Light-Weight Containers

Andreas Lochbihler

March 17, 2025

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

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*: $insert\ a\ A \gg= f = f\ a \cup (A \gg= f)$
⟨*proof*⟩

lemma *set-bind-iff*:
$set\ (List.bind\ xs\ f) = Set.bind\ (set\ xs)\ (set \circ f)$
⟨*proof*⟩

lemma *set-bind-conv-fold*: $set\ xs \gg= f = fold\ ((\cup) \circ f)\ xs\ \{\}$
⟨*proof*⟩

lemma *card-gt-1D*:
assumes $card\ A > 1$
shows $\exists\, x\ y.\ x \in A \wedge y \in A \wedge x \neq y$
⟨*proof*⟩

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

lemma *card-eq-Suc-0-ex1*: $card\ A = Suc\ 0 \longleftrightarrow (\exists!x.\ x \in A)$
⟨*proof*⟩

context *linorder* **begin**

lemma *sorted-last*: $[\![\ sorted\ xs;\ x \in set\ xs\ ]\!] \Longrightarrow x \leq last\ xs$
⟨*proof*⟩

end

lemma *empty-filter-conv*: $[] = filter\ P\ xs \longleftrightarrow (\forall\, x{\in}set\ xs.\ \neg\ P\ x)$
⟨*proof*⟩

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

lemma *ID-code* [*code*, *code-unfold*]: *ID* = $(\lambda x.\ x)$
⟨*proof*⟩

lemma *ID-Some*: *ID* (*Some x*) = *Some x*
⟨*proof*⟩

lemma *ID-None*: *ID None* = *None*
⟨*proof*⟩

lexicographic order on pairs

context
  fixes *leq-a* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** ‹$\sqsubseteq_a$› *50*)
  and *less-a* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** ‹$\sqsubset_a$› *50*)
  and *leq-b* :: $'b \Rightarrow 'b \Rightarrow bool$ (**infix** ‹$\sqsubseteq_b$› *50*)
  and *less-b* :: $'b \Rightarrow 'b \Rightarrow bool$ (**infix** ‹$\sqsubset_b$› *50*)
begin

definition *less-eq-prod* :: $('a \times 'b) \Rightarrow ('a \times 'b) \Rightarrow bool$ (**infix** ‹$\sqsubseteq$› *50*)
where *less-eq-prod* = $(\lambda(x1, x2)\ (y1, y2).\ x1 \sqsubset_a y1 \lor x1 \sqsubseteq_a y1 \land x2 \sqsubseteq_b y2)$

definition *less-prod* :: $('a \times 'b) \Rightarrow ('a \times 'b) \Rightarrow bool$ (**infix** ‹$\sqsubset$› *50*)
where *less-prod* = $(\lambda(x1, x2)\ (y1, y2).\ x1 \sqsubset_a y1 \lor x1 \sqsubseteq_a y1 \land x2 \sqsubset_b y2)$

lemma *less-eq-prod-simps* [*simp*]:
  $(x1, x2) \sqsubseteq (y1, y2) \longleftrightarrow x1 \sqsubset_a y1 \lor x1 \sqsubseteq_a y1 \land x2 \sqsubseteq_b y2$
⟨*proof*⟩

lemma *less-prod-simps* [*simp*]:
  $(x1, x2) \sqsubset (y1, y2) \longleftrightarrow x1 \sqsubset_a y1 \lor x1 \sqsubseteq_a y1 \land x2 \sqsubset_b y2$
⟨*proof*⟩

end

context
  fixes *leq-a* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** ‹$\sqsubseteq_a$› *50*)
  and *less-a* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** ‹$\sqsubset_a$› *50*)
  and *leq-b* :: $'b \Rightarrow 'b \Rightarrow bool$ (**infix** ‹$\sqsubseteq_b$› *50*)
  and *less-b* :: $'b \Rightarrow 'b \Rightarrow bool$ (**infix** ‹$\sqsubset_b$› *50*)
  assumes *lin-a*: *class.linorder leq-a less-a*
  and *lin-b*: *class.linorder leq-b less-b*
begin

abbreviation (*input*) *less-eq-prod'* :: $('a \times 'b) \Rightarrow ('a \times 'b) \Rightarrow bool$ (**infix** ‹$\sqsubseteq$› *50*)
where *less-eq-prod'* ≡ *less-eq-prod leq-a less-a leq-b*

abbreviation (*input*) *less-prod′* :: (′*a* × ′*b*) ⇒ (′*a* × ′*b*) ⇒ *bool* (**infix** ‹⊏› *50*)
where *less-prod′* ≡ *less-prod leq-a less-a less-b*

lemma *linorder-prod*:
class.linorder (⊑) (⊏)
⟨*proof*⟩

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* ⇒ (′*a* ⇒ ′*a*) ⇒ ′*a set*
where *rangeIt x f* = *range* (λ*n*. (*f* ^^ *n*) *x*)

definition *rangeC* :: (′*a* ⇒ ′*b*) *set* ⇒ ′*b set*
where *rangeC F* = (⋃*f* ∈ *F*. *range f*)

lemma *infinite-rangeIt*:
assumes *inj*: *inj f*
and *x*: ∀ *y*. *x* ≠ *f y*
shows ¬ *finite* (*rangeIt x f*)
⟨*proof*⟩

lemma *in-rangeC*: *f* ∈ *A* ⟹ *f x* ∈ *rangeC A*
⟨*proof*⟩

lemma *in-rangeCE*: **assumes** *y* ∈ *rangeC A*
obtains *f x* **where** *f* ∈ *A*   *y* = *f x*
⟨*proof*⟩

lemma *in-rangeC-singleton*: *f x* ∈ *rangeC* {*f*}
⟨*proof*⟩

lemma *in-rangeC-singleton-const*: *x* ∈ *rangeC* {λ-. *x*}
⟨*proof*⟩

lemma *rangeC-rangeC*: *f* ∈ *rangeC A* ⟹ *f x* ∈ *rangeC* (*rangeC A*)

⟨*proof*⟩

lemma *rangeC-eq-empty*: *rangeC* $A = \{\} \longleftrightarrow A = \{\}$
⟨*proof*⟩

lemma *Ball-rangeC-iff*:
$(\forall x \in \mathit{rangeC}\ A.\ P\ x) \longleftrightarrow (\forall f \in A.\ \forall x.\ P\ (f\ x))$
⟨*proof*⟩

lemma *Ball-rangeC-singleton*:
$(\forall x \in \mathit{rangeC}\ \{f\}.\ P\ x) \longleftrightarrow (\forall x.\ P\ (f\ x))$
⟨*proof*⟩

lemma *Ball-rangeC-rangeC*:
$(\forall x \in \mathit{rangeC}\ (\mathit{rangeC}\ A).\ P\ x) \longleftrightarrow (\forall f \in \mathit{rangeC}\ A.\ \forall x.\ P\ (f\ x))$
⟨*proof*⟩

lemma *finite-rangeC*:
assumes *inj*: $\forall f \in A.\ \mathit{inj}\ f$
and *disjoint*: $\forall f \in A.\ \forall g \in A.\ f \neq g \longrightarrow (\forall x\ y.\ f\ x \neq g\ y)$
shows $\mathit{finite}\ (\mathit{rangeC}\ (A :: ('a \Rightarrow 'b)\ \mathit{set})) \longleftrightarrow \mathit{finite}\ A \wedge (A \neq \{\} \longrightarrow \mathit{finite}$
$(\mathit{UNIV} :: 'a\ \mathit{set}))$
(**is** $?lhs \longleftrightarrow ?rhs$)
⟨*proof*⟩

lemma *finite-rangeC-singleton-const*:
$\mathit{finite}\ (\mathit{rangeC}\ \{\lambda\text{-}.\ x\})$
⟨*proof*⟩

lemma *card-Un*:
$[\![\mathit{finite}\ A; \mathit{finite}\ B]\!] \Longrightarrow \mathit{card}\ (A \cup B) = \mathit{card}\ (A) + \mathit{card}\ (B) - \mathit{card}(A \cap B)$
⟨*proof*⟩

lemma *card-rangeC-singleton-const*:
$\mathit{card}\ (\mathit{rangeC}\ \{\lambda\text{-}.\ f\}) = \mathit{1}$
⟨*proof*⟩

lemma *card-rangeC*:
assumes *inj*: $\forall f \in A.\ \mathit{inj}\ f$
and *disjoint*: $\forall f \in A.\ \forall g \in A.\ f \neq g \longrightarrow (\forall x\ y.\ f\ x \neq g\ y)$
shows $\mathit{card}\ (\mathit{rangeC}\ (A :: ('a \Rightarrow 'b)\ \mathit{set})) = \mathit{CARD}('a) * \mathit{card}\ A$
(**is** $?lhs = ?rhs$)
⟨*proof*⟩

lemma *rangeC-Int-rangeC*:
$[\![\forall f \in A.\ \forall g \in B.\ \forall x\ y.\ f\ x \neq g\ y]\!] \Longrightarrow \mathit{rangeC}\ A \cap \mathit{rangeC}\ B = \{\}$
⟨*proof*⟩

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*⟩

definition *card-fun* :: *nat* ⇒ *nat* ⇒ *nat*
where *card-fun a b* = (*if a* ≠ *0* ∧ *b* ≠ *0* ∨ *b* = *1 then b* ^ *a else 0*)

lemma *CARD-fun* [*card-simps*]:
CARD(*'a* ⇒ *'b*) = *card-fun CARD*(*'a*) *CARD*(*'b*)
⟨*proof*⟩

definition *card-sum* :: *nat* ⇒ *nat* ⇒ *nat*
where *card-sum a b* = (*if a* = *0* ∨ *b* = *0 then 0 else a* + *b*)

lemma *CARD-sum* [*card-simps*]:
CARD(*'a* + *'b*) = *card-sum CARD*(*'a*) *CARD*(*'b*)
⟨*proof*⟩

definition *card-option* :: *nat* ⇒ *nat*
where *card-option n* = (*if n* = *0 then 0 else Suc n*)

lemma *CARD-option* [*card-simps*]:
CARD(*'a option*) = *card-option CARD*(*'a*)
⟨*proof*⟩

definition *card-prod* :: *nat* ⇒ *nat* ⇒ *nat*

where *card-prod a b* = *a* ∗ *b*

lemma *CARD-prod* [*card-simps*]:
  CARD(′*a* ∗ ′*b*) = *card-prod CARD*(′*a*) *CARD*(′*b*)
⟨*proof*⟩

definition *card-list* :: *nat* ⇒ *nat*
where *card-list* - = *0*

lemma *CARD-list* [*card-simps*]: *CARD*(′*a list*) = *card-list CARD*(′*a*)
⟨*proof*⟩

end

theory *List-Fusion*
imports
  Main
begin

2.3 Shortcut fusion for lists

lemma *Option-map-mono* [*partial-function-mono*]:
  mono-option f ⟹ *mono-option* (λ*x*. *map-option g* (*f x*))
⟨*proof*⟩

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*′: ⋀*xs ys*. *X xs ys* ⟹ *xs* = [] ⟷ *ys* = []
  and *Cons*′: ⋀*xs ys*. ⟦ *X xs ys*; *xs* ≠ []; *ys* ≠ [] ⟧ ⟹ *A* (*hd xs*) (*hd ys*) ∧ (*X* (*tl xs*) (*tl ys*) ∨ *list-all2 A* (*tl xs*) (*tl ys*))
  shows *list-all2 A xs ys*
⟨*proof*⟩

2.3.1 The type of generators for finite lists

type-synonym (′*a*, ′*s*) *raw-generator* = (′*s* ⇒ *bool*) × (′*s* ⇒ ′*a* × ′*s*)

inductive-set *terminates-on* :: (′*a*, ′*s*) *raw-generator* ⇒ ′*s set*
  for *g* :: (′*a*, ′*s*) *raw-generator*
where
  stop: ¬ *fst g s* ⟹ *s* ∈ *terminates-on g*
| *unfold*: ⟦ *fst g s*; *snd* (*snd g s*) ∈ *terminates-on g* ⟧ ⟹ *s* ∈ *terminates-on g*

definition *terminates* :: (′*a*, ′*s*) *raw-generator* ⇒ *bool*
where *terminates g* ⟷ (*terminates-on g* = *UNIV*)

lemma *terminatesI* [*intro?*]:

$(\bigwedge s.\ s \in \textit{terminates-on } g) \Longrightarrow \textit{terminates } g$
⟨*proof*⟩

lemma *terminatesD*:
$\textit{terminates } g \Longrightarrow s \in \textit{terminates-on } g$
⟨*proof*⟩

lemma *terminates-on-stop*:
$\textit{terminates-on } (\lambda\text{-}.\ \textit{False}, \textit{next}) = \textit{UNIV}$
⟨*proof*⟩

lemma *wf-terminates*:
assumes *wf R*
and *step*: $\bigwedge s.\ \textit{fst } g\ s \Longrightarrow (\textit{snd } (\textit{snd } g\ s), s) \in R$
shows *terminates g*
⟨*proof*⟩

lemma *terminates-wfD*:
assumes *terminates g*
shows $\textit{wf } \{(\textit{snd } (\textit{snd } g\ s), s) \mid s .\ \textit{fst } g\ s\}$
⟨*proof*⟩

lemma *terminates-wfE*:
assumes *terminates g*
obtains *R* **where** *wf R* $\bigwedge s.\ \textit{fst } g\ s \Longrightarrow (\textit{snd } (\textit{snd } g\ s), s) \in R$
⟨*proof*⟩

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

partial-function (*option*) *terminates-within* $:: 's \Rightarrow \textit{nat option}$
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*⟩

end

lemma *terminates-within-unfold*:
$\textit{has-next } s \Longrightarrow$
terminates-within (*has-next*, *next*) *s* = *map-option* ($\lambda n.\ n + 1$) (*terminates-within* (*has-next*, *next*) (*snd* (*next s*)))
⟨*proof*⟩

typedef ('*a*, '*s*) *generator* = {*g* :: ('*a*, '*s*) *raw-generator*. *terminates g*}
morphisms *generator Generator*
⟨*proof*⟩

setup-lifting *type-definition-generator*

lemma *terminates-on-generator-eq-UNIV*:
terminates-on (*generator g*) = *UNIV*
⟨*proof*⟩

lemma *terminates-within-stop*:
terminates-within (λ-. *False*, *next*) *s* = *Some 0*
⟨*proof*⟩

lemma *terminates-within-generator-neq-None*:
terminates-within (*generator g*) *s* ≠ *None*
⟨*proof*⟩

locale *list* =
fixes *g* :: ('*a*, '*s*) *generator* **begin**

definition *has-next* :: '*s* ⇒ *bool*
where *has-next* = *fst* (*generator g*)

definition *next* :: '*s* ⇒ '*a* × '*s*
where *next* = *snd* (*generator g*)

function *unfoldr* :: '*s* ⇒ '*a list*
where *unfoldr s* = (*if has-next s then let* (*a*, *s'*) = *next s in a* # *unfoldr s' else* [])
⟨*proof*⟩
termination
⟨*proof*⟩

declare *unfoldr.simps* [*simp del*]

lemma *unfoldr-simps*:
has-next s ⟹ *unfoldr s* = *fst* (*next s*) # *unfoldr* (*snd* (*next s*))
¬ *has-next s* ⟹ *unfoldr s* = []
⟨*proof*⟩

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* (=) (=) ===> (=)) *fst list.has-next*
⟨*proof*⟩

lemma *generator-next-transfer* [*transfer-rule*]:
(*pcr-generator* (=) (=) ===> (=)) *snd list.next*
⟨*proof*⟩

end

lemma *unfoldr-eq-Nil-iff* [*iff*]:
list.unfoldr g s = [] ⟷ ¬ *list.has-next g s*
⟨*proof*⟩

lemma *Nil-eq-unfoldr-iff* [*simp*]:
[] = *list.unfoldr g s* ⟷ ¬ *list.has-next g s*
⟨*proof*⟩

2.3.2 Generators for *'a list*

primrec *list-has-next* :: *'a list* ⇒ *bool*
where
list-has-next [] ⟷ *False*
| *list-has-next* (*x* # *xs*) ⟷ *True*

primrec *list-next* :: *'a list* ⇒ *'a* × *'a list*
where
list-next (*x* # *xs*) = (*x*, *xs*)

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

lift-definition *list-generator* :: (*'a*, *'a list*) *generator*
is (*list-has-next*, *list-next*)
⟨*proof*⟩

lemma *has-next-list-generator* [*simp*]:
list.has-next list-generator = *list-has-next*
⟨*proof*⟩

lemma *next-list-generator* [*simp*]:
list.next list-generator = *list-next*
⟨*proof*⟩

lemma *unfoldr-list-generator*:
list.unfoldr list-generator xs = *xs*
⟨*proof*⟩

lemma *terminates-replicate-generator*:
terminates (λn :: *nat*. *0* < n, λn. (a, n − *1*))
⟨*proof*⟩

lift-definition *replicate-generator* :: *'a* ⇒ (*'a*, *nat*) *generator*
is λa. (λn. *0* < n, λn. (a, n − *1*))
⟨*proof*⟩

lemma *has-next-replicate-generator* [*simp*]:
list.has-next (*replicate-generator a*) n ⟷ *0* < n
⟨*proof*⟩

lemma *next-replicate-generator* [*simp*]:
list.next (*replicate-generator a*) n = (a, n − *1*)
⟨*proof*⟩

lemma *unfoldr-replicate-generator*:
list.unfoldr (*replicate-generator a*) n = *replicate n a*
⟨*proof*⟩

context fixes *f* :: *'a* ⇒ *'b* **begin**

lift-definition *map-generator* :: (*'a*, *'s*) *generator* ⇒ (*'b*, *'s*) *generator*
is λ(*has-next*, *next*). (*has-next*, λs. *let* (a, s') = *next s in* (*f a*, s'))
⟨*proof*⟩

lemma *has-next-map-generator* [*simp*]:
list.has-next (*map-generator g*) = *list.has-next g*
⟨*proof*⟩

lemma *next-map-generator* [*simp*]:
list.next (*map-generator g*) = *apfst f* ∘ *list.next g*
⟨*proof*⟩

lemma *unfoldr-map-generator*:
list.unfoldr (*map-generator g*) = *map f* ∘ *list.unfoldr g*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

end

context fixes *g1* :: (*'a*, *'s1*) *raw-generator*
and *g2* :: (*'a*, *'s2*) *raw-generator*
begin

fun *append-has-next* :: *'s1* × *'s2* + *'s2* ⇒ *bool*
where
append-has-next (*Inl* (*s1*, *s2*)) ⟷ *fst g1 s1* ∨ *fst g2 s2*
| *append-has-next* (*Inr s2*) ⟷ *fst g2 s2*

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

end

lift-definition *append-generator* :: $('a, 's1)$ *generator* $\Rightarrow$ $('a, 's2)$ *generator* $\Rightarrow$ $('a, 's1 \times 's2 + 's2)$ *generator*
  is $\lambda g1\ g2.$ (*append-has-next g1 g2*, *append-next g1 g2*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *unfoldr-append-generator-Inr*:
  list.unfoldr (*append-generator g1 g2*) (*Inr s2*) = *list.unfoldr g2 s2*
⟨*proof*⟩

lemma *unfoldr-append-generator-Inl*:
  list.unfoldr (*append-generator g1 g2*) (*Inl* (*s1*, *s2*)) =
   list.unfoldr g1 s1 @ *list.unfoldr g2 s2*
⟨*proof*⟩

lemma *unfoldr-append-generator*:
  list.unfoldr (*append-generator g1 g2*) (*append-init s1 s2*) =
   list.unfoldr g1 s1 @ *list.unfoldr g2 s2*
⟨*proof*⟩

lift-definition *zip-generator* :: $('a, 's1)$ *generator* $\Rightarrow$ $('b, 's2)$ *generator* $\Rightarrow$ $('a \times$

′b, ′s1 × ′s2) generator
is *λ(has-next1, next1) (has-next2, next2).*
(λ(s1, s2). has-next1 s1 ∧ has-next2 s2,
λ(s1, s2). let (x, s1′) = next1 s1; (y, s2′) = next2 s2
in ((x, y), (s1′, s2′)))
⟨proof⟩

abbreviation *(input) zip-init :: ′s1 ⇒ ′s2 ⇒ ′s1 × ′s2*
where *zip-init ≡ Pair*

lemma *has-next-zip-generator [simp]:*
list.has-next (zip-generator g1 g2) (s1, s2) ⟷
list.has-next g1 s1 ∧ list.has-next g2 s2
⟨proof⟩

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)))
⟨proof⟩

lemma *unfoldr-zip-generator:*
list.unfoldr (zip-generator g1 g2) (zip-init s1 s2) =
zip (list.unfoldr g1 s1) (list.unfoldr g2 s2)
⟨proof⟩

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

lift-definition *upt-generator :: (nat, nat) generator*
is *(λn. n < bound, λn. (n, Suc n))*
⟨proof⟩

lemma *has-next-upt-generator [simp]:*
list.has-next upt-generator n ⟷ n < bound
⟨proof⟩

lemma *next-upt-generator [simp]:*
list.next upt-generator n = (n, Suc n)
⟨proof⟩

lemma *unfoldr-upt-generator:*
list.unfoldr upt-generator n = [n..<bound]
⟨proof⟩

end

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

lift-definition *upto-generator :: (int, int) generator*

is ($\lambda n.\ n \le bound,\ \lambda n.\ (n,\ n + 1)$)
⟨*proof*⟩

lemma *has-next-upto-generator* [*simp*]:
list.has-next upto-generator $n \longleftrightarrow n \le bound$
⟨*proof*⟩

lemma *next-upto-generator* [*simp*]:
list.next upto-generator $n = (n,\ n + 1)$
⟨*proof*⟩

lemma *unfoldr-upto-generator*:
list.unfoldr upto-generator $n = [n..bound]$
⟨*proof*⟩

end

context
fixes $P :: {'a} \Rightarrow bool$
begin

context
fixes $g :: ({'a},\ {'s})$ *raw-generator*
begin

inductive *filter-has-next* $:: {'s} \Rightarrow bool$
where
⟦ *fst g s*; *P* (*fst* (*snd g s*)) ⟧ $\Longrightarrow$ *filter-has-next s*
| ⟦ *fst g s*; ¬ *P* (*fst* (*snd g s*)); *filter-has-next* (*snd* (*snd g s*)) ⟧ $\Longrightarrow$ *filter-has-next s*

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

end

lift-definition *filter-generator* :: (′*a*, ′*s*) *generator* ⇒ (′*a*, ′*s*) *generator*
is λ*g*. (*filter-has-next g*, *filter-next g*)
⟨*proof*⟩

lemma *has-next-filter-generator*:
list.has-next (*filter-generator g*) *s* ⟷
list.has-next g s ∧ (*let* $(x,\ s')$ = *list.next g s in if P x then True else list.has-next*
(*filter-generator g*) s')
⟨*proof*⟩

lemma *next-filter-generator*:
list.next (*filter-generator g*) *s* =
(*let* $(x,\ s')$ = *list.next g s*

in if P x then (x, s') *else list.next* (*filter-generator g*) s')
⟨*proof*⟩

lemma *has-next-filter-generator-induct* [*consumes 1*, *case-names find step*]:
assumes *list.has-next* (*filter-generator g*) *s*
and *find*: ⋀*s*. ⟦ *list.has-next g s*; *P* (*fst* (*list.next g s*)) ⟧ ⟹ *Q s*
and *step*: ⋀*s*. ⟦ *list.has-next g s*; ¬ *P* (*fst* (*list.next g s*)); *Q* (*snd* (*list.next g s*)) ⟧ ⟹ *Q s*
shows *Q s*
⟨*proof*⟩

lemma *filter-generator-empty-conv*:
list.has-next (*filter-generator g*) *s* ⟷ (∃ *x*∈*set* (*list.unfoldr g s*). *P x*) (**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *unfoldr-filter-generator*:
list.unfoldr (*filter-generator g*) *s* = *filter P* (*list.unfoldr g s*)
⟨*proof*⟩

end

2.3.3 Destroying lists

definition *hd-fusion* :: ('*a*, '*s*) *generator* ⇒ '*s* ⇒ '*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*)
⟨*proof*⟩

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

definition *fold-fusion* :: ('*a*, '*s*) *generator* ⇒ ('*a* ⇒ '*b* ⇒ '*b*) ⇒ '*s* ⇒ '*b* ⇒ '*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') = *list.next g s*
in fold-fusion g f s' (*f x b*)
else b)
⟨*proof*⟩

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

definition *gen-length-fusion* :: ('*a*, '*s*) *generator* ⇒ *nat* ⇒ '*s* ⇒ *nat*
where *gen-length-fusion g n s* = *n* + *length* (*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* (*Suc n*) (*snd* (*list.next g s*)) *else n*)
⟨*proof*⟩

definition *length-fusion* :: (′*a*, ′*s*) *generator* ⇒ ′*s* ⇒ *nat*
where *length-fusion g s* = *length* (*list.unfoldr g s*)

lemma *length-fusion-code* [*code*]:
  length-fusion g = *gen-length-fusion g 0*
⟨*proof*⟩

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

definition *map-fusion* :: (′*a* ⇒ ′*b*) ⇒ (′*a*, ′*s*) *generator* ⇒ ′*s* ⇒ ′*b list*
where *map-fusion f g s* = *map f* (*list.unfoldr g s*)

lemma *map-fusion-code* [*code*]:
  map-fusion f g s =
  (*if list.has-next g s then*
    let (*x*, *s*′) = *list.next g s*
    in f x # *map-fusion f g s*′
   else [])
⟨*proof*⟩

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

definition *append-fusion* :: (′*a*, ′*s1*) *generator* ⇒ (′*a*, ′*s2*) *generator* ⇒ ′*s1* ⇒ ′*s2* ⇒ ′*a list*
where *append-fusion g1 g2 s1 s2* = *list.unfoldr g1 s1* @ *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)
⟨*proof*⟩

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

definition *zip-fusion* :: (′*a*, ′*s1*) *generator* ⇒ (′*b*, ′*s2*) *generator* ⇒ ′*s1* ⇒ ′*s2* ⇒ (′*a* × ′*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* ∧ *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 [])
⟨proof⟩
```

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

definition *list-all-fusion* :: (*'a*, *'s*) *generator* ⇒ (*'a* ⇒ *bool*) ⇒ *'s* ⇒ *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 ⟷
 (list.has-next g s ⟶
  (let (x, s') = list.next g s
   in P x ∧ list-all-fusion g P s'))
⟨proof⟩
```

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

definition *list-all2-fusion* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ (*'a*, *'s1*) *generator* ⇒ (*'b*, *'s2*) *generator* ⇒ *'s1* ⇒ *'s2* ⇒ *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 ∧
    (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)
⟨proof⟩
```

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))
⟨proof⟩
```

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*⟩
lemma *lexordp-lex*: (*xs*, *ys*) ∈ *lex* {(*xs*, *ys*). *xs* < *ys*} ⟷ *lexordp xs ys* ∧ *length xs* = *length ys*
⟨*proof*⟩

end

2.4.1 Setup for list fusion

context *ord* **begin**

definition *lexord-fusion* :: (*'a*, *'s1*) *generator* ⇒ (*'a*, *'s2*) *generator* ⇒ *'s1* ⇒ *'s2* ⇒ *bool*
where [*code del*]: *lexord-fusion g1 g2 s1 s2* = *lexordp* (*list.unfoldr g1 s1*) (*list.unfoldr g2 s2*)

definition *lexord-eq-fusion* :: (*'a*, *'s1*) *generator* ⇒ (*'a*, *'s2*) *generator* ⇒ *'s1* ⇒ *'s2* ⇒ *bool*
where [*code del*]: *lexord-eq-fusion g1 g2 s1 s2* = *lexordp-eq* (*list.unfoldr g1 s1*) (*list.unfoldr g2 s2*)

lemma *lexord-fusion-code*:
lexord-fusion g1 g2 s1 s2 ⟷
(*if list.has-next g1 s1 then*
if list.has-next g2 s2 then
let (*x*, *s1'*) = *list.next g1 s1*;
(*y*, *s2'*) = *list.next g2 s2*
in x < *y* ∨ ¬ *y* < *x* ∧ *lexord-fusion g1 g2 s1' s2'*
else False
else list.has-next g2 s2)
⟨*proof*⟩

lemma *lexord-eq-fusion-code*:

lexord-eq-fusion g1 g2 s1 s2 ⟷
(*list.has-next g1 s1* ⟶
list.has-next g2 s2 ∧
(*let* (*x*, *s1'*) = *list.next g1 s1*;
(*y*, *s2'*) = *list.next g2 s2*
in $x < y \lor \neg\, y < x \land$ *lexord-eq-fusion g1 g2 s1' s2'*))
⟨*proof*⟩

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*: ⟦ $x \in C$; $C \in$ *Chains r*; $y \in C$ ⟧ ⟹ $(x, y) \in r \lor (y, x) \in r$
⟨*proof*⟩

lemma *Chains-Field*: ⟦ $C \in$ *Chains r*; $x \in C$ ⟧ ⟹ $x \in$ *Field r*
⟨*proof*⟩

lemma *total-onD*:
⟦ *total-on A r*; $x \in A$; $y \in A$ ⟧ ⟹ $(x, y) \in r \lor x = y \lor (y, x) \in r$
⟨*proof*⟩

lemma *linear-order-imp-linorder*: *linear-order* {(*A*, *B*). *leq A B*} ⟹ *class.linorder leq* (λ*x y*. *leq x y* ∧ ¬ *leq y x*)
⟨*proof*⟩

lemma (**in** *linorder*) *linear-order*: *linear-order* {(*A*, *B*). $A \le B$}
⟨*proof*⟩

definition *order-consistent* :: ('*a* × '*a*) *set* ⇒ ('*a* × '*a*) *set* ⇒ *bool*
where *order-consistent r s* ⟷ (∀ *a a'*. $(a, a') \in r \longrightarrow (a', a) \in s \longrightarrow a = a'$)

lemma *order-consistent-sym*:
  order-consistent r s $\Longrightarrow$ *order-consistent s r*
⟨*proof*⟩

lemma *antisym-order-consistent-self*:
  antisym r $\Longrightarrow$ *order-consistent r r*
⟨*proof*⟩

lemma *refl-on-trancl*:
  assumes *refl-on A r*
  shows *refl-on A* $(r\hat{\ }+)$
⟨*proof*⟩

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*⟩

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*: $(x, y) \in (r \cup s)\hat{\ }+$
  obtains $(x, y) \in r$
    | *u v* **where** $(x, u) \in r$ $(u, v) \in s$ $(v, y) \in r$
⟨*proof*⟩

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*: $B \subseteq A$
  shows *antisym* $((r \cup s)\hat{\ }+)$
⟨*proof*⟩

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* $((r \cup s)\hat{\ }+)$
  ⟨*proof*⟩

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*⟩

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*) ⟹ *A* ∈ *infinite-complement-partition*
complement-partition: ¬ *finite* (*UNIV* :: *'a set*)
⟹ (*A* :: *'a set*) ∈ *infinite-complement-partition* ⟷ − *A* ∉ *infinite-complement-partition*
⟨*proof*⟩

lemma *not-in-complement-partition*:
¬ *finite* (*UNIV* :: *'a set*)
⟹ (*A* :: *'a set*) ∉ *infinite-complement-partition* ⟷ − *A* ∈ *infinite-complement-partition*
⟨*proof*⟩

lemma *not-in-complement-partition-False*:
⟦ (*A* :: *'a set*) ∈ *infinite-complement-partition*; ¬ *finite* (*UNIV* :: *'a set*) ⟧
⟹ − *A* ∈ *infinite-complement-partition* = *False*
⟨*proof*⟩

lemma *infinite-complement-partition-finite* [*simp*]:
finite (*UNIV* :: *'a set*) ⟹ *infinite-complement-partition* = (*UNIV* :: *'a set set*)
⟨*proof*⟩

lemma *Compl-eq-empty-iff*: − *A* = {} ⟷ *A* = *UNIV*
⟨*proof*⟩

A lexicographic-style order on finite subsets

context *ord* **begin**

definition *set-less-aux* :: $'a\ set \Rightarrow 'a\ set \Rightarrow bool$ (**infix** ‹⊏′′› *50*)
where $A \sqsubset' B \longleftrightarrow$ *finite* $A \wedge$ *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** ‹⊑′′› *50*)
where $A \sqsubseteq' B \longleftrightarrow A \in$ *infinite-complement-partition* $\wedge A = B \vee A \sqsubset' B$

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

lemma *set-less-eq-aux-refl* [*iff*]: $A \sqsubseteq' A \longleftrightarrow A \in$ *infinite-complement-partition*
⟨*proof*⟩

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

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

lemma *set-less-aux-antisym*: $[\![\ A \sqsubset' B;\ B \sqsubset' A\ ]\!] \Longrightarrow$ *False*
⟨*proof*⟩

lemma *set-less-aux-conv-set-less-eq-aux*:
$A \sqsubset' B \longleftrightarrow A \sqsubseteq' B \wedge \neg B \sqsubseteq' A$
⟨*proof*⟩

lemma *set-less-eq-aux-antisym*: $[\![\ A \sqsubseteq' B;\ B \sqsubseteq' A\ ]\!] \Longrightarrow A = B$
⟨*proof*⟩

lemma *set-less-aux-finiteD*: $A \sqsubset' B \Longrightarrow$ *finite* $A \wedge B \in$ *infinite-complement-partition*
⟨*proof*⟩

lemma *set-less-eq-aux-infinite-complement-partitionD*:
$A \sqsubseteq' B \Longrightarrow A \in$ *infinite-complement-partition* $\wedge B \in$ *infinite-complement-partition*
⟨*proof*⟩

lemma *Compl-set-less-aux-Compl*:
finite (*UNIV* :: $'a\ set$) $\Longrightarrow -A \sqsubset' -B \longleftrightarrow B \sqsubset' A$
⟨*proof*⟩

lemma *Compl-set-less-eq-aux-Compl*:
finite (*UNIV* :: $'a\ set$) $\Longrightarrow -A \sqsubseteq' -B \longleftrightarrow B \sqsubseteq' A$
⟨*proof*⟩

lemma *set-less-aux-insert-same*:
$x \in A \longleftrightarrow x \in B \Longrightarrow$ *insert* $x\ A \sqsubset'$ *insert* $x\ B \longleftrightarrow A \sqsubset' B$

⟨*proof*⟩

lemma *set-less-eq-aux-insert-same*:
$[\![A \in$ *infinite-complement-partition*; *insert* x $B \in$ *infinite-complement-partition*;
$x \in A \longleftrightarrow x \in B]\!]$
$\Longrightarrow$ *insert* x $A \sqsubseteq'$ *insert* x $B \longleftrightarrow A \sqsubseteq' B$
⟨*proof*⟩

end

context *order* **begin**

lemma *set-less-aux-singleton-iff*: $A \sqsubset' \{x\} \longleftrightarrow$ *finite* $A \wedge (\forall\, a{\in}A.\ x < a)$
⟨*proof*⟩

end

context *linorder* **begin**

lemma *wlog-le* [*case-names sym le*]:
assumes $\bigwedge a\ b.\ P\ a\ b \Longrightarrow P\ b\ a$
and $\bigwedge a\ b.\ a \leq b \Longrightarrow P\ a\ b$
shows $P\ b\ a$
⟨*proof*⟩

lemma *empty-set-less-aux* [*simp*]: $\{\} \sqsubset' A \longleftrightarrow A \neq \{\} \wedge$ *finite* A
⟨*proof*⟩

lemma *empty-set-less-eq-aux* [*simp*]: $\{\} \sqsubseteq' A \longleftrightarrow$ *finite* A
⟨*proof*⟩

lemma *set-less-aux-trans*:
assumes AB: $A \sqsubset' B$ **and** BC: $B \sqsubset' C$
shows $A \sqsubset' C$
⟨*proof*⟩

lemma *set-less-eq-aux-trans* [*trans*]:
$[\![A \sqsubseteq' B;\ B \sqsubseteq' C]\!] \Longrightarrow A \sqsubseteq' C$
⟨*proof*⟩

lemma *set-less-trans-set-less-eq* [*trans*]:
$[\![A \sqsubset' B;\ B \sqsubseteq' C]\!] \Longrightarrow A \sqsubset' C$
⟨*proof*⟩

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

lemma *psubset-finite-imp-set-less-aux*:

assumes *AsB*: $A \subset B$ **and** *B*: *finite* B
shows $A \sqsubset' B$
⟨*proof*⟩

lemma *subset-finite-imp-set-less-eq-aux*:
$[\![A \subseteq B;\ \textit{finite}\ B]\!] \Longrightarrow A \sqsubseteq' B$
⟨*proof*⟩

lemma *empty-set-less-aux-finite-iff*:
$\textit{finite}\ A \Longrightarrow \{\} \sqsubset' A \longleftrightarrow A \neq \{\}$
⟨*proof*⟩

lemma *set-less-aux-finite-total*:
assumes *A*: *finite* A **and** *B*: *finite* B
shows $A \sqsubset' B \vee A = B \vee B \sqsubset' A$
⟨*proof*⟩

lemma *set-less-eq-aux-finite-total*:
$[\![\textit{finite}\ A;\ \textit{finite}\ B]\!] \Longrightarrow A \sqsubseteq' B \vee A = B \vee B \sqsubseteq' A$
⟨*proof*⟩

lemma *set-less-eq-aux-finite-total2*:
$[\![\textit{finite}\ A;\ \textit{finite}\ B]\!] \Longrightarrow A \sqsubseteq' B \vee B \sqsubseteq' A$
⟨*proof*⟩

lemma *set-less-aux-rec*:
assumes *A*: *finite* A **and** *B*: *finite* B
and A': $A \neq \{\}$ **and** B': $B \neq \{\}$
shows $A \sqsubset' B \longleftrightarrow \textit{Min}\ B < \textit{Min}\ A \vee \textit{Min}\ A = \textit{Min}\ B \wedge A - \{\textit{Min}\ A\} \sqsubset' B - \{\textit{Min}\ A\}$
⟨*proof*⟩

lemma *set-less-eq-aux-rec*:
assumes *finite* A *finite* B $A \neq \{\}$ $B \neq \{\}$
shows $A \sqsubseteq' B \longleftrightarrow \textit{Min}\ B < \textit{Min}\ A \vee \textit{Min}\ A = \textit{Min}\ B \wedge A - \{\textit{Min}\ A\} \sqsubseteq' B - \{\textit{Min}\ A\}$
⟨*proof*⟩

lemma *set-less-aux-Min-antimono*:
$[\![\textit{Min}\ A < \textit{Min}\ B;\ \textit{finite}\ A;\ \textit{finite}\ B;\ A \neq \{\}]\!] \Longrightarrow B \sqsubset' A$
⟨*proof*⟩

lemma *sorted-Cons-Min*: $\textit{sorted}\ (x \mathbin{\#} xs) \Longrightarrow \textit{Min}\ (\textit{insert}\ x\ (\textit{set}\ xs)) = x$
⟨*proof*⟩

lemma *set-less-aux-code*:
$[\![\textit{sorted}\ xs;\ \textit{distinct}\ xs;\ \textit{sorted}\ ys;\ \textit{distinct}\ ys]\!]$
$\Longrightarrow \textit{set}\ xs \sqsubset' \textit{set}\ ys \longleftrightarrow \textit{ord.lexordp}\ (>)\ xs\ ys$
⟨*proof*⟩

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*⟩

end

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

context *ord* **begin**

definition *set-less-eq-aux'* :: *'a set* $\Rightarrow$ *'a set* $\Rightarrow$ *bool* (**infix** ‹$\sqsubseteq''$› *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*
⟨*proof*⟩

lemma *set-less-eq-aux'-antisym*: $[\![A \sqsubseteq'' B; B \sqsubseteq'' A]\!] \Longrightarrow A = B$
⟨*proof*⟩

lemma *set-less-eq-aux'-infinite-complement-partitionD*:
$A \sqsubseteq'' B \Longrightarrow A \in$ *infinite-complement-partition* $\wedge B \in$ *infinite-complement-partition*
⟨*proof*⟩

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

end

context *linorder* **begin**

lemma *set-less-eq-aux'-trans*: $[\![A \sqsubseteq'' B; B \sqsubseteq'' C]\!] \Longrightarrow A \sqsubseteq'' C$
⟨*proof*⟩

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

end

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

context *ord* **begin**

definition *set-less-eq-aux''* :: *'a set* $\Rightarrow$ *'a set* $\Rightarrow$ *bool* (**infix** ‹$\sqsubseteq'''$› *50*)
where *set-less-eq-aux''* =
(*SOME sleq*.

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

lemma *set-less-eq-aux''-spec*:
shows $linear\text{-}order\ \{(a,\ b).\ a \leq b\} \Longrightarrow linear\text{-}order\text{-}on\ infinite\text{-}complement\text{-}partition$ $\{(A,\ B).\ A \sqsubseteq'''\ B\}$
(**is** *PROP ?thesis1*)
and $order\text{-}consistent\ \{(A,\ B).\ A \sqsubseteq''\ B\}\ \{(A,\ B).\ A \sqsubseteq'''\ B\}$ (**is** *?thesis2*)
⟨*proof*⟩

end

context *linorder* **begin**

lemma *set-less-eq-aux''-linear-order*:
$linear\text{-}order\text{-}on\ infinite\text{-}complement\text{-}partition\ \{(A,\ B).\ A \sqsubseteq'''\ B\}$
⟨*proof*⟩

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

lemma *set-less-eq-aux'-into-set-less-eq-aux''*:
assumes $A \sqsubseteq''\ B$
shows $A \sqsubseteq'''\ B$
⟨*proof*⟩

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

lemma *set-less-eq-aux''-finite*:
$finite\ (UNIV :: 'a\ set) \Longrightarrow set\text{-}less\text{-}eq\text{-}aux'' = set\text{-}less\text{-}eq\text{-}aux$
⟨*proof*⟩

lemma *set-less-eq-aux''-antisym*:
$[\![\ A \sqsubseteq'''\ B;\ B \sqsubseteq'''\ A;$
$A \in infinite\text{-}complement\text{-}partition;\ B \in infinite\text{-}complement\text{-}partition\ ]\!]$
$\Longrightarrow A = B$
⟨*proof*⟩

lemma *set-less-eq-aux''-trans*: $[\![\ A \sqsubseteq'''\ B;\ B \sqsubseteq'''\ C\ ]\!] \Longrightarrow A \sqsubseteq'''\ C$
⟨*proof*⟩

lemma *set-less-eq-aux''-total*:
$[\![\ A \in infinite\text{-}complement\text{-}partition;\ B \in infinite\text{-}complement\text{-}partition\ ]\!]$
$\Longrightarrow A \sqsubseteq'''\ B \vee B \sqsubseteq'''\ A$
⟨*proof*⟩

end

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

context *ord* **begin**

definition *set-less-eq* :: $'a\ set \Rightarrow 'a\ set \Rightarrow bool$ (**infix** ‹$\sqsubseteq$› *50*)
where
$A \sqsubseteq B \longleftrightarrow$
(*if* $A \in$ *infinite-complement-partition then* $A \sqsubseteq''' B \vee B \notin$ *infinite-complement-partition*
else $B \notin$ *infinite-complement-partition* $\wedge - B \sqsubseteq''' - A$)

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* $\wedge - B \sqsubseteq''' - A$)
⟨*proof*⟩

end

context *linorder* **begin**

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

lemma *set-less-eq-antisym*: $[\![A \sqsubseteq B; B \sqsubseteq A]\!] \Longrightarrow A = B$
⟨*proof*⟩

lemma *set-less-eq-trans*: $[\![A \sqsubseteq B; B \sqsubseteq C]\!] \Longrightarrow A \sqsubseteq C$
⟨*proof*⟩

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

lemma *set-less-eq-linorder*: *class.linorder* ($\sqsubseteq$) ($\sqsubset$)
⟨*proof*⟩

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

lemma *Compl-set-less-eq-Compl*: $- A \sqsubseteq - B \longleftrightarrow B \sqsubseteq A$
⟨*proof*⟩

lemma *Compl-set-less-Compl*: $- A \sqsubset - B \longleftrightarrow B \sqsubset A$
⟨*proof*⟩

lemma *set-less-eq-finite-iff*: $[\![\textit{finite } A; \textit{finite } B]\!] \Longrightarrow A \sqsubseteq B \longleftrightarrow A \sqsubseteq' B$
⟨*proof*⟩

lemma *set-less-finite-iff*: $[\![\textit{finite } A; \textit{finite } B]\!] \Longrightarrow A \sqsubset B \longleftrightarrow A \sqsubset' B$
⟨*proof*⟩

lemma *infinite-set-less-eq-Complement*:
$[\![\textit{finite } A; \textit{finite } B; \neg \textit{ finite } (\textit{UNIV} :: 'a \textit{ set})]\!] \Longrightarrow A \sqsubseteq - B$
⟨*proof*⟩

lemma *infinite-set-less-Complement*:
$[\![\textit{finite } A; \textit{finite } B; \neg \textit{ finite } (\textit{UNIV} :: 'a \textit{ set})]\!] \Longrightarrow A \sqsubset - B$
⟨*proof*⟩

lemma *infinite-Complement-set-less-eq*:
$[\![\textit{finite } A; \textit{finite } B; \neg \textit{ finite } (\textit{UNIV} :: 'a \textit{ set})]\!] \Longrightarrow \neg - A \sqsubseteq B$
⟨*proof*⟩

lemma *infinite-Complement-set-less*:
$[\![\textit{finite } A; \textit{finite } B; \neg \textit{ finite } (\textit{UNIV} :: 'a \textit{ set})]\!] \Longrightarrow \neg - A \sqsubset B$
⟨*proof*⟩

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

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

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

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

lemma *set-less-eq-UNIV* [*iff*]: $A \sqsubseteq \textit{UNIV}$
⟨*proof*⟩

lemma *UNIV-set-less-eq* [*iff*]: $\textit{UNIV} \sqsubseteq A \longleftrightarrow A = \textit{UNIV}$
⟨*proof*⟩

lemma *set-less-UNIV-iff* [*iff*]: $A \sqsubset \textit{UNIV} \longleftrightarrow A \neq \textit{UNIV}$
⟨*proof*⟩

lemma *not-UNIV-set-less* [*simp*]: $\neg \textit{UNIV} \sqsubset A$
⟨*proof*⟩

end

2.6.2 Implementation based on sorted lists

type-synonym *'a proper-interval* = *'a option* ⇒ *'a option* ⇒ *bool*

class *proper-intrvl* = *ord* +
  fixes *proper-interval* :: *'a proper-interval*

class *proper-interval* = *proper-intrvl* +
  assumes *proper-interval-simps*:
  proper-interval None None = *True*
  proper-interval None (*Some y*) = (∃ *z*. *z* < *y*)
  proper-interval (*Some x*) *None* = (∃ *z*. *x* < *z*)
  proper-interval (*Some x*) (*Some y*) = (∃ *z*. *x* < *z* ∧ *z* < *y*)

context *proper-intrvl* **begin**

function *set-less-eq-aux-Compl* :: *'a option* ⇒ *'a list* ⇒ *'a list* ⇒ *bool*
where
  set-less-eq-aux-Compl ao [] *ys* ⟷ *True*
| *set-less-eq-aux-Compl ao xs* [] ⟷ *True*
| *set-less-eq-aux-Compl ao* (*x* # *xs*) (*y* # *ys*) ⟷
  (*if x* < *y then proper-interval ao* (*Some x*) ∨ *set-less-eq-aux-Compl* (*Some x*) *xs* (*y* # *ys*)
   else if y < *x then proper-interval ao* (*Some y*) ∨ *set-less-eq-aux-Compl* (*Some y*) (*x* # *xs*) *ys*
   else proper-interval ao (*Some y*))
⟨*proof*⟩
termination ⟨*proof*⟩

fun *Compl-set-less-eq-aux* :: *'a option* ⇒ *'a list* ⇒ *'a list* ⇒ *bool*
where
  Compl-set-less-eq-aux ao [] [] ⟷ ¬ *proper-interval ao None*
| *Compl-set-less-eq-aux ao* [] (*y* # *ys*) ⟷ ¬ *proper-interval ao* (*Some y*) ∧ *Compl-set-less-eq-aux* (*Some y*) [] *ys*
| *Compl-set-less-eq-aux ao* (*x* # *xs*) [] ⟷ ¬ *proper-interval ao* (*Some x*) ∧ *Compl-set-less-eq-aux* (*Some x*) *xs* []
| *Compl-set-less-eq-aux ao* (*x* # *xs*) (*y* # *ys*) ⟷
  (*if x* < *y then* ¬ *proper-interval ao* (*Some x*) ∧ *Compl-set-less-eq-aux* (*Some x*) *xs* (*y* # *ys*)
   else if y < *x then* ¬ *proper-interval ao* (*Some y*) ∧ *Compl-set-less-eq-aux* (*Some y*) (*x* # *xs*) *ys*
   else ¬ *proper-interval ao* (*Some y*))

fun *set-less-aux-Compl* :: *'a option* ⇒ *'a list* ⇒ *'a list* ⇒ *bool* **where**
  set-less-aux-Compl ao [] [] ⟷ *proper-interval ao None*
| *set-less-aux-Compl ao* [] (*y* # *ys*) ⟷ *proper-interval ao* (*Some y*) ∨ *set-less-aux-Compl* (*Some y*) [] *ys*
| *set-less-aux-Compl ao* (*x* # *xs*) [] ⟷ *proper-interval ao* (*Some x*) ∨ *set-less-aux-Compl* (*Some x*) *xs* []
| *set-less-aux-Compl ao* (*x* # *xs*) (*y* # *ys*) ⟷

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

function *Compl-set-less-aux* :: *'a option* ⇒ *'a list* ⇒ *'a list* ⇒ *bool* **where**
Compl-set-less-aux ao [] *ys* ⟷ *False*
| *Compl-set-less-aux ao xs* [] ⟷ *False*
| *Compl-set-less-aux ao* (*x* # *xs*) (*y* # *ys*) ⟷
(*if* $x < y$ *then* ¬ *proper-interval ao* (*Some x*) ∧ *Compl-set-less-aux* (*Some x*) *xs* (*y* # *ys*)
else if $y < x$ *then* ¬ *proper-interval ao* (*Some y*) ∧ *Compl-set-less-aux* (*Some y*) (*x* # *xs*) *ys*
else ¬ *proper-interval ao* (*Some y*))
⟨*proof*⟩
termination ⟨*proof*⟩

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 ⊑′ − *set ys* ⟷ *set-less-eq-aux-Compl None xs ys* (**is** *?Compl2*)
and *Compl1-set-less-eq-aux-conv-Compl-set-less-eq-aux*:
− *set xs* ⊑′ *set ys* ⟷ *Compl-set-less-eq-aux None xs ys* (**is** *?Compl1*)
⟨*proof*⟩

lemma *set-less-aux-Compl-iff*:
set-less-aux-Compl ao xs ys ⟷ *set-less-eq-aux-Compl ao xs ys* ∧ ¬ *Compl-set-less-eq-aux ao ys xs*
⟨*proof*⟩

lemma *Compl-set-less-aux-Compl-iff*:
Compl-set-less-aux ao xs ys ⟷ *Compl-set-less-eq-aux ao xs ys* ∧ ¬ *set-less-eq-aux-Compl ao ys xs*
⟨*proof*⟩

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 ⊏′ − *set ys* ⟷ *set-less-aux-Compl None xs ys* (**is** *?Compl2*)
and *Compl1-set-less-aux-conv-Compl-set-less-aux*:
− *set xs* ⊏′ *set ys* ⟷ *Compl-set-less-aux None xs ys* (**is** *?Compl1*)
⟨*proof*⟩

end

2.6.3 Implementation of proper intervals for sets

definition *length-last* :: ′*a list* ⇒ *nat* × ′*a*
where *length-last xs* = (*length xs*, *last xs*)

lemma *length-last-Nil* [*code*]: *length-last* [] = (*0*, *undefined*)
⟨*proof*⟩

lemma *length-last-Cons-code* [*symmetric*, *code*]:
fold (λ*x* (*n*, -) . (*n* + *1*, *x*)) *xs* (*1*, *x*) = *length-last* (*x* # *xs*)
⟨*proof*⟩

context *proper-intrvl* **begin**

fun *exhaustive-above* :: ′*a* ⇒ ′*a list* ⇒ *bool* **where**
exhaustive-above x [] ⟷ ¬ *proper-interval* (*Some x*) *None*
| *exhaustive-above x* (*y* # *ys*) ⟷ ¬ *proper-interval* (*Some x*) (*Some y*) ∧ *exhaustive-above y ys*

fun *exhaustive* :: ′*a list* ⇒ *bool* **where**
exhaustive [] = *False*
| *exhaustive* (*x* # *xs*) ⟷ ¬ *proper-interval None* (*Some x*) ∧ *exhaustive-above x xs*

fun *proper-interval-set-aux* :: ′*a list* ⇒ ′*a list* ⇒ *bool*
where
proper-interval-set-aux xs [] ⟷ *False*
| *proper-interval-set-aux* [] (*y* # *ys*) ⟷ *ys* ≠ [] ∨ *proper-interval* (*Some y*) *None*
| *proper-interval-set-aux* (*x* # *xs*) (*y* # *ys*) ⟷
(*if x* < *y then False*
else if y < *x then proper-interval* (*Some y*) (*Some x*) ∨ *ys* ≠ [] ∨ ¬ *exhaustive-above x xs*
else proper-interval-set-aux xs ys)

fun *proper-interval-set-Compl-aux* :: ′*a option* ⇒ *nat* ⇒ ′*a list* ⇒ ′*a list* ⇒ *bool*
where
proper-interval-set-Compl-aux ao n [] [] ⟷
CARD(′*a*) > *n* + *1*
| *proper-interval-set-Compl-aux ao n* [] (*y* # *ys*) ⟷

```
  (let m = CARD('a) − n; (len-y, y') = length-last (y # ys)
   in m ≠ len-y ∧ (m = len-y + 1 ⟶ ¬ proper-interval (Some y') None))
| proper-interval-set-Compl-aux ao n (x # xs) [] ⟷
  (let m = CARD('a) − n; (len-x, x') = length-last (x # xs)
   in m ≠ len-x ∧ (m = len-x + 1 ⟶ ¬ proper-interval (Some x') None))
| proper-interval-set-Compl-aux ao n (x # xs) (y # ys) ⟷
 (if x < y then
    proper-interval ao (Some x) ∨
    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*⟩

lemma *exhaustive-correct*:
 assumes *sorted xs* *distinct xs*
 shows *exhaustive xs* ⟷ *set xs* = *UNIV*
⟨*proof*⟩

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* ⟷ (∃ *A*. *set xs* ⊏′ *A* ∧ *A* ⊏′ *set ys*)
⟨*proof*⟩

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* ⟷ (∃ *A*. *set xs* ⊏′ *A* ∧ *A* ⊏′ − *set ys*)
⟨*proof*⟩

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* ⟷ (∃ *A*. − *set xs* ⊏′ *A* ∧ *A* ⊏′ *set ys*)
⟨*proof*⟩

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 ⟨*proof*⟩
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 ⟨*proof*⟩
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*) ⟷ *x* > *0*
| *proper-interval-nat* (*Some x*) (*Some y*) ⟷ *y* − *x* > *1*
instance ⟨*proof*⟩

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 ⟨*proof*⟩
end

instantiation *integer* :: *proper-interval* **begin**
context includes *integer.lifting* **begin**
lift-definition *proper-interval-integer* :: *integer proper-interval* **is** *proper-interval*
⟨*proof*⟩
instance ⟨*proof*⟩
end
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*
⟨*proof*⟩

instantiation *natural* :: *proper-interval* **begin**
context includes *natural.lifting* **begin**
lift-definition *proper-interval-natural* :: *natural proper-interval* **is** *proper-interval*
⟨*proof*⟩
instance ⟨*proof*⟩
end
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*
⟨*proof*⟩

lemma *char-less-iff-nat-of-char*: *x* < *y* $\longleftrightarrow$ *of-char x* < (*of-char y* :: *nat*)
  ⟨*proof*⟩

lemma *nat-of-char-inject* [*simp*]: *of-char x* = (*of-char y* :: *nat*) $\longleftrightarrow$ *x* = *y*
  ⟨*proof*⟩

lemma *char-le-iff-nat-of-char*: *x* ≤ *y* $\longleftrightarrow$ *of-char x* ≤ (*of-char y* :: *nat*)
  ⟨*proof*⟩

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*⟩

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 = *None* $\wedge$ y = *None*
instance ⟨*proof*⟩
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 $\longleftrightarrow$ *True*
| *proper-interval-finite-2 None* (*Some x*) $\longleftrightarrow$ x = *finite-2*.a_2
| *proper-interval-finite-2* (*Some x*) *None* $\longleftrightarrow$ x = *finite-2*.a_1
| *proper-interval-finite-2* (*Some x*) (*Some y*) $\longleftrightarrow$ *False*
instance ⟨*proof*⟩
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 $\longleftrightarrow$ *True*
| *proper-interval-finite-3 None* (*Some x*) $\longleftrightarrow$ $x \neq$ *finite-3*.a_1
| *proper-interval-finite-3* (*Some x*) *None* $\longleftrightarrow$ $x \neq$ *finite-3*.a_3
| *proper-interval-finite-3* (*Some x*) (*Some y*) $\longleftrightarrow$ x = *finite-3*.a_1 $\wedge$ y = *finite-3*.a_3
instance
⟨*proof*⟩
end

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

definition *length-last-fusion* :: (*'a*, *'s*) *generator* $\Rightarrow$ *'s* $\Rightarrow$ *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*, *undefined*))
⟨*proof*⟩

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

context *proper-intrvl* **begin**

definition *set-less-eq-aux-Compl-fusion* :: *('a, 's1) generator* ⇒ *('a, 's2) generator* ⇒ *'a option* ⇒ *'s1* ⇒ *'s2* ⇒ *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* ⇒ *('a, 's2) generator* ⇒ *'a option* ⇒ *'s1* ⇒ *'s2* ⇒ *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* ⇒ *('a, 's2) generator* ⇒ *'a option* ⇒ *'s1* ⇒ *'s2* ⇒ *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* ⇒ *('a, 's2) generator* ⇒ *'a option* ⇒ *'s1* ⇒ *'s2* ⇒ *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* ⇒ *'a* ⇒ *'s* ⇒ *bool*
where *exhaustive-above-fusion g a s* = *exhaustive-above a* (*list.unfoldr g s*)

definition *exhaustive-fusion* :: *('a, 's) generator* ⇒ *'s* ⇒ *bool*
where *exhaustive-fusion g s* = *exhaustive* (*list.unfoldr g s*)

definition *proper-interval-set-aux-fusion* :: *('a, 's1) generator* ⇒ *('a, 's2) generator* ⇒ *'s1* ⇒ *'s2* ⇒ *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 ⇒ *('a, 's2) generator* ⇒ *'a option* ⇒ *nat* ⇒ *'s1* ⇒ *'s2* ⇒ *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 ⇒ *('a, 's2) generator* ⇒ *'a option* ⇒ *'s1* ⇒ *'s2* ⇒ *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*) ∨ *set-less-eq-aux-Compl-fusion g1 g2* (*Some x*) *s1' s2*
else if y < *x then proper-interval ao* (*Some y*) ∨ *set-less-eq-aux-Compl-fusion g1 g2* (*Some y*) *s1 s2'*
else proper-interval ao (*Some y*)))
⟨*proof*⟩

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* ¬ *proper-interval ao* (*Some x*) ∧ *Compl-set-less-eq-aux-fusion g1 g2* (*Some x*) *s1' s2*
else if y < *x then* ¬ *proper-interval ao* (*Some y*) ∧ *Compl-set-less-eq-aux-fusion g1 g2* (*Some y*) *s1 s2'*
else ¬ *proper-interval ao* (*Some y*)
else ¬ *proper-interval ao* (*Some x*) ∧ *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 ¬ *proper-interval ao* (*Some y*) ∧ *Compl-set-less-eq-aux-fusion g1 g2* (*Some y*) *s1 s2'*
else ¬ *proper-interval ao None*)
⟨*proof*⟩

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*) ∨ *set-less-aux-Compl-fusion g1 g2* (*Some x*) *s1' s2*
else if y < *x then proper-interval ao* (*Some y*) ∨ *set-less-aux-Compl-fusion g1 g2* (*Some y*) *s1 s2'*
else proper-interval ao (*Some y*)
else proper-interval ao (*Some x*) ∨ *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*) ∨ *set-less-aux-Compl-fusion g1 g2* (*Some y*) *s1 s2′*
else proper-interval ao None)
⟨*proof*⟩

lemma *Compl-set-less-aux-fusion-code*:
Compl-set-less-aux-fusion g1 g2 ao s1 s2 ⟷
list.has-next g1 s1 ∧ *list.has-next g2 s2* ∧
(*let* (*x*, *s1′*) = *list.next g1 s1*;
(*y*, *s2′*) = *list.next g2 s2*
in if x < *y then* ¬ *proper-interval ao* (*Some x*) ∧ *Compl-set-less-aux-fusion g1 g2* (*Some x*) *s1′ s2*
else if y < *x then* ¬ *proper-interval ao* (*Some y*) ∧ *Compl-set-less-aux-fusion g1 g2* (*Some y*) *s1 s2′*
else ¬ *proper-interval ao* (*Some y*))
⟨*proof*⟩

lemma *exhaustive-above-fusion-code*:
exhaustive-above-fusion g y s ⟷
(*if list.has-next g s then*
let (*x*, *s′*) = *list.next g s*
in ¬ *proper-interval* (*Some y*) (*Some x*) ∧ *exhaustive-above-fusion g x s′*
else ¬ *proper-interval* (*Some y*) *None*)
⟨*proof*⟩

lemma *exhaustive-fusion-code*:
exhaustive-fusion g s =
(*list.has-next g s* ∧
(*let* (*x*, *s′*) = *list.next g s*
in ¬ *proper-interval None* (*Some x*) ∧ *exhaustive-above-fusion g x s′*))
⟨*proof*⟩

lemma *proper-interval-set-aux-fusion-code*:
proper-interval-set-aux-fusion g1 g2 s1 s2 ⟷
list.has-next g2 s2 ∧
(*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*) ∨ *list.has-next g2 s2′* ∨ ¬ *exhaustive-above-fusion g1 x s1′*
else proper-interval-set-aux-fusion g1 g2 s1′ s2′
else list.has-next g2 s2′ ∨ *proper-interval* (*Some y*) *None*)
⟨*proof*⟩

lemma *proper-interval-set-Compl-aux-fusion-code*:
proper-interval-set-Compl-aux-fusion g1 g2 ao n s1 s2 ⟷
(*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) ∨
                proper-interval-set-Compl-aux-fusion g1 g2 (Some x) (n + 1) s1' s2
              else if y < x then
                proper-interval ao (Some y) ∨
                proper-interval-set-Compl-aux-fusion g1 g2 (Some y) (n + 1) s1 s2'
              else
                proper-interval ao (Some x) ∧
                (let m = CARD('a) − n
                 in m − length-fusion g2 s2' ≠ 2 ∨ m − length-fusion g1 s1' ≠ 2)
         else
           let m = CARD('a) − n; (len-x, x') = length-last-fusion g1 s1
           in m ≠ len-x ∧ (m = len-x + 1 ⟶ ¬ 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 ≠ len-y ∧ (m = len-y + 1 ⟶ ¬ proper-interval (Some y') None)
    else CARD('a) > n + 1)
⟨proof⟩
```

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

```
  proper-interval-Compl-set-aux-fusion g1 g2 ao s1 s2 ⟷
    list.has-next g1 s1 ∧ list.has-next g2 s2 ∧
    (let (x, s1') = list.next g1 s1;
         (y, s2') = list.next g2 s2
     in if x < y then
          ¬ proper-interval ao (Some x) ∧ proper-interval-Compl-set-aux-fusion g1 g2
(Some x) s1' s2
        else if y < x then
          ¬ proper-interval ao (Some y) ∧ proper-interval-Compl-set-aux-fusion g1 g2
(Some y) s1 s2'
        else ¬ proper-interval ao (Some x) ∧ (list.has-next g2 s2' ∨ list.has-next g1
s1'))
⟨proof⟩
```

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*⟩

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*: ⋀ *comp. ccompare* = *Some comp* ⟹ *comparator comp*
begin
abbreviation *ccomp* :: *'a comparator* **where** *ccomp* ≡ *the* (*ID ccompare*)
abbreviation *cless* :: *'a* ⇒ *'a* ⇒ *bool* **where** *cless* ≡ *lt-of-comp* (*the* (*ID ccompare*))
abbreviation *cless-eq* :: *'a* ⇒ *'a* ⇒ *bool* **where** *cless-eq* ≡ *le-of-comp* (*the* (*ID ccompare*))
end

lemma (**in** *ccompare*) *ID-ccompare'*:
  ⋀*c. ID ccompare* = *Some c* ⟹ *comparator c*
  ⟨*proof*⟩

lemma (**in** *ccompare*) *ID-ccompare*:
  ⋀*c. ID ccompare* = *Some c* ⟹ *class.linorder* (*le-of-comp c*) (*lt-of-comp c*)
  ⟨*proof*⟩

syntax *-CCOMPARE* :: *type* => *logic* (‹(*1CCOMPARE*/(*1'*(-')))›)

syntax-consts *-CCOMPARE* == *ccompare*

⟨*ML*⟩

definition *is-ccompare* :: *'a* :: *ccompare itself* ⇒ *bool*
where *is-ccompare* - ⟷ *ID CCOMPARE*(*'a*) ≠ *None*

context *ccompare*
begin

lemma *cless-eq-conv-cless*:
  fixes $a\ b :: 'a$
  assumes $ID\ CCOMPARE('a) \neq None$
  shows $cless\text{-}eq\ a\ b \longleftrightarrow cless\ a\ b \vee a = b$
⟨*proof*⟩
end

3.1.2 Generator for the *ccompare*–class

This generator registers itself at the derive-manager for the class *ccompare*. 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*: $class.linorder\ le\ lt \Longrightarrow a = Some\ (le,\ lt) \Longrightarrow a = Some\ (le',lt') \Longrightarrow$
  $class.linorder\ le'\ lt'$ ⟨*proof*⟩

lemma *comparator-subst*: $c1 = c2 \Longrightarrow comparator\ c1 \Longrightarrow comparator\ c2$ ⟨*proof*⟩

lemma (**in** *compare*) *compare-subst*: $\bigwedge\ comp.\ compare = comp \Longrightarrow comparator\ comp$
  ⟨*proof*⟩

⟨*ML*⟩

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*]: ¬ *is-ccompare TYPE*(*'a* ⇒ *'b*)
⟨*proof*⟩

instantiation *set* :: (*ccompare*) *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 ⟨*proof*⟩
end

lemma *is-ccompare-set* [*simp*, *code-post*]:
is-ccompare TYPE(*'a set*) ⟷ *is-ccompare TYPE*(*'a* :: *ccompare*)
⟨*proof*⟩

definition *cless-eq-set* :: *'a* :: *ccompare set* ⇒ *'a set* ⇒ *bool*
where [*simp*, *code del*]: *cless-eq-set* = *le-of-comp* (*the* (*ID CCOMPARE*(*'a set*)))

definition *cless-set* :: *'a* :: *ccompare set* ⇒ *'a set* ⇒ *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* ⇒ *None* | *Some* - ⇒ *Some* (*comp-of-ords cless-eq-set cless-set*))
⟨*proof*⟩

derive (*no*) *ccompare Predicate.pred*

3.1.4 Proper intervals

class *cproper-interval* = *ccompare* +
fixes *cproper-interval* :: *'a option* ⇒ *'a option* ⇒ *bool*
assumes *cproper-interval*:
⟦ *ID CCOMPARE*(*'a*) ≠ *None*; *finite* (*UNIV* :: *'a set*) ⟧
⟹ *class.proper-interval cless cproper-interval*
begin

lemma *ID-ccompare-interval*:
⟦ *ID CCOMPARE('a) = Some c*; *finite (UNIV :: 'a set)* ⟧
⟹ *class.linorder-proper-interval (le-of-comp c) (lt-of-comp c) cproper-interval*
⟨*proof*⟩

end

instantiation *unit* :: *cproper-interval* **begin**
definition *cproper-interval* = (*proper-interval* :: *unit proper-interval*)
instance ⟨*proof*⟩
end

instantiation *bool* :: *cproper-interval* **begin**
definition *cproper-interval* = (*proper-interval* :: *bool proper-interval*)
instance ⟨*proof*⟩
end

instantiation *nat* :: *cproper-interval* **begin**
definition *cproper-interval* = (*proper-interval* :: *nat proper-interval*)
instance ⟨*proof*⟩
end

instantiation *int* :: *cproper-interval* **begin**
definition *cproper-interval* = (*proper-interval* :: *int proper-interval*)
instance ⟨*proof*⟩
end

instantiation *integer* :: *cproper-interval* **begin**
definition *cproper-interval* = (*proper-interval* :: *integer proper-interval*)
instance ⟨*proof*⟩
end

instantiation *natural* :: *cproper-interval* **begin**
definition *cproper-interval* = (*proper-interval* :: *natural proper-interval*)
instance ⟨*proof*⟩
end

instantiation *char* :: *cproper-interval* **begin**
definition *cproper-interval* = (*proper-interval* :: *char proper-interval*)
instance ⟨*proof*⟩
end

instantiation *Enum.finite-1* :: *cproper-interval* **begin**
definition *cproper-interval* = (*proper-interval* :: *Enum.finite-1 proper-interval*)
instance ⟨*proof*⟩
end

instantiation *Enum.finite-2* :: *cproper-interval* **begin**

definition *cproper-interval* = (*proper-interval* :: *Enum.finite-2 proper-interval*)
instance ⟨*proof*⟩
end

instantiation *Enum.finite-3* :: *cproper-interval* **begin**
definition *cproper-interval* = (*proper-interval* :: *Enum.finite-3 proper-interval*)
instance ⟨*proof*⟩
end

instantiation *Enum.finite-4* :: *cproper-interval* **begin**
definition (*cproper-interval* :: *Enum.finite-4 proper-interval*) - - = *undefined*
instance ⟨*proof*⟩
end

instantiation *Enum.finite-5* :: *cproper-interval* **begin**
definition (*cproper-interval* :: *Enum.finite-5 proper-interval*) - - = *undefined*
instance ⟨*proof*⟩
end

lemma *lt-of-comp-sum*: *lt-of-comp* (*comparator-sum ca cb*) *sx sy* = (
  case sx of Inl x ⇒ (*case sy of Inl y* ⇒ *lt-of-comp ca x y* | *Inr y* ⇒ *True*)
  | *Inr x* ⇒ (*case sy of Inl y* ⇒ *False* | *Inr y* ⇒ *lt-of-comp cb x y*))
  ⟨*proof*⟩

instantiation *sum* :: (*cproper-interval*, *cproper-interval*) *cproper-interval* **begin**
fun *cproper-interval-sum* :: (*'a* + *'b*) *proper-interval* **where**
  cproper-interval-sum None None ⟷ *True*
| *cproper-interval-sum None* (*Some* (*Inl x*)) ⟷ *cproper-interval None* (*Some x*)
| *cproper-interval-sum None* (*Some* (*Inr y*)) ⟷ *True*
| *cproper-interval-sum* (*Some* (*Inl x*)) *None* ⟷ *True*
| *cproper-interval-sum* (*Some* (*Inl x*)) (*Some* (*Inl y*)) ⟷ *cproper-interval* (*Some x*) (*Some y*)
| *cproper-interval-sum* (*Some* (*Inl x*)) (*Some* (*Inr y*)) ⟷ *cproper-interval* (*Some x*) *None* ∨ *cproper-interval None* (*Some y*)
| *cproper-interval-sum* (*Some* (*Inr y*)) *None* ⟷ *cproper-interval* (*Some y*) *None*
| *cproper-interval-sum* (*Some* (*Inr y*)) (*Some* (*Inl x*)) ⟷ *False*
| *cproper-interval-sum* (*Some* (*Inr x*)) (*Some* (*Inr y*)) ⟷ *cproper-interval* (*Some x*) (*Some y*)
instance
⟨*proof*⟩
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*)
  ⟨*proof*⟩

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* ∨ *le-of-comp c-a x1 y1* ∧ *lt-of-comp c-b x2 y2*)

⟨*proof*⟩

instantiation *prod* :: (*cproper-interval*, *cproper-interval*) *cproper-interval* **begin**
fun *cproper-interval-prod* :: (′*a* × ′*b*) *proper-interval* **where**
cproper-interval-prod None None ⟷ *True*
| *cproper-interval-prod None* (*Some* (*y1*, *y2*)) ⟷ *cproper-interval None* (*Some y1*) ∨ *cproper-interval None* (*Some y2*)
| *cproper-interval-prod* (*Some* (*x1*, *x2*)) *None* ⟷ *cproper-interval* (*Some x1*) *None* ∨ *cproper-interval* (*Some x2*) *None*
| *cproper-interval-prod* (*Some* (*x1*, *x2*)) (*Some* (*y1*, *y2*)) ⟷
cproper-interval (*Some x1*) (*Some y1*) ∨
cless x1 y1 ∧ (*cproper-interval* (*Some x2*) *None* ∨ *cproper-interval None* (*Some y2*)) ∨
¬ *cless y1 x1* ∧ *cproper-interval* (*Some x2*) (*Some y2*)
instance
⟨*proof*⟩
end

instantiation *list* :: (*ccompare*) *cproper-interval* **begin**
definition *cproper-interval-list* :: ′*a list proper-interval*
where *cproper-interval-list xso yso* = *undefined*
instance ⟨*proof*⟩
end

lemma *infinite-UNIV-literal*:
infinite (*UNIV* :: *String.literal set*)
⟨*proof*⟩

instantiation *String.literal* :: *cproper-interval* **begin**
definition *cproper-interval-literal* :: *String.literal proper-interval*
where *cproper-interval-literal xso yso* = *undefined*
instance ⟨*proof*⟩
end

lemma *lt-of-comp-option*: *lt-of-comp* (*comparator-option c*) *sx sy* = (
case sx of None ⇒ (*case sy of None* ⇒ *False* | *Some y* ⇒ *True*)
| *Some x* ⇒ (*case sy of None* ⇒ *False* | *Some y* ⇒ *lt-of-comp c x y*))
⟨*proof*⟩

instantiation *option* :: (*cproper-interval*) *cproper-interval* **begin**
fun *cproper-interval-option* :: ′*a option proper-interval* **where**
cproper-interval-option None None ⟷ *True*
| *cproper-interval-option None* (*Some x*) ⟷ *x* ≠ *None*
| *cproper-interval-option* (*Some x*) *None* ⟷ *cproper-interval x None*
| *cproper-interval-option* (*Some x*) (*Some None*) ⟷ *False*
| *cproper-interval-option* (*Some x*) (*Some* (*Some y*)) ⟷ *cproper-interval x* (*Some y*)

instance
⟨*proof*⟩
end

instantiation *set* :: (*cproper-interval*) *cproper-interval* **begin**
fun *cproper-interval-set* :: *'a set proper-interval* **where**
[*code*]: *cproper-interval-set None None* ⟷ *True*
| [*code*]: *cproper-interval-set None* (*Some B*) ⟷ ($B \neq \{\}$)
| [*code*]: *cproper-interval-set* (*Some A*) *None* ⟷ ($A \neq UNIV$)
| *cproper-interval-set-Some-Some* [*code del*]: — Refine for concrete implementations
cproper-interval-set (*Some A*) (*Some B*) ⟷ *finite* (*UNIV* :: *'a set*) ∧ (∃ *C*. *cless A C* ∧ *cless C B*)
instance
⟨*proof*⟩

lemma *Complement-cproper-interval-set-Complement*:
fixes *A B* :: *'a set*
assumes *corder*: *ID CCOMPARE*(*'a*) ≠ *None*
shows *cproper-interval* (*Some* (− *A*)) (*Some* (− *B*)) = *cproper-interval* (*Some B*) (*Some A*)
⟨*proof*⟩

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* ⇒ *'b*) *proper-interval*)
instance ⟨*proof*⟩
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* ⟷ *xs* ≠ []
⟨*proof*⟩

lemma *less-append-same-iff*:

fixes xs :: $'a$ :: $preorder\ list$
shows $xs < xs\ @\ ys \longleftrightarrow [] < ys$
⟨*proof*⟩

lemma *less-append-same2-iff*:
fixes xs :: $'a$ :: $preorder\ list$
shows $xs\ @\ ys < xs\ @\ zs \longleftrightarrow ys < zs$
⟨*proof*⟩

lemma *Cons-less-iff*:
fixes x :: $'a$ :: *preorder* **shows**
$x \# xs < ys \longleftrightarrow (\exists\, y\ ys'.\ ys = y \# ys' \land (x < y \lor x = y \land xs < ys'))$
⟨*proof*⟩

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 $xs\ ys \longleftrightarrow (\exists\, zs.\ xs < zs \land zs < ys)$

lemma *proper-interval-list-aux-simps* [*code*]:
proper-interval-list-aux $xs\ [] \longleftrightarrow False$
proper-interval-list-aux $[]\ (y \# ys) \longleftrightarrow ys \neq [] \lor$ *proper-interval None* (*Some* y)
proper-interval-list-aux $(x \# xs)\ (y \# ys) \longleftrightarrow x < y \lor x = y \land$ *proper-interval-list-aux* $xs\ ys$
⟨*proof*⟩

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*⟩
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* :: $('a \Rightarrow 'a \Rightarrow bool)\ option$
assumes *ceq*: $ceq = Some\ eq \Longrightarrow eq = (=)$

begin

lemma *ceq-equality*: *ceq* = *Some eq* ⟹ *equality eq*
⟨*proof*⟩

lemma *ID-ceq*: *ID ceq* = *Some eq* ⟹ *eq* = (=)
⟨*proof*⟩

abbreviation *ceq′* :: *′a* ⇒ *′a* ⇒ *bool* **where** *ceq′* ≡ *the* (*ID ceq*)

end

syntax -*CEQ* :: *type* => *logic* (‹(*1CEQ*/(*1′*(-′)))›)

syntax-consts -*CEQ* == *ceq*

⟨*ML*⟩

definition *is-ceq* :: *′a* :: *ceq itself* ⇒ *bool*
where *is-ceq* - ⟷ *ID CEQ*(*′a*) ≠ *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* ⟹ *equality c1* ⟹ *equality c2* ⟨*proof*⟩

⟨*ML*⟩

3.3.2 Type class instances for HOL types

derive (*eq*) *ceq unit*
lemma [*code*]: *CEQ*(*unit*) = *Some* (λ- -. *True*)

⟨*proof*⟩
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*]: ¬ *is-ceq TYPE*(*'a* ⇒ *'b*)
⟨*proof*⟩

definition *set-eq* :: *'a set* ⇒ *'a set* ⇒ *bool*
where [*code del*]: *set-eq* = (=)

lemma *set-eq-code*:
shows [*code*]: *set-eq A B* ⟷ *A* ⊆ *B* ∧ *B* ⊆ *A*
and [*code-unfold*]: (=) = *set-eq*
⟨*proof*⟩

instantiation *set* :: (*ceq*) *ceq* **begin**
definition *CEQ*(*'a set*) = (*case ID CEQ*(*'a*) *of None* ⇒ *None* | *Some* - ⇒ *Some set-eq*)
instance ⟨*proof*⟩
end

lemma *is-ceq-set* [*simp*, *code-post*]: *is-ceq TYPE*(*'a set*) ⟷ *is-ceq TYPE*(*'a* :: *ceq*)
⟨*proof*⟩

lemma *ID-ceq-set-not-None-iff* [*simp*]: *ID CEQ*(*'a set*) ≠ *None* ⟷ *ID CEQ*(*'a* :: *ceq*) ≠ *None*
⟨*proof*⟩

Instantiation for *'a Predicate.pred*

context fixes *eq* :: *'a* ⇒ *'a* ⇒ *bool* **begin**

definition *member-pred* :: *'a Predicate.pred* ⇒ *'a* ⇒ *bool*
where *member-pred P x* ⟷ (∃ *y*. *eq x y* ∧ *Predicate.eval P y*)

definition *member-seq* :: *'a Predicate.seq* ⇒ *'a* ⇒ *bool*

where *member-seq xp* = *member-pred* (*Predicate.pred-of-seq xp*)

lemma *member-seq-code* [*code*]:
  member-seq seq.Empty x ⟷ *False*
  member-seq (*seq.Insert y P*) *x* ⟷ *eq x y* ∨ *member-pred P x*
  member-seq (*seq.Join Q xq*) *x* ⟷ *member-pred Q x* ∨ *member-seq xq x*
⟨*proof*⟩

lemma *member-pred-code* [*code*]:
  member-pred (*Predicate.Seq f*) = *member-seq* (*f* ())
⟨*proof*⟩

definition *leq-pred* :: *'a Predicate.pred* ⇒ *'a Predicate.pred* ⇒ *bool*
where *leq-pred P Q* ⟷ (∀ *x*. *Predicate.eval P x* ⟶ (∃ *y*. *eq x y* ∧ *Predicate.eval Q y*))

definition *leq-seq* :: *'a Predicate.seq* ⇒ *'a Predicate.pred* ⇒ *bool*
where *leq-seq xp Q* ⟷ *leq-pred* (*Predicate.pred-of-seq xp*) *Q*

lemma *leq-seq-code* [*code*]:
  leq-seq seq.Empty Q ⟷ *True*
  leq-seq (*seq.Insert x P*) *Q* ⟷ *member-pred Q x* ∧ *leq-pred P Q*
  leq-seq (*seq.Join P xp*) *Q* ⟷ *leq-pred P Q* ∧ *leq-seq xp Q*
⟨*proof*⟩

lemma *leq-pred-code* [*code*]:
  leq-pred (*Predicate.Seq f*) *Q* ⟷ *leq-seq* (*f* ()) *Q*
⟨*proof*⟩

definition *predicate-eq* :: *'a Predicate.pred* ⇒ *'a Predicate.pred* ⇒ *bool*
where *predicate-eq P Q* ⟷ *leq-pred P Q* ∧ *leq-pred Q P*

context assumes *eq*: *eq* = (=) **begin**

lemma *member-pred-eq*: *member-pred* = *Predicate.eval*
⟨*proof*⟩

lemma *member-seq-eq*: *member-seq* = *Predicate.member*
⟨*proof*⟩

lemma *leq-pred-eq*: *leq-pred* = (≤)
⟨*proof*⟩

lemma *predicate-eq-eq*: *predicate-eq* = (=)
⟨*proof*⟩

end
end

instantiation *Predicate.pred* :: (*ceq*) *ceq* **begin**
definition *CEQ*(*'a Predicate.pred*) = *map-option predicate-eq* (*ID CEQ*(*'a*))
instance ⟨*proof*⟩
end

end

theory *Collection-Enum* **imports**
  Containers-Auxiliary
  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*
⟨*proof*⟩

lemma *in-cenum*: *ID cEnum* = *Some* (*enum*, *rest*) ⟹ *f* ∈ *set enum*
⟨*proof*⟩

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*

⟨*ML*⟩

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*⟩

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* (λ*x. all-n-lists* (λ*xs. 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 ⟷ (*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* ⟷ (∀ *xs* ∈ *set* (*List.n-lists n cenum*). *P xs*)
⟨*proof*⟩

lemma *ex-n-lists-iff*: **fixes** *cenum* **shows**
ex-n-lists (*Bex* (*set cenum*)) *P n* ⟷ (∃ *xs* ∈ *set* (*List.n-lists n cenum*). *P xs*)
⟨*proof*⟩

instantiation *fun* :: (*cenum*, *cenum*) *cenum* **begin**

definition
CENUM(*'a* ⇒ *'b*) =
(*case ID CENUM*(*'a*) *of None* ⇒ *None* | *Some* (*enum-a*, *enum-all-a*, *enum-ex-a*) ⇒
case ID CENUM(*'b*) *of None* ⇒ *None* | *Some* (*enum-b*, *enum-all-b*, *enum-ex-b*) ⇒ *Some*

(*map* (*λys. the o map-of* (*zip enum-a ys*)) (*List.n-lists* (*length enum-a*) *enum-b*),
λP. all-n-lists enum-all-b (*λbs. P* (*the o map-of* (*zip enum-a bs*))) (*length enum-a*),
λP. ex-n-lists enum-ex-b (*λbs. P* (*the o map-of* (*zip enum-a bs*))) (*length enum-a*)))
instance ⟨*proof*⟩
end

instantiation *set* :: (*cenum*) *cenum* **begin**
definition
CENUM(*'a set*) =
(*case ID CENUM*(*'a*) *of None* ⇒ *None* | *Some* (*enum-a*, *enum-all-a*, *enum-ex-a*) ⇒ *Some*
(*map set* (*subseqs enum-a*),
λP. list-all P (*map set* (*subseqs enum-a*)),
λP. list-ex P (*map set* (*subseqs enum-a*))))
instance
⟨*proof*⟩
end

instantiation *unit* :: *cenum* **begin**
definition *CENUM*(*unit*) = *Some* (*enum-class.enum*, *enum-class.enum-all*, *enum-class.enum-ex*)
instance ⟨*proof*⟩
end

instantiation *bool* :: *cenum* **begin**
definition *CENUM*(*bool*) = *Some* (*enum-class.enum*, *enum-class.enum-all*, *enum-class.enum-ex*)
instance ⟨*proof*⟩
end

instantiation *prod* :: (*cenum*, *cenum*) *cenum* **begin**
definition
CENUM(*'a* × *'b*) =
(*case ID CENUM*(*'a*) *of None* ⇒ *None* | *Some* (*enum-a*, *enum-all-a*, *enum-ex-a*) ⇒
case ID CENUM(*'b*) *of None* ⇒ *None* | *Some* (*enum-b*, *enum-all-b*, *enum-ex-b*) ⇒ *Some*
(*List.product enum-a enum-b*,
λP. enum-all-a (%*x. enum-all-b* (%*y. P* (*x*, *y*))),
λP. enum-ex-a (%*x. enum-ex-b* (%*y. P* (*x*, *y*)))))
instance
⟨*proof*⟩
end

instantiation *sum* :: (*cenum*, *cenum*) *cenum* **begin**
definition
CENUM(*'a* + *'b*) =
(*case ID CENUM*(*'a*) *of None* ⇒ *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 Inl enum-a* @ *map Inr enum-b*,
     λP. *enum-all-a* (λx. *P* (*Inl x*)) $\wedge$ *enum-all-b* (λx. *P* (*Inr x*)),
     λP. *enum-ex-a* (λx. *P* (*Inl x*)) $\vee$ *enum-ex-b* (λx. *P* (*Inr x*))))
instance
 ⟨*proof*⟩
end

instantiation *option* :: (*cenum*) *cenum* **begin**
definition
 CENUM('a option) =
 (*case ID CENUM('a) of None* $\Rightarrow$ *None* | *Some* (*enum-a*, *enum-all-a*, *enum-ex-a*)
$\Rightarrow$ *Some*
  (*None* # *map Some enum-a*,
   λP. *P None* $\wedge$ *enum-all-a* (λx. *P* (*Some x*)),
   λP. *P None* $\vee$ *enum-ex-a* (λx. *P* (*Some x*))))
instance
 ⟨*proof*⟩
end

instantiation *Enum.finite-1* :: *cenum* **begin**
definition *CENUM(Enum.finite-1)* = *Some* (*enum-class.enum*, *enum-class.enum-all*, *enum-class.enum-ex*)
instance ⟨*proof*⟩
end

instantiation *Enum.finite-2* :: *cenum* **begin**
definition *CENUM(Enum.finite-2)* = *Some* (*enum-class.enum*, *enum-class.enum-all*, *enum-class.enum-ex*)
instance ⟨*proof*⟩
end

instantiation *Enum.finite-3* :: *cenum* **begin**
definition *CENUM(Enum.finite-3)* = *Some* (*enum-class.enum*, *enum-class.enum-all*, *enum-class.enum-ex*)
instance ⟨*proof*⟩
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 ⟨*proof*⟩
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 ⟨*proof*⟩
end

instantiation *char* :: *cenum* **begin**
definition *CENUM*(*char*) = *Some* (*enum-class.enum*, *enum-class.enum-all*, *enum-class.enum-ex*)
instance ⟨*proof*⟩
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* :: $'a \Rightarrow 'a \Rightarrow$ *bool*

locale *equal* = *equal-base* +
assumes *equal-eq*: *equal* = (=)
begin

lemma *equal-conv-eq*: *equal x y* $\longleftrightarrow$ *x* = *y*
⟨*proof*⟩

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 $\Longrightarrow$ *is-rbt* (*RBT-Impl.fold rbt-insert t′ t*)
⟨*proof*⟩

lemma *rbt-sorted-fold-insert*: *rbt-sorted t* ⟹ *rbt-sorted* (*RBT-Impl.fold rbt-insert t′ t*)
⟨*proof*⟩

lemma *rbt-lookup-rbt-insert′*: *rbt-sorted t* ⟹ *rbt-lookup* (*rbt-insert k v t*) = (*rbt-lookup t*)(*k* ↦ *v*)
⟨*proof*⟩

lemma *rbt-lookup-fold-rbt-insert-impl*:
rbt-sorted t2 ⟹
rbt-lookup (*RBT-Impl.fold rbt-insert t1 t2*) = *rbt-lookup t2* ++ *map-of* (*rev* (*RBT-Impl.entries t1*))
⟨*proof*⟩

end

3.6.2 Build the cross product of two RBTs

context fixes *f* :: ′*a* ⇒ ′*b* ⇒ ′*c* ⇒ ′*d* ⇒ ′*e* **begin**

definition *alist-product* :: (′*a* × ′*b*) *list* ⇒ (′*c* × ′*d*) *list* ⇒ ((′*a* × ′*c*) × ′*e*) *list*
where *alist-product xs ys* = *concat* (*map* (λ(*a*, *b*). *map* (λ(*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* (λ(*c*, *d*). ((*a*, *c*), *f a b c d*)) *ys* @ *alist-product xs ys*
⟨*proof*⟩

lemma *append-alist-product-conv-fold*:
zs @ *alist-product xs ys* = *rev* (*fold* (λ(*a*, *b*). *fold* (λ(*c*, *d*) *rest*. ((*a*, *c*), *f a b c d*) # *rest*) *ys*) *xs* (*rev zs*))
⟨*proof*⟩

lemma *alist-product-code* [*code*]:
alist-product xs ys =
rev (*fold* (λ(*a*, *b*). *fold* (λ(*c*, *d*) *rest*. ((*a*, *c*), *f a b c d*) # *rest*) *ys*) *xs* [])
⟨*proof*⟩

lemma *set-alist-product*:
set (*alist-product xs ys*) =
(λ((*a*, *b*), (*c*, *d*)). ((*a*, *c*), *f a b c d*)) ‘ (*set xs* × *set ys*)
⟨*proof*⟩

lemma *distinct-alist-product*:
⟦ *distinct* (*map fst xs*); *distinct* (*map fst ys*) ⟧
⟹ *distinct* (*map fst* (*alist-product xs ys*))

⟨*proof*⟩

lemma *map-of-alist-product*:
map-of (*alist-product xs ys*) (*a*, *c*) =
(*case map-of xs a of None* ⇒ *None*
| *Some b* ⇒ *map-option* (*f a b c*) (*map-of ys c*))
⟨*proof*⟩

definition *rbt-product* :: ('*a*, '*b*) *rbt* ⇒ ('*c*, '*d*) *rbt* ⇒ ('*a* × '*c*, '*e*) *rbt*
where
rbt-product rbt1 rbt2 = *rbtreeify* (*alist-product* (*RBT-Impl.entries rbt1*) (*RBT-Impl.entries rbt2*))

lemma *rbt-product-code* [*code*]:
rbt-product rbt1 rbt2 =
rbtreeify (*rev* (*RBT-Impl.fold* (λ*a b*. *RBT-Impl.fold* (λ*c d rest*. ((*a*, *c*), *f a b c d*) # *rest*) *rbt2*) *rbt1* []))
⟨*proof*⟩

end

context
fixes *leq-a* :: '*a* ⇒ '*a* ⇒ *bool* (**infix** ‹$\sqsubseteq_a$› *50*)
and *less-a* :: '*a* ⇒ '*a* ⇒ *bool* (**infix** ‹$\sqsubset_a$› *50*)
and *leq-b* :: '*b* ⇒ '*b* ⇒ *bool* (**infix** ‹$\sqsubseteq_b$› *50*)
and *less-b* :: '*b* ⇒ '*b* ⇒ *bool* (**infix** ‹$\sqsubset_b$› *50*)
assumes *lin-a*: *class.linorder leq-a less-a*
and *lin-b*: *class.linorder leq-b less-b*
begin

abbreviation (*input*) *less-eq-prod'* :: ('*a* × '*b*) ⇒ ('*a* × '*b*) ⇒ *bool* (**infix** ‹⊑› *50*)
where *less-eq-prod'* ≡ *less-eq-prod leq-a less-a leq-b*

abbreviation (*input*) *less-prod'* :: ('*a* × '*b*) ⇒ ('*a* × '*b*) ⇒ *bool* (**infix** ‹⊏› *50*)
where *less-prod'* ≡ *less-prod leq-a less-a less-b*

lemmas *linorder-prod* = *linorder-prod*[*OF lin-a lin-b*]

lemma *sorted-alist-product*:
assumes *xs*: *linorder.sorted leq-a* (*map fst xs*) *distinct* (*map fst xs*)
and *ys*: *linorder.sorted* ($\sqsubseteq_b$) (*map fst ys*)
shows *linorder.sorted* (⊑) (*map fst* (*alist-product f xs ys*))
⟨*proof*⟩

lemma *is-rbt-rbt-product*:
⟦ *ord.is-rbt* ($\sqsubset_a$) *rbt1*; *ord.is-rbt* ($\sqsubset_b$) *rbt2* ⟧
⟹ *ord.is-rbt* (⊏) (*rbt-product f rbt1 rbt2*)
⟨*proof*⟩

lemma *rbt-lookup-rbt-product*:
$[\![$ *ord.is-rbt* ($\sqsubset_a$) *rbt1*; *ord.is-rbt* ($\sqsubset_b$) *rbt2* $]\!]$
$\Longrightarrow$ *ord.rbt-lookup* ($\sqsubset$) (*rbt-product f rbt1 rbt2*) (*a*, *c*) =
(*case ord.rbt-lookup* ($\sqsubset_a$) *rbt1 a of None* $\Rightarrow$ *None*
| *Some b* $\Rightarrow$ *map-option* (*f a b c*) (*ord.rbt-lookup* ($\sqsubset_b$) *rbt2 c*))
⟨*proof*⟩

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* $\Rightarrow$ (′*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* (λ(*k*, *v*). ((*k*, *k*), *v*)) (*RBT-Impl.entries t*)
⟨*proof*⟩

lemma *keys-RBT-Impl-diag*:
RBT-Impl.keys (*RBT-Impl-diag t*) = *map* (λ*k*. (*k*, *k*)) (*RBT-Impl.keys t*)
⟨*proof*⟩

lemma *rbt-sorted-RBT-Impl-diag*:
ord.rbt-sorted lt t $\Longrightarrow$ *ord.rbt-sorted* (*less-prod leq lt lt*) (*RBT-Impl-diag t*)
⟨*proof*⟩

lemma *bheight-RBT-Impl-diag*:
bheight (*RBT-Impl-diag t*) = *bheight t*
⟨*proof*⟩

lemma *inv-RBT-Impl-diag*:
assumes *inv1 t* *inv2 t*
shows *inv1* (*RBT-Impl-diag t*) *inv2* (*RBT-Impl-diag t*)
and *color-of t* = *color.B* $\Longrightarrow$ *color-of* (*RBT-Impl-diag t*) = *color.B*
⟨*proof*⟩

lemma *is-rbt-RBT-Impl-diag*:
ord.is-rbt lt t $\Longrightarrow$ *ord.is-rbt* (*less-prod leq lt lt*) (*RBT-Impl-diag t*)
⟨*proof*⟩

lemma (**in** *linorder*) *rbt-lookup-RBT-Impl-diag*:
ord.rbt-lookup (*less-prod* (≤) (<) (<)) (*RBT-Impl-diag t*) =
(λ(*k*, *k*′). *if k* = *k*′ *then ord.rbt-lookup* (<) *t k else None*)

⟨*proof*⟩

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′*)))
⟨*proof*⟩

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* = ($\forall (k,\ v) \in$ *set* (*RBT-Impl.entries rbt*). *P k v*)

lemma *RBT-Impl-rbt-all-simps* [*simp*, *code*]:
RBT-Impl-rbt-all P rbt.Empty $\longleftrightarrow$ *True*
RBT-Impl-rbt-all P (*Branch c l k v r*) $\longleftrightarrow$ *P k v* $\wedge$ *RBT-Impl-rbt-all P l* $\wedge$ *RBT-Impl-rbt-all P r*
⟨*proof*⟩

definition *RBT-Impl-rbt-ex* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a,\ 'b)$ *rbt* $\Rightarrow$ *bool*
where [*code del*]: *RBT-Impl-rbt-ex P rbt* = ($\exists (k,\ v) \in$ *set* (*RBT-Impl.entries rbt*). *P k v*)

lemma *RBT-Impl-rbt-ex-simps* [*simp*, *code*]:
RBT-Impl-rbt-ex P rbt.Empty $\longleftrightarrow$ *False*
RBT-Impl-rbt-ex P (*Branch c l k v r*) $\longleftrightarrow$ *P k v* $\vee$ *RBT-Impl-rbt-ex P l* $\vee$ *RBT-Impl-rbt-ex P r*
⟨*proof*⟩

3.6.5 List fusion for RBTs

type-synonym $('a,\ 'b,\ 'c)$ *rbt-generator-state* = $('c \times ('a,\ 'b)$ *RBT-Impl.rbt*) *list* $\times$ $('a,\ 'b)$ *RBT-Impl.rbt*

fun *rbt-has-next* :: $('a,\ 'b,\ 'c)$ *rbt-generator-state* $\Rightarrow$ *bool*
where
rbt-has-next ([], *rbt.Empty*) = *False*
| *rbt-has-next* - = *True*

fun *rbt-keys-next* :: $('a,\ 'b,\ 'a)$ *rbt-generator-state* $\Rightarrow 'a \times ('a,\ 'b,\ 'a)$ *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 $\bigwedge k\ t\ kts.\ P\ (kts,\ t) \Longrightarrow P\ ((k,\ t)\ \#\ kts,\ rbt.Empty)$
and $\bigwedge kts\ c\ l\ k\ v\ r.\ P\ ((f\ k\ v,\ r)\ \#\ kts,\ l) \Longrightarrow P\ (kts,\ Branch\ c\ l\ k\ v\ r)$
shows *P ktst*
⟨*proof*⟩

lemma *terminates-rbt-keys-generator*:
terminates (*rbt-has-next*, *rbt-keys-next*)
⟨*proof*⟩

lift-definition *rbt-keys-generator* :: ('*a*, ('*a*, '*b*, '*a*) *rbt-generator-state*) *generator*
is (*rbt-has-next*, *rbt-keys-next*)
⟨*proof*⟩

definition *rbt-init* :: ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*, '*c*) *rbt-generator-state*
where *rbt-init* = *Pair* []

lemma *has-next-rbt-keys-generator* [*simp*]:
list.has-next rbt-keys-generator = *rbt-has-next*
⟨*proof*⟩

lemma *next-rbt-keys-generator* [*simp*]:
list.next rbt-keys-generator = *rbt-keys-next*
⟨*proof*⟩

lemma *unfoldr-rbt-keys-generator-aux*:
list.unfoldr rbt-keys-generator (*kts*, *t*) =
RBT-Impl.keys t @ *concat* (*map* (λ(*k*, *t*). *k* # *RBT-Impl.keys t*) *kts*)
⟨*proof*⟩

corollary *unfoldr-rbt-keys-generator*:
list.unfoldr rbt-keys-generator (*rbt-init t*) = *RBT-Impl.keys t*
⟨*proof*⟩

fun *rbt-entries-next* ::
('*a*, '*b*, '*a* × '*b*) *rbt-generator-state* ⇒ ('*a* × '*b*) × ('*a*, '*b*, '*a* × '*b*) *rbt-generator-state*
where
rbt-entries-next ((*kv*, *t*) # *kts*, *rbt.Empty*) = (*kv*, *kts*, *t*)
| *rbt-entries-next* (*kts*, *rbt.Branch c l k v r*) = *rbt-entries-next* (((*k*, *v*), *r*) # *kts*, *l*)

lemma *terminates-rbt-entries-generator*:
terminates (*rbt-has-next*, *rbt-entries-next*)
⟨*proof*⟩

lift-definition *rbt-entries-generator* :: ('*a* × '*b*, ('*a*, '*b*, '*a* × '*b*) *rbt-generator-state*) *generator*
is (*rbt-has-next*, *rbt-entries-next*)

⟨*proof*⟩

lemma *has-next-rbt-entries-generator* [*simp*]:
list.has-next rbt-entries-generator = rbt-has-next
⟨*proof*⟩

lemma *next-rbt-entries-generator* [*simp*]:
list.next rbt-entries-generator = rbt-entries-next
⟨*proof*⟩

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*⟩

corollary *unfoldr-rbt-entries-generator*:
list.unfoldr rbt-entries-generator (*rbt-init t*) = *RBT-Impl.entries t*
⟨*proof*⟩

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*: *distinct* (*map f xs*) ⟹ *distinct* (*map f* (*filter P xs*))
⟨*proof*⟩

lemma *map-of-filter-apply*:
distinct (*map fst xs*)
⟹ *map-of* (*filter P xs*) *k* =
(*case map-of xs k of None* ⇒ *None* | *Some v* ⇒ *if P* (*k*, *v*) *then Some v else None*)
⟨*proof*⟩

3.7.1 Type definition

typedef (**overloaded**) ('*a*, '*b*) *mapping-rbt*
= {*t* :: ('*a* :: *ccompare*, '*b*) *RBT-Impl.rbt*. *ord.is-rbt cless t* ∨ *ID CCOMPARE*('*a*) = *None*}
morphisms *impl-of Mapping-RBT'*
⟨*proof*⟩

definition *Mapping-RBT* :: ('*a* :: *ccompare*, '*b*) *rbt* ⇒ ('*a*, '*b*) *mapping-rbt*
where
Mapping-RBT t = *Mapping-RBT'*
(*if ord.is-rbt cless t* ∨ *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* ∈ {*t*. *ord.is-rbt cless t* ∨ *ID CCOMPARE*('*a*) = *None*}
shows *impl-of* (*Mapping-RBT y*) = *y*
⟨*proof*⟩

lemma *impl-of-inverse*: *Mapping-RBT* (*impl-of t*) = *t*
⟨*proof*⟩

lemma *type-definition-mapping-rbt'*:
type-definition impl-of Mapping-RBT
{*t* :: ('*a*, '*b*) *rbt*. *ord.is-rbt cless t* ∨ *ID CCOMPARE*('*a* :: *ccompare*) = *None*}
⟨*proof*⟩

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* ⟷ *impl-of t1* = *impl-of t2*
⟨*proof*⟩

lemma *rbt-eqI*:
impl-of t1 = *impl-of t2* ⟹ *t1* = *t2*
⟨*proof*⟩

lemma *Mapping-RBT-impl-of* [*simp*]:
Mapping-RBT (*impl-of t*) = *t*
⟨*proof*⟩

3.7.2 Operations

setup-lifting *type-definition-mapping-rbt'*

context fixes *dummy* :: '*a* :: *ccompare* **begin**

lift-definition *lookup* :: ('*a*, '*b*) *mapping-rbt* ⇒ '*a* ⇀ '*b* **is** *rbt-comp-lookup ccomp*
⟨*proof*⟩

lift-definition *empty* :: ('*a*, '*b*) *mapping-rbt* **is** *RBT-Impl.Empty*
⟨*proof*⟩

lift-definition *insert* :: $'a \Rightarrow 'b \Rightarrow ('a, 'b)$ *mapping-rbt* $\Rightarrow ('a, 'b)$ *mapping-rbt* **is**
rbt-comp-insert ccomp
⟨*proof*⟩

lift-definition *delete* :: $'a \Rightarrow ('a, 'b)$ *mapping-rbt* $\Rightarrow ('a, 'b)$ *mapping-rbt* **is**
rbt-comp-delete ccomp
⟨*proof*⟩

lift-definition *bulkload* :: $('a \times 'b)$ *list* $\Rightarrow ('a, 'b)$ *mapping-rbt* **is**
rbt-comp-bulkload ccomp
⟨*proof*⟩

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
⟨*proof*⟩

lift-definition *map* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a, 'b)$ *mapping-rbt* $\Rightarrow ('a, 'c)$ *mapping-rbt* **is** *RBT-Impl.map*
⟨*proof*⟩

lift-definition *entries* :: $('a, 'b)$ *mapping-rbt* $\Rightarrow ('a \times 'b)$ *list* **is** *RBT-Impl.entries*
⟨*proof*⟩

lift-definition *keys* :: $('a, 'b)$ *mapping-rbt* $\Rightarrow 'a$ *set* **is** *set* ∘ *RBT-Impl.keys* ⟨*proof*⟩

lift-definition *fold* :: $('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'c) \Rightarrow ('a, 'b)$ *mapping-rbt* $\Rightarrow 'c \Rightarrow 'c$ **is** *RBT-Impl.fold* ⟨*proof*⟩

lift-definition *is-empty* :: $('a, 'b)$ *mapping-rbt* $\Rightarrow$ *bool* **is** *case-rbt True* (λ- - - - -. *False*) ⟨*proof*⟩

lift-definition *filter* :: $('a \times 'b \Rightarrow$ *bool*$) \Rightarrow ('a, 'b)$ *mapping-rbt* $\Rightarrow ('a, 'b)$ *mapping-rbt* **is**
λ*P t. rbtreeify* (*List.filter P* (*RBT-Impl.entries t*))
⟨*proof*⟩

lift-definition *join* ::
$('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a, 'b)$ *mapping-rbt* $\Rightarrow ('a, 'b)$ *mapping-rbt* $\Rightarrow ('a, 'b)$ *mapping-rbt*
is *rbt-comp-union-with-key ccomp*
⟨*proof*⟩

lift-definition *meet* ::
$('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a, 'b)$ *mapping-rbt* $\Rightarrow ('a, 'b)$ *mapping-rbt* $\Rightarrow ('a, 'b)$ *mapping-rbt*
is *rbt-comp-inter-with-key ccomp*
⟨*proof*⟩

lift-definition *all* :: (′*a* ⇒ ′*b* ⇒ *bool*) ⇒ (′*a*, ′*b*) *mapping-rbt* ⇒ *bool*
is *RBT-Impl-rbt-all* ⟨*proof*⟩

lift-definition *ex* :: (′*a* ⇒ ′*b* ⇒ *bool*) ⇒ (′*a*, ′*b*) *mapping-rbt* ⇒ *bool*
is *RBT-Impl-rbt-ex* ⟨*proof*⟩

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*
⟨*proof*⟩

lift-definition *diag* ::
(′*a*, ′*b*) *mapping-rbt* ⇒ (′*a* × ′*a*, ′*b*) *mapping-rbt*
is *RBT-Impl-diag*
⟨*proof*⟩

lift-definition *init* :: (′*a*, ′*b*) *mapping-rbt* ⇒ (′*a*, ′*b*, ′*c*) *rbt-generator-state*
is *rbt-init* ⟨*proof*⟩

end

3.7.3 Properties

lemma *unfoldr-rbt-entries-generator*:
list.unfoldr rbt-entries-generator (*init t*) = *entries t*
⟨*proof*⟩

lemma *lookup-RBT*:
ord.is-rbt cless t ⟹
lookup (*Mapping-RBT t*) = *rbt-comp-lookup ccomp t*
⟨*proof*⟩

lemma *lookup-impl-of*:
rbt-comp-lookup ccomp (*impl-of t*) = *lookup t*
⟨*proof*⟩

lemma *entries-impl-of*:
RBT-Impl.entries (*impl-of t*) = *entries t*
⟨*proof*⟩

lemma *keys-impl-of*:
set (*RBT-Impl.keys* (*impl-of t*)) = *keys t*
⟨*proof*⟩

lemma *lookup-empty* [*simp*]:
lookup empty = *Map.empty*
⟨*proof*⟩

lemma *fold-conv-fold*:
fold f t = *List.fold* (*case-prod f*) (*entries t*)
⟨*proof*⟩

lemma *is-empty-empty* [*simp*]:
is-empty t ⟷ *t* = *empty*
⟨*proof*⟩

context assumes *ID-ccompare-neq-None*: *ID CCOMPARE*(′*a* :: *ccompare*) ≠ *None*
begin

lemma *mapping-linorder*: *class.linorder* (*cless-eq* :: ′*a* ⇒ ′*a* ⇒ *bool*) *cless*
⟨*proof*⟩

lemma *mapping-comparator*: *comparator* (*ccomp* :: ′*a comparator*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *lookup-insert* [*simp*]:
lookup (*insert* (*k* :: ′*a*) *v t*) = (*lookup t*)(*k* ↦ *v*)
⟨*proof*⟩

lemma *lookup-delete* [*simp*]:
lookup (*delete* (*k* :: ′*a*) *t*) = (*lookup t*)(*k* := *None*)
⟨*proof*⟩

lemma *map-of-entries* [*simp*]:
map-of (*entries* (*t* :: (′*a*, ′*b*) *mapping-rbt*)) = *lookup t*
⟨*proof*⟩

lemma *entries-lookup*:
entries (*t1* :: (′*a*, ′*b*) *mapping-rbt*) = *entries t2* ⟷ *lookup t1* = *lookup t2*
⟨*proof*⟩

lemma *lookup-bulkload* [*simp*]:
lookup (*bulkload xs*) = *map-of* (*xs* :: (′*a* × ′*b*) *list*)
⟨*proof*⟩

lemma *lookup-map-entry* [*simp*]:
lookup (*map-entry* (*k* :: ′*a*) *f t*) = (*lookup t*)(*k* := *map-option f* (*lookup t k*))
⟨*proof*⟩

lemma *lookup-map* [*simp*]:
lookup (*map f t*) (*k* :: *'a*) = *map-option* (*f k*) (*lookup t k*)
⟨*proof*⟩

lemma *RBT-lookup-empty* [*simp*]:
ord.rbt-lookup cless (*t* :: (*'a*, *'b*) *RBT-Impl.rbt*) = *Map.empty* ⟷ *t* = *RBT-Impl.Empty*
⟨*proof*⟩

lemma *lookup-empty-empty* [*simp*]:
lookup t = *Map.empty* ⟷ (*t* :: (*'a*, *'b*) *mapping-rbt*) = *empty*
⟨*proof*⟩

lemma *finite-dom-lookup* [*simp*]: *finite* (*dom* (*lookup* (*t* :: (*'a*, *'b*) *mapping-rbt*)))
⟨*proof*⟩

lemma *card-com-lookup* [*unfolded length-map*, *simp*]:
card (*dom* (*lookup* (*t* :: (*'a*, *'b*) *mapping-rbt*))) = *length* (*List.map fst* (*entries t*))
⟨*proof*⟩

lemma *lookup-join*:
lookup (*join f* (*t1* :: (*'a*, *'b*) *mapping-rbt*) *t2*) =
(*λk. case lookup t1 k of None* ⇒ *lookup t2 k* | *Some v1* ⇒ *Some* (*case lookup t2 k of None* ⇒ *v1* | *Some v2* ⇒ *f k v1 v2*))
⟨*proof*⟩

lemma *lookup-meet*:
lookup (*meet f* (*t1* :: (*'a*, *'b*) *mapping-rbt*) *t2*) =
(*λk. case lookup t1 k of None* ⇒ *None* | *Some v1* ⇒ *case lookup t2 k of None* ⇒ *None* | *Some v2* ⇒ *Some* (*f k v1 v2*))
⟨*proof*⟩

lemma *lookup-filter* [*simp*]:
lookup (*filter P* (*t* :: (*'a*, *'b*) *mapping-rbt*)) *k* =
(*case lookup t k of None* ⇒ *None* | *Some v* ⇒ *if P* (*k*, *v*) *then Some v else None*)
⟨*proof*⟩

lemma *all-conv-all-lookup*:
all P t ⟷ (∀(*k* :: *'a*) *v. lookup t k* = *Some v* ⟶ *P k v*)
⟨*proof*⟩

lemma *ex-conv-ex-lookup*:
ex P t ⟷ (∃(*k* :: *'a*) *v. lookup t k* = *Some v* ∧ *P k v*)
⟨*proof*⟩

lemma *diag-lookup*:
lookup (*diag t*) = (*λ*(*k* :: *'a*, *k'*). *if k* = *k' then lookup t k else None*)
⟨*proof*⟩

context assumes *ID-ccompare-neq-None'*: *ID CCOMPARE*(*'b* :: *ccompare*) ≠

None
begin

lemma *mapping-linorder'*: *class.linorder* (*cless-eq* :: *'b* ⇒ *'b* ⇒ *bool*) *cless*
⟨*proof*⟩

lemma *mapping-comparator'*: *comparator* (*ccomp* :: *'b comparator*)
⟨*proof*⟩

lemmas *rbt-comp'*[*simp*] = *rbt-comp-simps*[*OF mapping-comparator'*]

lemma *ccomp-comparator-prod*:
  ccomp = (*comparator-prod ccomp ccomp* :: (*'a* × *'b*)*comparator*)
  ⟨*proof*⟩

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*))
⟨*proof*⟩
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* ⇒ *'key* ⇒ (*'val* ⇒ *'val*) ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'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* ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'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* ⇒ *'a list* ⇒ (*nat* × *'a*) *list* **where**
zip-with-index-from n xs = *zip* [*n*..<*n* + *length xs*] *xs*

abbreviation *zip-with-index* :: *'a list* ⇒ (*nat* × *'a*) *list* **where**
zip-with-index ≡ *zip-with-index-from 0*

lemma *update-conv-update-with-aux*:
AList.update k v xs = *update-with-aux v k* (λ-. *v*) *xs*
⟨*proof*⟩

lemma *map-of-update-with-aux'*:
map-of (*update-with-aux v k f ps*) *k'* = ((*map-of ps*)(*k* ↦ (*case map-of ps k of None* ⇒ *f v* | *Some v* ⇒ *f v*))) *k'*
⟨*proof*⟩

lemma *map-of-update-with-aux*:
map-of (*update-with-aux v k f ps*) = (*map-of ps*)(*k* ↦ (*case map-of ps k of None* ⇒ *f v* | *Some v* ⇒ *f v*))
⟨*proof*⟩

lemma *dom-update-with-aux*: *fst* ‘ *set* (*update-with-aux v k f ps*) = {*k*} ∪ *fst* ‘ *set ps*
⟨*proof*⟩

lemma *distinct-update-with-aux* [*simp*]:
distinct (*map fst* (*update-with-aux v k f ps*)) = *distinct* (*map fst ps*)
⟨*proof*⟩

lemma *set-update-with-aux*:
distinct (*map fst xs*)
⟹ *set* (*update-with-aux v k f xs*) = (*set xs* − {*k*} × *UNIV* ∪ {(*k*, *f* (*case map-of xs k of None* ⇒ *v* | *Some v* ⇒ *v*))})
⟨*proof*⟩

lemma *set-delete-aux*: *distinct* (*map fst xs*) ⟹ *set* (*delete-aux k xs*) = *set xs* − {*k*} × *UNIV*
⟨*proof*⟩

lemma *dom-delete-aux*: *distinct* (*map fst ps*) ⟹ *fst* ‘ *set* (*delete-aux k ps*) = *fst* ‘ *set ps* − {*k*}
⟨*proof*⟩

lemma *distinct-delete-aux* [*simp*]:
distinct (*map fst ps*) ⟹ *distinct* (*map fst* (*delete-aux k ps*))
⟨*proof*⟩

lemma *map-of-delete-aux′*:
distinct (*map fst xs*) $\Longrightarrow$ *map-of* (*delete-aux k xs*) = (*map-of xs*)(*k* := *None*)
⟨*proof*⟩

lemma *map-of-delete-aux*:
distinct (*map fst xs*) $\Longrightarrow$ *map-of* (*delete-aux k xs*) *k′* = ((*map-of xs*)(*k* := *None*)) *k′*
⟨*proof*⟩

lemma *delete-aux-eq-Nil-conv*: *delete-aux k ts* = [] $\longleftrightarrow$ *ts* = [] ∨ (∃ *v*. *ts* = [(*k*, *v*)])
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *zip-with-index-from-conv-nth*:
zip-with-index-from n xs = *map* (λ*i*. (*n* + *i*, *xs* ! *i*)) [*0*..<*length xs*]
⟨*proof*⟩

lemma *map-of-zip-with-index-from* [*simp*]:
map-of (*zip-with-index-from n xs*) *i* = (*if i* ≥ *n* ∧ *i* < *n* + *length xs then Some* (*xs* ! (*i* − *n*)) *else None*)
⟨*proof*⟩

lemma *map-of-map′*: *map-of* (*map* (λ(*k*, *v*). (*k*, *f k v*)) *xs*) *x* = *map-option* (*f x*) (*map-of xs x*)
⟨*proof*⟩

3.8.2 Operations on the abstract type (′*a*, ′*b*) *alist*

lift-definition *update-with* :: ′*v* ⇒ ′*k* ⇒ (′*v* ⇒ ′*v*) ⇒ (′*k*, ′*v*) *alist* ⇒ (′*k*, ′*v*) *alist*
is *update-with-aux* ⟨*proof*⟩

lift-definition *delete* :: ′*k* ⇒ (′*k*, ′*v*) *alist* ⇒ (′*k*, ′*v*) *alist* **is** *delete-aux*
⟨*proof*⟩

lift-definition *keys* :: (′*k*, ′*v*) *alist* ⇒ ′*k set* **is** *set* ∘ *map fst* ⟨*proof*⟩

lift-definition *set* :: (′*key*, ′*val*) *alist* ⇒ (′*key* × ′*val*) *set*
is *List.set* ⟨*proof*⟩

lift-definition *map-values* :: $('key \Rightarrow 'val \Rightarrow 'val') \Rightarrow ('key, 'val)$ *alist* $\Rightarrow ('key, 'val')$ *alist* **is**
$\lambda f.$ *map* $(\lambda(x,y).\ (x, f\ x\ y))$
⟨*proof*⟩

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* | *Some v* $\Rightarrow$ *f v*)
⟨*proof*⟩

lemma *lookup-delete* [*simp*]: *DAList.lookup* (*delete k al*) = (*DAList.lookup al*)(*k* := *None*)
⟨*proof*⟩

lemma *finite-dom-lookup* [*simp*, *intro*!]: *finite* (*dom* (*DAList.lookup m*))
⟨*proof*⟩

lemma *update-conv-update-with*: *DAList.update k v* = *update-with v k* (λ-. *v*)
⟨*proof*⟩

lemma *lookup-update* [*simp*]: *DAList.lookup* (*DAList.update k v al*) = (*DAList.lookup al*)($k \mapsto v$)
⟨*proof*⟩

lemma *dom-lookup-keys*: *dom* (*DAList.lookup al*) = *keys al*
⟨*proof*⟩

lemma *keys-empty* [*simp*]: *keys DAList.empty* = {}
⟨*proof*⟩

lemma *keys-update-with* [*simp*]: *keys* (*update-with v k f al*) = *insert k* (*keys al*)
⟨*proof*⟩

lemma *keys-update* [*simp*]: *keys* (*DAList.update k v al*) = *insert k* (*keys al*)
⟨*proof*⟩

lemma *keys-delete* [*simp*]: *keys* (*delete k al*) = *keys al* − {*k*}
⟨*proof*⟩

lemma *set-empty* [*simp*]: *set DAList.empty* = {}
⟨*proof*⟩

lemma *set-update-with*:
set (*update-with v k f al*) =
(*set al* − {*k*} × *UNIV* ∪ {(*k*, *f* (*case DAList.lookup al k of None* $\Rightarrow$ *v* | *Some v* $\Rightarrow$ *v*))})
⟨*proof*⟩

lemma *set-update*: *set* (*DAList.update* *k* *v* *al*) = (*set* *al* − {*k*} × *UNIV* ∪ {(*k*, *v*)})
⟨*proof*⟩

lemma *set-delete*: *set* (*delete* *k* *al*) = *set* *al* − {*k*} × *UNIV*
⟨*proof*⟩

lemma *size-dalist-transfer* [*transfer-rule*]:
  includes *lifting-syntax*
  shows (*pcr-alist* (=) (=) ===> (=)) *length* *size*
⟨*proof*⟩

lemma *size-eq-card-dom-lookup*: *size* *al* = *card* (*dom* (*DAList.lookup* *al*))
⟨*proof*⟩

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* ⇒ ′*a* ⇒ *bool*
where
  list-member [] *y* ⟷ *False*
| *list-member* (*x* # *xs*) *y* ⟷ *equal* *x* *y* ∨ *list-member* *xs* *y*

primrec *list-distinct* :: ′*a* *list* ⇒ *bool*
where
  list-distinct [] ⟷ *True*
| *list-distinct* (*x* # *xs*) ⟷ ¬ *list-member* *xs* *x* ∧ *list-distinct* *xs*

definition *list-insert* :: ′*a* ⇒ ′*a* *list* ⇒ ′*a* *list* **where**
  list-insert *x* *xs* = (*if* *list-member* *xs* *x* *then* *xs* *else* *x* # *xs*)

primrec *list-remove1* :: ′*a* ⇒ ′*a* *list* ⇒ ′*a* *list* **where**
  list-remove1 *x* [] = []
| *list-remove1* *x* (*y* # *xs*) = (*if* *equal* *x* *y* *then* *xs* *else* *y* # *list-remove1* *x* *xs*)

primrec *list-remdups* :: ′*a* *list* ⇒ ′*a* *list* **where**

list-remdups [] = []
| *list-remdups* (*x* # *xs*) = (*if list-member xs x then list-remdups xs else x* # *list-remdups xs*)

lemma *list-member-filterD*: *list-member* (*filter P xs*) *x* ⟹ *list-member xs x*
⟨*proof*⟩

lemma *list-distinct-filter* [*simp*]: *list-distinct xs* ⟹ *list-distinct* (*filter P xs*)
⟨*proof*⟩

lemma *list-distinct-tl* [*simp*]: *list-distinct xs* ⟹ *list-distinct* (*tl xs*)
⟨*proof*⟩

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*⟩

lemma *list-distinct-conv-distinct* [*simp*]:
equal-base.list-distinct (=) = *List.distinct*
⟨*proof*⟩

lemma *list-insert-conv-insert* [*simp*]:
equal-base.list-insert (=) = *List.insert*
⟨*proof*⟩

lemma *list-remove1-conv-remove1* [*simp*]:
equal-base.list-remove1 (=) = *List.remove1*
⟨*proof*⟩

lemma *list-remdups-conv-remdups* [*simp*]:
equal-base.list-remdups (=) = *List.remdups*
⟨*proof*⟩

context *equal* **begin**

lemma *member-insert* [*simp*]: *list-member* (*list-insert x xs*) *y* ⟷ *equal x y* ∨ *list-member xs y*
⟨*proof*⟩

lemma *member-remove1* [*simp*]:
¬ *equal x y* ⟹ *list-member* (*list-remove1 x xs*) *y* = *list-member xs y*
⟨*proof*⟩

lemma *distinct-remove1*:
list-distinct xs ⟹ *list-distinct* (*list-remove1 x xs*)
⟨*proof*⟩

lemma *distinct-member-remove1* [*simp*]:
list-distinct xs ⟹ *list-member* (*list-remove1 x xs*) = (*list-member xs*)(*x* := *False*)
⟨*proof*⟩

end

lemma *ID-ceq*:
ID CEQ(*'a* :: *ceq*) = *Some eq* ⟹ *equal eq*
⟨*proof*⟩

3.9.2 The type of distinct lists

typedef (**overloaded**) *'a* :: *ceq set-dlist* =
{*xs*::*'a list*. *equal-base.list-distinct ceq' xs* ∨ *ID CEQ*(*'a*) = *None*}
morphisms *list-of-dlist Abs-dlist'*
⟨*proof*⟩

definition *Abs-dlist* :: *'a* :: *ceq list* ⇒ *'a set-dlist*
where
Abs-dlist xs = *Abs-dlist'*
(*if equal-base.list-distinct ceq' xs* ∨ *ID CEQ*(*'a*) = *None then xs*
else equal-base.list-remdups ceq' xs)

lemma *Abs-dlist-inverse*:
fixes *y* :: *'a* :: *ceq list*
assumes *y* ∈ {*xs*. *equal-base.list-distinct ceq' xs* ∨ *ID CEQ*(*'a*) = *None*}
shows *list-of-dlist* (*Abs-dlist y*) = *y*
⟨*proof*⟩

lemma *list-of-dlist-inverse*: *Abs-dlist* (*list-of-dlist dxs*) = *dxs*
⟨*proof*⟩

lemma *type-definition-set-dlist'*:
type-definition list-of-dlist Abs-dlist
{*xs* :: *'a* :: *ceq list*. *equal-base.list-distinct ceq' xs* ∨ *ID CEQ*(*'a*) = *None*}

⟨*proof*⟩

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** []
⟨*proof*⟩

lift-definition *insert* :: *'a* :: *ceq* ⇒ *'a set-dlist* ⇒ *'a set-dlist* **is**
equal-base.list-insert ceq'
⟨*proof*⟩

lift-definition *remove* :: *'a* :: *ceq* ⇒ *'a set-dlist* ⇒ *'a set-dlist* **is**
equal-base.list-remove1 ceq'
⟨*proof*⟩

lift-definition *filter* :: (*'a* :: *ceq* ⇒ *bool*) ⇒ *'a set-dlist* ⇒ *'a set-dlist* **is** *List.filter*
⟨*proof*⟩

Derived operations:

lift-definition *null* :: *'a* :: *ceq set-dlist* ⇒ *bool* **is** *List.null* ⟨*proof*⟩

lift-definition *member* :: *'a* :: *ceq set-dlist* ⇒ *'a* ⇒ *bool* **is** *equal-base.list-member ceq'* ⟨*proof*⟩

lift-definition *length* :: *'a* :: *ceq set-dlist* ⇒ *nat* **is** *List.length* ⟨*proof*⟩

lift-definition *fold* :: (*'a* :: *ceq* ⇒ *'b* ⇒ *'b*) ⇒ *'a set-dlist* ⇒ *'b* ⇒ *'b* **is** *List.fold*
⟨*proof*⟩

lift-definition *foldr* :: (*'a* :: *ceq* ⇒ *'b* ⇒ *'b*) ⇒ *'a set-dlist* ⇒ *'b* ⇒ *'b* **is** *List.foldr*
⟨*proof*⟩

lift-definition *hd* :: *'a* :: *ceq set-dlist* ⇒ *'a* **is** *List.hd* ⟨*proof*⟩

lift-definition *tl* :: *'a* :: *ceq set-dlist* ⇒ *'a set-dlist* **is** *List.tl*
⟨*proof*⟩

lift-definition *dlist-all* :: (*'a* ⇒ *bool*) ⇒ *'a* :: *ceq set-dlist* ⇒ *bool* **is** *list-all* ⟨*proof*⟩

lift-definition *dlist-ex* :: (*'a* ⇒ *bool*) ⇒ *'a* :: *ceq set-dlist* ⇒ *bool* **is** *list-ex* ⟨*proof*⟩

definition *union* :: *'a* :: *ceq set-dlist* ⇒ *'a set-dlist* ⇒ *'a set-dlist* **where**
union = *fold insert*

lift-definition *product* :: *'a* :: *ceq set-dlist* ⇒ *'b* :: *ceq set-dlist* ⇒ (*'a* × *'b*) *set-dlist*
is *λxs ys. rev* (*concat* (*map* (*λx. map* (*Pair x*) *ys*) *xs*))
⟨*proof*⟩

lift-definition *Id-on* :: *'a* :: *ceq set-dlist* ⇒ (*'a* × *'a*) *set-dlist*
is *map* (*λx.* (*x*, *x*))
⟨*proof*⟩

3.9.4 Properties

lemma *member-empty* [*simp*]: *member empty* = (*λ-. False*)
⟨*proof*⟩

lemma *null-iff* [*simp*]: *null xs* ⟷ *xs* = *empty*
⟨*proof*⟩

lemma *list-of-dlist-empty* [*simp*]: *list-of-dlist DList-Set.empty* = []
⟨*proof*⟩

lemma *list-of-dlist-insert* [*simp*]: ¬ *member dxs x* ⟹ *list-of-dlist* (*insert x dxs*) = *x* # *list-of-dlist dxs*
⟨*proof*⟩

lemma *list-of-dlist-eq-Nil-iff* [*simp*]: *list-of-dlist dxs* = [] ⟷ *dxs* = *empty*
⟨*proof*⟩

lemma *fold-empty* [*simp*]: *DList-Set.fold f empty b* = *b*
⟨*proof*⟩

lemma *fold-insert* [*simp*]: ¬ *member dxs x* ⟹ *DList-Set.fold f* (*insert x dxs*) *b* = *DList-Set.fold f dxs* (*f x b*)
⟨*proof*⟩

lemma *no-memb-fold-insert*:
¬ *member dxs x* ⟹ *fold f* (*insert x dxs*) *b* = *fold f dxs* (*f x b*)
⟨*proof*⟩

lemma *set-fold-insert*: *set* (*List.fold List.insert xs1 xs2*) = *set xs1* ∪ *set xs2*
⟨*proof*⟩

lemma *list-of-dlist-eq-singleton-conv*:
list-of-dlist dxs = [*x*] ⟷ *dxs* = *DList-Set.insert x DList-Set.empty*
⟨*proof*⟩

lemma *product-code* [*code abstract*]:
list-of-dlist (*product dxs1 dxs2*) = *fold* (*λa. fold* (*λc rest.* (*a*, *c*) # *rest*) *dxs2*)

dxs1 []
⟨*proof*⟩

lemma *set-list-of-dlist-Abs-dlist*:
  set (*list-of-dlist* (*Abs-dlist xs*)) = *set xs*
⟨*proof*⟩

context assumes *ID-ceq-neq-None*: *ID CEQ*(′*a* :: *ceq*) ≠ *None*
begin

lemma *equal-ceq*: *equal* (*ceq*′ :: ′*a* ⇒ ′*a* ⇒ *bool*)
⟨*proof*⟩

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*: ⋀*a dxs*. ⟦ ¬ *member dxs a*; *P dxs* ⟧ ⟹ *P* (*insert a dxs*)
  shows *P dxs*
⟨*proof*⟩

context includes *lifting-syntax*
begin

lemma *fold-transfer2* [*transfer-rule*]:
  assumes *is-equality A*
  shows ((*A* ===> *pcr-set-dlist* (=) ===> *pcr-set-dlist* (=)) ===>
   (*pcr-set-dlist* (=) :: ′*a list* ⇒ ′*a set-dlist* ⇒ *bool*) ===> *pcr-set-dlist* (=) ===> *pcr-set-dlist* (=))
     List.fold DList-Set.fold
⟨*proof*⟩

end

lemma *distinct-list-of-dlist*:
  distinct (*list-of-dlist* (*dxs* :: ′*a set-dlist*))
⟨*proof*⟩

lemma *member-empty-empty*: (∀ *x* :: ′*a*. ¬ *member dxs x*) ⟷ *dxs* = *empty*
⟨*proof*⟩

lemma *Collect-member*: *Collect* (*member* (*dxs* :: ′*a set-dlist*)) = *set* (*list-of-dlist dxs*)
⟨*proof*⟩

lemma *member-insert*: *member* (*insert* (*x* :: ′*a*) *xs*) = (*member xs*)(*x* := *True*)

⟨*proof*⟩

lemma *member-remove*:
member (*remove* (x :: $'a$) *xs*) = (*member xs*)(x := *False*)
⟨*proof*⟩

lemma *member-union*: *member* (*union* (*xs1* :: $'a$ *set-dlist*) *xs2*) x ⟷ *member xs1* x ∨ *member xs2* x
⟨*proof*⟩

lemma *member-fold-insert*: *member* (*List.fold insert xs dxs*) (x :: $'a$) ⟷ *member dxs* x ∨ x ∈ *set xs*
⟨*proof*⟩

lemma *card-eq-length* [*simp*]:
card (*Collect* (*member* (*dxs* :: $'a$ *set-dlist*))) = *length dxs*
⟨*proof*⟩

lemma *finite-member* [*simp*]:
finite (*Collect* (*member* (*dxs* :: $'a$ *set-dlist*)))
⟨*proof*⟩

lemma *member-filter* [*simp*]: *member* (*filter P xs*) = (λx :: $'a$. *member xs* x ∧ P x)
⟨*proof*⟩

lemma *dlist-all-conv-member*: *dlist-all P dxs* ⟷ (∀ x :: $'a$. *member dxs* x ⟶ P x)
⟨*proof*⟩

lemma *dlist-ex-conv-member*: *dlist-ex P dxs* ⟷ (∃ x :: $'a$. *member dxs* x ∧ P x)
⟨*proof*⟩

lemma *member-Id-on*: *member* (*Id-on dxs*) = (λ(x :: $'a$, y). x = y ∧ *member dxs* x)
⟨*proof*⟩

end

lemma *product-member*:
assumes *ID CEQ*($'a$ :: *ceq*) ≠ *None*   *ID CEQ*($'b$ :: *ceq*) ≠ *None*
shows *member* (*product dxs1 dxs2*) = (λ(a :: $'a$, b :: $'b$). *member dxs1* a ∧ *member dxs2* b)
⟨*proof*⟩

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*:
$map\text{-}of\ (map\ (\lambda x.\ (x,\ v))\ xs) = (\lambda x.\ if\ x \in set\ xs\ then\ Some\ v\ else\ None)$
⟨*proof*⟩

lemma *map-of-rev-unit* [*simp*]:
fixes $xs :: ('a * unit)\ list$
shows $map\text{-}of\ (rev\ xs) = map\text{-}of\ xs$
⟨*proof*⟩

lemma *fold-split-conv-map-fst*: $fold\ (\lambda(x,\ y).\ f\ x)\ xs = fold\ f\ (map\ fst\ xs)$
⟨*proof*⟩

lemma *foldr-split-conv-map-fst*: $foldr\ (\lambda(x,\ y).\ f\ x)\ xs = foldr\ f\ (map\ fst\ xs)$
⟨*proof*⟩

lemma *set-foldr-Cons*:
$set\ (foldr\ (\lambda x\ xs.\ if\ P\ x\ xs\ then\ x\ \#\ xs\ else\ xs)\ as\ []) \subseteq set\ as$
⟨*proof*⟩

lemma *distinct-fst-foldr-Cons*:
$distinct\ (map\ f\ as) \Longrightarrow distinct\ (map\ f\ (foldr\ (\lambda x\ xs.\ if\ P\ x\ xs\ then\ x\ \#\ xs\ else\ xs)\ as\ []))$
⟨*proof*⟩

lemma *filter-conv-foldr*:
$filter\ P\ xs = foldr\ (\lambda x\ xs.\ if\ P\ x\ then\ x\ \#\ xs\ else\ xs)\ xs\ []$
⟨*proof*⟩

lemma *map-of-filter*: $map\text{-}of\ (filter\ (\lambda x.\ P\ (fst\ x))\ xs) = map\text{-}of\ xs\ |`\ Collect\ P$
⟨*proof*⟩

lemma *map-of-map-Pair-key*: $map\text{-}of\ (map\ (\lambda k.\ (k,\ f\ k))\ xs)\ x = (if\ x \in set\ xs\ then\ Some\ (f\ x)\ else\ None)$
⟨*proof*⟩

lemma *neq-Empty-conv*: $t \neq rbt.Empty \longleftrightarrow (\exists\ c\ l\ k\ v\ r.\ t = Branch\ c\ l\ k\ v\ r)$
⟨*proof*⟩

context *linorder* **begin**

lemma *is-rbt-RBT-fold-rbt-insert* [*simp*]:
is-rbt t ⟹ *is-rbt* (*fold* (λ(*k*, *v*). *rbt-insert k v*) *xs t*)
⟨*proof*⟩

lemma *rbt-lookup-RBT-fold-rbt-insert* [*simp*]:
is-rbt t ⟹ *rbt-lookup* (*fold* (λ(*k*, *v*). *rbt-insert k v*) *xs t*) = *rbt-lookup t* ++ *map-of* (*rev xs*)
⟨*proof*⟩

lemma *is-rbt-fold-rbt-delete* [*simp*]:
is-rbt t ⟹ *is-rbt* (*fold rbt-delete xs t*)
⟨*proof*⟩

lemma *rbt-lookup-fold-rbt-delete* [*simp*]:
is-rbt t ⟹ *rbt-lookup* (*fold rbt-delete xs t*) = *rbt-lookup t* |‘ (− *set xs*)
⟨*proof*⟩

lemma *is-rbt-fold-rbt-insert*: *is-rbt t* ⟹ *is-rbt* (*fold* (λ*k*. *rbt-insert k* (*f k*)) *xs t*)
⟨*proof*⟩

lemma *rbt-lookup-fold-rbt-insert*:
is-rbt t ⟹
rbt-lookup (*fold* (λ*k*. *rbt-insert k* (*f k*)) *xs t*) =
rbt-lookup t ++ *map-of* (*map* (λ*k*. (*k*, *f k*)) *xs*)
⟨*proof*⟩

end

definition *fold-rev* :: (′*a* ⇒ ′*b* ⇒ ′*c* ⇒ ′*c*) ⇒ (′*a*, ′*b*) *rbt* ⇒ ′*c* ⇒ ′*c*
where *fold-rev f t* = *List.foldr* (λ(*k*, *v*). *f k v*) (*RBT-Impl.entries t*)

lemma *fold-rev-simps* [*simp*, *code*]:
fold-rev f RBT-Impl.Empty = *id*
fold-rev f (*Branch c l k v r*) = *fold-rev f l o f k v o fold-rev f r*
⟨*proof*⟩

context *linorder* **begin**

lemma *sorted-fst-foldr-Cons*:
sorted (*map f as*) ⟹ *sorted* (*map f* (*foldr* (λ*x xs*. *if P x xs then x* # *xs else xs*) *as* []))
⟨*proof*⟩

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* ⇒ *'a set-rbt*
where *Set-RBT* ≡ *Mapping-RBT*

3.10.2 Primitive operations

lift-definition *member* :: *'a* :: *ccompare set-rbt* ⇒ *'a* ⇒ *bool* **is**
λ*t x*. *x* ∈ *dom* (*rbt-comp-lookup ccomp t*) ⟨*proof*⟩

abbreviation *empty* :: *'a* :: *ccompare set-rbt*
where *empty* ≡ *RBT-Mapping2.empty*

abbreviation *insert* :: *'a* :: *ccompare* ⇒ *'a set-rbt* ⇒ *'a set-rbt*
where *insert k* ≡ *RBT-Mapping2.insert k* ()

abbreviation *remove* :: *'a* :: *ccompare* ⇒ *'a set-rbt* ⇒ *'a set-rbt*
where *remove* ≡ *RBT-Mapping2.delete*

lift-definition *bulkload* :: *'a* :: *ccompare list* ⇒ *'a set-rbt* **is**
rbt-comp-bulkload ccomp ∘ *map* (λ*x*. (*x*, ()))
⟨*proof*⟩

abbreviation *is-empty* :: *'a* :: *ccompare set-rbt* ⇒ *bool*
where *is-empty* ≡ *RBT-Mapping2.is-empty*

abbreviation *union* :: *'a* :: *ccompare set-rbt* ⇒ *'a set-rbt* ⇒ *'a set-rbt*
where *union* ≡ *RBT-Mapping2.join* (λ- -. *id*)

abbreviation *inter* :: *'a* :: *ccompare set-rbt* ⇒ *'a set-rbt* ⇒ *'a set-rbt*
where *inter* ≡ *RBT-Mapping2.meet* (λ- -. *id*)

lift-definition *inter-list* :: *'a* :: *ccompare set-rbt* ⇒ *'a list* ⇒ *'a set-rbt* **is**
λ*t xs*. *fold* (λ*k*. *rbt-comp-insert ccomp k* ()) [*x* ← *xs*. *rbt-comp-lookup ccomp t x* ≠ *None*] *RBT-Impl.Empty*
⟨*proof*⟩

lift-definition *minus* :: *'a* :: *ccompare set-rbt* ⇒ *'a set-rbt* ⇒ *'a set-rbt* **is**
rbt-comp-minus ccomp
⟨*proof*⟩

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** λ*f*. *RBT-Impl.fold* (λ*a* -. *f a*) ⟨*proof*⟩

lift-definition *fold1* :: (*'a* :: *ccompare* ⇒ *'a* ⇒ *'a*) ⇒ *'a set-rbt* ⇒ *'a* **is** *RBT-Impl-fold1*

⟨*proof*⟩

lift-definition *keys* :: $'a$ :: *ccompare set-rbt* ⇒ $'a$ *list* **is** *RBT-Impl.keys* ⟨*proof*⟩

abbreviation *all* :: ($'a$ :: *ccompare* ⇒ *bool*) ⇒ $'a$ *set-rbt* ⇒ *bool*
where *all P* ≡ *RBT-Mapping2.all* (λ*k* -. *P k*)

abbreviation *ex* :: ($'a$ :: *ccompare* ⇒ *bool*) ⇒ $'a$ *set-rbt* ⇒ *bool*
where *ex P* ≡ *RBT-Mapping2.ex* (λ*k* -. *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*)
⟨*proof*⟩

lemma *fold-conv-fold-keys*: *RBT-Set2.fold f rbt b* = *List.fold f* (*RBT-Set2.keys rbt*) *b*
⟨*proof*⟩

lemma *fold-conv-fold-keys'*:
fold f t = *List.fold f* (*RBT-Impl.keys* (*RBT-Mapping2.impl-of t*))
⟨*proof*⟩

lemma *member-lookup* [*code*]: *member t x* ⟷ *RBT-Mapping2.lookup t x* = *Some* ()
⟨*proof*⟩

lemma *unfoldr-rbt-keys-generator*:
list.unfoldr rbt-keys-generator (*init t*) = *keys t*
⟨*proof*⟩

lemma *keys-eq-Nil-iff* [*simp*]: *keys rbt* = [] ⟷ *rbt* = *empty*
⟨*proof*⟩

lemma *fold1-conv-fold*: *fold1 f rbt* = *List.fold f* (*tl* (*keys rbt*)) (*hd* (*keys rbt*))
⟨*proof*⟩

context assumes *ID-ccompare-neq-None*: *ID CCOMPARE*($'a$ :: *ccompare*) ≠ *None*

begin

lemma *set-linorder*: *class.linorder* (*cless-eq* :: *'a* ⇒ *'a* ⇒ *bool*) *cless*
⟨*proof*⟩

lemma *ccomp-comparator*: *comparator* (*ccomp* :: *'a comparator*)
⟨*proof*⟩

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 cless* (*RBT-Mapping2.impl-of t*)
⟨*proof*⟩

lemma *member-RBT*:
ord.is-rbt cless t ⟹ *member* (*Set-RBT t*) (*x* :: *'a*) ⟷ *ord.rbt-lookup cless t x* = *Some* ()
⟨*proof*⟩

lemma *member-impl-of*:
ord.rbt-lookup cless (*RBT-Mapping2.impl-of t*) (*x* :: *'a*) = *Some* () ⟷ *member t x*
⟨*proof*⟩

lemma *member-insert* [*simp*]:
member (*insert x* (*t* :: *'a set-rbt*)) = (*member t*)(*x* := *True*)
⟨*proof*⟩

lemma *member-fold-insert* [*simp*]:
member (*List.fold insert xs* (*t* :: *'a set-rbt*)) = (λ*x*. *member t x* ∨ *x* ∈ *set xs*)
⟨*proof*⟩

lemma *member-remove* [*simp*]:
member (*remove* (*x* :: *'a*) *t*) = (*member t*)(*x* := *False*)
⟨*proof*⟩

lemma *member-bulkload* [*simp*]:
member (*bulkload xs*) (*x* :: *'a*) ⟷ *x* ∈ *set xs*
⟨*proof*⟩

lemma *member-conv-keys*: *member t* = (λ*x* :: *'a*. *x* ∈ *set* (*keys t*))
⟨*proof*⟩

lemma *is-empty-empty* [*simp*]:
is-empty t ⟷ *t* = *empty*
⟨*proof*⟩

lemma *RBT-lookup-empty* [*simp*]:
ord.rbt-lookup cless (*t* :: (*'a*, *unit*) *rbt*) = *Map.empty* ⟷ *t* = *RBT-Impl.Empty*
⟨*proof*⟩

lemma *member-empty-empty* [*simp*]:
member t = (λ-. *False*) ⟷ (*t* :: *'a set-rbt*) = *empty*
⟨*proof*⟩

lemma *member-union* [*simp*]:
member (*union* (*t1* :: *'a set-rbt*) *t2*) = (λ*x*. *member t1 x* ∨ *member t2 x*)
⟨*proof*⟩

lemma *member-minus* [*simp*]:
member (*minus* (*t1* :: *'a set-rbt*) *t2*) = (λ*x*. *member t1 x* ∧ ¬ *member t2 x*)
⟨*proof*⟩

lemma *member-inter* [*simp*]:
member (*inter* (*t1* :: *'a set-rbt*) *t2*) = (λ*x*. *member t1 x* ∧ *member t2 x*)
⟨*proof*⟩

lemma *member-inter-list* [*simp*]:
member (*inter-list* (*t* :: *'a set-rbt*) *xs*) = (λ*x*. *member t x* ∧ *x* ∈ *set xs*)
⟨*proof*⟩

lemma *member-filter* [*simp*]:
member (*filter P* (*t* :: *'a set-rbt*)) = (λ*x*. *member t x* ∧ *P x*)
⟨*proof*⟩

lemma *distinct-keys* [*simp*]:
distinct (*keys* (*rbt* :: *'a set-rbt*))
⟨*proof*⟩

lemma *all-conv-all-member*:
all P t ⟷ (∀ *x* :: *'a*. *member t x* ⟶ *P x*)
⟨*proof*⟩

lemma *ex-conv-ex-member*:
ex P t ⟷ (∃ *x* :: *'a*. *member t x* ∧ *P x*)
⟨*proof*⟩

lemma *finite-member*: *finite* (*Collect* (*RBT-Set2.member* (*t* :: *'a set-rbt*)))
⟨*proof*⟩

lemma *member-Id-on*: *member* (*Id-on t*) = (λ(*k* :: *'a*, *k'*). *k* = *k'* ∧ *member t k*)
⟨*proof*⟩

context assumes *ID-ccompare-neq-None'*: *ID CCOMPARE*(*'b* :: *ccompare*) ≠ *None*
begin

lemma *set-linorder'*: *class.linorder* (*cless-eq* :: $'b \Rightarrow 'b \Rightarrow bool$) *cless*
⟨*proof*⟩

lemma *member-product*:
member (*product rbt1 rbt2*) = (λab :: $'a \times 'b$. $ab \in$ *Collect* (*member rbt1*) $\times$ *Collect* (*member rbt2*))
⟨*proof*⟩

end

end

lemma *sorted-RBT-Set-keys*:
ID CCOMPARE('a :: *ccompare*) = *Some c*
$\Longrightarrow$ *linorder.sorted* (*le-of-comp c*) (*RBT-Set2.keys rbt*)
⟨*proof*⟩

context assumes *ID-ccompare-neq-None*: *ID CCOMPARE*('a :: {*ccompare*, *lattice*}) $\neq$ *None*
begin

lemma *set-linorder2*: *class.linorder* (*cless-eq* :: $'a \Rightarrow 'a \Rightarrow bool$) *cless*
⟨*proof*⟩

end

lemma *set-keys-Mapping-RBT*: *set* (*keys* (*Mapping-RBT t*)) = *set* (*RBT-Impl.keys t*)
⟨*proof*⟩

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* :: $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow 'a \Rightarrow 'b$
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*
⟨*proof*⟩

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* ⇒ *'a list* ⇒ *'a list*
and *quicksort-part* :: *'a list* ⇒ *'a* ⇒ *'a list* ⇒ *'a list* ⇒ *'a list* ⇒ *'a list* ⇒ *'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*)
⟨*proof*⟩

lemma *length-quicksort-accp*:
quicksort-acc-quicksort-part-dom (*Inl* (*ac*, *xs*)) ⟹ *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*))
⟹ *length* (*quicksort-part ac x lts eqs gts zs*) = *length ac* + *1* + *length lts* + *length eqs* + *length gts* + *length zs*
⟨*proof*⟩

termination

⟨*proof*⟩

definition *quicksort* :: *'a list* ⇒ *'a list*
where *quicksort* = *quicksort-acc* []

lemma *set-quicksort-acc* [*simp*]: *set* (*quicksort-acc ac xs*) = *set ac* ∪ *set xs*
and *set-quicksort-part* [*simp*]:
set (*quicksort-part ac x lts eqs gts zs*) =
set ac ∪ {*x*} ∪ *set lts* ∪ *set eqs* ∪ *set gts* ∪ *set zs*
⟨*proof*⟩

lemma *set-quicksort* [*simp*]: *set* (*quicksort xs*) = *set xs*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *distinct-quicksort* [*simp*]: *distinct* (*quicksort xs*) = *distinct xs*
⟨*proof*⟩

end

lemmas [*code*] =
ord.quicksort-acc.simps quicksort-acc.simps
ord.quicksort-part.simps quicksort-part.simps
ord.quicksort-def quicksort-def

context *linorder* **begin**

lemma *sorted-quicksort-acc*:
⟦ *sorted ac*; ∀ *x* ∈ *set xs*. ∀ *a* ∈ *set ac*. *x* < *a* ⟧
⟹ *sorted* (*quicksort-acc ac xs*)
and *sorted-quicksort-part*:
⟦ *sorted ac*; ∀ *y* ∈ *set lts* ∪ {*x*} ∪ *set eqs* ∪ *set gts* ∪ *set zs*. ∀ *a* ∈ *set ac*. *y* < *a*;
∀ *y* ∈ *set lts*. *y* < *x*; ∀ *y* ∈ *set eqs*. *y* = *x*; ∀ *y* ∈ *set gts*. *y* > *x* ⟧
⟹ *sorted* (*quicksort-part ac x lts eqs gts zs*)
⟨*proof*⟩

lemma *sorted-quicksort* [*simp*]: *sorted* (*quicksort xs*)
⟨*proof*⟩

lemma *insort-key-append1*:
∀ *y* ∈ *set ys*. *f x* < *f y* ⟹ *insort-key f x* (*xs* @ *ys*) = *insort-key f x xs* @ *ys*
⟨*proof*⟩

lemma *insort-key-append2*:
$\forall y \in set\ xs.\ f\ x > f\ y \Longrightarrow insort\text{-}key\ f\ x\ (xs\ @\ ys) = xs\ @\ insort\text{-}key\ f\ x\ ys$
⟨*proof*⟩

lemma *sort-key-append*:
$\forall x{\in}set\ xs.\ \forall y{\in}set\ ys.\ f\ x < f\ y \Longrightarrow sort\text{-}key\ f\ (xs\ @\ ys) = sort\text{-}key\ f\ xs\ @\ sort\text{-}key\ f\ ys$
⟨*proof*⟩

definition *single-list* :: $'a \Rightarrow 'a\ list$ **where** $single\text{-}list\ a = [a]$

lemma *to-single-list*: $x\ \#\ xs = single\text{-}list\ x\ @\ xs$
⟨*proof*⟩

lemma *sort-snoc*: $sort\ (xs\ @\ [x]) = insort\ x\ (sort\ xs)$
⟨*proof*⟩

lemma *sort-append-swap*: $sort\ (xs\ @\ ys) = sort\ (ys\ @\ xs)$
⟨*proof*⟩

lemma *sort-append-swap2*: $sort\ (xs\ @\ ys\ @\ zs) = sort\ (ys\ @\ xs\ @\ zs)$
⟨*proof*⟩

lemma *sort-Cons-append-swap*: $sort\ (x\ \#\ xs) = sort\ (xs\ @\ [x])$
⟨*proof*⟩

lemma *sort-append-Cons-swap*: $sort\ (ys\ @\ x\ \#\ xs) = sort\ (ys\ @\ xs\ @\ [x])$
⟨*proof*⟩

lemma *quicksort-acc-conv-sort*:
$quicksort\text{-}acc\ ac\ xs = sort\ xs\ @\ ac$
and *quicksort-part-conv-sort*:
$[\![\ \forall y \in set\ lts.\ y < x;\ \forall y \in set\ eqs.\ y = x;\ \forall y \in set\ gts.\ y > x\ ]\!]$
$\Longrightarrow quicksort\text{-}part\ ac\ x\ lts\ eqs\ gts\ zs = sort\ (lts\ @\ eqs\ @\ gts\ @\ x\ \#\ zs)\ @\ ac$
⟨*proof*⟩

lemma *quicksort-conv-sort*: $quicksort\ xs = sort\ xs$
⟨*proof*⟩

lemma *sort-remdups*: $sort\ (remdups\ xs) = remdups\ (sort\ xs)$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *distinct-remdups-sorted* [*simp*]: *sorted xs* ⟹ *distinct* (*remdups-sorted xs*)
⟨*proof*⟩

lemma *remdups-sorted-conv-remdups*: *sorted xs* ⟹ *remdups-sorted xs* = *remdups xs*
⟨*proof*⟩

end

An specialised operation to convert a finite set into a sorted list

definition *csorted-list-of-set* :: *'a* :: *ccompare set* ⇒ *'a list*
where [*code del*]:
  csorted-list-of-set A =
  (*if ID CCOMPARE('a)* = *None* ∨ ¬ *finite A then undefined else linorder.sorted-list-of-set cless-eq A*)

lemma *csorted-list-of-set-set* [*simp*]:
  ⟦ *ID CCOMPARE('a* :: *ccompare*) = *Some c*; *linorder.sorted* (*le-of-comp c*) *xs*; *distinct xs* ⟧
  ⟹ *linorder.sorted-list-of-set* (*le-of-comp c*) (*set xs*) = *xs*
⟨*proof*⟩

lemma *csorted-list-of-set-split*:
  fixes *A* :: *'a* :: *ccompare set* **shows**
  P (*csorted-list-of-set A*) ⟷
  (∀ *xs*. *ID CCOMPARE('a)* ≠ *None* ⟶ *finite A* ⟶ *A* = *set xs* ⟶ *distinct xs* ⟶ *linorder.sorted cless-eq xs* ⟶ *P xs*) ∧
  (*ID CCOMPARE('a)* = *None* ∨ ¬ *finite A* ⟶ *P undefined*)
⟨*proof*⟩

code-identifier code-module *Set* ⇀ (*SML*) *Set-Impl*

| **code-module** *Set-Impl* ⇀ (*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*
⟨*proof*⟩

definition *is-UNIV* :: ′*a set* ⇒ *bool*
where [*code del*]: *is-UNIV A* ⟷ *A* = *UNIV*

lemma *is-UNIV-unfold* [*code-unfold*]:
A = *UNIV* ⟷ *is-UNIV A*
UNIV = *A* ⟷ *is-UNIV A*
set-eq A UNIV ⟷ *is-UNIV A*
set-eq UNIV A ⟷ *is-UNIV A*
⟨*proof*⟩

declare [[*code drop*:
Set.empty
Set.is-empty
uminus-set-inst.uminus-set
Set.member
Set.insert
Set.remove
UNIV
Set.filter
image
Set.subset-eq
Ball
Bex
Set.union
minus-set-inst.minus-set
Set.inter
card
Set.bind
the-elem
Pow
sum
Gcd
Lcm
Product-Type.product
Id-on
Image
trancl
relcomp
wf-code
Min

Inf-fin
Max
Sup-fin
Inf :: *'a set set* ⇒ *'a set*
Sup :: *'a set set* ⇒ *'a set*
sorted-list-of-set
List.map-project
Sup-pred-inst.Sup-pred
finite
card
Inf-pred-inst.Inf-pred
pred-of-set
Wellfounded.acc
Bleast
can-select

irrefl-on
bacc
set-of-pred
set-of-seq
]]

3.12.3 Set implementations

definition *Collect-set* :: (*'a* ⇒ *bool*) ⇒ *'a set*
where [*simp*]: *Collect-set* = *Collect*

definition *DList-set* :: *'a* :: *ceq set-dlist* ⇒ *'a set*
where *DList-set* = *Collect o DList-Set.member*

definition *RBT-set* :: *'a* :: *ccompare set-rbt* ⇒ *'a set*
where *RBT-set* = *Collect o RBT-Set2.member*

definition *Complement* :: *'a set* ⇒ *'a set*
where [*simp*]: *Complement A* = − *A*

definition *Set-Monad* :: *'a list* ⇒ *'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* = {}
⟨*proof*⟩

lemma *RBT-set-empty* [*simp*]: *RBT-set RBT-Set2.empty* = {}
⟨*proof*⟩

lemma *RBT-set-conv-keys*:
ID CCOMPARE(*'a* :: *ccompare*) ≠ *None*

$\Longrightarrow$ *RBT-set* (*t* :: *'a set-rbt*) = *set* (*RBT-Set2.keys t*)
⟨*proof*⟩

3.12.4 Set operations

A collection of all the theorems about *Complement*.

⟨*ML*⟩

Various fold operations over sets

typedef (*'a*, *'b*) *comp-fun-commute* = {*f* :: *'a* ⇒ *'b* ⇒ *'b*. *comp-fun-commute f*}
morphisms *comp-fun-commute-apply Abs-comp-fun-commute*
⟨*proof*⟩

setup-lifting *type-definition-comp-fun-commute*

lemma *comp-fun-commute-apply'* [*simp*]:
comp-fun-commute-on UNIV (*comp-fun-commute-apply f*)
⟨*proof*⟩

lift-definition *set-fold-cfc* :: (*'a*, *'b*) *comp-fun-commute* ⇒ *'b* ⇒ *'a set* ⇒ *'b* **is** *Finite-Set.fold* ⟨*proof*⟩

declare [[*code drop*: *set-fold-cfc*]]

lemma *set-fold-cfc-code* [*code*]:
fixes *xs* :: *'a* :: *ceq list*
and *dxs* :: *'a* :: *ceq set-dlist*
and *rbt* :: *'b* :: *ccompare set-rbt*
shows *set-fold-cfc-Complement*[*set-complement-code*]:
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*))
and
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*))
set-fold-cfc f b (*Set-Monad xs*) =
(*case ID CEQ*(*'a*) *of None* ⇒ *Code.abort* (*STR ''set-fold-cfc Set-Monad*: *ceq = None''*) (λ-. *set-fold-cfc f b* (*Set-Monad xs*))
| *Some eq* ⇒ *List.fold* (*comp-fun-commute-apply f*) (*equal-base.list-remdups eq xs*) *b*)
(**is** *?Set-Monad*)
set-fold-cfc f' b (*DList-set dxs*) =
(*case ID CEQ*(*'a*) *of None* ⇒ *Code.abort* (*STR ''set-fold-cfc DList-set*: *ceq = None''*) (λ-. *set-fold-cfc f' b* (*DList-set dxs*))
| *Some* - ⇒ *DList-Set.fold* (*comp-fun-commute-apply f'*) *dxs b*)
(**is** *?DList-set*)
set-fold-cfc f'' b (*RBT-set rbt*) =
(*case ID CCOMPARE*(*'b*) *of None* ⇒ *Code.abort* (*STR ''set-fold-cfc RBT-set*: *ccompare = None''*) (λ-. *set-fold-cfc f'' b* (*RBT-set rbt*))
| *Some* - ⇒ *RBT-Set2.fold* (*comp-fun-commute-apply f''*) *rbt b*)

(**is** *?RBT-set*)
⟨*proof*⟩

typedef ('*a*, '*b*) *comp-fun-idem* = {*f* :: '*a* ⇒ '*b* ⇒ '*b*. *comp-fun-idem f*}
morphisms *comp-fun-idem-apply Abs-comp-fun-idem*
⟨*proof*⟩

setup-lifting *type-definition-comp-fun-idem*

lemma *comp-fun-idem-apply'* [*simp*]:
comp-fun-idem-on UNIV (*comp-fun-idem-apply f*)
⟨*proof*⟩

lift-definition *set-fold-cfi* :: ('*a*, '*b*) *comp-fun-idem* ⇒ '*b* ⇒ '*a set* ⇒ '*b* **is** *Finite-Set.fold* ⟨*proof*⟩

declare [[*code drop*: *set-fold-cfi*]]

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 (*Complement A*) = *Code.abort* (*STR ''set-fold-cfi not supported on Complement''*) (λ-. *set-fold-cfi f b* (*Complement A*))
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 (*Set-Monad xs*) = *List.fold* (*comp-fun-idem-apply f*) *xs b*
(**is** *?Set-Monad*)
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 (*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*)
⟨*proof*⟩

typedef '*a semilattice-set* = {*f* :: '*a* ⇒ '*a* ⇒ '*a*. *semilattice-set f*}
morphisms *semilattice-set-apply Abs-semilattice-set*
⟨*proof*⟩

setup-lifting *type-definition-semilattice-set*

lemma *semilattice-set-apply'* [*simp*]:
semilattice-set (*semilattice-set-apply f*)
⟨*proof*⟩

lemma *comp-fun-idem-semilattice-set-apply* [*simp*]:
comp-fun-idem-on UNIV (*semilattice-set-apply f*)
⟨*proof*⟩

lift-definition *set-fold1* :: *'a semilattice-set* ⇒ *'a set* ⇒ *'a* **is** *semilattice-set.F*
⟨*proof*⟩

lemma (**in** *semilattice-set*) *F-set-conv-fold*:
xs ≠ [] ⟹ *F* (*set xs*) = *Finite-Set.fold f* (*hd xs*) (*set* (*tl xs*))
⟨*proof*⟩

lemma *set-fold1-code* [*code*]:
fixes *rbt* :: *'a* :: {*ccompare*, *lattice*} *set-rbt*
and *dxs* :: *'b* :: {*ceq*, *lattice*} *set-dlist* **shows**
set-fold1-Complement[*set-complement-code*]:
set-fold1 f (*Complement A*) = *Code.abort* (*STR ''set-fold1*: *Complement''*) (λ-. *set-fold1 f* (*Complement A*))
and *set-fold1 f* (*Collect-set P*) = *Code.abort* (*STR ''set-fold1*: *Collect-set''*) (λ-. *set-fold1 f* (*Collect-set P*))
and *set-fold1 f* (*Set-Monad* (*x* # *xs*)) = *fold* (*semilattice-set-apply f*) *xs x* (**is** *?Set-Monad*)
and
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*)
and
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*)
⟨*proof*⟩

Implementation of set operations

lemma *Collect-code* [*code*]:
fixes *P* :: *'a* :: *cenum* ⇒ *bool* **shows**
Collect P =
(*case ID CENUM*(*'a*) *of None* ⇒ *Collect-set P*
| *Some* (*enum*, -) ⇒ *Set-Monad* (*filter P enum*))
⟨*proof*⟩

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* ⇒ *bool* **shows**
finite (*DList-set dxs*) =
(*case ID CEQ*(*'a*) *of None* ⇒ *Code.abort* (*STR ''finite DList-set*: *ceq* = *None''*) (λ-. *finite* (*DList-set dxs*))
| *Some* - ⇒ *True*)
finite (*RBT-set rbt*) =
(*case ID CCOMPARE*(*'b*) *of None* ⇒ *Code.abort* (*STR ''finite RBT-set*: *ccompare* = *None''*) (λ-. *finite* (*RBT-set rbt*))
| *Some* - ⇒ *True*)
and *finite-Complement* [*set-complement-code*]:
finite (*Complement A*) ⟷
(*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
finite (*Set-Monad xs*) = *True*
finite (*Collect-set P*) ⟷
of-phantom (*finite-UNIV* :: *'c finite-UNIV*) ∨ *Code.abort* (*STR ''finite Collect-set''*) (λ-. *finite* (*Collect-set P*))
⟨*proof*⟩

lemma *CARD-code* [*code-unfold*]:
CARD(*'a* :: *card-UNIV*) = *of-phantom* (*card-UNIV* :: *'a card-UNIV*)
⟨*proof*⟩

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 (*DList-set dxs*) =
(*case ID CEQ*(*'a*) *of None* ⇒ *Code.abort* (*STR ''card DList-set*: *ceq* = *None''*) (λ-. *card* (*DList-set dxs*))
| *Some* - ⇒ *DList-Set.length dxs*)
card (*RBT-set rbt*) =
(*case ID CCOMPARE*(*'b*) *of None* ⇒ *Code.abort* (*STR ''card RBT-set*: *ccompare* = *None''*) (λ-. *card* (*RBT-set rbt*))
| *Some* - ⇒ *length* (*RBT-Set2.keys rbt*))
card (*Set-Monad xs*) =
(*case ID CEQ*(*'a*) *of None* ⇒ *Code.abort* (*STR ''card Set-Monad*: *ceq* = *None''*) (λ-. *card* (*Set-Monad xs*))
| *Some eq* ⇒ *length* (*equal-base.list-remdups eq xs*))
and *card-Complement* [*set-complement-code*]:
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'') (λ-. card (Complement A)))
⟨*proof*⟩

lemma *is-UNIV-code* [*code*]:
fixes *rbt* :: *'a* :: {*cproper-interval*, *finite-UNIV*} *set-rbt*
and *A* :: *'b* :: *card-UNIV set* **shows**
is-UNIV A ⟷
(*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''*) (λ-. *is-UNIV A*))
(**is** *?generic*)
is-UNIV (*RBT-set rbt*) =
(*case ID CCOMPARE*(*'a*) *of None* ⇒ *Code.abort* (*STR ''is-UNIV RBT-set*: *ccompare* = *None''*) (λ-. *is-UNIV* (*RBT-set rbt*))
| *Some* - ⇒ *of-phantom* (*finite-UNIV* :: *'a finite-UNIV*) ∧ *proper-intrvl.exhaustive-fusion cproper-interval rbt-keys-generator* (*RBT-Set2.init rbt*))
(**is** *?rbt*)
⟨*proof*⟩

lemma *is-empty-code* [*code*]:
fixes *dxs* :: *'a* :: *ceq set-dlist*
and *rbt* :: *'b* :: *ccompare set-rbt*
and *A* :: *'c set* **shows**
Set.is-empty (*Set-Monad xs*) ⟷ *xs* = []
Set.is-empty (*DList-set dxs*) ⟷
(*case ID CEQ*(*'a*) *of None* ⇒ *Code.abort* (*STR ''is-empty DList-set*: *ceq* = *None''*) (λ-. *Set.is-empty* (*DList-set dxs*))
| *Some* - ⇒ *DList-Set.null dxs*) (**is** *?DList-set*)
Set.is-empty (*RBT-set rbt*) ⟷
(*case ID CCOMPARE*(*'b*) *of None* ⇒ *Code.abort* (*STR ''is-empty RBT-set*: *ccompare* = *None''*) (λ-. *Set.is-empty* (*RBT-set rbt*))
| *Some* - ⇒ *RBT-Set2.is-empty rbt*) (**is** *?RBT-set*)
and *is-empty-Complement* [*set-complement-code*]:
Set.is-empty (*Complement A*) ⟷ *is-UNIV A* (**is** *?Complement*)
⟨*proof*⟩

lemma *Set-insert-code* [*code*]:
fixes *dxs* :: *'a* :: *ceq set-dlist*
and *rbt* :: *'b* :: *ccompare set-rbt* **shows**
⋀*x*. *Set.insert x* (*Collect-set A*) =
(*case ID CEQ*(*'a*) *of None* ⇒ *Code.abort* (*STR ''insert Collect-set*: *ceq* = *None''*) (λ-. *Set.insert x* (*Collect-set A*))
| *Some eq* ⇒ *Collect-set* (*equal-base.fun-upd eq A x True*))

$\bigwedge x.$ *Set.insert x* (*Set-Monad xs*) = *Set-Monad* (x # *xs*)
$\bigwedge x.$ *Set.insert x* (*DList-set dxs*) =
(*case ID CEQ*('*a*) *of None* ⇒ *Code.abort* (*STR* ''*insert DList-set*: *ceq* = *None*'') (λ-. *Set.insert x* (*DList-set dxs*))
| *Some* - ⇒ *DList-set* (*DList-Set.insert x dxs*))
$\bigwedge x.$ *Set.insert x* (*RBT-set rbt*) =
(*case ID CCOMPARE*('*b*) *of None* ⇒ *Code.abort* (*STR* ''*insert RBT-set*: *ccompare* = *None*'') (λ-. *Set.insert x* (*RBT-set rbt*))
| *Some* - ⇒ *RBT-set* (*RBT-Set2.insert x rbt*))
and *insert-Complement* [*set-complement-code*]:
$\bigwedge x.$ *Set.insert x* (*Complement X*) = *Complement* (*Set.remove x X*)
⟨*proof*⟩

lemma *Set-member-code* [*code*]:
fixes *xs* :: '*a* :: *ceq list* **shows**
$\bigwedge x.$ $x \in$ *Collect-set A* ⟷ *A x*
$\bigwedge x.$ $x \in$ *DList-set dxs* ⟷ *DList-Set.member dxs x*
$\bigwedge x.$ $x \in$ *RBT-set rbt* ⟷ *RBT-Set2.member rbt x*
and *mem-Complement* [*set-complement-code*]:
$\bigwedge x.$ $x \in$ *Complement X* ⟷ $x \notin X$
and
$\bigwedge x.$ $x \in$ *Set-Monad xs* ⟷
(*case ID CEQ*('*a*) *of None* ⇒ *Code.abort* (*STR* ''*member Set-Monad*: *ceq* = *None*'') (λ-. $x \in$ *Set-Monad xs*)
| *Some eq* ⇒ *equal-base.list-member eq xs x*)
⟨*proof*⟩

lemma *Set-remove-code* [*code*]:
fixes *rbt* :: '*a* :: *ccompare set-rbt*
and *dxs* :: '*b* :: *ceq set-dlist* **shows**
$\bigwedge x.$ *Set.remove x* (*Collect-set A*) =
(*case ID CEQ*('*b*) *of None* ⇒ *Code.abort* (*STR* ''*remove Collect*: *ceq* = *None*'') (λ-. *Set.remove x* (*Collect-set A*))
| *Some eq* ⇒ *Collect-set* (*equal-base.fun-upd eq A x False*))
$\bigwedge x.$ *Set.remove x* (*DList-set dxs*) =
(*case ID CEQ*('*b*) *of None* ⇒ *Code.abort* (*STR* ''*remove DList-set*: *ceq* = *None*'') (λ-. *Set.remove x* (*DList-set dxs*))
| *Some* - ⇒ *DList-set* (*DList-Set.remove x dxs*))
$\bigwedge x.$ *Set.remove x* (*RBT-set rbt*) =
(*case ID CCOMPARE*('*a*) *of None* ⇒ *Code.abort* (*STR* ''*remove RBT-set*: *ccompare* = *None*'') (λ-. *Set.remove x* (*RBT-set rbt*))
| *Some* - ⇒ *RBT-set* (*RBT-Set2.remove x rbt*))
and *remove-Complement* [*set-complement-code*]:
$\bigwedge x\ A.$ *Set.remove x* (*Complement A*) = *Complement* (*Set.insert x A*)
⟨*proof*⟩

lemma *Set-uminus-code* [*code*, *set-complement-code*]:
− *A* = *Complement A*
− (*Collect-set P*) = *Collect-set* (λ*x*. ¬ *P x*)

$-$ (*Complement B*) = *B*
⟨*proof*⟩

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* ⇒ *Complement A*
| *Some* (*enum*, -) ⇒ *Set-Monad* (*filter* (λ*x*. *x* ∉ *A*) *enum*))
and $-$ (*Complement B*) = *B*
⟨*proof*⟩

lemma *Set-minus-code* [*code*]:
fixes *rbt1 rbt2* :: *'a* :: *ccompare set-rbt*
shows *A* − *B* = *A* ∩ (− *B*)
RBT-set rbt1 − *RBT-set rbt2* =
(*case ID CCOMPARE('a) of None* ⇒ *Code.abort* (*STR ''minus RBT-set RBT-set*: *ccompare* = *None''*) (λ-. *RBT-set rbt1* − *RBT-set rbt2*)
| *Some* - ⇒ *RBT-set* (*RBT-Set2.minus rbt1 rbt2*))
⟨*proof*⟩

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**
RBT-set rbt1 ∪ *RBT-set rbt2* =
(*case ID CCOMPARE('a) of None* ⇒ *Code.abort* (*STR ''union RBT-set RBT-set*: *ccompare* = *None''*) (λ-. *RBT-set rbt1* ∪ *RBT-set rbt2*)
| *Some* - ⇒ *RBT-set* (*RBT-Set2.union rbt1 rbt2*)) (**is** *?RBT-set-RBT-set*)
RBT-set rbt ∪ *DList-set dxs* =
(*case ID CCOMPARE('b) of None* ⇒ *Code.abort* (*STR ''union RBT-set DList-set*: *ccompare* = *None''*) (λ-. *RBT-set rbt* ∪ *DList-set dxs*)
| *Some* - ⇒
case ID CEQ('b) of None ⇒ *Code.abort* (*STR ''union RBT-set DList-set*: *ceq* = *None''*) (λ-. *RBT-set rbt* ∪ *DList-set dxs*)
| *Some* - ⇒ *RBT-set* (*DList-Set.fold RBT-Set2.insert dxs rbt*))
(**is** *?RBT-set-DList-set*)
DList-set dxs ∪ *RBT-set rbt* =
(*case ID CCOMPARE('b) of None* ⇒ *Code.abort* (*STR ''union DList-set RBT-set*: *ccompare* = *None''*) (λ-. *RBT-set rbt* ∪ *DList-set dxs*)
| *Some* - ⇒
case ID CEQ('b) of None ⇒ *Code.abort* (*STR ''union DList-set RBT-set*: *ceq* = *None''*) (λ-. *RBT-set rbt* ∪ *DList-set dxs*)
| *Some* - ⇒ *RBT-set* (*DList-Set.fold RBT-Set2.insert dxs rbt*))
(**is** *?DList-set-RBT-set*)

DList-set dxs1 ∪ *DList-set dxs2* =
(*case ID CEQ*(*'c*) *of None* ⇒ *Code.abort* (*STR ''union DList-set DList-set*: *ceq* = *None''*) (λ-. *DList-set dxs1* ∪ *DList-set dxs2*)
| *Some* - ⇒ *DList-set* (*DList-Set.union dxs1 dxs2*)) (**is** *?DList-set-DList-set*)
Set-Monad zs ∪ *RBT-set rbt2* =
(*case ID CCOMPARE*(*'a*) *of None* ⇒ *Code.abort* (*STR ''union Set-Monad RBT-set*: *ccompare* = *None''*) (λ-. *Set-Monad zs* ∪ *RBT-set rbt2*)
| *Some* - ⇒ *RBT-set* (*fold RBT-Set2.insert zs rbt2*)) (**is** *?Set-Monad-RBT-set*)
RBT-set rbt1 ∪ *Set-Monad zs* =
(*case ID CCOMPARE*(*'a*) *of None* ⇒ *Code.abort* (*STR ''union RBT-set Set-Monad*: *ccompare* = *None''*) (λ-. *RBT-set rbt1* ∪ *Set-Monad zs*)
| *Some* - ⇒ *RBT-set* (*fold RBT-Set2.insert zs rbt1*)) (**is** *?RBT-set-Set-Monad*)
Set-Monad ws ∪ *DList-set dxs2* =
(*case ID CEQ*(*'c*) *of None* ⇒ *Code.abort* (*STR ''union Set-Monad DList-set*: *ceq* = *None''*) (λ-. *Set-Monad ws* ∪ *DList-set dxs2*)
| *Some* - ⇒ *DList-set* (*fold DList-Set.insert ws dxs2*)) (**is** *?Set-Monad-DList-set*)
DList-set dxs1 ∪ *Set-Monad ws* =
(*case ID CEQ*(*'c*) *of None* ⇒ *Code.abort* (*STR ''union DList-set Set-Monad*: *ceq* = *None''*) (λ-. *DList-set dxs1* ∪ *Set-Monad ws*)
| *Some* - ⇒ *DList-set* (*fold DList-Set.insert ws dxs1*)) (**is** *?DList-set-Set-Monad*)
Set-Monad xs ∪ *Set-Monad ys* = *Set-Monad* (*xs* @ *ys*)
Collect-set A ∪ *B* = *Collect-set* (λ*x*. *A x* ∨ *x* ∈ *B*)
B ∪ *Collect-set A* = *Collect-set* (λ*x*. *A x* ∨ *x* ∈ *B*)
and *Set-union-Complement* [*set-complement-code*]:
Complement B ∪ *B'* = *Complement* (*B* ∩ − *B'*)
B' ∪ *Complement B* = *Complement* (− *B'* ∩ *B*)
⟨*proof*⟩

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
Collect-set A'' ∩ *J* = *Collect-set* (λ*x*. *A'' x* ∧ *x* ∈ *J*) (**is** *?collect1*)
J ∩ *Collect-set A''* = *Collect-set* (λ*x*. *A'' x* ∧ *x* ∈ *J*) (**is** *?collect2*)

Set-Monad xs'' ∩ *I* = *Set-Monad* (*filter* (λ*x*. *x* ∈ *I*) *xs''*) (**is** *?monad1*)
I ∩ *Set-Monad xs''* = *Set-Monad* (*filter* (λ*x*. *x* ∈ *I*) *xs''*) (**is** *?monad2*)

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*)
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*)

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*)
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*)
and *Set-inter-Complement* [*set-complement-code*]:
Complement B'' ∩ *Complement B'''* = *Complement* (*B''* ∪ *B'''*) (**is** *?complement*)
and
Set-Monad xs ∩ *RBT-set rbt1* =
(*case ID CCOMPARE('a) of None* ⇒ *Code.abort* (*STR ''inter Set-Monad RBT-set: ccompare = None''*) (*λ-. RBT-set rbt1* ∩ *Set-Monad xs*)
| *Some -* ⇒ *RBT-set* (*RBT-Set2.inter-list rbt1 xs*)) (**is** *?monad-rbt*)
Set-Monad xs' ∩ *DList-set dxs2* =
(*case ID CEQ('c) of None* ⇒ *Code.abort* (*STR ''inter Set-Monad DList-set: ceq = None''*) (*λ-. Set-Monad xs'* ∩ *DList-set dxs2*)
| *Some eq* ⇒ *DList-set* (*DList-Set.filter* (*equal-base.list-member eq xs'*) *dxs2*)) (**is** *?monad-dlist*)
Set-Monad xs1 ∩ *Set-Monad xs2* =
(*case ID CEQ('c) of None* ⇒ *Code.abort* (*STR ''inter Set-Monad Set-Monad: ceq = None''*) (*λ-. Set-Monad xs1* ∩ *Set-Monad xs2*)
| *Some eq* ⇒ *Set-Monad* (*filter* (*equal-base.list-member eq xs2*) *xs1*)) (**is** *?monad*)

DList-set dxs ∩ *RBT-set rbt* =
(*case ID CCOMPARE('b) of None* ⇒ *Code.abort* (*STR ''inter DList-set RBT-set: ccompare = None''*) (*λ-. DList-set dxs* ∩ *RBT-set rbt*)
| *Some -* ⇒
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*)
DList-set dxs1 ∩ *DList-set dxs2* =
(*case ID CEQ('c) of None* ⇒ *Code.abort* (*STR ''inter DList-set DList-set: ceq = None''*) (*λ-. DList-set dxs1* ∩ *DList-set dxs2*)
| *Some -* ⇒ *DList-set* (*DList-Set.filter* (*DList-Set.member dxs2*) *dxs1*)) (**is** *?dlist*)

DList-set dxs1 ∩ *Set-Monad xs′* =
(*case ID CEQ*(*′c*) *of None* ⇒ *Code.abort* (*STR ′′inter DList-set Set-Monad*: *ceq* = *None′′*) (λ-. *DList-set dxs1* ∩ *Set-Monad xs′*)
| *Some eq* ⇒ *DList-set* (*DList-Set.filter* (*equal-base.list-member eq xs′*) *dxs1*)) (**is** *?dlist-monad*)

RBT-set rbt1 ∩ *RBT-set rbt2* =
(*case ID CCOMPARE*(*′a*) *of None* ⇒ *Code.abort* (*STR ′′inter RBT-set RBT-set*: *ccompare* = *None′′*) (λ-. *RBT-set rbt1* ∩ *RBT-set rbt2*)
| *Some* - ⇒ *RBT-set* (*RBT-Set2.inter rbt1 rbt2*)) (**is** *?rbt-rbt*)
RBT-set rbt ∩ *DList-set dxs* =
(*case ID CCOMPARE*(*′b*) *of None* ⇒ *Code.abort* (*STR ′′inter RBT-set DList-set*: *ccompare* = *None′′*) (λ-. *RBT-set rbt* ∩ *DList-set dxs*)
| *Some* - ⇒
case ID CEQ(*′b*) *of None* ⇒ *Code.abort* (*STR ′′inter RBT-set DList-set*: *ceq* = *None′′*) (λ-. *RBT-set rbt* ∩ *DList-set dxs*)
| *Some* - ⇒ *RBT-set* (*RBT-Set2.inter-list rbt* (*list-of-dlist dxs*)))
(**is** *?rbt-dlist*)
RBT-set rbt1 ∩ *Set-Monad xs* =
(*case ID CCOMPARE*(*′a*) *of None* ⇒ *Code.abort* (*STR ′′inter RBT-set Set-Monad*: *ccompare* = *None′′*) (λ-. *RBT-set rbt1* ∩ *Set-Monad xs*)
| *Some* - ⇒ *RBT-set* (*RBT-Set2.inter-list rbt1 xs*)) (**is** *?rbt-monad*)

⟨*proof*⟩

lemma *Set-bind-code* [*code*]:
fixes *dxs* :: *′a* :: *ceq set-dlist*
and *rbt* :: *′b* :: *ccompare set-rbt* **shows**
Set.bind (*Set-Monad xs*) *f* = *fold* ((∪) ∘ *f*) *xs* (*Set-Monad* []) (**is** *?Set-Monad*)
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 (*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*)
⟨*proof*⟩

lemma *UNIV-code* [*code*]: *UNIV* = − {}
⟨*proof*⟩

lift-definition *inf-sls* :: *′a* :: *lattice semilattice-set* **is** *inf* ⟨*proof*⟩

lemma *Inf-fin-code* [*code*]: *Inf-fin A* = *set-fold1 inf-sls A*
⟨*proof*⟩

lift-definition *sup-sls* :: *′a* :: *lattice semilattice-set* **is** *sup* ⟨*proof*⟩

lemma *Sup-fin-code* [*code*]: *Sup-fin A* = *set-fold1 sup-sls A*
⟨*proof*⟩

lift-definition *inf-cfi* :: ('*a* :: *lattice*, '*a*) *comp-fun-idem* **is** *inf*
⟨*proof*⟩

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*))
⟨*proof*⟩

lift-definition *sup-cfi* :: ('*a* :: *lattice*, '*a*) *comp-fun-idem* **is** *sup*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemmas *Inter-code* [*code*] = *Inf-code*[**where** *?'a* = - :: *type set*]
lemmas *Union-code* [*code*] = *Sup-code*[**where** *?'a* = - :: *type set*]
lemmas *Predicate-Inf-code* [*code*] = *Inf-code*[**where** *?'a* = - :: *type Predicate.pred*]
lemmas *Predicate-Sup-code* [*code*] = *Sup-code*[**where** *?'a* = - :: *type Predicate.pred*]
lemmas *Inf-fun-code* [*code*] = *Inf-code*[**where** *?'a* = - :: *type* ⇒ - :: *complete-lattice*]
lemmas *Sup-fun-code* [*code*] = *Sup-code*[**where** *?'a* = - :: *type* ⇒ - :: *complete-lattice*]

lift-definition *min-sls* :: '*a* :: *linorder semilattice-set* **is** *min* ⟨*proof*⟩

lemma *Min-code* [*code*]: *Min A* = *set-fold1 min-sls A*
⟨*proof*⟩

lift-definition *max-sls* :: '*a* :: *linorder semilattice-set* **is** *max* ⟨*proof*⟩

lemma *Max-code* [*code*]: *Max A* = *set-fold1 max-sls A*
⟨*proof*⟩

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*]:
fixes *rbt* :: '*a* :: *ccompare set-rbt*
and *dxs* :: '*b* :: *ceq set-dlist* **shows**
Ball (*Set-Monad xs*) *P* = *list-all P xs*
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 (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)
⟨*proof*⟩

lemma *Bex-code* [*code*]:
fixes *rbt* :: *'a* :: *ccompare set-rbt*
and *dxs* :: *'b* :: *ceq set-dlist* **shows**
Bex (Set-Monad xs) P = list-ex P xs
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 (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)
⟨*proof*⟩

lemma *csorted-list-of-set-code* [*code*]:
fixes *rbt* :: *'a* :: *ccompare set-rbt*
and *dxs* :: *'b* :: {*ccompare*, *ceq*} *set-dlist*
and *xs* :: *'a* :: *ccompare list* **shows**
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)
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.quicksort (lt-of-comp c) (list-of-dlist dxs))
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.quicksort (lt-of-comp c) xs))
⟨*proof*⟩

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 A B ⟷
(case ID CCOMPARE('a) of None ⇒ Code.abort (STR ''cless-set: ccompare =

None'') (λ-. *cless-set A B*)
| *Some c* ⇒
if finite A ∧ *finite B then ord.lexordp* (λ*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''*) (λ-. *cless-set A B*))
(**is** *?fin-fin*)
and *cless-set-Complement2* [*set-complement-code*]:
cless-set A' (*Complement B'*) ⟷
(*case ID CCOMPARE*(*'b*) *of None* ⇒ *Code.abort* (*STR ''cless-set Complement2*: *ccompare* = *None''*) (λ-. *cless-set A'* (*Complement B'*))
| *Some c* ⇒
if finite A' ∧ *finite B' then*
finite (*UNIV* :: *'b set*) ⟶
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''*) (λ-. *cless-set A'* (*Complement B'*)))
(**is** *?fin-Compl-fin*)
and *cless-set-Complement1* [*set-complement-code*]:
cless-set (*Complement A'*) *B'* ⟷
(*case ID CCOMPARE*(*'b*) *of None* ⇒ *Code.abort* (*STR ''cless-set Complement1*: *ccompare* = *None''*) (λ-. *cless-set* (*Complement A'*) *B'*)
| *Some c* ⇒
if finite A' ∧ *finite B' then*
finite (*UNIV* :: *'b set*) ∧
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''*) (λ-. *cless-set* (*Complement A'*) *B'*))
(**is** *?Compl-fin-fin*)
and *cless-set-Complement12* [*set-complement-code*]:
cless-set (*Complement A*) (*Complement B*) ⟷
(*case ID CCOMPARE*(*'a*) *of None* ⇒ *Code.abort* (*STR ''cless-set Complement Complement*: *ccompare* = *None''*) (λ-. *cless-set* (*Complement A*) (*Complement B*))
| *Some* - ⇒ *cless B A*) (**is** *?Compl-Compl*)
and
cless-set (*RBT-set rbt*) (*RBT-set rbt'*) ⟷
(*case ID CCOMPARE*(*'a*) *of None* ⇒ *Code.abort* (*STR ''cless-set RBT-set RBT-set*: *ccompare* = *None''*) (λ-. *cless-set* (*RBT-set rbt*) (*RBT-set rbt'*))
| *Some c* ⇒ *ord.lexord-fusion* (λ*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-rbt-Complement2* [*set-complement-code*]:
cless-set (*RBT-set rbt1*) (*Complement* (*RBT-set rbt2*)) ⟷
(*case ID CCOMPARE*(*'b*) *of None* ⇒ *Code.abort* (*STR ''cless-set RBT-set* (*Complement RBT-set*): *ccompare* = *None''*) (λ-. *cless-set* (*RBT-set rbt1*) (*Complement* (*RBT-set rbt2*)))
| *Some c* ⇒
finite (*UNIV* :: *'b set*) ⟶

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 *cless-set-rbt-Complement1* [*set-complement-code*]:
cless-set (*Complement* (*RBT-set rbt1*)) (*RBT-set rbt2*) ⟷
(*case ID CCOMPARE*(*'b*) *of None* ⇒ *Code.abort* (*STR ''cless-set* (*Complement RBT-set*) *RBT-set*: *ccompare* = *None''*) (λ-. *cless-set* (*Complement* (*RBT-set rbt1*)) (*RBT-set rbt2*))
| *Some c* ⇒
finite (*UNIV* :: *'b set*) ∧
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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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 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 (λ*x 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*)
and *cless-eq-set-Complement2* [*set-complement-code*]:
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 *cless-eq-set-Complement1* [*set-complement-code*]:
cless-eq-set (*Complement A'*) *B'* ⟷
(*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-Complement12* [*set-complement-code*]:
cless-eq-set (*Complement A*) (*Complement B*) ⟷
(*case ID CCOMPARE*(′*a*) *of None* ⇒ *Code.abort* (*STR* ′′*cless-eq-set Complement Complement*: *ccompare* = *None*′′) (λ-. *cless-eq* (*Complement A*) (*Complement B*))
| *Some c* ⇒ *cless-eq-set B A*)
(**is** *?Compl-Compl*)

cless-eq-set (*RBT-set rbt*) (*RBT-set rbt′*) ⟷
(*case ID CCOMPARE*(′*a*) *of None* ⇒ *Code.abort* (*STR* ′′*cless-eq-set RBT-set RBT-set*: *ccompare* = *None*′′) (λ-. *cless-eq-set* (*RBT-set rbt*) (*RBT-set rbt′*))
| *Some c* ⇒ *ord.lexord-eq-fusion* (λ*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-rbt-Complement2* [*set-complement-code*]:
cless-eq-set (*RBT-set rbt1*) (*Complement* (*RBT-set rbt2*)) ⟷
(*case ID CCOMPARE*(′*b*) *of None* ⇒ *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* ⇒
finite (*UNIV* :: ′*b set*) ⟶
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 *cless-eq-set-rbt-Complement1* [*set-complement-code*]:
cless-eq-set (*Complement* (*RBT-set rbt1*)) (*RBT-set rbt2*) ⟷
(*case ID CCOMPARE*(′*b*) *of None* ⇒ *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* ⇒
finite (*UNIV* :: ′*b set*) ∧
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*)
⟨*proof*⟩

lemma *cproper-interval-set-Some-Some-code* [*code*]:
fixes *rbt1 rbt2* :: ′*a* :: *cproper-interval set-rbt*
and *A B* :: ′*a set* **shows**

cproper-interval (*Some A*) (*Some B*) ⟷
(*case ID CCOMPARE*(′*a*) *of None* ⇒ *Code.abort* (*STR* ′′*cproper-interval*: *ccom-*

pare = None'') (λ-. *cproper-interval* (*Some A*) (*Some B*))
| *Some c* ⇒
finite (*UNIV* :: *'a set*) ∧ *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*)
and *cproper-interval-set-Some-Some-Complement* [*set-complement-code*]:
cproper-interval (*Some A*) (*Some* (*Complement B*)) ⟷
(*case ID CCOMPARE*(*'a*) *of None* ⇒ *Code.abort* (*STR ''cproper-interval Complement2*: *ccompare* = *None''*) (λ-. *cproper-interval* (*Some A*) (*Some* (*Complement B*)))
| *Some c* ⇒
finite (*UNIV* :: *'a set*) ∧ *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 *cproper-interval-set-Some-Complement-Some* [*set-complement-code*]:
cproper-interval (*Some* (*Complement A*)) (*Some B*) ⟷
(*case ID CCOMPARE*(*'a*) *of None* ⇒ *Code.abort* (*STR ''cproper-interval Complement1*: *ccompare* = *None''*) (λ-. *cproper-interval* (*Some* (*Complement A*)) (*Some B*))
| *Some c* ⇒
finite (*UNIV* :: *'a set*) ∧ *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-Complement-Some-Complement* [*set-complement-code*]:
cproper-interval (*Some* (*Complement A*)) (*Some* (*Complement B*)) ⟷
(*case ID CCOMPARE*(*'a*) *of None* ⇒ *Code.abort* (*STR ''cproper-interval Complement Complement*: *ccompare* = *None''*) (λ-. *cproper-interval* (*Some* (*Complement A*)) (*Some* (*Complement B*)))
| *Some* - ⇒ *cproper-interval* (*Some B*) (*Some A*))
(**is** *?Compl-Compl*)

cproper-interval (*Some* (*RBT-set rbt1*)) (*Some* (*RBT-set rbt2*)) ⟷
(*case ID CCOMPARE*(*'a*) *of None* ⇒ *Code.abort* (*STR ''cproper-interval RBT-set RBT-set*: *ccompare* = *None''*) (λ-. *cproper-interval* (*Some* (*RBT-set rbt1*)) (*Some* (*RBT-set rbt2*)))
| *Some c* ⇒
finite (*UNIV* :: *'a set*) ∧ *proper-intrvl.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-rbt-Some-Complement* [*set-complement-code*]:
cproper-interval (*Some* (*RBT-set rbt1*)) (*Some* (*Complement* (*RBT-set rbt2*))) ⟷
(*case ID CCOMPARE*(*'a*) *of None* ⇒ *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* ⇒
finite (*UNIV* :: *'a set*) ∧ *proper-intrvl.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 *cproper-interval-set-Some-Complement-Some-rbt* [*set-complement-code*]:
  cproper-interval (*Some* (*Complement* (*RBT-set rbt1*))) (*Some* (*RBT-set rbt2*)) $\longleftrightarrow$
  (*case ID CCOMPARE*('*a*) *of None* ⇒ *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* ⇒
  finite (*UNIV* :: '*a set*) ∧ *proper-intrvl.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*)
⟨*proof*⟩

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*
  ⟨*proof*⟩

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)
⟨*proof*⟩

end

lemmas [*code*] = *ord.sorted-list-subset-fusion-code*

lemma *subset-eq-code* [*code*]:
fixes *A1 A2* :: ′*a set*
and *rbt* :: ′*b* :: *ccompare set-rbt*
and *rbt1 rbt2* :: ′*d* :: {*ccompare*, *ceq*} *set-rbt*
and *dxs* :: ′*c* :: *ceq set-dlist*
and *xs* :: ′*c list* **shows**
RBT-set rbt ⊆ *B* ⟷
(*case ID CCOMPARE*(′*b*) *of None* ⇒ *Code.abort* (*STR* ′′*subset RBT-set1*: *ccompare* = *None*′′) (λ-. *RBT-set rbt* ⊆ *B*)
| *Some* - ⇒ *list-all-fusion rbt-keys-generator* (λ*x*. *x* ∈ *B*) (*RBT-Set2.init rbt*)) (**is** *?rbt*)
DList-set dxs ⊆ *C* ⟷
(*case ID CEQ*(′*c*) *of None* ⇒ *Code.abort* (*STR* ′′*subset DList-set1*: *ceq* = *None*′′) (λ-. *DList-set dxs* ⊆ *C*)
| *Some* - ⇒ *DList-Set.dlist-all* (λ*x*. *x* ∈ *C*) *dxs*) (**is** *?dlist*)
Set-Monad xs ⊆ *C* ⟷ *list-all* (λ*x*. *x* ∈ *C*) *xs* (**is** *?Set-Monad*)
and *Collect-subset-eq-Complement* [*set-complement-code*]:
Collect-set P ⊆ *Complement A* ⟷ *A* ⊆ {*x*. ¬ *P x*} (**is** *?Collect-set-Compl*)
and *Complement-subset-eq-Complement* [*set-complement-code*]:
Complement A1 ⊆ *Complement A2* ⟷ *A2* ⊆ *A1* (**is** *?Compl*)
and
RBT-set rbt1 ⊆ *RBT-set rbt2* ⟷
(*case ID CCOMPARE*(′*d*) *of None* ⇒ *Code.abort* (*STR* ′′*subset RBT-set RBT-set*: *ccompare* = *None*′′) (λ-. *RBT-set rbt1* ⊆ *RBT-set rbt2*)
| *Some c* ⇒
(*case ID CEQ*(′*d*) *of None* ⇒ *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* ⇒ *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*)
⟨*proof*⟩

lemma *set-eq-code* [*code*]:
fixes *rbt1 rbt2* :: ′*b* :: {*ccompare*, *ceq*} *set-rbt* **shows**
set-eq A B ⟷ *A* ⊆ *B* ∧ *B* ⊆ *A*
and *set-eq-Complement-Complement* [*set-complement-code*]:
set-eq (*Complement A*) (*Complement B*) = *set-eq A B*
and

set-eq (*RBT-set rbt1*) (*RBT-set rbt2*) =
(*case ID CCOMPARE*(′*b*) *of None* ⇒ *Code.abort* (*STR* ′′*set-eq RBT-set RBT-set*: *ccompare* = *None*′′) (λ-. *set-eq* (*RBT-set rbt1*) (*RBT-set rbt2*))
| *Some c* ⇒
(*case ID CEQ*(′*b*) *of None* ⇒ *list-all2-fusion* (λ *x y*. *c x y* = *Eq*) *rbt-keys-generator rbt-keys-generator* (*RBT-Set2.init rbt1*) (*RBT-Set2.init rbt2*)
| *Some eq* ⇒ *list-all2-fusion eq rbt-keys-generator rbt-keys-generator* (*RBT-Set2.init rbt1*) (*RBT-Set2.init rbt2*)))
(**is** *?rbt-rbt*)
⟨*proof*⟩

lemma *Set-project-code* [*code*]:
Set.filter P A = *A* ∩ *Collect-set P*
⟨*proof*⟩

lemma *Set-image-code* [*code*]:
fixes *dxs* :: ′*a* :: *ceq set-dlist*
and *rbt* :: ′*b* :: *ccompare set-rbt* **shows**
image f (*Set-Monad xs*) = *Set-Monad* (*map f xs*)
image f (*Collect-set A*) = *Code.abort* (*STR* ′′*image Collect-set*′′) (λ-. *image f* (*Collect-set A*))
and *image-Complement-Complement* [*set-complement-code*]:
image f (*Complement* (*Complement B*)) = *image f B*
and
image g (*DList-set dxs*) =
(*case ID CEQ*(′*a*) *of None* ⇒ *Code.abort* (*STR* ′′*image DList-set*: *ceq* = *None*′′) (λ-. *image g* (*DList-set dxs*))
| *Some* - ⇒ *DList-Set.fold* (*insert* ∘ *g*) *dxs* {})
(**is** *?dlist*)
image h (*RBT-set rbt*) =
(*case ID CCOMPARE*(′*b*) *of None* ⇒ *Code.abort* (*STR* ′′*image RBT-set*: *ccompare* = *None*′′) (λ-. *image h* (*RBT-set rbt*))
| *Some* - ⇒ *RBT-Set2.fold* (*insert* ∘ *h*) *rbt* {})
(**is** *?rbt*)
⟨*proof*⟩

lemma *the-elem-code* [*code*]:
fixes *dxs* :: ′*a* :: *ceq set-dlist*
and *rbt* :: ′*b* :: *ccompare set-rbt* **shows**
the-elem (*Set-Monad* [*x*]) = *x*
the-elem (*DList-set dxs*) =
(*case ID CEQ*(′*a*) *of None* ⇒ *Code.abort* (*STR* ′′*the-elem DList-set*: *ceq* = *None*′′) (λ-. *the-elem* (*DList-set dxs*))
| *Some* - ⇒
case list-of-dlist dxs of [*x*] ⇒ *x*
| - ⇒ *Code.abort* (*STR* ′′*the-elem DList-set*: *not unique*′′) (λ-. *the-elem* (*DList-set dxs*)))
the-elem (*RBT-set rbt*) =
(*case ID CCOMPARE*(′*b*) *of None* ⇒ *Code.abort* (*STR* ′′*the-elem RBT-set*: *ccom-*

pare = None'') (λ-. *the-elem* (*RBT-set rbt*))
| *Some* - ⇒
case RBT-Mapping2.impl-of rbt of RBT-Impl.Branch - RBT-Impl.Empty x - RBT-Impl.Empty ⇒ *x*
| - ⇒ *Code.abort* (*STR ''the-elem RBT-set: not unique''*) (λ-. *the-elem* (*RBT-set rbt*)))
⟨*proof*⟩

lemma *Pow-set-conv-fold*:
Pow (*set xs* ∪ *A*) = *fold* (λ*x A*. *A* ∪ *insert x* ‘ *A*) *xs* (*Pow A*)
⟨*proof*⟩

lemma *Pow-code* [*code*]:
fixes *dxs* :: *'a* :: *ceq set-dlist*
and *rbt* :: *'b* :: *ccompare set-rbt* **shows**
Pow A = *Collect-set* (λ*B*. *B* ⊆ *A*)
Pow (*Set-Monad xs*) = *fold* (λ*x A*. *A* ∪ *insert x* ‘ *A*) *xs* {{}}
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* (λ*x A*. *A* ∪ *insert x* ‘ *A*) *dxs* {{}})
Pow (*RBT-set rbt*) =
(*case ID CCOMPARE*(*'b*) *of None* ⇒ *Code.abort* (*STR ''Pow RBT-set: ccompare = None''*) (λ-. *Pow* (*RBT-set rbt*))
| *Some* - ⇒ *RBT-Set2.fold* (λ*x A*. *A* ∪ *insert x* ‘ *A*) *rbt* {{}})
⟨*proof*⟩

lemma *fold-singleton*: *Finite-Set.fold f x* {*y*} = *f y x*
⟨*proof*⟩

lift-definition *sum-cfc* :: (*'a* ⇒ *'b* :: *comm-monoid-add*) ⇒ (*'a*, *'b*) *comp-fun-commute*
is λ*f* :: *'a* ⇒ *'b*. *plus* ∘ *f*
⟨*proof*⟩

lemma *sum-code* [*code*]:
sum f A = (*if finite A then set-fold-cfc* (*sum-cfc f*) *0 A else 0*)
⟨*proof*⟩

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 A B = *Collect-set* (λ(*x*, *y*). *x* ∈ *A* ∧ *y* ∈ *B*)

Product-Type.product (*Set-Monad xs*) (*Set-Monad ys*) =
Set-Monad (*fold* (λ*x*. *fold* (λ*y rest*. (*x*, *y*) # *rest*) *ys*) *xs* [])
(**is** *?Set-Monad*)

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* (λ*x rest*. *Pair x* ‘ *B1* ∪ *rest*) *dxs* {})
(**is** *?dlist1*)

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* (λ*y rest*. (λ*x*. (*x*, *y*)) ‘ *A1* ∪ *rest*) *dys* {})
(**is** *?dlist2*)

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 (*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* (λ*x rest*. *Pair x* ‘ *B2* ∪ *rest*) *rbt1* {})
(**is** *?rbt1*)

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* (λ*y rest*. (λ*x*. (*x*, *y*)) ‘ *A2* ∪ *rest*) *rbt2* {})
(**is** *?rbt2*)

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*))
⟨*proof*⟩

lemma *Id-on-code* [*code*]:
fixes *A* :: '*a* :: *ceq set*
and *dxs* :: '*a set-dlist*
and *P* :: '*a* ⇒ *bool*
and *rbt* :: '*b* :: *ccompare set-rbt* **shows**
Id-on B = (λ*x*. (*x*, *x*)) ‘ *B*

and *Id-on-Complement* [*set-complement-code*]:
Id-on (*Complement A*) =
(*case ID CEQ*(*'a*) *of None* ⇒ *Code.abort* (*STR ''Id-on Complement*: *ceq* = *None''*) (λ-. *Id-on* (*Complement A*))
| *Some eq* ⇒ *Collect-set* (λ(*x*, *y*). *eq x y* ∧ *x* ∉ *A*))
and
Id-on (*Collect-set P*) =
(*case ID CEQ*(*'a*) *of None* ⇒ *Code.abort* (*STR ''Id-on Collect-set*: *ceq* = *None''*) (λ-. *Id-on* (*Collect-set P*))
| *Some eq* ⇒ *Collect-set* (λ(*x*, *y*). *eq x y* ∧ *P x*))
Id-on (*DList-set dxs*) =
(*case ID CEQ*(*'a*) *of None* ⇒ *Code.abort* (*STR ''Id-on DList-set*: *ceq* = *None''*) (λ-. *Id-on* (*DList-set dxs*))
| *Some* - ⇒ *DList-set* (*DList-Set.Id-on dxs*))
Id-on (*RBT-set rbt*) =
(*case ID CCOMPARE*(*'b*) *of None* ⇒ *Code.abort* (*STR ''Id-on RBT-set*: *ccompare* = *None''*) (λ-. *Id-on* (*RBT-set rbt*))
| *Some* - ⇒ *RBT-set* (*RBT-Set2.Id-on rbt*))
⟨*proof*⟩

lemma *Image-code* [*code*]:
fixes *dxs* :: (*'a* :: *ceq* × *'b* :: *ceq*) *set-dlist*
and *rbt* :: (*'c* :: *ccompare* × *'d* :: *ccompare*) *set-rbt* **shows**
X ‘‘ *Y* = *snd* ‘ *Set.filter* (λ(*x*, *y*). *x* ∈ *Y*) *X*
(**is** *?generic*)

Set-Monad rxs ‘‘ *A* = *Set-Monad* (*fold* (λ(*x*, *y*) *rest*. *if x* ∈ *A then y* # *rest else rest*) *rxs* [])
(**is** *?Set-Monad*)
DList-set dxs ‘‘ *B* =
(*case ID CEQ*(*'a*) *of None* ⇒ *Code.abort* (*STR ''Image DList-set*: *ceq1* = *None''*) (λ-. *DList-set dxs* ‘‘ *B*)
| *Some* - ⇒
case ID CEQ(*'b*) *of None* ⇒ *Code.abort* (*STR ''Image DList-set*: *ceq2* = *None''*) (λ-. *DList-set dxs* ‘‘ *B*)
| *Some* - ⇒
DList-Set.fold (λ(*x*, *y*) *acc*. *if x* ∈ *B then insert y acc else acc*) *dxs* {})
(**is** *?DList-set*)
RBT-set rbt ‘‘ *C* =
(*case ID CCOMPARE*(*'c*) *of None* ⇒ *Code.abort* (*STR ''Image RBT-set*: *ccompare1* = *None''*) (λ-. *RBT-set rbt* ‘‘ *C*)
| *Some* - ⇒
case ID CCOMPARE(*'d*) *of None* ⇒ *Code.abort* (*STR ''Image RBT-set*: *ccompare2* = *None''*) (λ-. *RBT-set rbt* ‘‘ *C*)
| *Some* - ⇒
RBT-Set2.fold (λ(*x*, *y*) *acc*. *if x* ∈ *C then insert y acc else acc*) *rbt* {})
(**is** *?RBT-set*)
⟨*proof*⟩

lemma *insert-relcomp*: *insert* (*a*, *b*) *A O B* = *A O B* ∪ {*a*} × {*c*. (*b*, *c*) ∈ *B*}
⟨*proof*⟩

lemma *trancl-code* [*code*]:
trancl A =
(*if finite A then ntrancl* (*card A* − *1*) *A else Code.abort* (*STR* ''*trancl*: *infinite set*'') (λ-. *trancl A*))
⟨*proof*⟩

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*⟩

lemma *If-not*: (*if* ¬ *a then b else c*) = (*if a then c else b*)
⟨*proof*⟩

lemma *relcomp-code* [*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 *dxs1* :: (′*d* × ′*e* :: *ceq*) *set-dlist*
and *dxs2* :: (′*e* × ′*d*) *set-dlist*
and *dxs3* :: (′*e* × ′*f* :: *ceq*) *set-dlist*
and *dxs4* :: (′*f* × ′*g* :: *ceq*) *set-dlist*
and *xs1* :: (′*h* × ′*i* :: *ceq*) *list*
and *xs2* :: (′*i* × ′*j*) *list*
and *xs3* :: (′*b* × ′*h*) *list*
and *xs4* :: (′*h* × ′*b*) *list*
and *xs5* :: (′*f* × ′*h*) *list*
and *xs6* :: (′*h* × ′*f*) *list*
shows
RBT-set rbt1 O RBT-set rbt2 =
(*case ID CCOMPARE*(′*a*) *of None* ⇒ *Code.abort* (*STR* ''*relcomp RBT-set RBT-set*: *ccompare1* = *None*'') (λ-. *RBT-set rbt1 O RBT-set rbt2*)
| *Some* - ⇒
case ID CCOMPARE(′*b*) *of None* ⇒ *Code.abort* (*STR* ''*relcomp RBT-set RBT-set*: *ccompare2* = *None*'') (λ-. *RBT-set rbt1 O RBT-set rbt2*)
| *Some c-b* ⇒
case ID CCOMPARE(′*c*) *of None* ⇒ *Code.abort* (*STR* ''*relcomp RBT-set RBT-set*: *ccompare3* = *None*'') (λ-. *RBT-set rbt1 O RBT-set rbt2*)
| *Some* - ⇒ *RBT-Set2.fold* (λ(*x*, *y*). *RBT-Set2.fold* (λ(*y*′, *z*) *A*. *if c-b y y*′ ≠ *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* ⇒ *Code.abort* (*STR* ''*relcomp RBT-set DList-set*:

ccompare1 = None'') (λ-. *RBT-set rbt3 O DList-set dxs1*)
| *Some - ⇒*
case ID CCOMPARE('d) of None ⇒ Code.abort (*STR ''relcomp RBT-set DList-set*: *ccompare2 = None''*) (λ-. *RBT-set rbt3 O DList-set dxs1*)
| *Some - ⇒*
case ID CEQ('d) of None ⇒ Code.abort (*STR ''relcomp RBT-set DList-set*: *ceq2 = None''*) (λ-. *RBT-set rbt3 O DList-set dxs1*)
| *Some eq ⇒*
case ID CEQ('e) of None ⇒ Code.abort (*STR ''relcomp RBT-set DList-set*: *ceq3 = None''*) (λ-. *RBT-set rbt3 O DList-set dxs1*)
| *Some - ⇒ RBT-Set2.fold* (λ(*x*, *y*). *DList-Set.fold* (λ(*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 ⇒ Code.abort* (*STR ''relcomp DList-set RBT-set*: *ceq1 = None''*) (λ-. *DList-set dxs2 O RBT-set rbt4*)
| *Some - ⇒*
case ID CCOMPARE('d) of None ⇒ Code.abort (*STR ''relcomp DList-set RBT-set*: *ceq2 = None''*) (λ-. *DList-set dxs2 O RBT-set rbt4*)
| *Some - ⇒*
case ID CEQ('d) of None ⇒ Code.abort (*STR ''relcomp DList-set RBT-set*: *ccompare2 = None''*) (λ-. *DList-set dxs2 O RBT-set rbt4*)
| *Some eq ⇒*
case ID CCOMPARE('a) of None ⇒ Code.abort (*STR ''relcomp DList-set RBT-set*: *ccompare3 = None''*) (λ-. *DList-set dxs2 O RBT-set rbt4*)
| *Some - ⇒ DList-Set.fold* (λ(*x*, *y*). *RBT-Set2.fold* (λ(*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 ⇒ Code.abort* (*STR ''relcomp DList-set DList-set*: *ceq1 = None''*) (λ-. *DList-set dxs3 O DList-set dxs4*)
| *Some - ⇒*
case ID CEQ('f) of None ⇒ Code.abort (*STR ''relcomp DList-set DList-set*: *ceq2 = None''*) (λ-. *DList-set dxs3 O DList-set dxs4*)
| *Some eq ⇒*
case ID CEQ('g) of None ⇒ Code.abort (*STR ''relcomp DList-set DList-set*: *ceq3 = None''*) (λ-. *DList-set dxs3 O DList-set dxs4*)
| *Some - ⇒ DList-Set.fold* (λ(*x*, *y*). *DList-Set.fold* (λ(*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 ⇒ Code.abort* (*STR ''relcomp Set-Monad Set-Monad*: *ceq = None''*) (λ-. *Set-Monad xs1 O Set-Monad xs2*)
| *Some eq ⇒ fold* (λ(*x*, *y*). *fold* (λ(*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* ⇒ *Code.abort* (*STR ''relcomp RBT-set Set-Monad*: *ccompare1* = *None''*) (λ-. *RBT-set rbt1 O Set-Monad xs3*)
| *Some* - ⇒
case ID CCOMPARE(*'b*) *of None* ⇒ *Code.abort* (*STR ''relcomp RBT-set Set-Monad*: *ccompare2* = *None''*) (λ-. *RBT-set rbt1 O Set-Monad xs3*)
| *Some c-b* ⇒ *RBT-Set2.fold* (λ(*x*, *y*). *fold* (λ(*y'*, *z*) *A*. *if c-b y y'* ≠ *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* ⇒ *Code.abort* (*STR ''relcomp Set-Monad RBT-set*: *ccompare1* = *None''*) (λ-. *Set-Monad xs4 O RBT-set rbt5*)
| *Some* - ⇒
case ID CCOMPARE(*'b*) *of None* ⇒ *Code.abort* (*STR ''relcomp Set-Monad RBT-set*: *ccompare2* = *None''*) (λ-. *Set-Monad xs4 O RBT-set rbt5*)
| *Some c-b* ⇒ *fold* (λ(*x*, *y*). *RBT-Set2.fold* (λ(*y'*, *z*) *A*. *if c-b y y'* ≠ *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* ⇒ *Code.abort* (*STR ''relcomp DList-set Set-Monad*: *ceq1* = *None''*) (λ-. *DList-set dxs3 O Set-Monad xs5*)
| *Some* - ⇒
case ID CEQ(*'f*) *of None* ⇒ *Code.abort* (*STR ''relcomp DList-set Set-Monad*: *ceq2* = *None''*) (λ-. *DList-set dxs3 O Set-Monad xs5*)
| *Some eq* ⇒ *DList-Set.fold* (λ(*x*, *y*). *fold* (λ(*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* ⇒ *Code.abort* (*STR ''relcomp Set-Monad DList-set*: *ceq1* = *None''*) (λ-. *Set-Monad xs6 O DList-set dxs4*)
| *Some eq* ⇒
case ID CEQ(*'g*) *of None* ⇒ *Code.abort* (*STR ''relcomp Set-Monad DList-set*: *ceq2* = *None''*) (λ-. *Set-Monad xs6 O DList-set dxs4*)
| *Some* - ⇒ *fold* (λ(*x*, *y*). *DList-Set.fold* (λ(*y'*, *z*) *A*. *if eq y y' then insert* (*x*, *z*) *A else A*) *dxs4*) *xs6* {})
(**is** *?monad-dlist*)
⟨*proof*⟩

lemma *irrefl-on-code* [*code*]:
fixes *r* :: (*'a* :: {*ceq*, *ccompare*} × *'a*) *set* **shows**
irrefl-on A r ⟷
(*case ID CEQ*(*'a*) *of Some eq* ⇒ (∀(*x*, *y*) ∈ *r*. *x* ∈ *A* ⟶ *y* ∈ *A* ⟶ ¬ *eq x y*) | *None* ⇒
case ID CCOMPARE(*'a*) *of None* ⇒ *Code.abort* (*STR ''irrefl-on*: *ceq* = *None* & *ccompare* = *None''*) (λ-. *irrefl-on A r*)

| *Some c* ⇒ (∀ (*x*, *y*) ∈ *r*. *x* ∈ *A* ⟶ *y* ∈ *A* ⟶ *c x y* ≠ *Eq*))
⟨*proof*⟩

lemma *wf-code* [*code*]:
fixes *rbt* :: (′*a* :: *ccompare* × ′*a*) *set-rbt*
and *dxs* :: (′*b* :: *ceq* × ′*b*) *set-dlist* **shows**
wf-code (*Set-Monad xs*) = *acyclic* (*Set-Monad xs*)
wf-code (*RBT-set rbt*) =
(*case ID CCOMPARE*(′*a*) *of None* ⇒ *Code.abort* (*STR* ′′*wf-code RBT-set*: *ccompare* = *None*′′) (λ-. *wf-code* (*RBT-set rbt*))
| *Some* - ⇒ *acyclic* (*RBT-set rbt*))
wf-code (*DList-set dxs*) =
(*case ID CEQ*(′*b*) *of None* ⇒ *Code.abort* (*STR* ′′*wf-code DList-set*: *ceq* = *None*′′) (λ-. *wf-code* (*DList-set dxs*))
| *Some* - ⇒ *acyclic* (*DList-set dxs*))
⟨*proof*⟩

lemma *bacc-code* [*code*]:
bacc R 0 = − *snd* ‘ *R*
bacc R (*Suc n*) = (*let rec* = *bacc R n in rec* ∪ − *snd* ‘ (*Set.filter* (λ(*y*, *x*). *y* ∉ *rec*) *R*))
⟨*proof*⟩

lemma *acc-code* [*code*]:
fixes *A* :: (′*a* :: {*finite*, *card-UNIV*} × ′*a*) *set* **shows**
Wellfounded.acc A = *bacc A* (*of-phantom* (*card-UNIV* :: ′*a card-UNIV*))
⟨*proof*⟩

lemma *sorted-list-of-set-code* [*code*]:
fixes *dxs* :: ′*a* :: {*linorder*, *ceq*} *set-dlist*
and *rbt* :: ′*b* :: {*linorder*, *ccompare*} *set-rbt*
shows
sorted-list-of-set (*Set-Monad xs*) = *sort* (*remdups xs*)
sorted-list-of-set (*DList-set dxs*) =
(*case ID CEQ*(′*a*) *of None* ⇒ *Code.abort* (*STR* ′′*sorted-list-of-set DList-set*: *ceq* = *None*′′) (λ-. *sorted-list-of-set* (*DList-set dxs*))
| *Some* - ⇒ *sort* (*list-of-dlist dxs*))
sorted-list-of-set (*RBT-set rbt*) =
(*case ID CCOMPARE*(′*b*) *of None* ⇒ *Code.abort* (*STR* ′′*sorted-list-of-set RBT-set*: *ccompare* = *None*′′) (λ-. *sorted-list-of-set* (*RBT-set rbt*))
| *Some* - ⇒ *sort* (*RBT-Set2.keys rbt*))
— We must sort the keys because *ccompare*'s ordering need not coincide with *linorder*'s.
⟨*proof*⟩

lemma *map-project-set*: *List.map-project f* (*set xs*) = *set* (*List.map-filter f xs*)
⟨*proof*⟩

lemma *map-project-simps*:
shows *map-project-empty*: *List.map-project f* {} = {}
and *map-project-insert*:
List.map-project f (*insert x A*) =
(*case f x of None* ⇒ *List.map-project f A*
| *Some y* ⇒ *insert y* (*List.map-project f A*))
⟨*proof*⟩

lemma *map-project-conv-fold*:
List.map-project f (*set xs*) =
fold (λ*x A*. *case f x of None* ⇒ *A* | *Some y* ⇒ *insert y A*) *xs* {}
⟨*proof*⟩

lemma *map-project-code* [*code*]:
fixes *dxs* :: ′*a* :: *ceq set-dlist*
and *rbt* :: ′*b* :: *ccompare set-rbt* **shows**
List.map-project f (*Set-Monad xs*) = *Set-Monad* (*List.map-filter f xs*)
List.map-project g (*DList-set dxs*) =
(*case ID CEQ*(′*a*) *of None* ⇒ *Code.abort* (*STR* ′′*map-project DList-set*: *ceq* = *None*′′) (λ-. *List.map-project g* (*DList-set dxs*))
| *Some* - ⇒ *DList-Set.fold* (λ*x A*. *case g x of None* ⇒ *A* | *Some y* ⇒ *insert y A*) *dxs* {})
(**is** *?dlist*)
List.map-project h (*RBT-set rbt*) =
(*case ID CCOMPARE*(′*b*) *of None* ⇒ *Code.abort* (*STR* ′′*map-project RBT-set*: *ccompare* = *None*′′) (λ-. *List.map-project h* (*RBT-set rbt*))
| *Some* - ⇒ *RBT-Set2.fold* (λ*x A*. *case h x of None* ⇒ *A* | *Some y* ⇒ *insert y A*) *rbt* {})
(**is** *?rbt*)
⟨*proof*⟩

lemma *Bleast-code* [*code*]:
Bleast A P =
(*if finite A then case filter P* (*sorted-list-of-set A*) *of* [] ⇒ *abort-Bleast A P* | *x* # *xs* ⇒ *x*
else abort-Bleast A P)
⟨*proof*⟩

lemma *can-select-code* [*code*]:
fixes *xs* :: ′*a* :: *ceq list*
and *dxs* :: ′*a* :: *ceq set-dlist*
and *rbt* :: ′*b* :: *ccompare set-rbt* **shows**
can-select P (*Set-Monad xs*) =
(*case ID CEQ*(′*a*) *of None* ⇒ *Code.abort* (*STR* ′′*can-select Set-Monad*: *ceq* = *None*′′) (λ-. *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*)

can-select Q (*DList-set dxs*) =
(*case ID CEQ*('*a*) *of None* ⇒ *Code.abort* (*STR* ''*can-select DList-set*: *ceq* = *None*'') (λ-. *can-select Q* (*DList-set dxs*))
| *Some* - ⇒ *DList-Set.length* (*DList-Set.filter Q dxs*) = *1*)
(**is** *?dlist*)
can-select R (*RBT-set rbt*) =
(*case ID CCOMPARE*('*b*) *of None* ⇒ *Code.abort* (*STR* ''*can-select RBT-set*: *ccompare* = *None*'') (λ-. *can-select R* (*RBT-set rbt*))
| *Some* - ⇒ *singleton-list-fusion* (*filter-generator R rbt-keys-generator*) (*RBT-Set2.init rbt*))
(**is** *?rbt*)
⟨*proof*⟩

lemma *pred-of-set-code* [*code*]:
fixes *dxs* :: '*a* :: *ceq set-dlist*
and *rbt* :: '*b* :: *ccompare set-rbt* **shows**
pred-of-set (*Set-Monad xs*) = *fold* (*sup* ∘ *Predicate.single*) *xs bot*
pred-of-set (*DList-set dxs*) =
(*case ID CEQ*('*a*) *of None* ⇒ *Code.abort* (*STR* ''*pred-of-set DList-set*: *ceq* = *None*'') (λ-. *pred-of-set* (*DList-set dxs*))
| *Some* - ⇒ *DList-Set.fold* (*sup* ∘ *Predicate.single*) *dxs bot*)
pred-of-set (*RBT-set rbt*) =
(*case ID CCOMPARE*('*b*) *of None* ⇒ *Code.abort* (*STR* ''*pred-of-set RBT-set*: *ccompare* = *None*'') (λ-. *pred-of-set* (*RBT-set rbt*))
| *Some* - ⇒ *RBT-Set2.fold* (*sup* ∘ *Predicate.single*) *rbt bot*)
⟨*proof*⟩

'*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* ⇒ '*a set* ⇒ '*a set*
where [*simp*]: *insert-monad* = *insert*

definition *union-monad* :: '*a set* ⇒ '*a set* ⇒ '*a set*
where [*simp*]: *union-monad* = (∪)

lemma *insert-monad-code* [*code*]:
insert-monad x (*Set-Monad xs*) = *Set-Monad* (*x* # *xs*)
⟨*proof*⟩

lemma *union-monad-code* [*code*]:
union-monad (*Set-Monad xs*) (*Set-Monad ys*) = *Set-Monad* (*xs* @ *ys*)
⟨*proof*⟩

lemma *set-of-pred-code* [*code*]:
set-of-pred (*Predicate.Seq f*) =
(*case f* () *of seq.Empty* ⇒ *Set-Monad* []
| *seq.Insert x P* ⇒ *insert-monad x* (*set-of-pred P*)
| *seq.Join P xq* ⇒ *union-monad* (*set-of-pred P*) (*set-of-seq xq*))

⟨*proof*⟩

lemma *set-of-seq-code* [*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*)
⟨*proof*⟩

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*]:
fixes *x* :: *set-impl*
shows *size x* = *0*
and *size-set-impl x* = *0*
⟨*proof*⟩

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-empty-choose* :: *'a* :: {*ceq*, *ccompare*} *set*) =
(*case CCOMPARE('a) of Some -* ⇒ *RBT-set RBT-Set2.empty*
| *None* ⇒ *case CEQ('a) of None* ⇒ *Set-Monad* [] | *Some -* ⇒ *DList-set* (*DList-Set.empty*))
⟨*proof*⟩

definition *set-impl-choose2* :: *set-impl* ⇒ *set-impl* ⇒ *set-impl*
where [*simp*]: *set-impl-choose2* = (λ- -. *Set-IMPL*)

lemma *set-impl-choose2-code* [*code*]:
set-impl-choose2 x y = *set-Choose*
set-impl-choose2 set-Collect set-Collect = *set-Collect*
set-impl-choose2 set-DList set-DList = *set-DList*

set-impl-choose2 set-RBT set-RBT = set-RBT
set-impl-choose2 set-Monad set-Monad = set-Monad
⟨*proof*⟩

definition *set-empty* :: *set-impl* ⇒ *'a set*
where [*simp*]: *set-empty* = (λ-. {})

lemma *set-empty-code* [*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*
⟨*proof*⟩

class *set-impl* =
fixes *set-impl* :: (*'a*, *set-impl*) *phantom*

syntax (*input*)
-SET-IMPL :: *type* => *logic* (‹(1SET'-IMPL/(1'(-')))›)

syntax-consts
-SET-IMPL == *set-impl*

⟨*ML*⟩

declare [[*code drop*: {}]]

lemma *empty-code* [*code*, *code-unfold*]:
({} :: *'a* :: *set-impl set*) = *set-empty* (*of-phantom SET-IMPL*(*'a*))
⟨*proof*⟩

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*⟩

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 ⟨*proof*⟩
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 ⟨*proof*⟩
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 ⟨*proof*⟩
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 ⟨*proof*⟩
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* ⇒ '*a list* ⇒ '*a set*
where [*simp*, *code del*]: *set-aux* - = *set*

lemma *set-aux-code* [*code*]:
defines *conv* ≡ *foldl* (λ*s* (*x* :: '*a*). *insert x s*)
shows
set-aux impl = *conv* (*set-empty impl*) (**is** *?thesis1*)
set-aux set-Choose =
(*case CCOMPARE*('*a* :: {*ccompare*, *ceq*}) *of Some* - ⇒ *conv* (*RBT-set RBT-Set2.empty*)
| *None* ⇒ *case CEQ*('*a*) *of None* ⇒ *Set-Monad*
| *Some* - ⇒ *conv* (*DList-set DList-Set.empty*)) (**is** *?thesis2*)

set-aux set-Monad = *Set-Monad*
⟨*proof*⟩

lemma *set-code* [*code*]:
fixes *xs* :: *'a* :: *set-impl list*
shows *set xs* = *set-aux* (*of-phantom* (*ID SET-IMPL*(*'a*))) *xs*
⟨*proof*⟩

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*
⟨*proof*⟩

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* ⇒ *'a* :: *ccompare set* ⇒ *'a set*
where *collapse-RBT-set r M* = *set* (*RBT-Set2.keys r*) ∪ *M*

lemma *RBT-set-collapse-RBT-set* [*code-post*]:
fixes *r* :: *'a* :: *ccompare set-rbt*
assumes *code-post* ⟹ *is-ccompare TYPE*(*'a*) **and** *code-post-set*
shows *RBT-set r* = *collapse-RBT-set r* {}
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *collapse-RBT-set-Empty* [*code-post*]:
collapse-RBT-set (*Mapping-RBT rbt.Empty*) *M* = *M*
⟨*proof*⟩

definition *collapse-DList-set* :: *'a* :: *ceq set-dlist* ⇒ *'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*
⟨*proof*⟩

lemma *collapse-DList-set-empty* [*code-post*]: *collapse-DList-set* (*Abs-dlist* []) = {}
⟨*proof*⟩

lemma *collapse-DList-set-Cons* [*code-post*]:
collapse-DList-set (*Abs-dlist* (*x* # *xs*)) = *insert x* (*collapse-DList-set* (*Abs-dlist* *xs*))
⟨*proof*⟩

lemma *Set-Monad-code-post* [*code-post*]:
assumes *code-post-set*
shows *Set-Monad* [] = {}
and *Set-Monad* (*x*#*xs*) = *insert x* (*Set-Monad xs*)
⟨*proof*⟩

end

theory *Mapping-Impl* **imports**
RBT-Mapping2
AssocList
HOL−Library.Mapping
Set-Impl
Containers-Generator
begin

3.13 Different implementations of maps

code-identifier
code-module *Mapping* ⇀ (*SML*) *Mapping-Impl*
| **code-module** *Mapping-Impl* ⇀ (*SML*) *Mapping-Impl*

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* :: *ccompare*, *'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

declare [[*code drop*: *Mapping.lookup*]]

lemma *lookup-Mapping-code* [*code*]:
Mapping.lookup (*Assoc-List-Mapping al*) = *DAList.lookup al*
Mapping.lookup (*RBT-Mapping t*) = *RBT-Mapping2.lookup t*
⟨*proof*⟩

declare [[*code drop*: *Mapping.is-empty*]]

lemma *is-empty-transfer* [*transfer-rule*]:
includes *lifting-syntax*
shows (*pcr-mapping* (=) (=) ===> (=)) (λ*m*. *m* = *Map.empty*) *Mapping.is-empty*
⟨*proof*⟩

lemma *is-empty-Mapping* [*code*]:
fixes *t* :: (′*a* :: *ccompare*, ′*b*) *mapping-rbt* **shows**
Mapping.is-empty (*Assoc-List-Mapping al*) ⟷ *al* = *DAList.empty*
Mapping.is-empty (*RBT-Mapping t*) ⟷
(*case ID CCOMPARE*(′*a*) *of None* ⇒ *Code.abort* (*STR* ′′*is-empty RBT-Mapping*: *ccompare* = *None*′′) (λ-. *Mapping.is-empty* (*RBT-Mapping t*))
| *Some* - ⇒ *RBT-Mapping2.is-empty t*)
⟨*proof*⟩

declare [[*code drop*: *Mapping.update*]]

lemma *update-Mapping* [*code*]:
fixes *t* :: (′*a* :: *ccompare*, ′*b*) *mapping-rbt* **shows**
Mapping.update k v (*Mapping m*) = *Mapping* (*m*(*k* ↦ *v*))
Mapping.update k v (*Assoc-List-Mapping al*) = *Assoc-List-Mapping* (*DAList.update k v al*)
Mapping.update k v (*RBT-Mapping t*) =
(*case ID CCOMPARE*(′*a*) *of None* ⇒ *Code.abort* (*STR* ′′*update RBT-Mapping*: *ccompare* = *None*′′) (λ-. *Mapping.update k v* (*RBT-Mapping t*))
| *Some* - ⇒ *RBT-Mapping* (*RBT-Mapping2.insert k v t*)) (**is** *?RBT*)
⟨*proof*⟩

declare [[*code drop*: *Mapping.delete*]]

lemma *delete-Mapping* [*code*]:
fixes *t* :: (′*a* :: *ccompare*, ′*b*) *mapping-rbt* **shows**
Mapping.delete k (*Mapping m*) = *Mapping* (*m*(*k* := *None*))
Mapping.delete k (*Assoc-List-Mapping al*) = *Assoc-List-Mapping* (*AssocList.delete k al*)
Mapping.delete k (*RBT-Mapping t*) =
(*case ID CCOMPARE*(′*a*) *of None* ⇒ *Code.abort* (*STR* ′′*delete RBT-Mapping*: *ccompare* = *None*′′) (λ-. *Mapping.delete k* (*RBT-Mapping t*))
| *Some* - ⇒ *RBT-Mapping* (*RBT-Mapping2.delete k t*))
⟨*proof*⟩

declare [[*code drop*: *Mapping.keys*]]

theorem *rbt-comp-lookup-map-const*: *rbt-comp-lookup c* (*RBT-Impl.map* (λ-. *f*) *t*) = *map-option f* ∘ *rbt-comp-lookup c t*
⟨*proof*⟩

lemma *keys-Mapping* [*code*]:
fixes *t* :: (′*a* :: *ccompare*, ′*b*) *mapping-rbt* **shows**
Mapping.keys (*Mapping m*) = *Collect* (λ*k*. *m k* ≠ *None*) (**is** *?Mapping*)
Mapping.keys (*Assoc-List-Mapping al*) = *AssocList.keys al* (**is** *?Assoc-List*)
Mapping.keys (*RBT-Mapping t*) = *RBT-set* (*RBT-Mapping2.map* (λ- -. ()) *t*) (**is** *?RBT*)
⟨*proof*⟩

declare [[*code drop*: *Mapping.size*]]

lemma *Mapping-size-transfer* [*transfer-rule*]:
includes *lifting-syntax*
shows (*pcr-mapping* (=) (=) ===> (=)) (*card* ∘ *dom*) *Mapping.size*
⟨*proof*⟩

lemma *size-Mapping* [*code*]:
fixes *t* :: (′*a* :: *ccompare*, ′*b*) *mapping-rbt* **shows**
Mapping.size (*Assoc-List-Mapping al*) = *size al*
Mapping.size (*RBT-Mapping t*) =
(*case ID CCOMPARE*(′*a*) *of None* ⇒ *Code.abort* (*STR* ′′*size RBT-Mapping*: *ccompare* = *None*′′) (λ-. *Mapping.size* (*RBT-Mapping t*))
| *Some* - ⇒ *length* (*RBT-Mapping2.entries t*))
⟨*proof*⟩

declare [[*code drop*: *Mapping.tabulate*]]
declare *tabulate-fold* [*code*]

declare [[*code drop*: *Mapping.ordered-keys*]]
declare *ordered-keys-def*[*code*]

declare [[*code drop*: *Mapping.lookup-default*]]
declare *Mapping.lookup-default-def*[*code*]

declare [[*code drop*: *Mapping.filter*]]
lemma *filter-code* [*code*]:
fixes *t* :: (′*a* :: *ccompare*, ′*b*) *mapping-rbt* **shows**
Mapping.filter P (*Mapping m*) = *Mapping* (λ*k*. *case m k of None* ⇒ *None* | *Some v* ⇒ *if P k v then Some v else None*)
Mapping.filter P (*Assoc-List-Mapping al*) = *Assoc-List-Mapping* (*DAList.filter* (λ(*k*, *v*). *P k v*) *al*)
Mapping.filter P (*RBT-Mapping t*) =
(*case ID CCOMPARE*(′*a*) *of None* ⇒ *Code.abort* (*STR* ′′*filter RBT-Mapping*: *ccompare* = *None*′′) (λ-. *Mapping.filter P* (*RBT-Mapping t*))

| *Some* - ⇒ *RBT-Mapping* (*RBT-Mapping2.filter* (λ(*k*, *v*). *P* *k v*) *t*))
⟨*proof*⟩

declare [[*code drop*: *Mapping.map*]]
lemma *map-values-code* [*code*]:
fixes *t* :: (′*a* :: *ccompare*, ′*b*) *mapping-rbt* **shows**
Mapping.map-values f (*Mapping m*) = *Mapping* (λ*k*. *map-option* (*f k*) (*m k*))
Mapping.map-values f (*Assoc-List-Mapping al*) = *Assoc-List-Mapping* (*AssocList.map-values f al*)
Mapping.map-values f (*RBT-Mapping t*) =
(*case ID CCOMPARE*(′*a*) *of None* ⇒ *Code.abort* (*STR* ′′*map-values RBT-Mapping*: *ccompare* = *None*′′) (λ-. *Mapping.map-values f* (*RBT-Mapping t*))
| *Some* - ⇒ *RBT-Mapping* (*RBT-Mapping2.map f t*))
⟨*proof*⟩

declare [[*code drop*: *Mapping.combine-with-key*]]
declare [[*code drop*: *Mapping.combine*]]

datatype *mapping-impl* = *Mapping-IMPL*
declare
mapping-impl.eq.simps [*code del*]
mapping-impl.rec [*code del*]
mapping-impl.case [*code del*]

lemma [*code*]:
fixes *x* :: *mapping-impl*
shows *size x* = *0*
and *size-mapping-impl x* = *0*
⟨*proof*⟩

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* - ⇒ *RBT-Mapping RBT-Mapping2.empty*
| *None* ⇒ *Assoc-List-Mapping DAList.empty*)

⟨*proof*⟩

definition *mapping-impl-choose2* :: *mapping-impl* ⇒ *mapping-impl* ⇒ *mapping-impl*
where [*simp*]: *mapping-impl-choose2* = (λ- -. *Mapping-IMPL*)

lemma *mapping-impl-choose2-code* [*code*]:
mapping-impl-choose2 x y = *mapping-Choose*
mapping-impl-choose2 mapping-Mapping mapping-Mapping = *mapping-Mapping*
mapping-impl-choose2 mapping-Assoc-List mapping-Assoc-List = *mapping-Assoc-List*
mapping-impl-choose2 mapping-RBT mapping-RBT = *mapping-RBT*
⟨*proof*⟩

definition *mapping-empty* :: *mapping-impl* ⇒ (′*a*, ′*b*) *mapping*
where [*simp*]: *mapping-empty* = (λ-. *Mapping.empty*)

lemma *mapping-empty-code* [*code*]:
mapping-empty mapping-Choose = *mapping-empty-choose*
mapping-empty mapping-Mapping = *Mapping* (λ-. *None*)
mapping-empty mapping-Assoc-List = *Assoc-List-Mapping DAList.empty*
mapping-empty mapping-RBT = *RBT-Mapping RBT-Mapping2.empty*
⟨*proof*⟩

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*

⟨*ML*⟩

declare [[*code drop*: *Mapping.empty*]]

lemma *Mapping-empty-code* [*code*, *code-unfold*]:
(*Mapping.empty* :: (′*a* :: *mapping-impl*, ′*b*) *mapping*) =
mapping-empty (*of-phantom MAPPING-IMPL*(′*a*))
⟨*proof*⟩

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*⟩

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 ⟨*proof*⟩
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 ⟨*proof*⟩
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 ⟨*proof*⟩
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 ⟨*proof*⟩
end

declare [[*code drop*: *Mapping.bulkload*]]
lemma *bulkload-code* [*code*]:
Mapping.bulkload vs = *RBT-Mapping* (*RBT-Mapping2.bulkload* (*zip-with-index vs*))
⟨*proof*⟩

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*⟩

Dynamic theorem list of rewrite rules that are applied after Transfer.transferred

⟨*ML*⟩

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*
⟨*proof*⟩

definition *map-apply* :: $('a \Rightarrow 'b$ *option*$) \Rightarrow 'a \Rightarrow 'b$ *option*
where [*code-unfold*]: *map-apply* = $(\lambda m.\ m)$

lemma *eq-map-apply*: $m\ x \equiv$ *map-apply* $m\ x$
⟨*proof*⟩

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.

⟨*ML*⟩

lemma *map-apply-parametric* [*transfer-rule*]:
((*A* ===> *B*) ===> *A* ===> *B*) *map-apply map-apply*
⟨*proof*⟩

lemma *map-apply-transfer* [*transfer-rule*]:
(*pcr-mapping A B* ===> *A* ===> *rel-option B*) *map-apply Mapping.lookup*
⟨*proof*⟩

definition *map-update* :: $'a \Rightarrow 'b$ *option* $\Rightarrow$ $('a \Rightarrow 'b$ *option*$)$ $\Rightarrow$ $('a \Rightarrow 'b$ *option*$)$
where *map-update x y f* = *f*(*x* := *y*)

lemma *map-update-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*A* ===> *rel-option B* ===> (*A* ===> *rel-option B*) ===> (*A* ===> *rel-option B*)) *map-update map-update*
⟨*proof*⟩

context begin
⟨*ML*⟩

lift-definition *update'* :: $'a \Rightarrow 'b$ *option* $\Rightarrow$ $('a, 'b)$ *mapping* $\Rightarrow$ $('a, 'b)$ *mapping*
is *map-update* **parametric** *map-update-parametric* ⟨*proof*⟩

lemma *update'-code* [*simp*, *code*, *code-unfold*]:
update' x None = *Mapping.delete x*
update' x (*Some y*) = *Mapping.update x y*
⟨*proof*⟩

end

declare *map-update-def*[*abs-def*, *containers-post*] *map-update-def*[*symmetric*, *containers-pre*]

definition *map-is-empty* :: $('a \Rightarrow 'b$ *option*$)$ $\Rightarrow$ *bool*
where *map-is-empty m* $\longleftrightarrow$ *m* = *Map.empty*

lemma *map-is-empty-folds*:
m = *map-empty* $\longleftrightarrow$ *map-is-empty m*
map-empty = *m* $\longleftrightarrow$ *map-is-empty m*
⟨*proof*⟩

declare *map-is-empty-folds*[*containers-pre*]
map-is-empty-def[*abs-def*, *containers-post*]

lemma *map-is-empty-transfer* [*transfer-rule*]:

assumes *bi-total A*
shows (*pcr-mapping A B* ===> (=)) *map-is-empty Mapping.is-empty*
⟨*proof*⟩

end

⟨*ML*⟩

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 *rexp*
derive *ceq rexp*
derive *ccompare rexp*
derive (*choose*) *set-impl rexp*

notepad begin
⟨*proof*⟩
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 *int*s 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 ⇒ vname set where
  vars (Var v) = {v}
| vars (Lit i) = {}
| vars (Add e1 e2) = vars e1 ∪ 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' ⇒ 'a set where
  vars' (Var' v) = {v}
| vars' (Lit' i) = {}
| vars' (Add' e1 e2) = vars' e1 ∪ 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.

[1] These type classes are only required for set complements (see §4.7.2).

[2] We 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.

[3] Technically, 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), $SET\text{-}IMPL('a)$ constrains the operation *set-impl*, (§4.3.3), $MAPPING\text{-}IMPL('a)$ (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 instantiates 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 ⟨*proof*⟩

instantiation simple-tycon :: ceq **begin**
definition CEQ(simple-tycon) = Some (=)
instance ⟨*proof*⟩
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 ⇒ ′b ⇒ bool **begin**
fun expr′-rel :: ′a expr′ ⇒ ′b expr′ ⇒ bool
where
  expr′-rel (Var′ v) (Var′ v′) ⟷ R v v′
| expr′-rel (Lit′ i) (Lit′ i′) ⟷ i = i′
| expr′-rel (Add′ e_1 e_2) (Add′ e_1′ e_2′) ⟷ expr′-rel e_1 e_1′ ∧ expr′-rel e_2 e_2′
| expr′-rel - - ⟷ False
end

If we give HOL equality as parameter, the relator is equality:

lemma expr′-rel-eq: expr′-rel (=) e_1 e_2 ⟷ e_1 = e_2
⟨*proof*⟩

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
⟨*proof*⟩
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 (⊆) 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, compare-class)
instance simple-tycon :: compare ⟨*proof*⟩

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.Phantom-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 ⟨*proof*⟩
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**
{} :: *expr set*	*RBT-set* (*Mapping-RBT Empty*)
empty :: (*expr*, *unit*) *mapping*	*RBT-Mapping* (*Mapping-RBT Empty*)
{} :: *string expr′ set*	*RBT-set* (*Mapping-RBT Empty*)
{} :: (*nat* ⇒ *nat*) *expr′ set*	*Set-Monad* []
{} :: *bool expr′ set*	*RBT-set* (*Mapping-RBT Empty*)
empty :: (*bool expr′*, *unit*) *mapping*	*Assoc-List-Mapping* (*Alist* [])

For *expr*, *mapping-Choose* picks RBTs, because *ccompare* provides a comparison operation for *expr*. For *′a expr′*, the effect of *set-Choose* is more pronounced: *ccompare* is not *None*, so neither is *ccompare*, and *set-Choose* picks RBTs. As *nat* ⇒ *nat* neither provides equality tests (*ceq*) nor comparisons (*ccompare*), neither does (*nat* ⇒ *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′* enherits 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 canoncial, see *Containers.Collection-Enum* for examples.

instantiation expr :: cenum **begin**
definition CENUM(expr) = None
instance ⟨*proof*⟩
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* ⇒ *'a option* ⇒ *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)
⟨*proof*⟩

lemma infinite-UNIV-expr: ¬ finite (UNIV :: expr set)
  including card-datatype
⟨*proof*⟩

instantiation expr :: card-UNIV **begin**
definition finite-UNIV = Phantom(expr) False
definition card-UNIV = Phantom(expr) 0
instance
  ⟨*proof*⟩
end

lemma inj-expr′ [simp]: inj Lit′   inj Var′   inj Add′   inj (Add′ e)
⟨*proof*⟩

lemma infinite-UNIV-expr′: ¬ finite (UNIV :: ′a expr′ set)
  including card-datatype
⟨*proof*⟩

instantiation expr′ :: (type) card-UNIV **begin**
definition finite-UNIV = Phantom(′a expr′) False
definition card-UNIV = Phantom(′a expr′) 0
instance
  ⟨*proof*⟩

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 ⟨*proof*⟩
end

instantiation expr' :: (ccompare) cproper-interval **begin**
definition cproper-interval-expr' :: 'a expr' proper-interval
  where cproper-interval-expr' - - = undefined
instance ⟨*proof*⟩
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 (*'a*, *'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 *bool*ans for the keys where the *bool*ean indicates whether we should go to the left or right child.

For brevity, we skip this step and rather assume that the type *'v trie-raw* of tries has following operations and properties:

type-synonym trie-key = bool list
axiomatization
  trie-empty :: 'v trie-raw **and**
  trie-update :: trie-key ⇒ 'v ⇒ 'v trie-raw ⇒ 'v trie-raw **and**
  trie-lookup :: 'v trie-raw ⇒ trie-key ⇒ 'v option **and**
  trie-keys :: 'v trie-raw ⇒ 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 ↦ 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 *bool*eans for the keys.

⟨*proof*⟩⟨*proof*⟩

4.4.2 Generalise the data structure

As (*'k*, *'v*) *mapping* store keys of arbitrary type *'k*, not just *trie-key*, we cannot use *'v trie-raw* directly. Instead, we must first convert arbitrary types *'k* into *trie-key*. Of course, this is not always possbile, 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 *ccompare*.

type-synonym 'a cbl = (('a ⇒ bool list) × (bool list ⇒ 'a)) option
class cbl =
  fixes cbl :: 'a cbl
  assumes inj-to-bl: ID cbl = Some (to-bl, from-bl) ⟹ inj to-bl
  and to-bl-inverse: ID cbl = Some (to-bl, from-bl) ⟹ from-bl (to-bl a) =
a
begin
abbreviation from-bl **where** from-bl ≡ snd (the (ID cbl))
abbreviation to-bl **where** to-bl ≡ 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 ⇒ 'b* does not.

instantiation unit :: cbl **begin**

definition cbl = Some (λ-. [], λ-. ())
instance ⟨*proof*⟩
end

instantiation fun :: (type, type) cbl **begin**
definition cbl = (None :: ('a ⇒ 'b) cbl)
instance ⟨*proof*⟩
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 ⊆ range (to-bl :: 'k ⇒ trie-key) ∨ ID (cbl :: 'k cbl) = None}
⟨*proof*⟩

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
 ⟨*proof*⟩

lift-definition lookup :: ('k :: cbl, 'v) trie ⇒ 'k ⇒ 'v option
 is λt. trie-lookup t ∘ to-bl ⟨*proof*⟩

lift-definition update :: 'k ⇒ 'v ⇒ ('k :: cbl, 'v) trie ⇒ ('k, 'v) trie
 is trie-update ∘ to-bl
 ⟨*proof*⟩

lift-definition keys :: ('k :: cbl, 'v) trie ⇒ 'k set
 is λt. from-bl ‘ trie-keys t ⟨*proof*⟩

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

⟨*proof*⟩

context
  fixes t :: ('k :: cbl, 'v) trie
  assumes ID-cbl: ID (cbl :: 'k cbl) ≠ None
begin

lemma lookup-update: lookup (update k v t) = (lookup t)(k ↦ v)
⟨*proof*⟩

lemma keys-conv-dom-lookup: keys t = dom (lookup t)
⟨*proof*⟩

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 ⇒ ('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* ⇒ (*unit* ⇒ *'a*) ⇒ *'a* with definition λ- *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*
⟨*proof*⟩

lemma update-Trie-Mapping [code]:
Mapping.update k v (Trie-Mapping t) =
(case ID cbl :: 'k cbl of
None ⇒ Code.abort (STR ''update Trie-Mapping: cbl = None'') (λ-. Mapping.update k v (Trie-Mapping t))
| Some - ⇒ Trie-Mapping (update k v t))
⟨*proof*⟩

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)
⟨*proof*⟩

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-foo A*, and you prove a specialised equation *foo* (*Trie-Mapping t*) = *trie-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 (∩), 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 mapping-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)
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

If your new data structure is better than the existing ones for some element type, you should reconfigure the type's preferene. 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
⟨*proof*⟩

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., (*'a*, (*'b*, *'c*) *mapping*) *mapping*. However, (*'a*, *'b*) *mapping* cannot currently be the key of a mapping, i.e., code generation fails for ((*'a*, *'b*) *mapping*, *'c*) *mapping*. Simiarly, you cannot have a set of mappings like (*'a*, *'b*) *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**
card-UNIV	§4.3.5	*HOL−Library.Cardinality*
cenum	§4.3.4	*Containers.Collection-Enum*
ceq	§4.3.1	*Containers.Collection-Eq*
ccompare	§4.3.2	*Containers.Collection-Order*
cproper-interval	§4.3.5	*Containers.Collection-Order*
finite-UNIV	§4.3.5	*HOL−Library.Cardinality*
mapping-impl	§4.3.3	*Containers.Mapping-Impl*
set-impl	§4.3.3	*Containers.Set-Impl*

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 delete the code equations involving *Complement*, the theorem list *set-complement-code* collects them. 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.

declare set-complement-code[code del]
code-datatype Collect-set DList-set RBT-set Set-Monad

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 $'a \Rightarrow 'b$. 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., (‘). If the element type is enumerable, provide an instance of *cenum* 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 $(\lambda i.\ i < 4)$ ` $\{i.\ 2 < i\}$, so this is a set comprehension over *int* rather than *bool*.

⟨*ML*⟩

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*: *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: {} = Set-Monad [] ⟨*proof*⟩

   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.