj-expr [simp]: inj Lit    inj Var    inj Add    inj (Add e)
by(simp-all add: fun-eq-iff inj-on-def)
```

```
lemma infinite-UNIV-expr:  $\neg$  finite (UNIV :: expr set)
including card-datatype
```

```
proof –
```

```
  have rangeIt (Lit 0) (Add (Lit 0))  $\subseteq$  UNIV by simp
  from finite-subset[OF this] show ?thesis by auto
```

```
qed
```

```
instantiation expr :: card-UNIV begin
```

```
definition finite-UNIV = Phantom(expr) False
```

```
definition card-UNIV = Phantom(expr) 0
```

```
instance
```

```
  by intro-classes
```

```
  (simp-all add: finite-UNIV-expr-def card-UNIV-expr-def infinite-UNIV-expr)
```

```
end
```

```
lemma inj-expr' [simp]: inj Lit'   inj Var'   inj Add'   inj (Add' e)
by(simp-all add: fun-eq-iff inj-on-def)
```

```
lemma infinite-UNIV-expr':  $\neg$  finite (UNIV :: 'a expr' set)
including card-datatype
```

```
proof –
```

```
  have rangeIt (Lit' 0) (Add' (Lit' 0))  $\subseteq$  UNIV by simp
```

```

from finite-subset[OF this] show ?thesis by auto
qed

```

```

instantiation expr' :: (type) card-UNIV begin
definition finite-UNIV = Phantom('a expr') False
definition card-UNIV = Phantom('a expr') 0
instance
  by intro-classes
  (simp-all add: finite-UNIV-expr'-def card-UNIV-expr'-def infinite-UNIV-expr')
end

```

As *expr* and *'a expr'* are infinite, instantiating *cproper-interval* is trivial, because *cproper-interval* only makes assumptions about its parameters for finite types. Nevertheless, it is important to actually define *cproper-interval*, because the code generator requires a code equation.

```

instantiation expr :: cproper-interval begin
definition cproper-interval-expr :: expr proper-interval
  where cproper-interval-expr - - = undefined
instance by(intro-classes)(simp add: infinite-UNIV-expr)
end

```

```

instantiation expr' :: (ccompare) cproper-interval begin
definition cproper-interval-expr' :: 'a expr' proper-interval
  where cproper-interval-expr' - - = undefined
instance by(intro-classes)(simp add: infinite-UNIV-expr')
end

```

Instantiation of *proper-interval*

To illustrate what to do with finite types, we instantiate *proper-interval* for *expr*. Like *ccompare* relates to *compare*, the class *cproper-interval* has a counterpart *proper-interval* without the finiteness assumption. Here, we first have to gather the simplification rules of the comparator from the *derive* invocation, especially, how the strict order of the comparator, *lt-of-comp*, can be defined.

Since the order on lists is not yet shown to be consistent with the comparators that are used for lists, this part of the userguide is currently not available.

4.4 New implementations for containers

This section explains how to add a new implementation for a container type. If you do so, please consider to add your implementation to this AFP entry.

4.4.1 Model and verify the data structure

First, you of course have to define the data structure and verify that it has the required properties. As our running example, we use a trie to implement $(\prime a, \prime b)$ mapping. A trie is a binary tree whose the nodes store the values, the keys are the paths from the root to the given node. We use lists of *boolans* for the keys where the *boolean* indicates whether we should go to the left or right child.

For brevity, we skip this step and rather assume that the type $\prime v$ *trie-raw* of tries has following operations and properties:

type-synonym *trie-key* = bool list

axiomatization

trie-empty :: $\prime v$ *trie-raw* **and**

trie-update :: *trie-key* $\Rightarrow \prime v \Rightarrow \prime v$ *trie-raw* $\Rightarrow \prime v$ *trie-raw* **and**

trie-lookup :: $\prime v$ *trie-raw* \Rightarrow *trie-key* $\Rightarrow \prime v$ option **and**

trie-keys :: $\prime v$ *trie-raw* \Rightarrow *trie-key* set

where *trie-lookup-empty*: *trie-lookup* *trie-empty* = Map.empty

and *trie-lookup-update*:

trie-lookup (*trie-update* *k v t*) = (*trie-lookup* *t*)(*k* \mapsto *v*)

and *trie-keys-dom-lookup*: *trie-keys* *t* = dom (*trie-lookup* *t*)

This is only a minimal example. A full-fledged implementation has to provide more operations and – for efficiency – should use more than just *boolans* for the keys.

4.4.2 Generalise the data structure

As $(\prime k, \prime v)$ mapping store keys of arbitrary type $\prime k$, not just *trie-key*, we cannot use $\prime v$ *trie-raw* directly. Instead, we must first convert arbitrary types $\prime k$ into *trie-key*. Of course, this is not always possible, but we only have to make sure that we pick tries as implementation only if the types do. This is similar to red-black trees which require an order. Hence, we introduce a type class to convert arbitrary keys into trie keys. We make the conversions optional such that every type can instantiate the type class, just as LC does for *ceq* and *compare*.

type-synonym $\prime a$ cbl = $((\prime a \Rightarrow$ bool list) \times (bool list $\Rightarrow \prime a))$ option

class cbl =

fixes cbl :: $\prime a$ cbl

assumes inj-to-bl: ID cbl = Some (to-bl, from-bl) \Longrightarrow inj to-bl

and to-bl-inverse: ID cbl = Some (to-bl, from-bl) \Longrightarrow from-bl (to-bl a) =

a

begin

abbreviation from-bl **where** from-bl \equiv snd (the (ID cbl))

abbreviation to-bl **where** to-bl \equiv fst (the (ID cbl))
end

It is best to immediately provide the instances for as many types as possible. Here, we only present two examples: *unit* provides conversion functions, $'a \Rightarrow 'b$ does not.

instantiation unit :: cbl **begin**
definition cbl = Some (λ -. [], λ -. ())
instance by(intro-classes)(auto simp add: cbl-unit-def ID-Some intro: injI)
end

instantiation fun :: (type, type) cbl **begin**
definition cbl = (None :: ($'a \Rightarrow 'b$) cbl)
instance by intro-classes(simp-all add: cbl-fun-def ID-None)
end

4.4.3 Hide the invariants of the data structure

Many data structures have invariants on which the operations rely. You must hide such invariants in a **typedef** before connecting to the container, because the code generator cannot handle explicit invariants. The type must be inhabited even if the types of the elements do not provide the required operations. The easiest way is often to ignore all invariants in that case.

In our example, we require that all keys in the trie represent encoded values.

typedef (**overloaded**) ($'k ::$ cbl, $'v$) trie =
 {t :: $'v$ trie-raw.
 trie-keys t \subseteq range (to-bl :: $'k \Rightarrow$ trie-key) \vee ID (cbl :: $'k$ cbl) = None}

proof

show trie-empty \in ?trie

by(simp add: trie-keys-dom-lookup trie-lookup-empty)

qed

Next, transfer the operations to the new type. The transfer package does a good job here.

setup-lifting type-definition-trie — also sets up code generation

lift-definition empty :: ($'k ::$ cbl, $'v$) trie
is trie-empty
by(simp add: trie-keys-empty)

lift-definition lookup :: ($'k ::$ cbl, $'v$) trie \Rightarrow $'k \Rightarrow 'v$ option
is λ t. trie-lookup t o to-bl .

lift-definition `update` :: $'k \Rightarrow 'v \Rightarrow ('k :: \text{cbl}, 'v) \text{trie} \Rightarrow ('k, 'v) \text{trie}$
is `trie-update` \circ `to-bl`
by `(auto simp add: trie-keys-dom-lookup trie-lookup-update)`

lift-definition `keys` :: $('k :: \text{cbl}, 'v) \text{trie} \Rightarrow 'k \text{set}$
is $\lambda t. \text{from-bl } ' \text{ trie-keys } t .$

And now we go for the properties. Note that some properties hold only if the type class operations are actually provided, i.e., `cbl` \neq `None` in our example.

lemma `lookup-empty`: `lookup empty = Map.empty`
by `transfer(simp add: trie-lookup-empty fun-eq-iff)`

context

fixes `t` :: $('k :: \text{cbl}, 'v) \text{trie}$
assumes `ID-cbl`: `ID (cbl :: 'k cbl) \neq None`

begin

lemma `lookup-update`: `lookup (update k v t) = (lookup t)(k \mapsto v)`
using `ID-cbl`
by `transfer(auto simp add: trie-lookup-update fun-eq-iff dest: inj-to-bl[THEN injD])`

lemma `keys-conv-dom-lookup`: `keys t = dom (lookup t)`
using `ID-cbl`
by `transfer(force simp add: trie-keys-dom-lookup to-bl-inverse intro: rev-image-eqI)`

end

4.4.4 Connecting to the container

Connecting to the container (*'a, 'b* mapping in our example) takes three steps:

1. Define a new pseudo-constructor
2. Implement the container operations for the new type
3. Configure the heuristics to automatically pick an implementation
4. Test thoroughly

Thorough testing is particularly important, because Isabelle does not check whether you have implemented all your operations, whether you have configured your heuristics sensibly, nor whether your implementation always terminates.

Define a new pseudo-constructor

Define a function that returns the abstract container view for a data structure value, and declare it as a datatype constructor for code generation with **code-datatype**. Unfortunately, you have to repeat all existing pseudo-constructors, because there is no way to extract the current set of pseudo-constructors from the code generator. We call them pseudo-constructors, because they do not behave like datatype constructors in the logic. For example, ours are neither injective nor disjoint.

definition `Trie-Mapping` :: (`'k` :: `cbl`, `'v`) `trie` \Rightarrow (`'k`, `'v`) `mapping`
where [`simp`, `code del`]: `Trie-Mapping t` = `Mapping.Mapping (lookup t)`

code-datatype `Assoc-List-Mapping` `RBT-Mapping` `Mapping` `Trie-Mapping`

Implement the operations

Next, you have to prove and declare code equations that implement the container operations for the new implementation. Typically, these just dispatch to the operations on the type from §4.4.3. Some operations depend on the type class operations from §4.4.2 being defined; then, the code equation must check that the operations are indeed defined. If not, there is usually no way to implement the operation, so the code should raise an exception. Logically, we use the function `Code.abort` of type `String.literal` \Rightarrow (`unit` \Rightarrow `'a`) \Rightarrow `'a` with definition $\lambda\text{-}f. f ()$, but the generated code raises an exception `Fail` with the given message (the unit closure avoids non-termination in strict languages). This function gets the exception message and the unit-closure of the equation's left-hand side as argument, because it is then trivial to prove equality.

Again, we only show a small set of operations; a realistic implementation should cover as many as possible.

context `fixes t` :: (`'k` :: `cbl`, `'v`) `trie` **begin**

lemma `lookup-Trie-Mapping` [`code`]:

`Mapping.lookup (Trie-Mapping t)` = `lookup t`

— `Lookup` does not need the check on `cbl`, because we have defined the pseudo-constructor `Trie-Mapping` in terms of `lookup`

by `simp(transfer, simp)`

lemma `update-Trie-Mapping` [`code`]:

`Mapping.update k v (Trie-Mapping t)` =

(`case ID cbl` :: `'k cbl` of

`None` \Rightarrow `Code.abort (STR "update Trie-Mapping: cbl = None")` ($\lambda\text{-}$
`Mapping.update k v (Trie-Mapping t)`)

```
| Some - => Trie-Mapping (update k v t))
by(simp split: option.split add: lookup-update Mapping.update.abs-eq)
```

lemma keys-Trie-Mapping [code]:

```
Mapping.keys (Trie-Mapping t) =
  (case ID cbl :: !k cbl of
    None => Code.abort (STR "keys Trie-Mapping: cbl = None") (λ.
Mapping.keys (Trie-Mapping t))
    | Some - => keys t)
by(simp add: Mapping.keys.abs-eq keys-conv-dom-lookup split: option.split)
```

end

These equations do not replace the existing equations for the other constructors, but they do take precedence over them. If there is already a generic implementation for an operation *foo*, say $foo\ A = gen\text{-}foo\ A$, and you prove a specialised equation $foo\ (Trie\text{-}Mapping\ t) = trie\text{-}foo\ t$, then when you call *foo* on some *Trie-Mapping t*, your equation will kick in. LC exploits this sequentiality especially for binary operators on sets like (\cap) , where there are generic implementations and faster specialised ones.

Configure the heuristics

Finally, you should setup the heuristics that automatically picks a container implementation based on the types of the elements (§4.3.3).

The heuristics uses a type with a single value, e.g., *mapping-impl* with value *Mapping-IMPL*, but there is one pseudo-constructor for each container implementation in the generated code. All these pseudo-constructors are the same in the logic, but they are different in the generated code. Hence, the generated code can distinguish them, but we do not have to commit to anything in the logic. This allows to reconfigure and extend the heuristic at any time.

First, define and declare a new pseudo-constructor for the heuristics. Again, be sure to redeclare all previous pseudo-constructors.

```
definition mapping-Trie :: mapping-impl
where [simp]: mapping-Trie = Mapping-IMPL
```

code-datatype

```
mapping-Choose mapping-Assoc-List mapping-RBT mapping-Mapping map-
ping-Trie
```

Then, adjust the implementation of the automatic choice. For every initial value of the container (such as the empty map or the empty set), there is one new constant (e.g., *mapping-empty-choose* and *set-empty-choose*) equivalent

to it. Its code equation, however, checks the available operations from the type classes and picks an appropriate implementation.

For example, the following prefers red-black trees over tries, but tries over associative lists:

lemma mapping-empty-choose-code [code]:

```
(mapping-empty-choose :: ('a :: {ccompare, cbl}, 'b) mapping) =
(case ID CCOMPARE('a) of Some - => RBT-Mapping RBT-Mapping2.empty
 | None =>
  case ID (cbl :: 'a cbl) of Some - => Trie-Mapping empty
  | None => Assoc-List-Mapping DAList.empty)
```

by(auto split: option.split simp add: DAList.lookup-empty[abs-def] Mapping.empty-def lookup-empty)

There is also a second function for every such initial value that dispatches on the pseudo-constructors for *mapping-impl*. This function is used to pick the right implementation for types that specify a preference.

lemma mapping-empty-code [code]:

```
mapping-empty mapping-Trie = Trie-Mapping empty
```

by(simp add: lookup-empty Mapping.empty-def)

For $('k, 'v)$ *mapping*, LC also has a function *mapping-impl-choose2* which is given two preferences and returns one (for *'a set*, it is called *set-impl-choose2*). Polymorphic type constructors like $'a + 'b$ use it to pick an implementation based on the preferences of *'a* and *'b*. By default, it returns *mapping-Choose*, i.e., ignore the preferences. You should add a code equation like the following that overrides this choice if both preferences are your new data structure:

lemma mapping-impl-choose2-Trie [code]:

```
mapping-impl-choose2 mapping-Trie mapping-Trie = mapping-Trie
```

by(simp add: mapping-Trie-def)

If your new data structure is better than the existing ones for some element type, you should reconfigure the type's preference. As all preferences are logically equal, you can prove (and declare) the appropriate code equation. For example, the following prefers tries for keys of type *unit*:

lemma mapping-impl-unit-Trie [code]:

```
MAPPING-IMPL(unit) = Phantom(unit) mapping-Trie
```

by(simp add: mapping-impl-unit-def)

value Mapping.empty :: (unit, int) mapping

You can also use your new pseudo-constructor with *derive* in instantiations, just give its name as option:

derive (mapping-Trie) mapping-impl simple-tycon

4.5 Changing the configuration

As containers are connected to data structures only by refinement in the code generator, this can always be adapted later on. You can add new data structures as explained in §4.4. If you want to drop one, you redeclare the remaining pseudo-constructors with **code-datatype** and delete all code equations that pattern-match on the obsolete pseudo-constructors. The command **code-thms** will tell you which constants have such code equations. You can also freely adapt the heuristics for picking implementations as described in §4.4.4.

One thing, however, you cannot change afterwards, namely the decision whether an element type supports an operation and if so how it does, because this decision is visible in the logic.

4.6 New containers types

We hope that the above explanations and the examples with sets and maps suffice to show what you need to do if you add a new container type, e.g., priority queues. There are three steps:

1. **Introduce a type constructor for the container.**

Your new container type must not be a composite type, like $'a \Rightarrow 'b$ *option* for maps, because refinement for code generation only works with a single type constructor. Neither should you reuse a type constructor that is used already in other contexts, e.g., do not use $'a$ *list* to model queues.

Introduce a new type constructor if necessary (e.g., $('a, 'b)$ *mapping* for maps) – if your container type already has its own type constructor, everything is fine.

2. **Implement the data structures**

and connect them to the container type as described in §4.4.

3. **Define a heuristics for picking an implementation.**

See [5] for an explanation.

4.7 Troubleshooting

This section describes some difficulties in using LC that we have come across, provides some background for them, and discusses how to overcome them. If you experience other difficulties, please contact the author.

4.7.1 Nesting of mappings

Mappings can be arbitrarily nested on the value side, e.g., $(\prime a, (\prime b, \prime c) \text{ mapping}) \text{ mapping}$. However, $(\prime a, \prime b) \text{ mapping}$ cannot currently be the key of a mapping, i.e., code generation fails for $((\prime a, \prime b) \text{ mapping}, \prime c) \text{ mapping}$. Similarly, you cannot have a set of mappings like $(\prime a, \prime b) \text{ mapping set}$ at the moment. There are no issues to make this work, we have just not seen the need for it. If you need to generate code for such types, please get in touch with the author.

4.7.2 Wellsortedness errors

LC uses its own hierarchy of type classes which is distinct from Isabelle/HOL's. This ensures that every type can be made an instance of LC's type classes. Consequently, you must instantiate these classes for your own types. The following lists where you can find information about the classes and examples how to instantiate them:

type class	user guide	theory
<i>card-UNIV</i>	§4.3.5	<i>HOL-Library.Cardinality</i>
<i>cenum</i>	§4.3.4	<i>Containers.Collection-Enum</i>
<i>ceq</i>	§4.3.1	<i>Containers.Collection-Eq</i>
<i>ccompare</i>	§4.3.2	<i>Containers.Collection-Order</i>
<i>cproper-interval</i>	§4.3.5	<i>Containers.Collection-Order</i>
<i>finite-UNIV</i>	§4.3.5	<i>HOL-Library.Cardinality</i>
<i>mapping-impl</i>	§4.3.3	<i>Containers.Mapping-Impl</i>
<i>set-impl</i>	§4.3.3	<i>Containers.Set-Impl</i>

The type classes *card-UNIV* and *cproper-interval* are only required to implement the operations on set complements. If your code does not need complements, you can manually declare the code equations not involving *Complement* using the theorem collection *set-base-code*. It is also recommended that you remove the pseudo-constructor *Complement* from the code generator. Note that some set operations like $A - B$ and *UNIV* have no code equations any more.

```
code-datatype Collect-set DList-set RBT-set Set-Monad
declare set-base-code [code]
```

4.7.3 Exception raised at run-time

Not all combinations of data and container implementation are possible. For example, you cannot implement a set of functions with a RBT, because there is no order on $\prime a \Rightarrow \prime b$. If you try, the code will raise an exception **Fail** (with an exception message) or **Match**. They can occur in three cases:

1. You have misconfigured the heuristics that picks implementations (§4.3.3), or you have manually picked an implementation that requires an operation that the element type does not provide. Printing a stack trace for the exception may help you in locating the error.
2. You are trying to invoke an operation on a set complement which cannot be implemented on a complement representation, e.g., (\cdot) . If the element type is enumerable, provide an instance of *enum* and choose to represent complements of sets of enumerable types by the elements rather than the elements of the complement (see §4.3.4 for how to do this).
3. You use set comprehensions on types which do not provide an enumeration (i.e., they are represented as closures) or you chose to represent a map as a closure.

A lot of operations are not implementable for closures, in particular those that return some element of the container

Inspect the code equations with **code-thms** and look for calls to *Collect-set* and *Mapping* which are LC's constructor for sets and maps as closures.

Note that the code generator preprocesses set comprehensions like $\{i < 4 \mid i. 2 < i\}$ to $(\lambda i. i < 4) \text{ ' } \{i. 2 < i\}$, so this is a set comprehension over *int* rather than *bool*.

4.7.4 LC slows down my code

Normally, this will not happen, because LC's data structures are more efficient than Isabelle's list-based implementations. However, in some rare cases, you can experience a slowdown:

1. **Your containers contain just a few elements.**

In that case, the overhead of the heuristics to pick an implementation outweighs the benefits of efficient implementations. You should identify the tiny containers and disable the heuristics locally. You do so by replacing the initial value like $\{\}$ and *Mapping.empty* with low-overhead constructors like *Set-Monad* and *Mapping*. For example, if *tiny-set-code*: $\text{tiny-set} = \{1, 2\}$ is your code equation with a tiny set, the following changes the code equation to directly use the list-based representation, i.e., disables the heuristics:

lemma empty-Set-Monad: $\{\} = \text{Set-Monad } []$ **by** simp

declare tiny-set-code[code del, unfolded empty-Set-Monad, code]

If you want to globally disable the heuristics, you can also declare an equation like *empty-Set-Monad* as [code].

2. The element type contains many type constructors and some type variables.

LC heavily relies on type classes, and type classes are implemented as dictionaries if the compiler cannot statically resolve them, i.e., if there are type variables. For type constructors with type variables (like $'a \times 'b$), LC's definitions of the type class parameters recursively calls itself on the type variables, i.e., $'a$ and $'b$. If the element type is polymorphic, the compiler cannot precompute these recursive calls and therefore they have to be constructed repeatedly at run time. If you wrap your complicated type in a new type constructor, you can define optimised equations for the type class parameters.

Bibliography

- [1] F. Haftmann and L. Bulwahn. Code generation from Isabelle/HOL theories. <http://isabelle.in.tum.de/dist/Isabelle/doc/codegen.pdf>, 2013.
- [2] F. Haftmann, A. Krauss, O. Kunčar, and T. Nipkow. Data refinement in Isabelle/HOL. In *ITP'13*, LNCS. Springer, 2013.
- [3] P. Lammich. Collections framework. *Archive of Formal Proofs*, 2009. <http://isa-afp.org/entries/Collections.shtml>, Formal proof development.
- [4] P. Lammich and A. Lochbihler. The Isabelle Collections Framework. In *ITP'10*, volume 6172 of *LNCS*, pages 339–354. Springer, 2010.
- [5] A. Lochbihler. Light-weight containers for Isabelle: efficient, extensible, nestable. In *ITP'13*, LNCS. Springer, 2013.